

*The genes are the master programmers, and they are programming for their lives.*  
— Richard Dawkins, *The Selfish Gene* (1976)



# Abstract

Random Boolean Networks are a generalisation of binary Cellular Automata, without a fixed topology. This thesis presents an RBN implementation using an instruction-based approach, and compares this to a traditional table-based approach. The implementations are used to evolve RBNs with maximum attractor lengths, in order to investigate the evolvability and the usefulness of an instruction-based implementation. The results show limited usefulness for  $K = 2$ , but the instruction-based implementation performs significantly better for  $K = 3$ . The instruction-based implementation is slower than the table-based implementation by a factor of  $\sim 10$ , but areas of improvement have been identified and discussed.

# Sammendrag

Tilfeldige Boolske Nettverk er en generalisering av binære Cellulære Automater, uten en fast topologi. Denne oppgaven presenterer en TBN-implementasjon med en instruksjonsbasert tilnærming, og sammenlikner denne med en tradisjonell tabellbasert tilnærming. Implementasjonene brukes til å evolvere TBN-er med maksimale attraktorlengder, for å undersøke evolverbarheten og nytteverdien av en instruksjonsbasert implementasjon. Resultatene viser begrenset nytteverdi for  $K = 2$ , men den instruksjonsbaserte implementasjonen yter vesentlig bedre for  $K = 3$ . Den instruksjonsbaserte implementasjonen er tregere enn den tabellbaserte implementasjonen med en faktor på  $\sim 10$ , men forbedringsområder blir identifisert og diskutert.



# Acknowledgements

First of all, I would like to thank my advisors, Gunnar Tufte, Stefano Nichele and Odd Rune Strømmen Lykkebø for their guidance and encouragement, especially after a long period of illness.

I would also like to thank Terje Schjelderup, Einar Johan Trøan Sømåen and Christer Bru for inspiring and productive discussions on the topic of RBNs, CAs and cellular computation-related knitting patterns. Finally, I would like to thank my family for their support, and of course, Jean Niklas L'orange for always being there for me. I don't think I could have done this without you.



# Contents

<b>Abstract</b>	<b>a</b>
<b>Acknowledgements</b>	<b>c</b>
<b>Contents</b>	<b>i</b>
<b>List of Tables</b>	<b>ii</b>
<b>List of Figures</b>	<b>ii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Motivation . . . . .	1
1.2 Terminology . . . . .	2
1.2.1 Cellular Automata and Random Boolean Networks . . . . .	2
1.2.2 Evolution and Evolutionary Algorithms . . . . .	2
<b>2 Background</b>	<b>4</b>
2.1 Random Boolean Networks . . . . .	4
2.2 Cellular Automata . . . . .	5
2.3 Instruction-Based and Table-Based Development . . . . .	6
2.4 Evolution and Evolutionary Algorithms . . . . .	6
2.4.1 Natural and Artificial Evolution . . . . .	7
2.4.2 Reproduction . . . . .	7
<b>3 Methodology</b>	<b>10</b>
3.1 Table-based RBN . . . . .	10
3.2 Instruction-based RBN . . . . .	11
3.3 Evolutionary strategy . . . . .	12
3.4 Testing . . . . .	14
3.5 Setup . . . . .	14
<b>4 Results and Discussion</b>	<b>15</b>
4.1 Table-based RBN . . . . .	15
4.2 Instruction-based RBN . . . . .	15
4.3 Running time . . . . .	21
<b>5 Conclusion</b>	<b>23</b>
5.1 Further work . . . . .	23

## List of Tables

---

2.1	Transition table for Rule 110 . . . . .	5
-----	---	---

## List of Figures

---

3.1	Structural permutation of two 2-input Boolean functions . . . . .	12
3.2	The implemented configuration . . . . .	12
3.3	Distribution of 3-input Boolean functions through combinations of two 2-input Boolean functions . . . . .	13
4.1	Fitness values for $K=2$ . . . . .	17
4.2	Fitness values for $K=3$ . . . . .	19





# CHAPTER 1

## Introduction

The natural world, with all its fascinating complexities, has, to the best of our knowledge, made itself happen, through simple processes on simple building blocks. Evolution is elegant in its simplicity — it has no end goal, no overarching plan, and still it has resulted in creatures such as ourselves. To result in such complexity, these methods must be very powerful in their simplicity, and evolutionary computation seeks to harness this power.

Dawkins, in his book “The Blind Watchmaker”, counts computers as honorary biological objects, because they derive their complexity and design from living objects[1, pp. 1-2]. With this in mind, applying biological principles to electronic computing is not quite so far-fetched.

A well-known example is the NASA *evolved antenna* project[2]. While traditionally designed antennae typically require a fair amount of work and specialised knowledge, evolved antennae do not. The design process is automated once the requirements and design constraints have been set. Typically, evolved antennae have asymmetrical designs that nonetheless conform very well to specification. This is particularly useful for projects that have unusual requirements, where the design would require a lot of specialised knowledge and take up a lot of time. Once the requirements have been described in sufficient detail, a fitness function to evaluate the candidates according to the requirements must be made, and the simulation is run.

Random Boolean Networks were originally introduced as a method of modelling genetic regulatory networks [3], but have also found use as models in a wide range of areas, such as mathematics, music, sociology and evolutionary theory[4]. Improving the methods for finding suitable RBNs may provide more and better solutions to these problems.

### 1.1 Motivation

The goal of this thesis is to compare an instruction-based Random Boolean Network (RBN) implementation to a traditional table-based implementation in order to investigate whether the instruction-based implementation improves timing and performance.

RBN evaluation is costly if the entire state space for each network is evaluated. Applying an evolutionary algorithm to finding a network that behaves in a certain way involves a large search space,  $\left(2^{2^K} \times N^K\right)^N$ , where  $K$  is the number of input nodes each node takes, and  $N$  is the number of nodes in a network. With instruction-based RBN, the hope is to reduce the  $2^{2^K}$  term without compromising performance. This term represents the number of possible Boolean operators, and even a small reduction would reduce the search space considerably.

Instruction-based methods have already been successfully applied to Cellular Automata [5], and the hope is that this may prove useful for RBN. In this thesis, this principle is applied to RBN construction and evolution, with the goal of studying whether this is an useful improvement over the traditional table-based implementation. The main difference between the two cases is that each node in an RBN uses a randomly chosen Boolean function, and that the cell states are binary.

## 1.2 Terminology

### 1.2.1 Cellular Automata and Random Boolean Networks

*State space* is the finite set of all possible states an RBN or CA may have. The state space of a CA is  $k^n$ , where  $k$  is the number of cell states, and  $n$  is the size of the lattice. The state space of an RBN consists of  $2^N$  states, where  $N$  is the number of nodes.

The *neighbourhood* of a cell in a CA consists of the set of cells whose states determine its new state. Neighbourhoods are within a certain distance, or radius  $r$ , from the central cell, and are identical in size for all cells in a CA. The set of *input nodes* in an RBN serves the same purpose.

The *transition rule* of a cellular automaton is analogous to the *Boolean operator* of a node in an RBN. It describes the new state of the cell or node as determined by the states of the neighbourhood or the input nodes.

An *attractor* is reached when the network reaches a stable state, and no new states are visited. A *point attractor* is a state that leads to itself, while a *cycle attractor* is a repeating set of states. The set of all states leading to an attractor is called an *attractor basin*. For a deterministic RBN, each state is in exactly one attractor basin.

A *Garden of Eden* state is a state that is not reachable by another state in the state space.

### 1.2.2 Evolution and Evolutionary Algorithms

The set of genetic information belonging to an organism is called the *genotype*. The genotype contains sufficient information to create the individual organism, and the

next generation inherits the genotype of the previous generation. A *genome* is a specific instance of a genotype.

The *phenotype* is the observable characteristics of an individual, and is a manifestation of its genotype. Selection mechanisms in the natural world and in EAs operate on the phenotype of an organism.

A *mutation* is a small random change in the genotype. It may or may not affect the phenotype of an organism.

*Fitness* is a measure of how well an individual fits to the given specification. An individual with a high fitness value is more likely to reproduce.

## CHAPTER 2

# Background

### 2.1 Random Boolean Networks

Random Boolean Networks (RBN), also known as N-K models or Kauffman networks, were first proposed by Stuart Kauffman in 1969 as a tool for modelling genetic regulatory networks[3]. An RBN is a randomly selected network of  $N$  nodes, with  $K$  inputs from other nodes each (including the node itself). Each node has two states, zero or one. New states at time  $t + 1$  are calculated with Boolean operator for that node and the state of the  $K$  input nodes at time  $t$ . Thus, RBNs can be seen as a generalisation of cellular automata (CA) [4]. As well as examining networks with  $K = 2$  and  $K = 3$ , as we will in this paper, Kauffman briefly discussed one connected nets (RBNs with  $K = 1$ ), and totally connected nets ( $K = N$ , also known as random maps) both of which he considered unrealistic and biologically impossible.

There are several types of Random Boolean Networks. Most of the literature studying RBNs deals with homogeneous RBNs, where  $K$  is constant for the network, all nodes having the same number of inputs. Gershenson [6] classified various types of homogeneous RBNs. First, there are Classical Random Boolean Networks (CRBNs), which are those described by Kauffman, where each node in the network updates synchronously with all the other nodes. Gershenson mentions Asynchronous Random Boolean Networks (ARBNS), where the updating of each node is random and asynchronous. These networks are non-deterministic. Gershenson goes on to describe Deterministic ARBNs (DARBNS), an adaptation where node updates are not random, but still asynchronous, Deterministic Generalised ARBNs, where all nodes that fulfil their updating condition are updated synchronously, and then, for completeness Generalised ARBNs, where these nodes are picked randomly instead of having a deterministic update condition.

Deterministic Random Boolean Networks have finite state spaces, and being deterministic, will eventually reach a state that will be repeated. These are called *attractors*. There are two types of attractors, *point attractors*, where a state leads to itself in the next time step, and *cyclic attractors*, where a number of states are repeated in sequence. Should a network be left to run once it has entered a cyclic attractor, it will continue this cycle indefinitely. States that lead to a given attractor can be said to be in that *attractor basin*.

RBNs may be used to model biological processes, even when these biological processes are not fully understood, providing a mechanistic model of the “black box”[7]. Huang [8] suggested that Boolean genetic networks may provide good models of “gene function” and improve understanding on the origins of neoplasia (tumour growth).

## 2.2 Cellular Automata

The main difference between Cellular Automata (CA) and Random Boolean Networks is that CAs have a fixed topology, and may have more than two cell states. Typically, the CAs studied are also homogeneous, meaning that all cells have the same *transition rule*. The transition rule is analogous to the Boolean operator in RBNs, and describes the behaviour of the cell in relation to the states of the cell itself, and the cells in its *neighbourhood*. When describing CAs,  $k$  is typically used to describe the number of states a cell may have, and  $N$  describes the neighbourhood. This may be confusing when discussing both RBNs and CAs.

The topology of a CA is determined by its *dimension*. In a 1D CA, each cell has neighbours only in one direction, i.e. to its left and right. For a 2D CA, the cell has neighbours in two directions. In this case, the neighbourhood may include only the direct neighbours of a cell (a von Neumann neighbourhood), or the direct neighbours as well as the diagonal neighbours (a Moore neighbourhood)- For all types, a neighbourhood radius  $r$ , determines the number of cells included in the neighbourhood as an expression of distance from the central cell.

The simplest type of cellular automata are called the *elementary cellular automata*, as described by Wolfram in 1983 [9]. These are one-dimensional, binary, homogeneous, and have the smallest type of neighbourhood, a nearest neighbourhood of  $N = 3$  (the cell itself, and the cells adjacent to it). There are 256 such elementary cellular automata, each with a different *rule*. These rules are numbered from 0 to 255, interpreting the transition rule result table (in descending order) as a  $k^N$ -digit binary number, expressed in decimal. This is known as a *Wolfram code*[10]. For example, the table for Rule 110 can be seen in Table 2.1,  $01101110_2 = 110_{10}$ . As a point of interest, Rule 110 is notable for being Turing complete.[11]

111	110	101	100	011	010	001	000
0	1	1	0	1	1	1	0

Table 2.1: Transition table for Rule 110

A well-known example of a CA is known as Conway’s Game of Life, or simply, “Life”<sup>1</sup>. “Life” is a two-dimensional (2D) binary CA with a set of rules that define whether a cell is “dead” or “alive”, in order to simulate life[10]. “Life” can produce patterns that self-replicate[13], and is Turing-complete[14].

<sup>1</sup>Early on, “Life” was considered an interesting mathematical solitaire game, and the suggested playing method used a chequerboard or Go board and flat chips of two colours[12]

## 2.3 Instruction-Based and Table-Based Development

The traditional way of implementing RBNs and CAs uses tables to specify the new node or cell state for all possible combinations of neighbourhood or input node states. By instead using an instruction-based method, as introduced by Bidlo and Vašíček[5] for CAs, the hope is to produce better results in a shorter period of time.

By using evolutionary algorithms, the search space for all combinations of CAs and RBNs with given specifications becomes a significant hurdle. Particularly large for non-binary CAs, the transition function needs to be defined for  $k^N$  different combinations for  $k$  cell states with a neighbourhood of  $N$ . For the CA, there are  $k^{k^N}$  possible transition functions. It is clear that the search space becomes very large, and fast. Bidlo and Vašíček showed that, particularly for non-binary complex CAs, an instruction-based implementation allowed them to “design complex cellular automata with higher success rate than the conventional table-based method”[5].

An instruction-based approach defines a set of instructions that are combined in order to represent the transition function, rather than doing so by generating the entire transition table for each function. It allows for a shorter genotype, since the transition table does not need to be encoded directly, and the search space is reduced considerably. There is still the problem of finding the right number of instructions to describe the best transition function, but using evolutionary growth has been proposed as a solution. [15]

## 2.4 Evolution and Evolutionary Algorithms

*Evolutionary algorithms* (EAs) are algorithms that are inspired by the principle of evolution in the natural world. These are usually based on the key points of natural evolution: Maintaining the population, creating diversity, having a selection mechanism, and implementing genetic inheritance [10]. EAs thus attempt to exploit the mechanism that caused the speciation of life on Earth.

To facilitate progressive development, an EA must implement a mechanism of change to create diversity. Normally, this means implementing some form of mutation. A *mutation* is a change in the genome, and it occurs randomly. Mutation is one of the mechanisms of change in the natural world, and increases genetic diversity. However, not all mutations are viable, nor are they necessarily useful, and mutation alone does not account for progressive change.

The mutations in the genome are essentially random, but their survival to the next generation is not. For this reason, EAs must implement a method to select a subset of the population for reproduction or propagation. This is done by implementing a fitness function, to select the members of the population that fit the requirements and purpose best. The fitness function evaluates the individual phenotype of each member of the population and compares it to the specified ideal, and this value is then used by the reproduction or propagation

The fact that the fitness function evaluates the *phenotype* rather than the *genotype* of an individual means that it does not need to know exactly how a successful individual is put together, only how it behaves. This is why evolutionary methods can be used to find solutions even for complex requirements, as the methods only evaluate a “black box”, rather than the inner workings. It then finds the inner workings that fit the black box model best.

### 2.4.1 Natural and Artificial Evolution

*Natural selection* as a selection method favours the “fittest” members of a population, but this fitness evaluation can not be said to be static. Rather, it is a product of the environment around the individual, as this determines whether the individual survives long enough to reproduce, and how frequently it does so.

Natural selection is neither tractable nor useful for evolutionary computation in electronic media, and is much too complex to implement. An adaptive approximation may be sought for purposes of biological study, but would be too resource-intensive for general computational purposes.

Evolution in the natural world also has other mechanisms of change besides natural selection and mutation; *migration* and *genetic drift* [16]. The closest example of the implementation of migration is in so-called *island models*, where diverse populations evolve in parallel, rather than in a single group [10]. Two groups of the same species on two different islands with limited contact will eventually develop differently, as in the famous example of the Galápagos finches<sup>2</sup>. Genetic drift is a random phenomenon that does not lead to increased adaptation, and is of limited usefulness as it may eliminate the individuals that are close to the desired end result.

Thus, a common solution is to implement a static fitness function as a selection method. Unlike evolution through natural selection, this means that there is a fixed end goal. This approach to evolutionary methods may be less powerful than natural evolution, as it limits us to what programmers can conceive as an ideal end result. However, this does not mean that artificial evolutionary methods are not very useful. Just because the goal must be formulated by a human, does not mean that the ideal solution needs to conform to human constraints and pattern ideas.

### 2.4.2 Reproduction

There are several ways of maintaining a population for reproduction and evolution. The EA must have a number of individual genomes to select from, and a sufficiently

---

<sup>2</sup>Also known as Darwin’s finches, which he wrote about in *On the Origin of Species*, in the chapter on geographical distribution.



large population is necessary to achieve this. A starting population may be randomly generated, but the next generation must necessarily depend on the previous generation in order for evolution to take place. There are several ways of doing this. A simple way is to replace the old generation entirely (generational replacement), with mutated and/or crossed genomes from the old generation, making the new generation inherit the abilities of the previous generation[10]. Another way is to preserve a selected number of the individuals with the highest fitness value and directly transfer them to the newest generation. This is called *elitism*, and may be useful where good individuals are rare and might be lost to the next generation.

A new individual in the new generation may be generated through mutating the genotypes of single parents, or through combining the genotypes of multiple parents. Typically, two parents are used, but more may increase performance [17]. The parents are then combined to create offspring. This is known as *recombination* or *crossover*, and consists of swapping chunks of the genome between the parents to create a child. A *one-point* crossover selects a single point in the genotype, and swaps the genomes after this point, for example. There are a number of different crossover operators, but they all work on the principle of the offspring receiving genes from multiple parents.

This alone is not enough to evolve the desirable characteristics. There must be some form of *selection pressure* that gives preference to the desirable genes of the parent generation, such that each successive generation improves. A high selection pressure means that only the fittest individuals of the previous generation pass on their genomes to the next generation, but this may quickly reduce diversity, and the population may converge to a local maximum. Greater diversity means that fitness will increase more slowly, but may result in a better result overall. In the natural world, a so-called *genetic bottleneck* may lead to the demise of a species[16], and in evolutionary computation it may leave us with a less than optimal solution.

In *proportionate selection*, also known as roulette wheel selection, the fitness values of all members of the population are summed up[10]. Then, the ratio between the fitness value of the individual to the total fitness value becomes the probability that that individual is selected for reproduction. It has often been described using a roulette wheel analogy, where the total fitness is the roulette wheel circumference, and each individual makes up a sector of the circle proportionate to their fitness value. For larger fitness values, the chance is higher that the roulette ball lands in that sector.

In *rank-based selection*, the population is ranked according to fitness value, and then selected based on a probability proportional to that rank. This has two benefits over proportionate selection: For many similar fitness values, the rank-based selection will definitely favour the individuals with slightly higher fitness values, and for a situation where a few individuals have very high fitness values, the individuals with very small fitness values still have a chance of reproducing, maintaining higher genetic diversity.

*Tournament selection* uses a method of organising a tournament between groups of individuals. For each offspring to be generated, a tournament of size  $k$  picks  $k$

---

randomly chosen individuals from the total population. Only the individual with the best fitness value is chosen for reproduction. This is then repeated until the new offspring has a sufficient number of parents. Each tournament selects participants from the total population, meaning that each individual may be selected multiple times.

## CHAPTER 3

# Methodology

The goal in this paper is to test the evolvability of Random Boolean Networks, and compare an instruction-based RBN implementation to a traditional table-based implementation in order to investigate whether the instruction-based implementation improves timing and performance.

To do this, an Evolutionary Algorithm was implemented and run, evaluating simulations of table-based and instruction-based RBNs. The running times and results were then compared and analysed for  $K = 2$  and  $K = 3$  and  $N = 10, 12, 14, 16, 18$ .

As an objective measure of network fitness, attractor length was chosen. However, as shown by Kauffman [3] these are difficult to achieve, and few randomly generated networks have long cycles, though networks with  $K = 3$  do have slightly longer cycles than those with  $K = 2$ . A point attractor has fitness 1, while a cyclic attractor has fitness equal to the number of states in the cycle. Evaluating the attractors of a random Boolean network requires repeatedly running it until it reaches a stable state, using every possible state combination as a start value. As the states are visited, they are added to a history buffer. Once an attractor is found, the states from the history buffer are then added to the list of visited states, and the attractor length is saved if it is better than the previous best attractor length. The history buffer is then emptied, and the computation starts from the next start state. If the computation for a network reaches a state it has visited previously, all the states leading up to that state are in the same attractor basin as the visited state, and these are added to the visited list.

### 3.1 Table-based RBN

The table-based Random Boolean Network implementation is quite straightforward compared to the instruction-based implementation, and easily implemented for variable  $K$ . The result column of the truth tables for all the possible Boolean functions is self-indexing, comprising all numbers from 0 to  $2^K - 1$  in binary. It may be interesting to note that this method of numbering was first used by Stephen Wolfram to name cellular automaton rules[9], and is often called a Wolfram code[10]. Additionally, the input columns are redundant and can be trimmed away. Thus, the result column can be directly encoded in the genotype. The implementation then looks up the states of

the current node's input nodes, and combines these into a single binary number. It then uses this to look up the result value at that index in the result column for the current node.

Including the result column directly in the genotype means that most mutations result in a fairly small Hamming distance between the original genotype and the mutated one. In the implementation, there is a given small chance of flipping a single arbitrarily chosen bit in the truth table for each node. The hope is that this will result in a small, uniform change in behaviour.

In addition to mutating the truth table, there is also the possibility of mutating the input nodes of each node. This resulting change may be large or small, but it is not possible to tell which without actually evaluating each possible combination. This may make guaranteeing a small change in the truth table redundant.

Networks are generated through randomly selecting  $K$  input indexes for each node, as well as randomly generating an integer array of length  $2^K$ , representing the truth table with a number between 0 and  $2^K - 1$ .

## 3.2 Instruction-based RBN

For an instruction-based RBN implementation, the  $K = 2$  implementation is trivial, achieved through directly implementing each of the  $2^{2^K}$  Boolean functions. However, for  $K = 3$  a similar approach would mean directly implementing 256 different Boolean functions, and for  $K = 4$  this number rises to 65536. This is not practical, and it defeats the purpose of using an instruction set rather than generating truth tables.

Thus, 2-input logic functions must be combined to make 3- and 4-input logic. It remains to be seen what methods are practical and reasonably efficient, and whether the result is representative of a roughly equal distribution of all possible Boolean functions.

The distribution of resulting functions for  $K = 3$  logic was tested using exactly two 12-input logic gates in all possible combinations and permutations. Because non-commutative functions were included, this means that there are four different structural permutations, as shown in Figure 3.1, as well as  $3! = 6$  ways to combine the three input values. Since, as mentioned in Section 3.1, the output column of all the possible truth tables results in the numbers 0 to 255, the test function searches through the 24 possible permutations of  $16^2$  combinations of two 2-input Boolean logic functions, and uses the resulting number as an index in a result array. A histogram of the result array is shown in Figure 3.3a. As can be seen in the figure, it is not possible to generate all 256 Boolean functions using only two 2-input logic gates, and the  $K > 2$  instruction-based implementation will therefore be functionally different from the table-based implementation.

Another, simpler option disregarded the multiple structural permutations and settled on a single configuration as seen in Figure 3.2, using a single order of input. As can

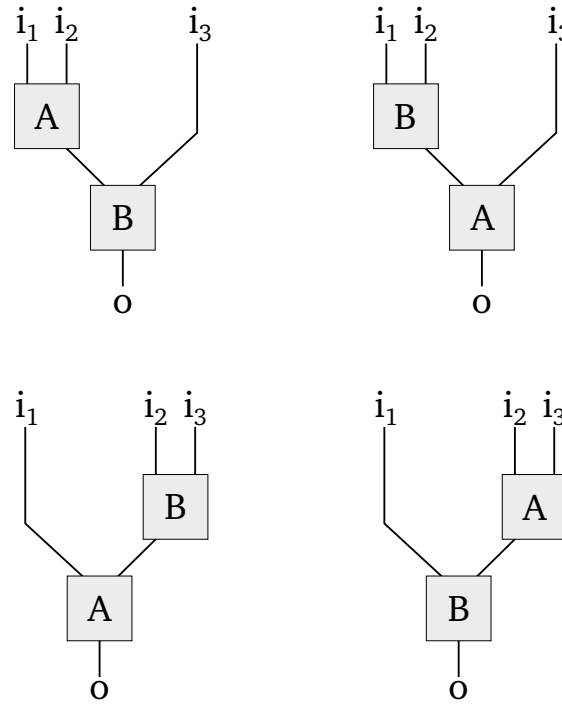


Figure 3.1: Structural permutation of two 2-input Boolean functions

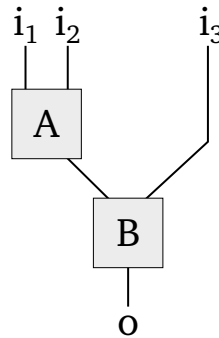
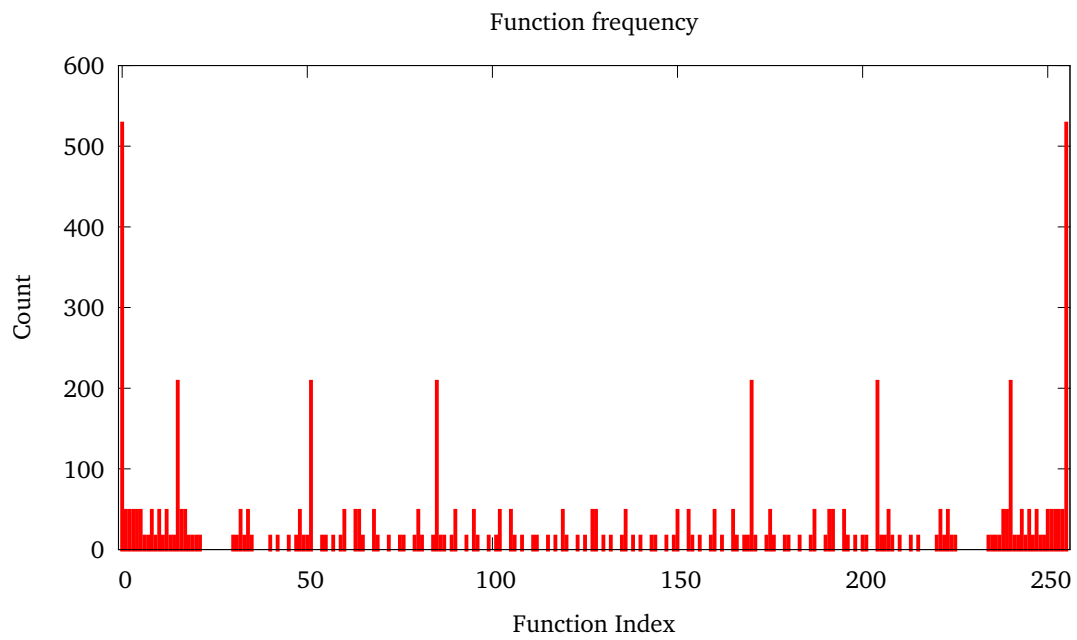


Figure 3.2: The implemented configuration

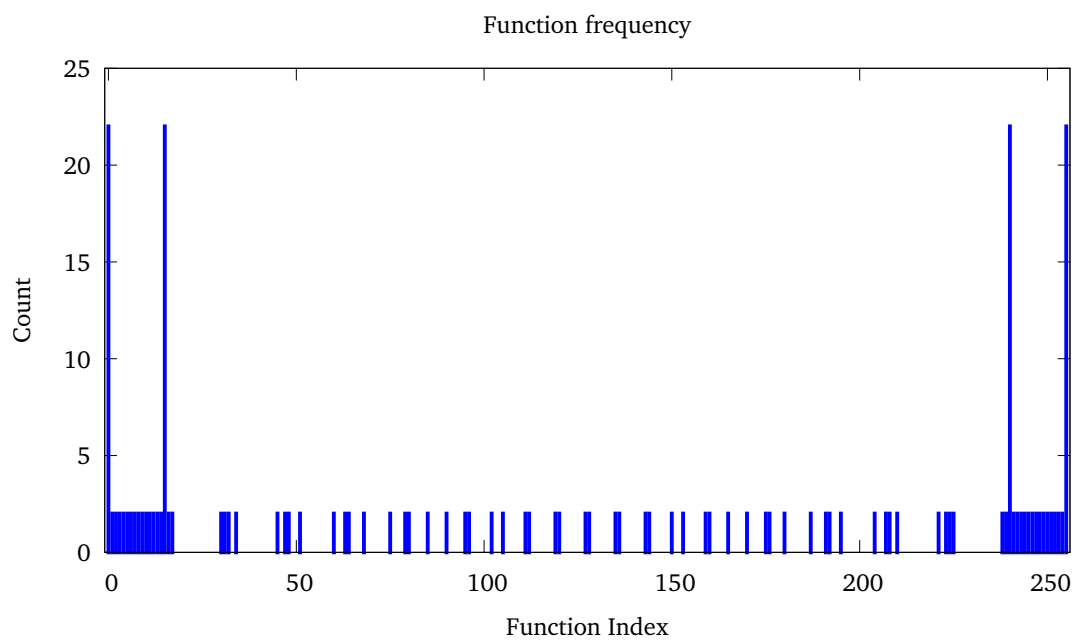
be seen in Figure 3.3b, the two results are not very different. Notably, both options result in a large number of combinations resulting in either contradiction or tautology. This was concerning, given Kauffman's observation that the elimination of these two Boolean functions results in longer cycle length, but initial testing runs showed that the EA was able to find good solutions. Given the similar results, the simpler option was chosen for the instruction-based implementation.

### 3.3 Evolutionary strategy

The implementation uses a fitness proportionate selection for population replacement, combined with elitism of 5 genotypes. The inclusion of elitism is to improve maximum fitness, because even small changes might eliminate an individual with good fitness, and earlier testing runs without elitism gave disappointing results. For the remaining



(a) Combination distribution, all permutations



(b) Combination distribution, simplified

Figure 3.3: Distribution of 3-input Boolean functions through combinations of two 2-input Boolean functions

genotypes, a new genome is created using a random one-point crossover from two parents (not necessarily distinct). These are then mutated, node for node, with a mutation rate of 0.05 for each node input, and 0.025 for each Boolean operator. In the instruction-based implementation, a new Boolean operator is randomly selected from all possible Boolean operators. In the table-based implementation, a randomly chosen cell in the truth table is flipped. This means that while the truth table is only changed minutely, there exists a possibility that the Boolean operator changes to the opposite. For an implementation that uses multiple Boolean operators per node, all of these may change, though the chance of this is quite small at  $0.025^2$  for just two operators.

## 3.4 Testing

Each version of the implementation was run for  $N = 10, 12, 14, 16, 18$ , with 20 individual runs each. Four threads were executed simultaneously. To ensure that all runs ran with different seeds, `/dev/urandom` was used. While this does not strictly guarantee random numbers generated with sufficient entropy for cryptographic applications [18], it is suitable for RBN testing purposes. Unlike when using wall-clock time as a seed for random number generation, the four threads executing simultaneously do not end up using the same seed.

For accurate timing, the implementation uses a monotonic clock, rather than a clock based on wall-clock time, and returns the total running time in nanoseconds for each run.

At every generation, the maximum, minimum, and mean fitness, as well as the standard deviation, is calculated, then printed. At the end of each run, the best network configuration is printed in a format suitable for graphing. If there is more than one configuration with the same fitness, only the first of these is printed.

## 3.5 Setup

The test runs were performed on a PC with a 4-core, 8-threaded 2.20GHz Intel Core i7-2670QM processor and 8 GB of RAM, running 64-bit Debian Jessie (Linux 3.2). The programs were compiled with Clang version 3.5.2, optimised with `-Ofast`, and using the GNU99 standard.

## CHAPTER 4

# Results and Discussion

### 4.1 Table-based RBN

The table-based RBN implementation performs as expected, given the difficulty of evolving RBNs, the large search space, and the fact that small changes in the genotype may cause a large change in the fitness. The average run gives a moderately good result, but may find close to ideal results occasionally. Kauffman found that RBNs with large cycle lengths are not common [3]. Occasionally exceptional results skew the average, as can be seen in 4.1d, where the best run found a very good solution quite early on, but on average the results are well under half of the theoretical maximum fitness value. Finding exceptional solutions requires an element of chance when the runs are small, but most runs find good results.

For  $K = 3$ , the table-based implementation finds on average significantly poorer solutions than it does for  $K = 2$  for networks of the same size, as can be seen in figure 4.2. One reason for this is the considerably larger search space. The size of the search space is  $(2^{2^K} \times N^K)^N$ . For  $N = 10$ ,  $K = 2$  the search space is  $1600^{10} \sim 10^{32}$ , and for  $K = 3$  it is  $256000^{10} \sim 10^{54}$ .

### 4.2 Instruction-based RBN

The instruction-based RBN implementation for  $K = 2$  performs as well as the table-based implementation. Since the two implementations are functionally identical apart from the mutation, as specified in Section 3.3, this is as expected.

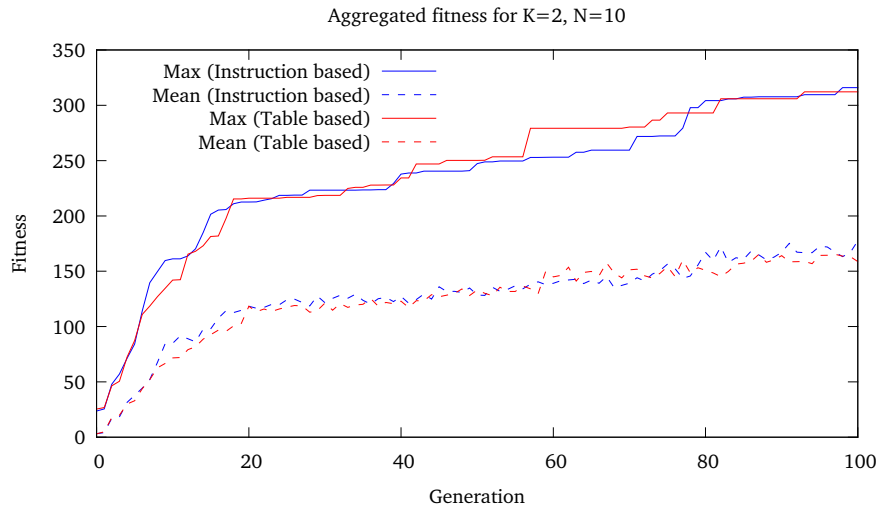
For  $K = 3$ , however, there is a significant difference between the table-based and instruction-based implementation, where the instruction-based implementation outperforms the table-based implementation by a large degree. For  $K = 3$ , the instruction-based implementation finds results that are similar to but still smaller than the results for  $K = 2$  in fitness.

The difference in the size of the search space between the table-based and the instruction-based implementation is large, and is likely to be a major cause of the big difference in performance. The original search space of  $(2^{2^K} \times N^K)^N$  is significantly

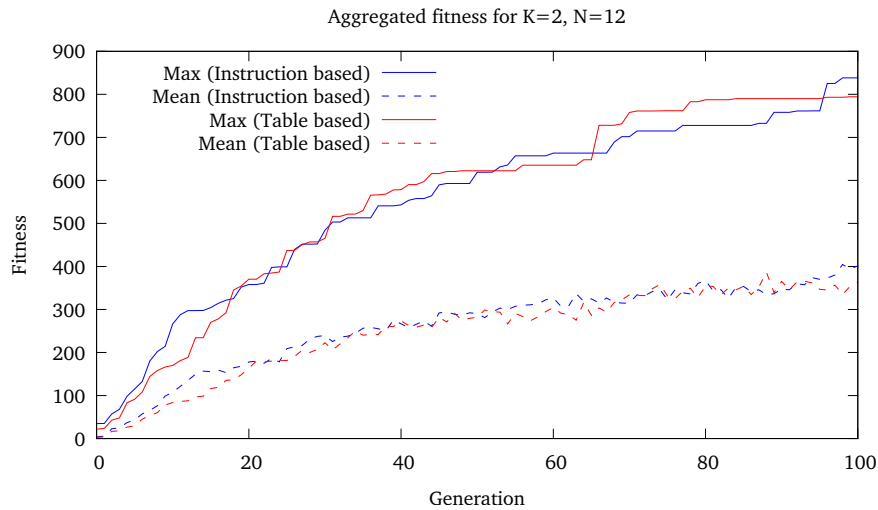


reduced when the  $2^{2^K}$  term is reduced. The  $K = 3$  implementation uses 88 different Boolean functions rather than the complete set of 256. For  $N = 10$  this is the difference between  $(256 \times 10^3)^{10} \sim 10^{54}$  and  $(88 \times 10^3)^{10} \sim 10^{49}$ .

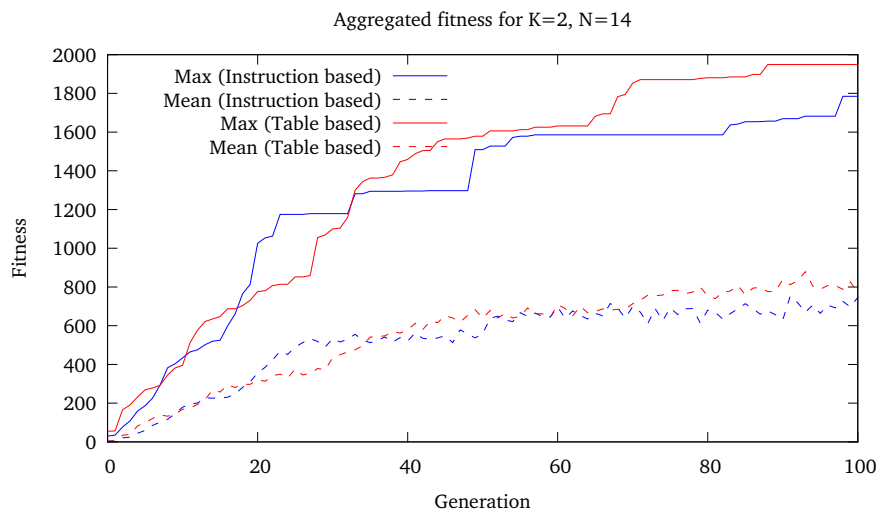
It is interesting to note that the different mutation schemes do not seem to impact the results of the  $K = 2$  run, where both implementations are otherwise identical. In the table-based implementation, the mutation of the truth table guarantees a Hamming distance of 1, while the instruction-based implementation replaces the instruction entirely with a randomly selected instruction, potentially giving the maximum Hamming distance of 4 (all bits flipped). Of course, the structural mutation (changing input nodes) also impacts performance a great deal, and is performed identically in both implementations. It is possible that this masks any difference caused by the different the Boolean function mutation schemes, but it is also possible that this difference does not impact performance overall.



(a) Fitness values for K=2, N=10, both implementations

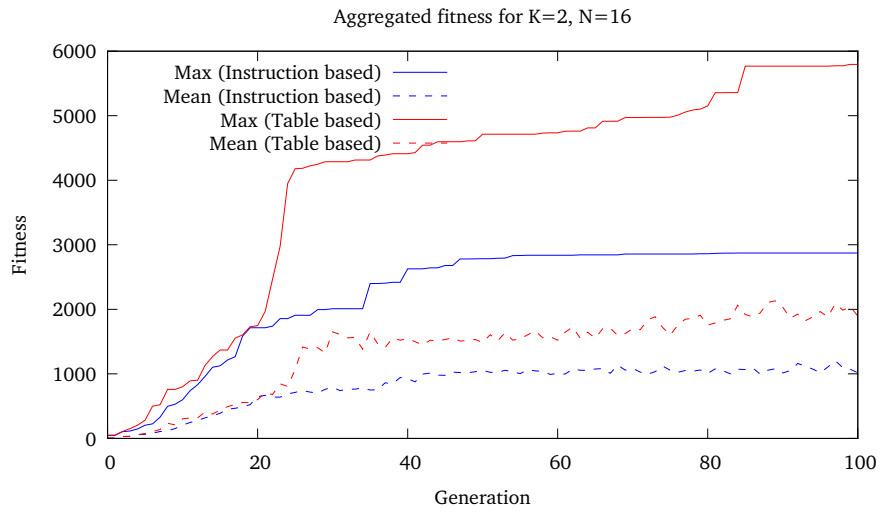


(b) Fitness values for K=2, N=12, both implementations

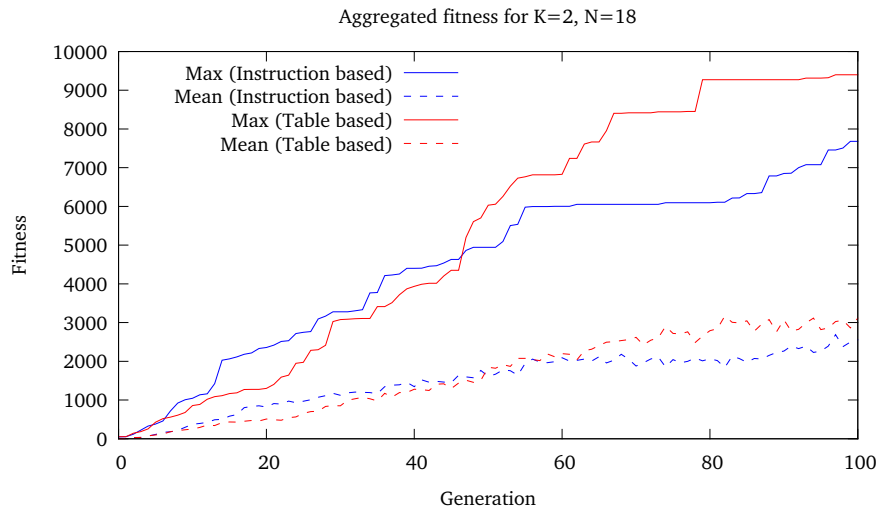


(c) Fitness values for K=2, N=14, both implementations

Figure 4.1: Fitness values for K=2

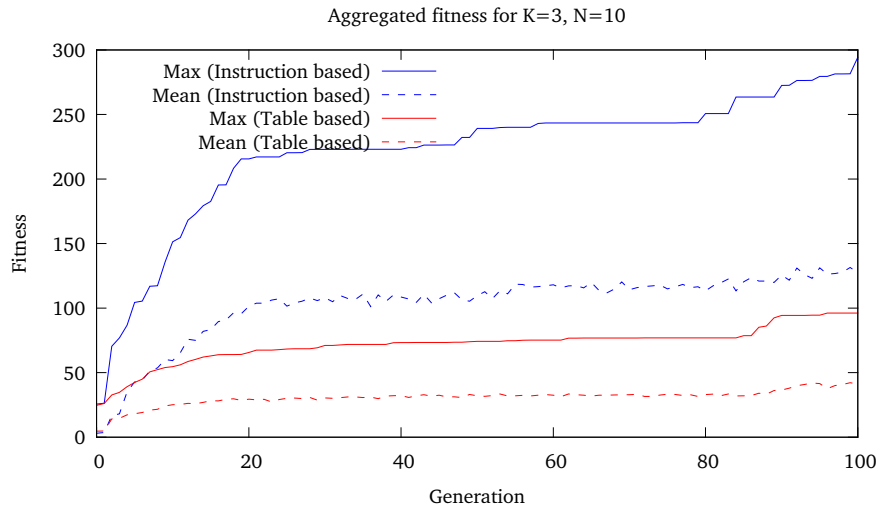


(d) Fitness values for K=2, N=16, both implementations

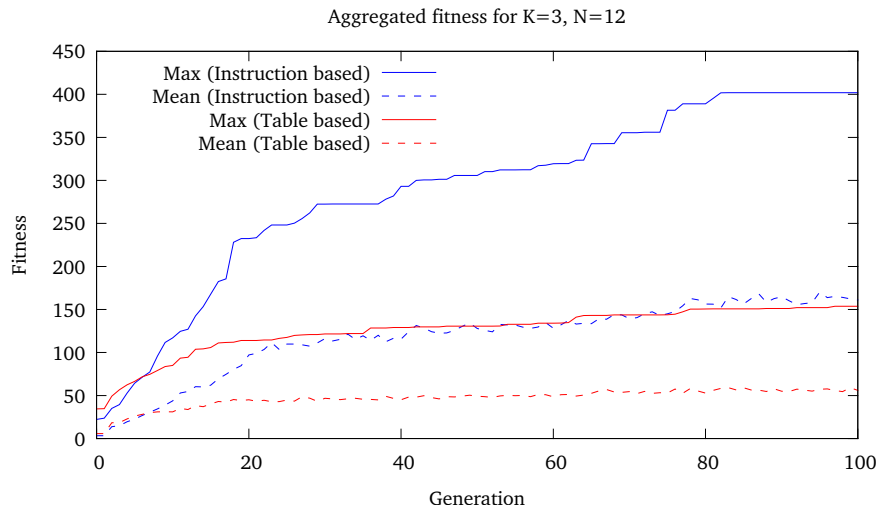


(e) Fitness values for K=2, N=18, both implementations

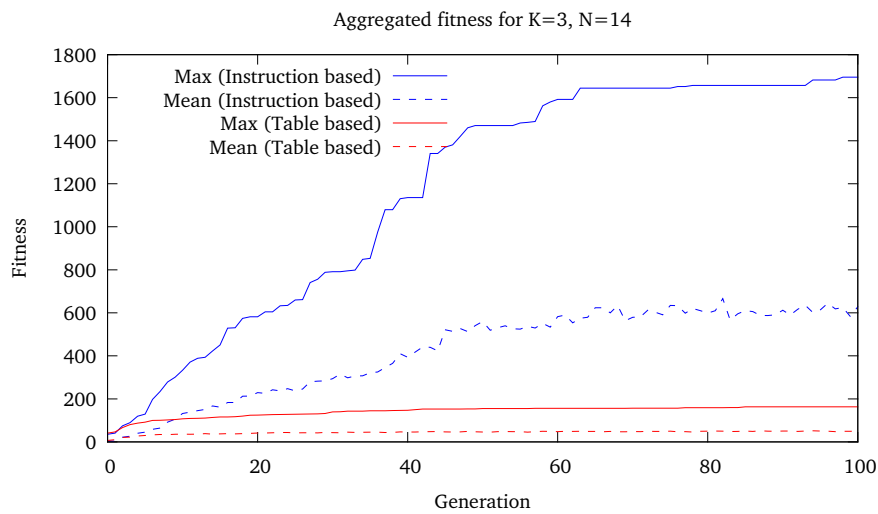
Figure 4.1: Fitness values for K=2



(a) Fitness values for K=3, N=10, both implementations

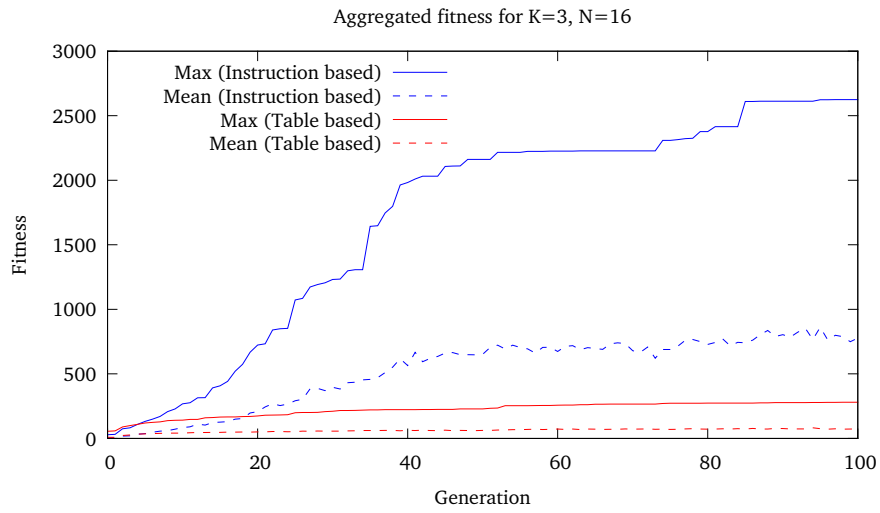


(b) Fitness values for K=3, N=12, both implementations

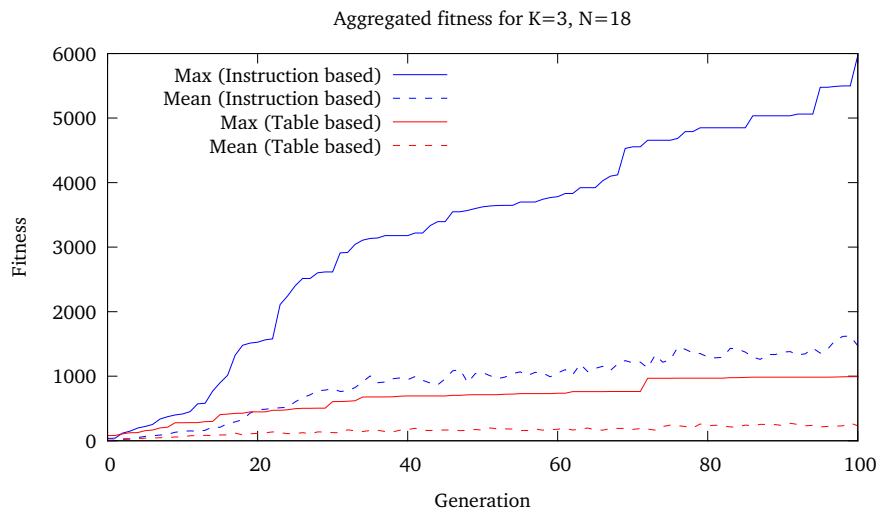


(c) Fitness values for K=3, N=14, both implementations

Figure 4.2: Fitness values for K=3



(d) Fitness values for K=3, N=16, both implementations



(e) Fitness values for K=3, N=18, both implementations

Figure 4.2: Fitness values for K=3

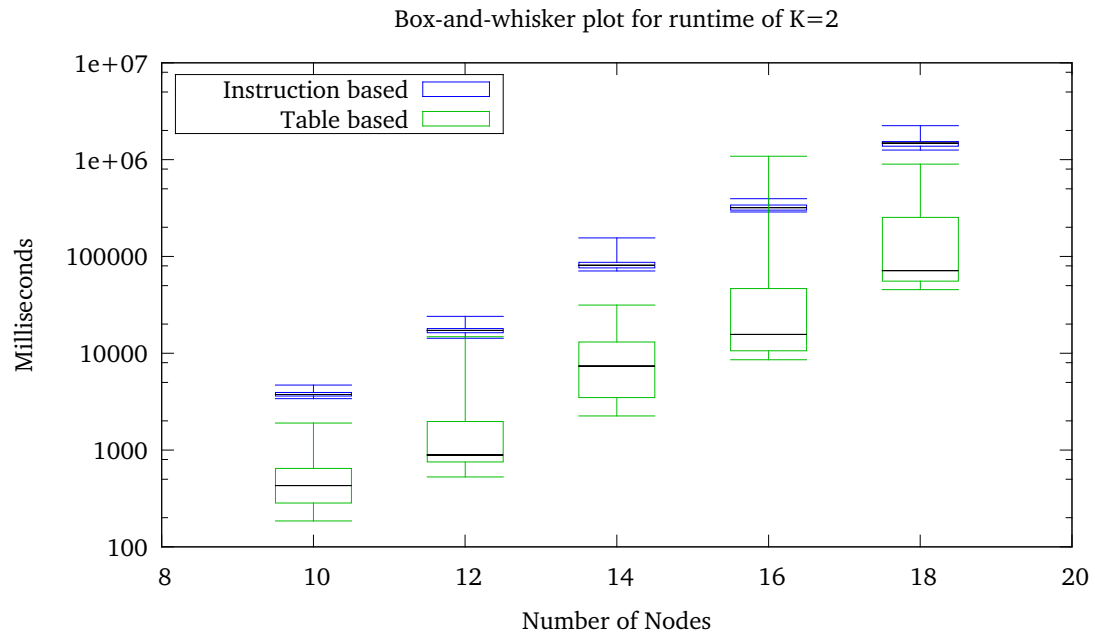
## 4.3 Running time

The running time of the instruction-based implementation is greater than the running time of the table-based implementation by a factor of  $\sim 10$ .

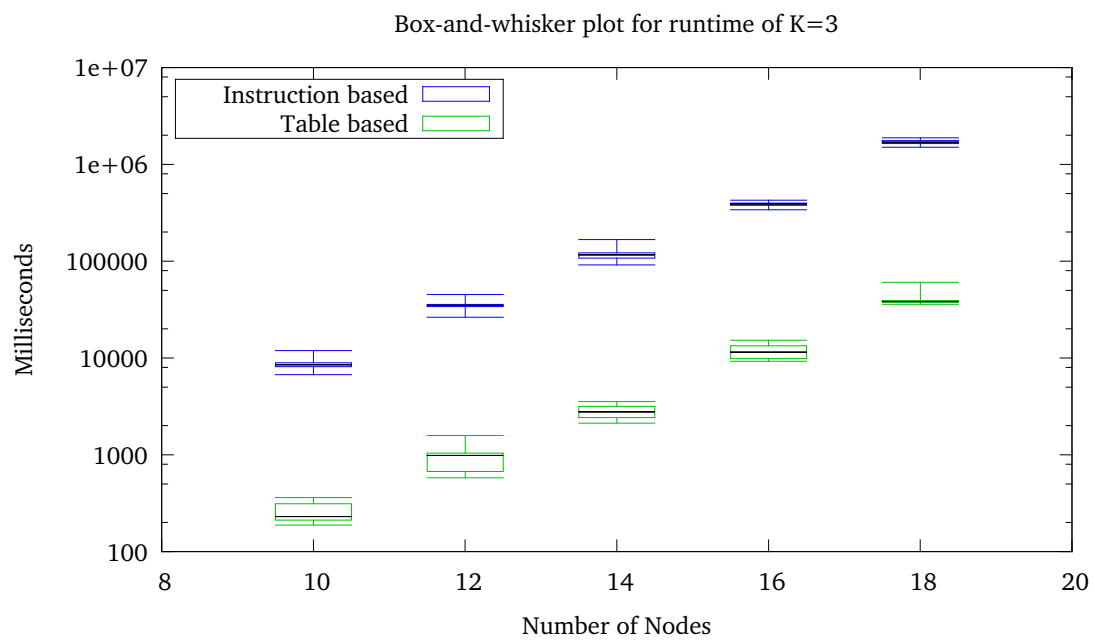
For  $K = 2, N = 10$ , the table-based and instruction-based implementation had an average running time of 577ms and 3820ms respectively. For  $K = 2, N = 18$ , running time can be measured in seconds, 208.1s and 1505.4s. For  $K = 3$ , the differences are larger. For  $N = 10$  the average running times are 256ms and 8687ms, and for  $N = 18$  they are 39.7s and 1701s. At first glance, it may seem odd that the table-based implementation for  $K = 3$  runs faster than that for  $K = 2$ , but this can be explained by the poorer fitness values for the  $K = 3$  runs. The worst-case running time complexity for the fitness evaluation is  $\mathcal{O}(n^2)$ , where  $n$  is the number of possible states. For large cycles this means that the running time increases quadratically.

The  $K = 2$  implementation shows large variation in running times for the table-based implementation, as can be seen in Figure 4.3a, particularly for  $N = 16$ . As we saw in 4.1d, there is a run that found a very good solution, which explains the longer running time. For  $K = 3$  (Figure 4.3b), running times are considerably more homogeneous.

The fitness evaluation complexity being the same for both implementations, the other major difference in time cost between the two comes down to the implementation of the instructions compared to the tables. The instruction-based implementation uses a switch statement to distinguish between instructions. This introduces branching, and occasional mis-predicted branch adds up over time. Table look-up, on the other hand, is a fairly low-cost operation, with no branching involved.



(a) Running time for K=2



(b) Running timefor K=3

Figure 4.3: Running time

## CHAPTER 5

# Conclusion

This thesis describes the implementation of an evolutionary algorithm applied to instruction-based random Boolean networks, and compares them with the same algorithm applied to traditionally implemented table-based RBNs. The goal was to increase performance and timing in order to improve the chances of finding an ideal RBN for a given problem. The limitations in this thesis include the simplified metric for RBN performance, and the small set of different configurations.

For  $K = 2$ , the implementations are functionally identical, and the table-based implementation is considerably faster. For  $K = 3$ , the table-based implementation is still considerably faster, but the instruction-based implementation outperforms it in finding RBNs with long attractor cycles. The probably cause for this is that the instruction-based implementation reduces the search space for  $K = 3$  considerably. This is consistent with Bidlo and Vašíček's finding that an instruction-based approach gives higher performance for more complex, non-binary cellular automata [5]. Compared to their results, it is clear that the instruction-based approach is more efficient for homogeneous CAs, where there goal is to find only one transition function, than for RBNs, where each node may have a different operator.

Unless a useful, significantly reduced instruction set is found, the instruction-based approach has limited usefulness for RBNs with  $K < 3$ , as these networks have a small set of Boolean operators already, and there is little room for pruning.

### 5.1 Further work

There are many ways to improve the implementation presented in this thesis. Primarily, the fitness evaluation has a time complexity of  $\mathcal{O}(n^2)$ , and reducing this to linear time will improve running time considerably. Additionally, the implementation of good heuristics for RBN behaviour may reduce this further. However, as RBNs are quite complex, these heuristics will be difficult to find.

Another improvement could be made by finding better-performing reduced instruction sets. One potentially useful reduction for  $K = 2$  eliminates True and False as well as



all the non-commutative Boolean operators, leaving a set of six instructions<sup>1</sup>. Earlier work showed that RBNs generated with this limited set produced long cycles[19], but there is no available information on timing, and combining multiple instructions is not implemented.

For  $K > 3$ , a better method of combining instructions would prove useful, as this implementation uses a single fixed configuration. In particular, the implementation of variable genome length as described in [15] would allow for a greater set of Boolean functions through combining a variable number of instructions for each node where necessary.

Additionally, the information provided by this thesis is limited by the single metric chosen. RBNs with long cyclic attractors are fairly rare, and the conclusion is not generalisable for all purposes. As a result, the reduced instruction set may not perform similarly for typical applications of RBNs. The application of other metrics to this problem may prove interesting, e.g. generating networks with many cycles of the same length, or finding networks that have an exact number of attractor basins.

---

<sup>1</sup>AND, NAND, OR, NOR, XOR, and XNOR

## Bibliography

- [1] R. Dawkins. *The Blind Watchmaker: Why the evidence of evolution reveals a universe without design*. WW Norton & Company, 1986.
- [2] G. S. Hornby, A. Globus, D. S. Linden, and J. D. Lohn. “Automated Antenna Design with Evolutionary Algorithms”. In: *AIAA Space*. 2006, pp. 19–21.
- [3] S. A. Kauffman. “Metabolic Stability and Epigenesis in Randomly Constructed Genetic Nets”. In: *Journal of Theoretical Biology* 22.3 (Mar. 1969), pp. 437–467.
- [4] C. Gershenson. “Introduction to Random Boolean Networks”. In: *Workshop and Tutorial Proceedings, Ninth International Conference on the Simulation and Synthesis of Living Systems (ALife IX)*. Ed. by M. Bedau, P. Husbands, T. Hutton, S. Kumar, and H. Suzuki. 2004, pp. 160–173.
- [5] M. Bidlo and Z. Vašíček. “Evolution of Cellular Automata Using Instruction-based Approach”. In: *Congress on Evolutionary Computation (CEC)*. IEEE. 2012, pp. 1–8.
- [6] C. Gershenson. “Classification of Random Boolean Networks”. In: *Artificial Life VIII: Proceedings of the Eight International Conference on Artificial Life*. Ed. by R. K. Standish, H. A. Abbass, and M. A. Bedau. MIT Press, 2002, pp. 1–8.
- [7] C. C. Santini, G. Tufte, and P. Haddow. “Bio-inspired Reverse Engineering of Regulatory Networks”. In: *Proceedings of the Eleventh Conference on Congress on Evolutionary Computation*. CEC’09. Trondheim, Norway: IEEE Press, 2009, pp. 2716–2723. ISBN: 978-1-4244-2958-5.
- [8] S. Huang. “Gene expression profiling, genetic networks, and cellular states: an integrating concept for tumorigenesis and drug discovery”. English. In: *Journal of Molecular Medicine* 77.6 (1999), pp. 469–480. ISSN: 0946-2716.
- [9] S. Wolfram. “Statistical Mechanics of Cellular Automata”. In: *Reviews of Modern Physics* 55 (July 1983), pp. 601–644.
- [10] D. Floreano and C. Mattiussi. “Bio-Inspired Artificial Intelligence: Theories, Methods and Technologies”. In: (2008).
- [11] M. Cook. “Universality in Elementary Cellular Automata”. In: *Complex Systems* 15.1 (2004), pp. 1–40.

- [12] M. Gardner. *The Fantastic Combinations of John Conway's New Solitaire Game "Life"*. 1970. URL: [http://ddi.cs.uni-potsdam.de/HyFISCH/Produzieren/lis\\_projekt/proj\\_gamelife/ConwayScientificAmerican.htm](http://ddi.cs.uni-potsdam.de/HyFISCH/Produzieren/lis_projekt/proj_gamelife/ConwayScientificAmerican.htm) (visited on June 10, 2015).
- [13] J. Aron. *First Replicating Creature Spawned in Life Simulator*. 2010. URL: <http://www.newscientist.com/article/mg20627653.800-first-replicating-creature-spawned-in-life-simulator.html> (visited on June 21, 2015).
- [14] P. Rendell. *A Turing Machine in Conway's Game Life*. 2001.
- [15] S. Nichele and G. Tufte. "Evolutionary growth of genomes for the development and replication of multicellular organisms with indirect encoding". In: *International Conference on Evolvable Systems (ICES)*. IEEE. Dec. 2014, pp. 141–148.
- [16] University of California Museum of Paleontology. *Understanding Evolution*. 2015. URL: <http://evolution.berkeley.edu> (visited on June 3, 2015).
- [17] Á. E. Eiben, C. H. van Kemenade, and J. N. Kok. *Orgy in the Computer: Multi-Parent Reproduction in Genetic Algorithms*. 1995.
- [18] M. Kerrisk et al. *urandom(4) Linux User's Manual*. Mar. 2013. URL: <http://linux.die.net/man/4/urandom> (visited on May 9, 2015).
- [19] C. A. Sæhle. *Evolvability of Random Boolean Networks*. 2013.