Technical Report CSTN-039

Simulating Large Random Boolean Networks

K.A. Hawick, H.A. James and C.J. Scogings
Institute of Information & Mathematical Sciences
Massey University – Albany
North Shore 102-904, Auckland, New Zealand
email: {k.a.hawick, h.a.james, c.scogings}@massey.ac.nz

20 May 2007

ABSTRACT

The Kauffman N-K, or random boolean network, model is an important tool for exploring the properties of large scale complex systems. There are computational challenges in simulating large networks with high connectivities. We describe some high-performance data structures and algorithms for implementing large-scale simulations of the random boolean network model using various storage types provided by the D programming language. We discuss the memory complexity of an optimised simulation code and present some measured properties of large networks.

KEY WORDS
random boolean network; time series analysis; high memory; simulation.

1 Introduction

The classical random boolean network (RBN) model was introduced by Kauffman to represent genetic regulatory networks [10,11]. This model is also commonly known as the N-K model or as Kauffman networks and consists of a network of N nodes, connected to K neighbours. Each node is decorated by a boolean state which is initialised randomly, and also a boolean function of K inputs, each of which is randomly assigned from one of the possible boolean functions of K inputs.

The model is studied by evolving the nodes synchronously, based on the state of their inputs. It is known that a transition occurs at $K = 2$ and that for smaller K the system rapidly moves to an attractor regime [9, 13] with nodes converging to a fixed value. Above the transition the nodes are chaotic, changing their values with no discernible pattern. Kauffman networks have been widely studied [5] and a variety of software tools are available for investigating small-sized networks [6]. The model can be thought of as an extension of the binary cellular automaton model of Wolfram [12], and indeed for $K = 1$ the model shows similar properties as can be seen in figure 1.

Figure 1 shows the time evolution of four random configurations of the classical random boolean network model, each with 128 nodes, connected with $K = 1, 2, 3, 4$. In the case $K = 1$ most nodes quickly become locked and no longer change. In the cases $K = 3$ and $K = 4$ the system remains chaotic. Case $K = 2$ is known to be at the critical edge of chaos [3].

The model has important uses in the study of networks and recently there has been interest in studying scale-free Kauffman networks [1]. Unlike the fixed-K normal Kauffman networks, a scale-free network has an exponential distribution of connectivities, so although most nodes may have a small connectivity, some can have large values. Due to the way N-K networks are constructed and stored there are several practical implementation problems associated with simulating large N-K networks. In this paper we explore these and investigate possible data structures and algorithms for managing such large scale simulations.

In section 2 we outline the core algorithms for constructing the structural network and the random boolean functions. We give a number of code fragments in the D programming language [2] to illustrate these ideas. In section 3 we present some performance analysis results and discuss some of the measured properties of large N-K networks in section 4. We offer some conclusions for the future simulation of large network models in section 5.

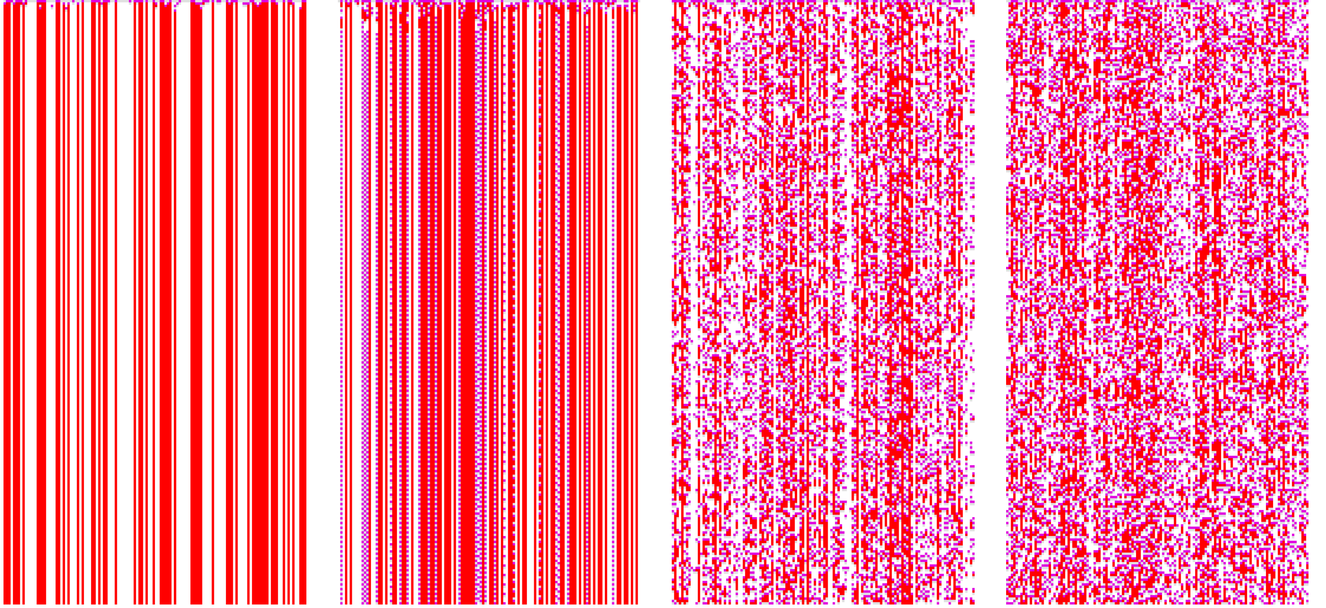

Figure 1: Classic random boolean network with 128 nodes (horizontally), evolved for 256 time steps (top to bottom) from a random initialisation, for $K = 1, 2, 3, 4$ (left to right). Red/white denotes sites of value 0/1 and mauve/grey denotes 0/1 sites that are changing respectively.

2 Simulation Algorithms and Experiments

The network is represented as an array of neighbour lists, and each of the N nodes is decorated by a boolean state and a random boolean function. The state is initialised randomly with 50 percent probability of being a zero or a one. A random boolean function is assigned statically (i.e. once) to each node.

The connectivity of the network itself can be stored in a number of ways. It is fundamentally a directed graph and a convenient storage structure makes use of the dynamic D arrays. Our `neighbours[i][n]` structure lists the n neighbours of the i'th node. D arrays have the '`length`' property which can be assigned a value to invoke an appropriate dynamic reallocation of memory (and copying of old values) as necessary. This structure is suited to fixed-K networks as we discuss here, or those with a distribution of K values, added dynamically.

The algorithm for assigning the network neighbours array so that each node has exactly K inputs is shown below. While this algorithm avoids self-arcs it does allow multi-arc neighbours – i.e. the same node can be picked as an input more than once.

```
int [][] neighbours;
neighbours.length = N;
for(int i=0;i<N;i++){
  neighbours[i].length = K;
  for(int n=0;n<neighbours[i].length;n++){
    int choice;  // avoid self-arcs
    do{
      choice = randomInt() % N;
    }while( choice == i );
    neighbours[i][n] = choice;
  }
}
```

The type system of the D programming language is somewhat stronger than that of C/C++ and is therefore helpful in developing less error-prone simulation codes. The critical data type required for the random boolean network simulation is that of a logical bit. The original D language proposal had a `bit` as a fundamental data type, but due to implementation difficulties that was unfortunately demoted. A `bool` in D is stored as a whole byte and is type-checked for boolean logical operations. An `int` is the usual way C and (old fashioned) C++ programs implement a logical bit. The standard library accompanying D, called 'Phobos', comes with a `BitArray` data structure that has appropriate implementations of the `opIndex` and `opAssignIndex` operators that make the conventional array indexing syntax with square brackets(`[ ]`) work as expected.

The efficiency tradeoffs are obvious: an `int` or `uint`

is the same size as the machine architecture register width and is the fundamental unit of addressing, albeit wasting 31 bits on a 32-bit architecture; a `bool` stored as a `byte` wastes only 7 bits of storage but in a modern machine it may be almost as fast to address as a machine word. A packed data structure such as the `BitArray` is fully efficient as regards memory utilisation but requires extra addressing arithmetic and bit-shifting operations to insert/extract individual logical bit values.

The code fragments below are written in terms of a type definition for `bits_t` that can be one of the following type definitions:

```
typedef int[] bits_t;

typedef bool[] bits_t;

typedef BitArray bits_t;
```

The performance consequences of using these different bit storage representations are discussed in section 3.

The random boolean functions of K-inputs are represented by a truth table. An example encoding of this for the simple case of $K = 1$ and $K = 2$ are shown in tables 1 and 3. Each column of the table represents the complete specification of a possible boolean function of K inputs.

Although modern programming languages such as D do support higher dimensional arrays, it would be inefficient to use a separate programming dimension '`[ ][ ][ ][ ]...`' of the truth table for each of the K boolean inputs i_n, so they are encoded as an integer $k = i_0 \times 2^0 + i_1 \times 2^1 + i_3 \times 2^2 + ... + i_n \times 2^n$ for $n = 0, 1, 2, ..., K - 1$ and implemented as:

```
// compose bits_t vector to a k-coding:
uint bits_to_k( bits_t bits ){
  int nBits = bits.length;
  uint k = 0;
  for(int n=0;n<nBits;n++)
    if( bits[n] )
      k += powersOf2[b];
  return k;
}
```

We can also write the inverse function to decode an integer into its bits, which is implemented as:

```
// decompose k-coding to nBits-length
// bits_t vector:
bits_t k_to_bits( uint k, int nBits ){
  bits_t bit; bits.length = nBits;
  for(int n=0;n<nBits;n++){
    bits[n] = (k & powersOf2[n]) != 0;
  }
  return bits;
}
```

Input	Output			
i_0	o_{00}	o_{01}	o_{02}	o_{03}
0	0	1	0	1
1	0	0	1	1

Table 1: Truth table of the $2^{2^K} = 2^2 = 4$ random boolean functions for the example case $K = 1$.

K	Input States 2^K	Possible Boolean Functions $N_{\text{BF}} = 2^{2^K}$
1	2	4
2	4	16
3	8	256
4	16	65,536
5	32	4,294,967,296
6	64	18,446,744,073,709,551,616

Table 2: Exponential growth of the number of inputs states and number of possible boolean functions with connectivity K.

There are other syntactic ways to encode these functions in C/C++ or D, but these forms are type-safe against any of the three D typedefs described above.

Using these function primitives, it is easy to construct each column of the truth table using a bit representation of j. `TT[j][k]` is the truth table for the j'th boolean function and the k'th encoded bit-vector of the input bits and is implemented as:

```
for(int j=0;j<nBFuncs;j++){
  TT[j] = k_to_bits( j, nBits );
}
```

The update algorithm for the random boolean network model of N nodes, numbered $i = 0, 1, 2, ..., N - 1$ is then:

```
1  bits_t newState, swapState, bits;
2  newState.length = N;
3  bits.length = maxK;
4  for(int step=0;step<nSteps;step++){
5    for(int i=0;i<N;i++){
6      for(int n=0;n<neighbours[i].length;n++){
7        bits[n] = state[ neighbours[i][n] ];
8      }
9      newState[i] =
10       TT[ bFunc[i] ][ bits_to_k(bits) ];
11   }
12 }
```

This uses a synchronous update whereby a new state vector is built at each time step and all N nodes are updated effectively together at once through the use of the swapping of state arrays `newState` and `state`. As

k	Input		Output															
k	i_1	i_0	o_{00}	o_{01}	o_{02}	o_{03}	o_{04}	o_{05}	o_{06}	o_{07}	o_{08}	o_{09}	o_{10}	o_{11}	o_{12}	o_{13}	o_{14}	o_{15}
0	0	0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
1	0	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
2	1	0	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
3	1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1

Table 3: Truth table of the $2^{2^K} = 2^4 = 16$ random boolean functions for the example case $K = 2$. The $K = 2$ inputs i_1 and i_0 are encoded as an integer k which can index into the `TT[j][k]` to give the boolean output value o for the j'th boolean function's truth table, where $j = 0, 1, ...2^{2^K}$.

might be expected there are various sweeping effects and other artifacts that arise if a non-synchronous update algorithm is used [8].

Our code is written in terms of an integer lookup `bFunc[i]` that specifies which of the possible boolean functions is assigned to node i. The number of inputs states grows as 2^K with the number of inputs K. Table 2 shows that the number of possible boolean functions $N_{\mathrm{BF}} = 2^{2^K}$ grows very rapidly and that it is not feasible to sample them all in any realisable network simulated on affordable computers except for K smaller than around 5.

If the number of possible boolean functions ($N_{\mathrm{BF}} = 2^{2^K}$) is very much greater that the number of nodes (N) then it is more effective to assign a truth table entry to each node directly and index `TT[i]` directly. In this case, `TT` is set up accordingly and the algorithm lines 9 and 10 are replaced by the line:

```
newState[i] = TT[ i ][ bits_to_k(bits) ];
```

We can encode our simulation to deal with $K = 5$ by using 64-bit unsigned integers for calculations involving powers of two. We can in fact extend this to $K = 6$ *providing* we accept we are not uniformly sampling all possible boolean functions.

The D language has provision for a 128-bit unsigned integer (the `ucent`), but this is not yet supported by available distributions [4, 7]. We have been able to obtain timings for our random boolean network simulation up to values of $K = 1, 2, 3, 4, 6$ with the proviso that although the data structures are correct and therefore the timings are correctly representative, the sample space of possible boolean function is truncated for $K = 6$.

3 Performance Results and Analysis

For most of the sizes of system reported here we can quite easily use simple integers or bytes for the states rather than packed bits, as memory is typically dominated by the storage space required to specify the random boolean function at each node. The truth tables for the random boolean functions occupy occupy 2^{2^K} storage elements. In general the number of nodes N in our network is less than the possible number of boolean functions which grows very rapidly with K. In particular for $K = 5$, $N_{\mathrm{BF}} = 2^{2^5} = 2^{32}$ which presents interesting problems for programs running on 32-bit architectures.

Phobos provides a `BitArray` structure which can conveniently be used to make packed bit representations of the `state` and `TT` arrays. Using the overloaded [ ] array indexing operators in the D programming language, the `BitArray` can be used as a direct code substitute for any other typed array representation for the bits. Memory is then no longer a concern, but the manipulation of the k indices as 32- or 64-bit fixed entities still poses a limitation on our code. Table 4 shows some timing analyses for the random boolean network simulation code on various platforms.

As table 4 indicates, the use of the `bool` data type is fastest in most cases for "medium" memory utilisation. We carried out these timing experiments on a variety of platforms and generally for "normal" program memory utilisation of less than 2GBytes the timings show a practical time savings using the direct single byte `bool` type. At the time of writing 64-bit operating systems support is becoming available and does work, but does not necessarily have appropriate cache and page management sizes.

In table 5 we show that for large systems that explore more than the 31-bit signed-integer addressable range of memory (up to 2GBytes) the timing reflects the architecture's cache size and page memory management system rather than being directly indicative of the simulation algorithm. In these cases the packed storage `BitArray` shows a marked improvement and does allow us to usefully simulate large systems of up to $N \approx 10^7$ network nodes.

We can compute the memory complexity of the simulation algorithm and verify that the platform is genuinely allocating the correct amounts of (packed-bits) memory by recording the resident memory size as reported by a

N	K	bit_t	Time	RSIZE
128,000	1	int[]	1.2	6.1
	1	bool[]	1.1	4.9
	1	Bitarray	2.9	4.7
128,000	2	int[]	2.0	5.7
	2	bool[]	1.9	4.9
	2	BitArray	4.2	5.1
128,000	3	bool[]	2.4	4.9
	4	bool[]	5.0	9.5
	5	bool[]	5.1	15.0
	6	bool[]	7.0	23.0
1,280,000	1	bool[]	16.1	43.4
	2	bool[]	25.6	43.4
	3	bool[]	30.8	43.4
	4	bool[]	68.6	66.2
	5	bool[]	66.9	144
	6	bool[]	86.8	225
1,280,000	1	int[]	31.8	50.9
	2	int[]	47.2	50.9
	3	int[]	62.0	50.9
	4	int[]	122.2	80.0
	5	int[]	121.8	394
	6	int[]	147.3	717
1,280,000	1	BitArray	30.9	41.0
	2	BitArray	48.7	41.0
	3	BitArray	64.9	41.0
	4	BitArray	98.8	62.8
	5	BitArray	109.9	81.4
	6	BitArray	121.0	81.4
2,560,000	1	bool[]	43.7	86.1
	2	bool[]	67.7	86.1
	3	bool[]	83.5	86.1
	4	bool[]	183.5	129
	5	bool[]	171.7	287
	6	bool[]	219.8	449

Table 4: Timing measurements of the random boolean network simulation code for **256 steps**, using GNU gdc D Compiler Version 0.23, with optimization level -O6 set. Timing is measured in seconds on a 2.66GHz 64-bin Xeon with 4GB of available RAM. RSIZE is the resident memory size, as reported by the Unix top utility, measured in MB.

N	K	bit_t	Time	RSIZE
5,120,000	1	bool[]	9.3	171
	2	bool[]	12.3	171
	3	bool[]	15.1	171
	4	bool[]	24.9	254
	5	bool[]	48.8	575
	6	bool[]	83.5	898
10,240,000	1	bool[]	24.4	342
	2	bool[]	31.0	342
	3	bool[]	38.6	342
	4	bool[]	61.6	506
	5	bool[]	172.4	1,120
	6	bool[]	325	1,750
20,480,000	1	bool[]	65.0	683
	2	bool[]	78.0	683
	3	bool[]	97.1	683
	4	bool[]	165.9	1,008
	5	bool[]	737.6	>2,000
	6	bool[]	1437	>3,000
20,480,000	1	BitArray	63.0	648
	2	BitArray	77.9	648
	3	BitArray	92.4	648
	4	BitArray	168.4	973
	5	BitArray	306.1	1,260
	6	BitArray	325.7	1,260

Table 5: Timing measurements of the random boolean network simulation code for **8 steps**, using GNU gdc D Compiler Version 0.23, with optimization level -O6 set. Timing is measured in seconds on a 2.66GHz 64-bin Xeon with 4GB of available RAM. RSIZE is the resident memory size, as reported by the Unix top utility, measured in MB.

systems monitoring tool such as the Unix `top` utility.

Variable Name	Number of Elements	Data type
`state`	N	`bit_t`
`newState`	N	`bit_t`
`neighbours`	$N \times K$ N	`int` `void *`
`TT`	$N \times 2^K$ N	`bit_t` `void *`

Table 6: Memory complexity of the random boolean network simulator code. For `neighbours` and `TT` it is necessary to consider the all pointers required to efficiently implement a two-dimensional array.

The memory utilisation is shown in table 6. In the case in which `bit_t` is implemented as an `int` we have $N + N + N.K + N.2^K + N = 4N + NK + 2^K$ and if $K = 2$, we have 10N `words` which requires 51MBytes (assuming a 4-byte machine word size). This is consistent with the `RSIZE` values reported in table 4 and 5.

4 Some RBN Properties

As shown in figure 1, the random boolean network model has a varying degree of autocorrelation in time. Nodes' states will lock to fixed values below a critical value of K and will remain highly chaotic above it. A simple measure of the regime is given by the first-order correlation moment, which can be simply measured by:

```
int nSame = 0;
for(int i=0;i<N;i++)
  if( state[i] == newState[i] ) nSame++;
```

Other moments of the auto-correlation function can be computed to obtain a time-series analysis that identifies the set of attractors present in the network. For a statistically meaningful analysis we need to consider how the system self-averages. Ideally we would require to sample over completely independently generated large networks that individually properly sample all possible boolean functions. This is not computationally feasible for $K > 4$. Nevertheless it is feasible to perform reasonably unbiased sample of possible boolean functions for $K = 5$ that may be representative of the large scale bulk network behaviour.

The results of this are shown in figure 2. At $K < 2$ the nodes tend to order, fairly rapidly, whereas for $K > 2$ the system tends to remain decorrelated to an extend dependent upon K. For higher K the tendency is for more and more nodes to remain changing. The curves shown

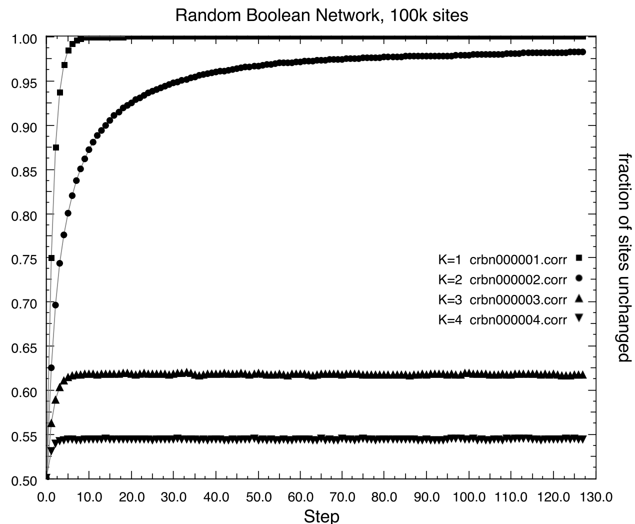

Figure 2: The 1-step correlation fraction for an RBN for different connectivities $K = 1, 2, 3, 4$ over 128 steps.

are reasonably smooth and show convergence to definite values. This is due to the large network size, relative to the number of possible boolean functions.

Figure 3 shows the cluster size distribution for a system generated with 128,000 nodes for $K = 1$. At higher K values the system is fully connected with a single cluster. Using this algorithm at $K = 1$ however, there is a distribution of islands of nodes, clustered as shown. A log-log plot reveals that cluster size population goes as cluster size s to the power of ≈ -1.5.

5 Discussion and Conclusions

We have shown how relatively large scale random boolean networks can be simulated for $K = 1, ..., 5$ and that a suitable data structure can be managed for $K = 6$ providing some means of generating unbiased and representative samples of the possible boolean functions can be provided. One such mechanism is to store a copy of the particular K-input boolean function for each network node. An arbitrary precision arithmetic package could then be used to generate the (long) look-up tables required for a high connectivity node. This will be feasible to implement in memory providing most nodes have a relatively small connectivity, K. This then provides us with a platform for investigating scale-free networks, where the distribution of connectivities is indeed exponential in K.

We have shown the memory complexity and typical performance achieved using various bit storage types in

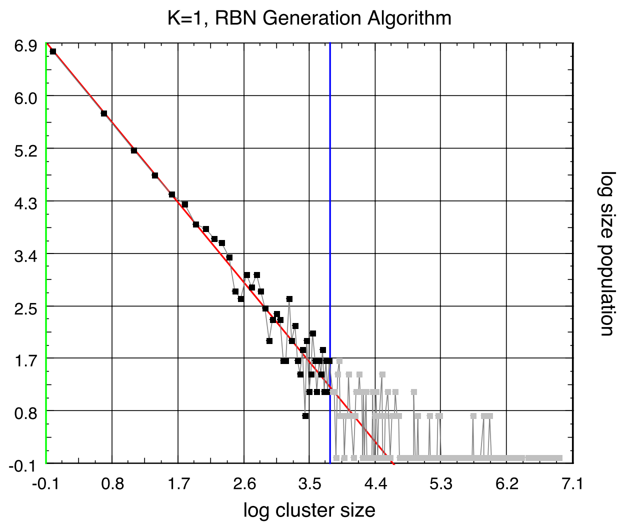

Figure 3: The cluster size distribution for a $K = 1$ network, generated using the algorithm described.

the D programming language. We believe this language has great promise for numerical simulations such as the study of random boolean networks.

Acknowledgments

This work has benefited from the use of Massey University's "Monte" compute cluster and thanks also to the Allan Wilson centre for Molecular Biology for the use of the "Helix" Supercomputers.

References

[1] Aldana, Maximino, Boolean dynamics of networks with scale-free topology, Physica D 185 (2003) 45-66.

[2] The D Programming Language, http://-www.prowiki.org/wiki4d/wiki.cgi?-LanguageSpecification, accessed May 2007.

[3] Derrida, B., and Pomeau, Y., Random Networks of Automata: A simple annealed approximation, Europhys. Lett. 1(2) 45-49, 1986.

[4] Digital Mars, D Compiler, http://www.digitalmars.-com/d/, accessed May 2007.

[5] Gershenson, Carlos, Introduction to Random Boolean Networks, arXiv:nlin/0408006v3 12 August, 2004.

[6] Gershenson, Carlos, RBNLab, http://homepages.-vub.ac.be/˜cgershen/rbn/, accessed May 2007.

[7] GNU D Compiler, http://dgcc.sourceforge.net/, accessed May 2007.

[8] Harvey, I. and Bossomaier, T., Time out of joint: Attractors in Asynchronous Random Boolean Networks, In Proc Fourth European Conference on Artificial Life (ECAL97), pp67-75, MIT Press, ed. P. Husbands and Harvey, I., 1997.

[9] Kadanoff, Leo, Coppersmith, Susan and Aldana, Maximino, Boolean Dynamics with Random Couplings, arXiv:nlin/0204062v2, 2002.

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

[11] Kauffman, S. A., The Origins of Order, Oxford University Press, 1993.

[12] Wolfram, Stephen, Theory and Applications of Cellular Automata, Pub. World Scientific, 1986.

[13] Wuensche, A., Discrete Dynamical networks and their attractor basins, In Proc. Complex Systems 1998 pp3-21. ed R.Standish et al., UNSW Sydney, Australia.