

# Evolving Transition Rules for Cellular Automata with multiple dimensions

Ron Breukelaar  
Leiden Institute of Advanced Computer Science  
Universiteit Leiden, P.O. Box 9512, 2300 RA Leiden, The Netherlands  
`rbreukel@liacs.nl`

September 22, 2004

# Contents

|          |  |           |
|----------|--|-----------|
| <b>1</b> | <b>Introduction</b>                              | <b>2</b>  |
| 1.1      | Cellular Automata . . . . .                      | 2         |
| 1.2      | Genetic Algorithms . . . . .                     | 4         |
| 1.2.1    | Fitness function . . . . .                       | 5         |
| 1.2.2    | Selection . . . . .                              | 6         |
| 1.2.3    | Mutation . . . . .                               | 6         |
| 1.2.4    | Crossover . . . . .                              | 6         |
| 1.2.5    | Gene mapping . . . . .                           | 7         |
| <b>2</b> | <b>Evolving one dimensional CA</b>               | <b>8</b>  |
| 2.1      | Majority Problem . . . . .                       | 8         |
| 2.2      | Experiment Details . . . . .                     | 11        |
| 2.3      | Their results . . . . .                          | 12        |
| 2.4      | Copying the experiment . . . . .                 | 13        |
| <b>3</b> | <b>Evolving multi dimensional CA</b>             | <b>19</b> |
| 3.1      | Majority Problem . . . . .                       | 22        |
| 3.2      | AND and XOR Problem . . . . .                    | 26        |
| 3.2.1    | Experiment details . . . . .                     | 26        |
| 3.2.2    | The XOR Problem . . . . .                        | 29        |
| 3.2.3    | Crossover and two dimensional mappings . . . . . | 31        |
| 3.3      | Evolving bitmaps . . . . .                       | 33        |
| <b>4</b> | <b>Summary and Outlook</b>                       | <b>37</b> |

# Chapter 1

## Introduction

Genetic Algorithms have been used before to evolve transition rules for one dimensional Cellular Automata (CA) to solve e.g. the majority problem and investigate communication processes within such CA [4]. In this thesis, the principle is extended to multi dimensional CA, and it is demonstrated how the approach evolves transition rules for the two dimensional case with a von Neumann neighborhood. In particular, the method is applied to the binary AND and XOR problems by using the GA to optimize the corresponding rules. Moreover, it is shown how the approach can also be used for more general patterns, and therefore how it can serve as a method for calibrating and designing CA for real-world applications.

### 1.1 Cellular Automata

According to [8] Cellular Automata (CA) are mathematical idealisations of physical systems in which space and time are discrete, and physical quantities take on a finite set of discrete values. The simplest CA is one dimensional and looks like an array of ones and zeros of width  $N$ , where the first position of the array is linked to the last position. In other words, defining a row of positions  $C = \{a_1, a_2, \dots, a_N\}$  where  $C$  is a CA of width  $N$ ,  $a_N$  is adjacent to  $a_1$  and  $a_i \in \{0, 1\}$ .

The neighborhood  $s_i$  of  $a_i$  is defined as the local set of positions with a distance to  $a_i$  along the connected chain which is no more than a certain radius ( $r$ ). This for instance means that  $s_2 = \{a_{148}, a_{149}, a_1, a_2, a_3, a_4, a_5\}$  for  $r = 3$  and  $N = 149$ . Please note that for one dimensional CA the size of the neighborhood is always equal to  $2r + 1$ .

The values in a CA can be altered all at the same time (synchronous) or at different times (asynchronous). Only synchronous CA are considered in this document. In the synchronous approach at every timestep ( $t$ ) every cell state in the CA is recalculated according to the states of the neighborhood using a certain transition rule  $\Theta : \{0, 1\}^{2r+1} \rightarrow \{0, 1\}$ ,  $s_i \rightarrow \Theta(s_i)$ . This rule basically is a one-to-one mapping that defines an output value for every possible set of input values, the input values being the ‘state’ of a neighborhood. The state of  $a_i$  at time  $t$  is written as  $a_i^t$ , the state of  $s_i$  at time  $t$  as  $s_i^t$  and the state of the entire CA  $C$  at time  $t$  as  $C^t$  so that  $C^0$  is the initial state and  $\forall n = 1, \dots, N$   $a_i^{t+1} = \Theta(s_i^t)$ . Given  $C^t = \{a_1^t, \dots, a_N^t\}$ ,  $C^{t+1}$  can be defined as  $\{\Theta(s_1^t), \dots, \Theta(s_N^t)\}$ .

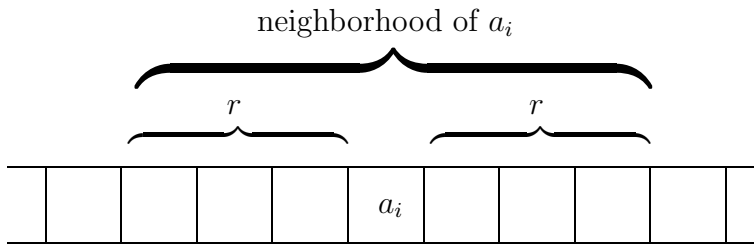


Figure 1.1: This figure shows how the neighborhood relates to  $r$  and cell position in a one dimensional CA.

Because  $a_i \in \{0, 1\}$  the number of possible states of  $s_i$  equals  $2^{2r+1}$ . Because all possible binary representations of  $m$  with  $0 \leq m < 2^{2r+1}$  can be mapped to a unique state of the neighborhood,  $\Theta$  can be written as a row of ones and zeros  $R = \{b_1, b_2, \dots, b_{2^{2r+1}}\}$  where  $b_m$  is the output value of the rule for the input state that maps to the binary representation of  $m - 1$ . A rule therefore has a length that equals  $2^{2r+1}$  and so there are  $2^{2^{2r+1}}$  possible rules for a binary one dimensional CA. This is a huge number of possible rules (if  $r = 3$  this sums up to about  $3,4 \times 10^{28}$ ) each with a different behaviour.

One of the interesting things about these and other CA is that certain rules tend to exhibit organisational behaviour, independently of the initial state of the CA. This behaviour also demonstrates there is some form of communication going on in the CA over longer distances than the neighborhood allows directly. In [4] the authors examine if these simple CA are able to perform tasks that need positions in a CA to work together and use some form of communication. One problem where such a communication seems required in order to give a good answer is the Majority Problem (as described in section 2.1). A genetic algorithm is used to evolve rules for one dimensional CA that do a rather good job of solving the Majority Problem [4] and it is shown how these rules seem to send “particles” and communicate by using these particles [5]. These results imply that even very simple cells in one dimensional cellular automata can communicate and work together to form more complex and powerful behavior.

It is not unthinkable that the capabilities of these one dimensional CA are restricted by the number of directions in which information can “travel” through a CA and that using multiple dimensions might remove these restriction and therefore improve performance. The possibilities of evolving two dimensional CA are explored in [3] on three different problems. It reports that the GA does give some CA rules that perform communicational tasks, but it also reports that random searching the rule space yields almost the same results and goes on to show that totalistic rules outperform all other rules, totalistic rules being rules that trigger on the number of ones in the neighborhood and the state of the center cell instead on the state of the entire neighborhood. These rules have a much smaller search space. Totalistic rules are defined by 18 bits [3] and that means there are “only”  $2^{18} = 262144$  possible rules. It looks feasible to try every possible totalistic rule instead of using a GA.

Every totalistic rule can be represented by a full rule table and can therefore in theory be generated by a GA. Taking that in mind, [3] shows surprisingly good results for totalistic rules, but their claim that totalistic rules have a greater power than full rules seems a bit premature. Totalistic rules in theory are very restricted and have limited possibilities.

In this document only full rules will be considered.

Multiple dimensional CA yield great possibilities in applications such as Parallel Computing and modelling social and biological processes. The goal of this research is to find a generalization and report phenomena observed on a higher level, with the future goal to use this research for identification and calibration of higher-dimensional CA applications to real world systems like parallel computing and modelling social and biological processes. The approach is described and results are reported on simple problems such as the Majority Problem, AND, XOR, extending into how it can be applied to pattern generation processes.

## 1.2 Genetic Algorithms

A problem in general often has a number of possible answers, sometimes called a search space. A problem then often consist of finding the best answer or at least a pretty good one. A trivial approach of finding the best answer is of course trying every answer and remembering the best one. But this can take a long time if the search space is very large and is impossible if the search space is infinite (for instance a real number). A lot of problems can be logically reduced to simpler problems by having additional information about the search space. For example searching the position of a number in an ordered list is possible in  $\log_2 N$  steps using a binary search algorithm, because every step in the algorithm half of the possible positions become impossible due to the fact that the list is ordered.

But sometimes there is no additional information about the search space available to easily reduce the problem. When this coincides with very large search spaces it is often impossible to define an algorithm that always gives the right answer and is also solvable in polynomial time as opposed to exponential time. That is where Genetic Algorithms (GAs) can be used to explore the search space and come up with a good answer. Note that proving an answer is the best one is equal to proving that there is no better one in the search space and that can only be done after having tested all the possible answers or by having additional information about the search space. Therefore if both are impossible (as is usually the case when using a GA) it is quite impossible to know what the best answer is. GAs are quite capable of finding a good answer, but they will not really help much in proving that the best answer is found.

The process behind GAs is quite similar to genetic processes that are found in nature. In nature a species evolves by giving its genes to its offspring. Often a member of the species (or individual) can only reproduce if it survives until it matures. That means that if it has bad genes it will probably not survive long enough to reproduce and the genes are lost. This is often called “survival of the fittest” where the “fitness” of an individual is determined by how well he succeeds in producing offspring.

To make this process flexible and able to withstand dramatic changes in the surroundings of a species, the reproduction of the genes is never an exact copy. These small “errors” (or “mutations”) that are made in the reproduction process ensure that the offspring looks like its parents, but is no exact copy or merge. This process makes it possible for a species to develop long over the course of numerous generations if the water gets too crowded and a pair of wings if the same happens with the land.

In a GA the rules are pretty much the same as in nature. In the initial state there exist a number of possible answers called individuals. This group of individuals is called “population” or “pool”. The individuals all have a certain “fitness” according to a piece of logic called “the fitness function”. Some individuals will have a higher fitness than others but it is probable the best possible answer is not among them. As in nature individuals are defined by there genes. This means that for every possible answer to the problem there exists an unique “genetic description” that defines that answer and every legal “genetic description” maps on a unique possible answer to the problem. This relation between “genetic description” and answer is also called a “mapping”.

To evolve better answers from the existing ones a GA repeats a number of steps over and over again. Such a step is often called a generation. There are many different GAs, but one generation often looks something like this:

- The first phase is calculating the fitness. This is done by the fitness function which calculates the fitness of an individual. This is often where the problem that needs to be solved is defined, because defining what is a good individual and what is not is equal to defining what the characteristics are of the desired answer.
- Then comes the selection phase. This is usually the phase in which the good individuals are selected to reproduce and bad individuals are removed.
- And after that comes the reproduction phase. In this phase the selected individuals get copied, merged and/or mutated to be placed into the next generation.

Usually this process is repeated until a good answer is found or the user runs out of patients. GAs can be stopped whenever this seems necessary and the best individuals so far can be viewed while the algorithm is running. This makes a GA very flexible to restrictions such as time or computation power. A problem with GAs though is that there is no guarantee that the algorithm will find a good or better answer and it is not at all obvious what needs to be changed if a GA performs poorly. Because most reproduction schemes use random factors it is often difficult to predict the results and success always depends on luck.

### 1.2.1 Fitness function

In a GA every individual has a fitness value, usually denoted by  $f$ . This value is usually calculated using a fitness function. This function defines what the GA must solve in order to be succesful, that is why designing a good fitness function is very important. An important aspect of a good fitness function is that it ranks partial solutions higher than no solution, meaning that it should be better to give a reasonable answer instead of a bad one. If the fitness function is only able to reward very good answers, there might not be any propagation towards those answers and evolution might not even get started.

## 1.2.2 Selection

A GA selects the individuals according to their fitness value, but that doesn't mean that having the highest fitness ensures survival. There are many different selection methods and they all have a very different impact on the population.

A very common selection method is *proportional selection*. In this selection method every individual has a chance that it will survive proportional to its fitness. That means that even the weakest in a population has a chance to survive.

In this document though only *truncation selection* is used. This selection method just lets a few best individuals survive. This is usually a certain percentage of the total pool size. Although this is not a very standard selection method, it is necessary in order to compare our results with earlier findings and it seems to be sufficient to get results.

## 1.2.3 Mutation

As stated earlier mutation generates small “errors” in the copied genes of offspring. These changes generate diversity in the population and are the driving force behind evolution. There are different ways to mutate an individual, but when an individual is a binary string there are two simple mutation types that are often used in GAs.

- *Probabilistic mutation* where every bit in the individual is flipped with a certain probability called the “mutation rate”.
- *Fixed bit mutation* where a fixed number of randomly chosen bits are flipped every time an individual is mutated.

In this document both of these methods will be used.

## 1.2.4 Crossover

Crossover is a special way to generate offspring. Instead of copying a selected individual as is done when not using crossover, two individuals are merged to form the offspring. Again there are a number of ways to do this all having very different characteristics. The best known are:

- *Single-point crossover* where a single position in the bit string is chosen at random, both of the parents bitstrings are “cut” at that position and “glued” back together so that an offspring has a part of both its parents. Normaly only one offspring is generated from two parents.
- *N-point crossover* where multiple points in the bitstring are chosen at random, both of the parents bit strings are “cut” at those positions and the offspring is generated from the pieces by alternating the parent at every chosen position.
- *Uniform crossover* where every bit in the offsprings bitstring is equal to the bit at that same position in one of the parents, chosen at random for every bit.

Crossover is often used in combination with mutation to make it possible for a GA to generate bitstrings that have parts that don't exist in the “gene-pool”.

### 1.2.5 Gene mapping

Because not everything in the world is represented by bitstrings, it is very important to understand that evolving bitstrings with a GA evolves representations of answers, not the answers directly. Changing the representation of an answer slightly might have a very big impact on the answer it represents. That is why it is important to have a good mapping and to choose the right mutation operators and crossover methods to suite that mapping. Section 3.2.3 elaborates more on what mapping to use in a two dimensional CA.

# Chapter 2

## Evolving one dimensional CA

As described in the introduction, a simple one dimensional Cellular Automata (CA) can be described as a row of ones and zeros that behave according to a rule. This rule changes the values in the row according to the neighbors of these values. As described in the introduction a rule can be defined by a row of ones and zeros with a length equal to  $2^{2r+1}$  where  $r$  is the number of positions the neighborhood of a value extends to the left and right of that value.

Note that every possible input state of the neighborhood is represented by only one output value in the bitstring and every output value can be linked to only one input state of the neighborhood, making this an efficient way to represent a rule.

In the case of a synchronous CA the rule together with the initial state of a CA define the behaviour of that CA in time. Because there are a huge number of possible rules (being  $2^{2^{2r+1}}$ ), it is quite impossible to examine all behaviours of all the possible rules if  $r > 1$ . So if one wants to find a rule that exhibits certain behaviour, one has to search in a smart way. A genetic algorithm (GA) seems to do the trick.

By evolving rules represented by bitstrings with a GA the authors in [4, 5] tried to find a rule that looks to solve a problem called “the Majority Problem”. This chapter will look into and elaborate on these findings.

### 2.1 Majority Problem

The Majority Problem can be defined as follows:

*Given a set  $A = \{a_1, \dots, a_n\}$  with  $n$  odd and  $a_m \in \{0, 1\}$  for all  $1 \leq m \leq n$ , answer the question: ‘Are there more ones than zeros in  $A$ ?’.*

The Majority Problem first does not seem to be a very difficult problem to solve. It seems only a matter of counting the ones in the set and then comparing them to the number of zeros. Yet when this problem is converted to the dimensions of a CA it becomes a lot more difficult. This is because the rule in a CA does not let a position look past its neighborhood and that is why the cells all have to work together and use some form of communication.

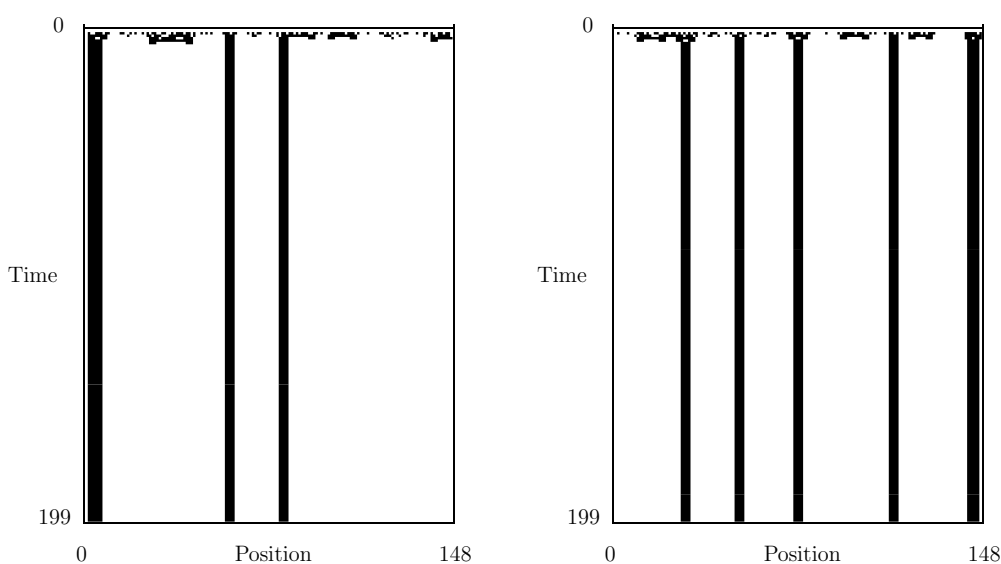


Figure 2.1: These are examples of majority problem classification by the Majority Rule. The pictures show how the rule gets stuck on “thick lines” in the time plot. Time  $t$  proceeds from top to bottom and every row corresponds to  $C^t$ .

Given that the relative number of ones in  $C^0$  is written as  $\lambda$ , in a simple binary CA the Majority Problem can be defined as:

*Find a rule that, given an initial state of a CA with  $N$  odd and a finite number of iterations to run ( $I$ ), will result in an ‘all zero’ state if  $\lambda < 0.5$  and an ‘all one’ state otherwise. The ‘all zero’ state being the state in which every cell in the CA is zero and the ‘all one’ state being a the state in which every cell is one.*

The first intuitive rule to come up with is the ‘majority rule’. This being the rule where the calculated value is 1 if the number of ones in the neighborhood is more than the number of zeros, and a zero otherwise. Surprising as it may seem this does not at all solve the problem (as is shown in Figure 2.1). The majority rule gets stuck on the problem that on the boundary thick line in the time plot the cell can’t “agree” on the global answer. The cell just left of such a thick line is zero and because all other cells left of it in the neighborhood are also zero, it “decides” to stay that way. Yet its neighbor to the right is one and sees only ones on its right and therefore decides to stay one. This way the information fails to propagate through the CA and classification fails.

A lot of people have come up with different rules to solve this problem, one such rule is the GKL rule after Gacs, Kurdyumov and Levin [2]. This rule is pretty good at classifying the majority problem and does it for 81.6% of the test cases with a width of 149 cells. For 17 years this was the best rule and then Lawrence Davis found a better one in 1995 which did 81.8%. In the same year Rajarshi Das found a rule that did 82.178%. Then in 1996 David, Forrest and Koza found a rule by cleverly using genetic programming that was able to classify 82.326% correctly [1]. In Table 2.1 a list of these historical rules is given.

Although these rules are very impressive it is believed that there is no definite solution for the problem as long as the neighborhood is smaller than the size of the CA. If there

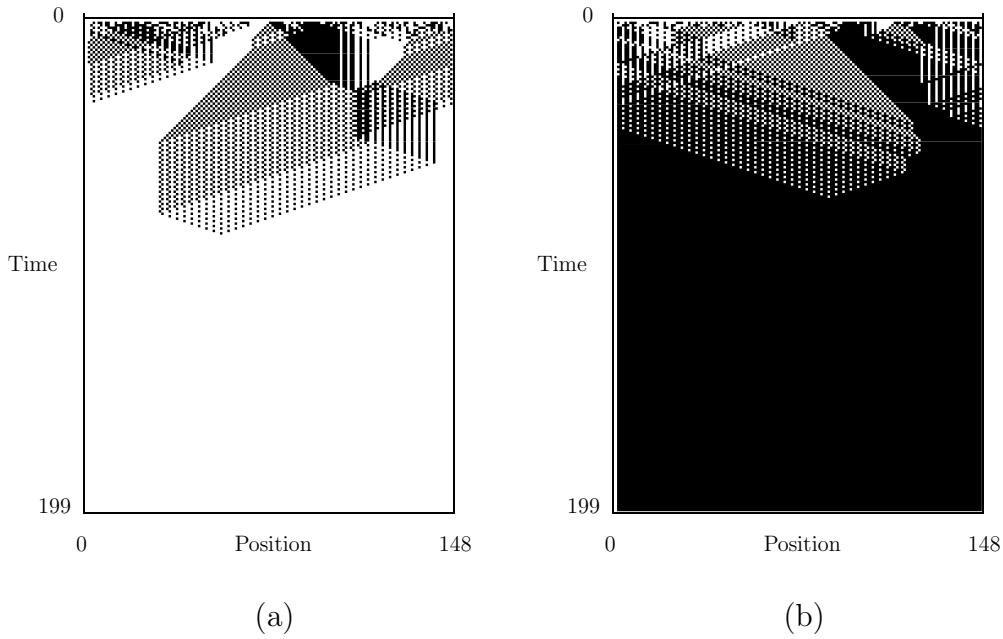


Figure 2.2: These are examples of majority problem classification by the rule found by David, Forrest and Koza [1]. Both are correct classifications (a) with 74 ones in the initial state, (b) with 75.

| <i>Rule name</i>                         | <i>Bit string</i>  |
|--|--|
| Majority rule                            | 00000000 00000001 00000001 00010111 00000001 00010111 00010111 01111111<br>00000001 00010111 00010111 01111111 00010111 01111111 01111111 11111111 |
| GKL 1978<br><i>human-written</i>         | 00000000 01011111 00000000 01011111 00000000 01011111 00000000 01011111<br>00000000 01011111 11111111 01011111 00000000 01011111 11111111 01011111 |
| Davis 1995<br><i>human-written</i>       | 00000000 00101111 00000011 01011111 00000000 00011111 11001111 00011111<br>00000000 00101111 11111100 01011111 00000000 00011111 11111111 00011111 |
| Das 1995<br><i>human-written</i>         | 00000111 00000000 00000111 11111111 00001111 00000000 00001111 11111111<br>00001111 00000000 00000111 11111111 00001111 00110001 00001111 11111111 |
| David 1996<br><i>genetic programming</i> | 00000101 00000000 01010101 00000101 00000101 00000000 01010101 00000101<br>01010101 11111111 01010101 11111111 01010101 11111111 01010101 11111111 |

Table 2.1: Bitstrings of some important rules.

is, nobody has found it yet. It is already a big accomplishment for a CA to get 70 percent of all random initial states correct, for this shows there is some kind of communication going on; some kind of emerging behaviour.

## 2.2 Experiment Details

This section will briefly introduce the work done in [4, 5] by Mitchell, Crutchfield and Hraber. The GA used is fairly simple.

The approach defines the fitness ( $f$ ) of a rule as the relative number of correct answers to 100 randomly chosen initial states and defines ‘a correct answer’ as an ‘all one’ state if there are more ones than zeros in the initial state and an ‘all zero’ state otherwise. To make sure the number of ones is never equal to the number of zeros the width of the CA is always chosen odd. They used a one dimensional CA with a width of 149 and linked ends.

Rules with  $r = 3$  were used to iterate the CA and therefore the bit strings are 128 bits in length. A pool of 100 rules was used and these rules were evolved using a genetic algorithm (GA). This algorithm used crossover and mutation to try to improve the fitness of the rules. The maximum number of iterations ( $M$ ) was set to approximately 320, which was not a fixed number, but was recalculated every time a rule was evaluated. This prevented the algorithm from specialising on a fixed  $M$  as described in [7]. The algorithm was run for 100 generations where one generation step looked like this:

- Randomize all the initial states.
- Recalculate all the fitness values of the individuals using these states.
- Select the top best 20 percent of the individuals as ‘elite rules’ and copy these rules unchanged to the next generation.
- Override the other 80 percent with crossovers of randomly chosen ‘elite rules’ and mutate 2 random bits in their bit strings (which is 128 bit long).

To select the top 20 percent of the population the population was sorted on fitness value and then took the best 20 percent. But because there often would be a lot of rules with the same fitness value, rules with the same fitness were sorted at random.

Because the algorithm did not produce any good results at first, the distribution of the number of ones in the initial states was changed from normal to uniform. It was believed (rightly so) that in a normal distribution of ones in the initial states the probability that the number of ones would be near to the number of zeros is too high. The closer the number of ones is to the number of zeros the more difficult the task of finding a correct answer is going to be and therefore it was believed the algorithm had problems finding anything at all. By using a uniform distribution in the number of ones the algorithm had a lot more ‘obvious’ initial states to start with and this increased the chance that the algorithm would find ‘intelligent’ solutions. However, the Majority Problem uses a normal distribution over the number of ones in an initial state and therefore individuals generated using a uniform distribution might solve a totally different problem.

| <i>Rule name</i>            | <i>Fitness for different widths</i> |                |                |
|-----------------------------|-------------------------------------|----------------|----------------|
|                             | $F_{149,10^4}$                      | $F_{599,10^4}$ | $F_{999,10^4}$ |
| Best block-expand algorithm | 0.652                               | 0.515          | 0.503          |
| A particle-based algorithm  | 0.742                               | 0.718          | 0.701          |
| A particle-based algorithm  | 0.755                               | 0.696          | 0.670          |
| A particle-based algorithm  | 0.769                               | 0.725          | 0.714          |

Table 2.2: Fitness values of rules found in [5].

All the rules were also initialised with this ‘uniform distribution in the number of ones’. With other words, the number of rules with less than 10 ones in their bit string was more or less equal to the number of rules with for example between 60 and 70 ones in the bit string. The importance of this distribution will be discussed later in the document.

## 2.3 Their results

The algorithm was run for 300 runs for 100 generations each. Then the best rules were tested on 10000 initial states with a normal distribution over the number of ones. This distribution has a high peak around initial states with  $\lambda \approx 0.5$ . These initial states are the hardest to solve for a CA and therefore the number of correct answers a rule gives in this last test is a lot lower than when using the uniform distribution. The relative number of correct answers to these tests is denoted as  $F_{n,m}$  where  $n$  is the width of the CA and  $m$  is the number of tests conducted.

Most of the best rules reached  $f > 0.9$  before 100 generations and generated test results with  $0.6 < F_{149,m} < 0.7$ . These rules all seemed to have a comparable algorithm to solve the majority problem. They triggered on large blocks of ones or zeros and tried to expand these. The probability of these blocks occurring is fairly small in initial states with  $\lambda \approx 0.5$ , but a lot higher otherwise. When these results were tested on larger CA this approach began to struggle and  $F_{n,m}$  would approach 0.5. If the width of the CA increases, the chance that a large ‘gap’ occurs increases and therefore the number of wrong classifications using this algorithm does too.

Some of the results did not reach a  $f$  above 0.9 and got stuck at  $F_{149,m} \approx 0.5$ . These rules ignored the initial state and always result in either the ‘all one’ or the ‘all zero’ state. It was suggested that these rules will eventually result in rules with  $f > 0.9$  if given enough time [4].

Out of the 300 rules 7 used a more intelligent algorithm. These rules reached  $f > 0.95$  easily and  $0.7 < F_{149,m} < 0.76$ . The rules seem to use some kind of ‘communication method’ to ‘signal’ the positions and sizes of ‘blocks’ to other such ‘blocks’. This kind of rule is called ‘particle based’ [4] and a number of different particles are identified that seem to ‘transport’ information. Because these rules do not rely on the big blocks, performance deteriorates less on wider CA than the ‘block expanding’ rules do.

The behaviour of different rules with different CA widths are documented in [4]. Table 2.2 copies these results. Note that the particle-based algorithms have a higher fitness than the block-expand algorithms. Also note that the fitness of the block-expanding rule drops dramatically when the width of the CA is increased.

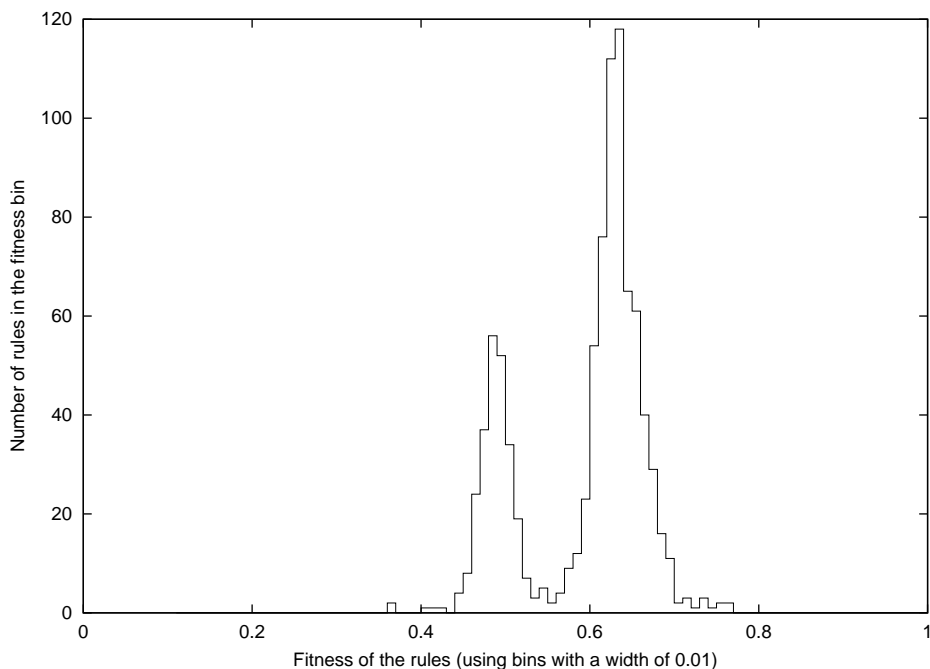


Figure 2.3: This figure displays the frequency with which rules have a certain fitness value in the one dimensional experiment. The fitness bins are 0.01 in width and 900 rules are displayed.

## 2.4 Copying the experiment

This section describes how the experiment described in section 2.2 was copied and what the results were. No variables of the algorithm were changed although there were some things that might need mentioning. They are listed below:

- The two ‘elite’ rules that are chosen for the crossover are chosen randomly with replacement. This means that a ‘pair’ can consist of two times the same rule. The crossover then effectively is a copy.
- When mutating a rule, the two bits that are to be mutated are also chosen randomly with replacement. This means that the same bit can be chosen twice and effectively will not change.

These are interpretations of the experiment description [4, 5] and are no real changes. Although earlier experiments suggested that without these interpretations the algorithm performs worse.

The algorithm was run for 900 runs. Note that this is three times as many runs as was calculated in the original experiment. Afterwards  $F_{149,10^3}$  was calculated for every best rule of a run. It was assumed that the best rule of a run was the rule ranked the highest at generation 100. Note that there might be lower ranked ‘elite’ rules in the rule pool at generation 100 that will get a higher overall fitness than this top ranked rule. This is due to the fact that  $f$  (the fitness of a rule during evolution) is calculated with 100 initial states and is therefore only a rough estimate.

In Figure 2.3 the fitness values of the 900 runs are displayed in a frequency graph. All the fitness values are grouped into bins with a width of 0.01. The peak around  $F_{149,10^3} \approx 0.5$  shows all the rules that did not make it further than an ‘always all ones’ or an ‘always all zeros’ strategy. The biggest peak is situated around  $F_{149,10^3} \approx 0.63$  and corresponds to the ‘block expanding’ algorithms (as shown in Figure 2.4) and the ‘particle based’ rules are situated where  $F_{149,10^3} > 0.71$ .

Out of 900 runs 12 ‘particle based’ rules were found, that means that 1.3% of the total runs had evolved to a ‘particle based’ rule. This is less than the percentage M. Mitchell et al. have found [4] which was 7 out of 300 or 2.3%. This could be contributed to chance or a different definition of what a ‘particle based’ rule exactly is. The 12 rules that are counted as ‘particle based’ rules in this document all have a  $F_{149,10^3} > 0.7$  and are clearly doing something more than just expanding large blocks. There seem to be a lot of different ways to send ‘particles’ from one side of the CA to the other. The inner workings of one rule and its different ‘particles’ are studied in [5], but it is not unthinkable that other rules use a totally different approach.

$F_{n,10^4}$  was calculated for different  $n$  of the CA. As stated in [4] ‘particle based’ rules not only perform better than the ‘block expanding’ rules, but their performance also is less affected by an increase of the width of the CA. Figure 2.6 shows that the ‘block expanding’ rules shift further to the left than the small amount of ‘particle based’ rules. This is all consistent with the experiments conducted in the original experiment.

The duration of a run is defined as the number of iterations needed in a CA to reach a ‘all ones’ or an ‘all zeros’ state and is denoted by  $D_{N,M}$  where  $N$  is the number of cells in the CA and  $M$  is the number of runs used to calculate the average. If a rule does not reach an ‘all ones’ or an ‘all zeros’ state the maximum duration is counted instead.  $D_{149,10^3}$  was calculated for all the 900 rules that were found. Figure 2.7 shows the average duration of these rules against their fitness ( $F_{149,10^3}$ ).

Note that the different types of rules can clearly be seen. There is a big group of “always all ones” and “always all zeros” rules around a fitness of 0.5. These rules don’t really care about the initial state and don’t have to communicate, that is why they have a very low average duration. Next to that group the block expanding rules are situated around a fitness of 0.63 with average durations ranging from 50 to 175. The particle based rules are right next to the large group. This is a small group and can barely be seen, but it is situated roughly around a fitness of 0.74 and has an average duration of about 80.

These results suggest that particle based rules all have roughly the same average duration time, whereas block expanding rules can have a lot of different duration times. The bigger complexity of particle based rules might be the reason for the clustering of duration times of particle based rules, but there are not enough of these rules to conclude anything, it could still be attributed to chance.

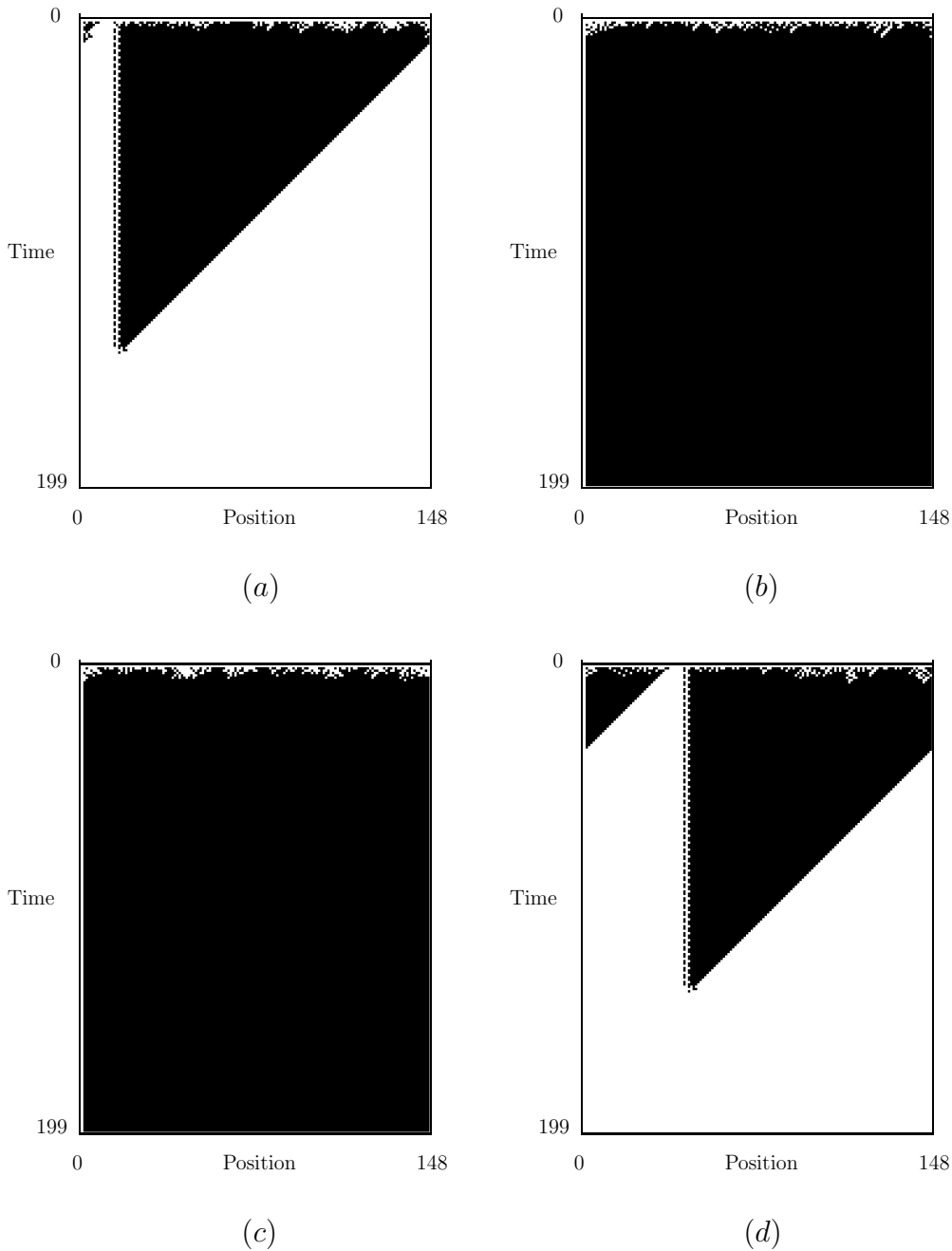
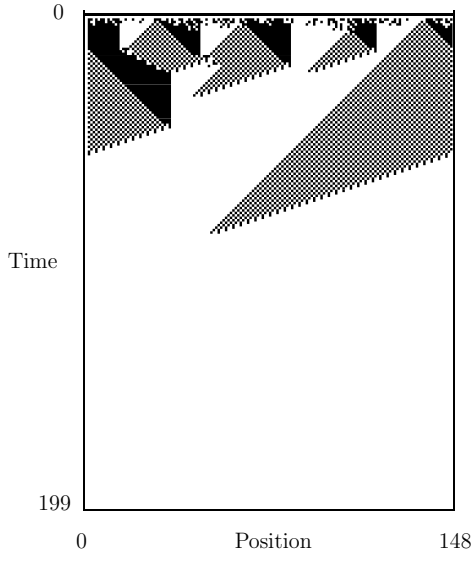
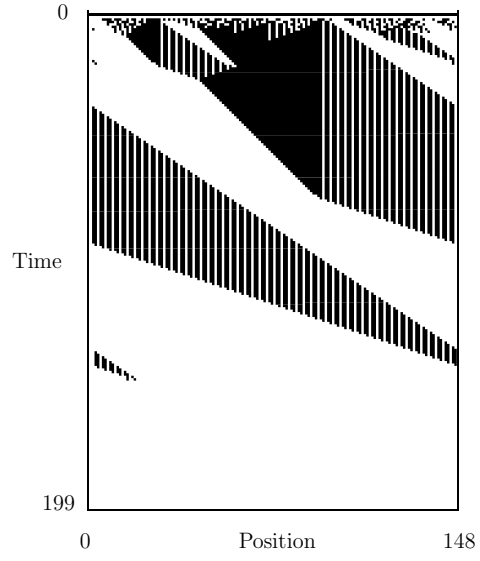


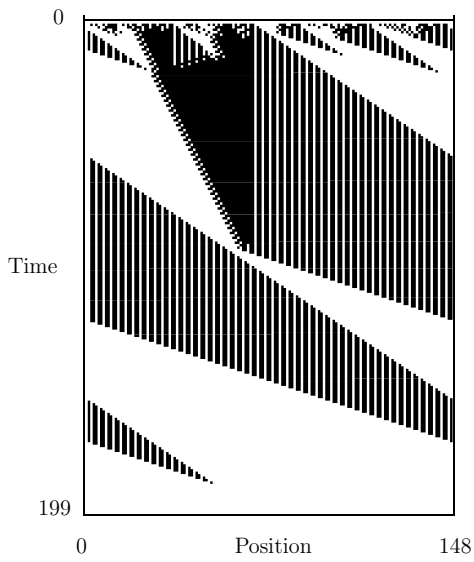
Figure 2.4: These are examples of majority problem classification by a typical block expanding rule with  $N = 149$  and  $F_{149,10^4} \approx 6.5$ . Both (a) and (b) are correct classifications (a) with 74 ones in the initial state, (b) with 75. Note that in (a) there emerges a block of zeros right at the beginning. This block is then extended throughout the CA, if the block is not found (as in (b)) the algorithm assumes it is an ‘all ones’ classification. The chance a block of zeros occurs is bigger with more zeros in the initial state and that is why this approach works. In (c) and (d) the algorithm has incorrectly classified initial states with (c) 65 and (d) 85 ones.



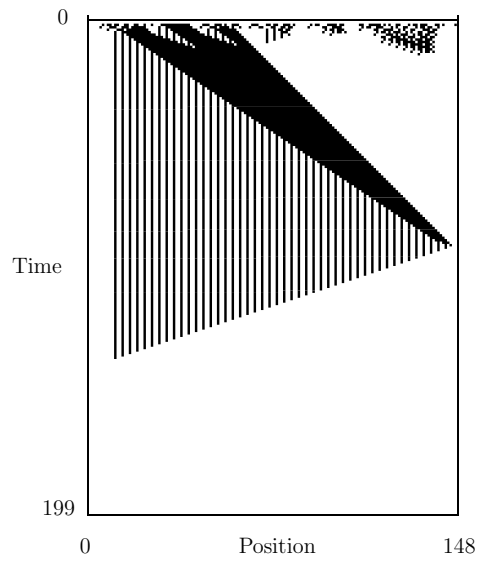
(a)



(b)



(c)



(d)

Figure 2.5: This figure displays four correct classification of the majority problem by four different particle based rules. (a) and (c) both have  $F_{149,10^4} \approx 0.76$ , (b) has  $F_{149,10^4} \approx 0.75$  and (d) has  $F_{149,10^4} \approx 0.73$  with  $N = 149$ .

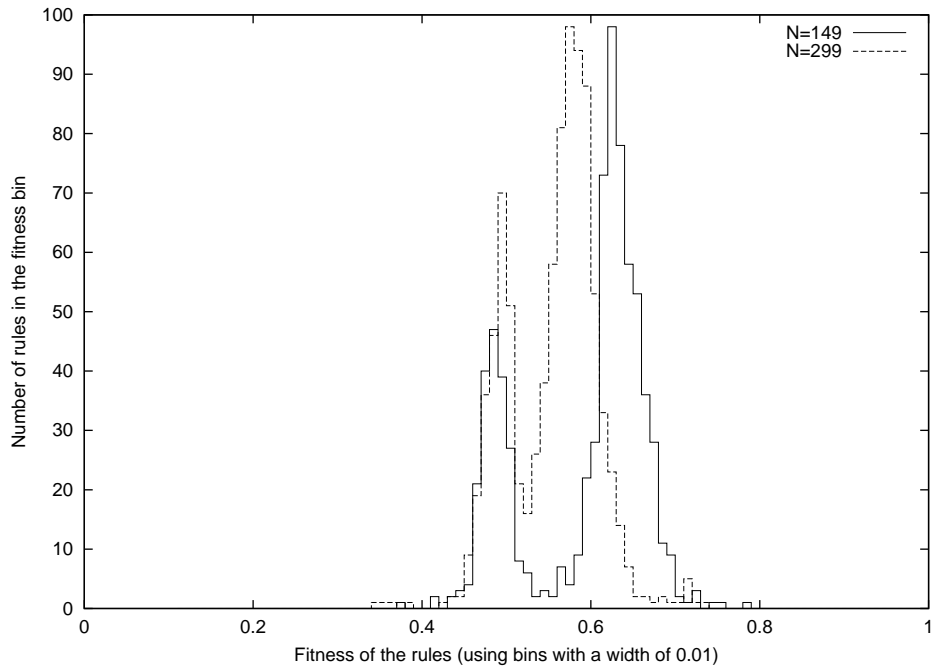


Figure 2.6: This figure shows the effect of increasing the width of the CA ( $N$ ) in the one dimensional experiment. The fitness bins are 0.01 in width and 900 rules are displayed.

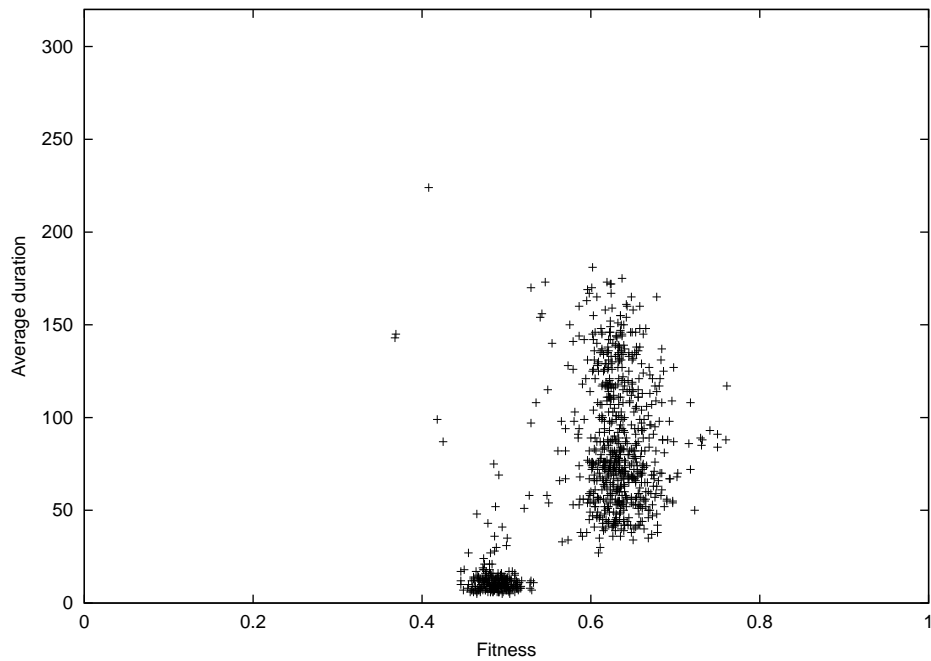


Figure 2.7: This figure displays the average duration ( $D_{N,M}$ ) of runs for the one dimensional algorithm. For this algorithm  $N = 149$  and  $M = 10^3 = 1000$ .

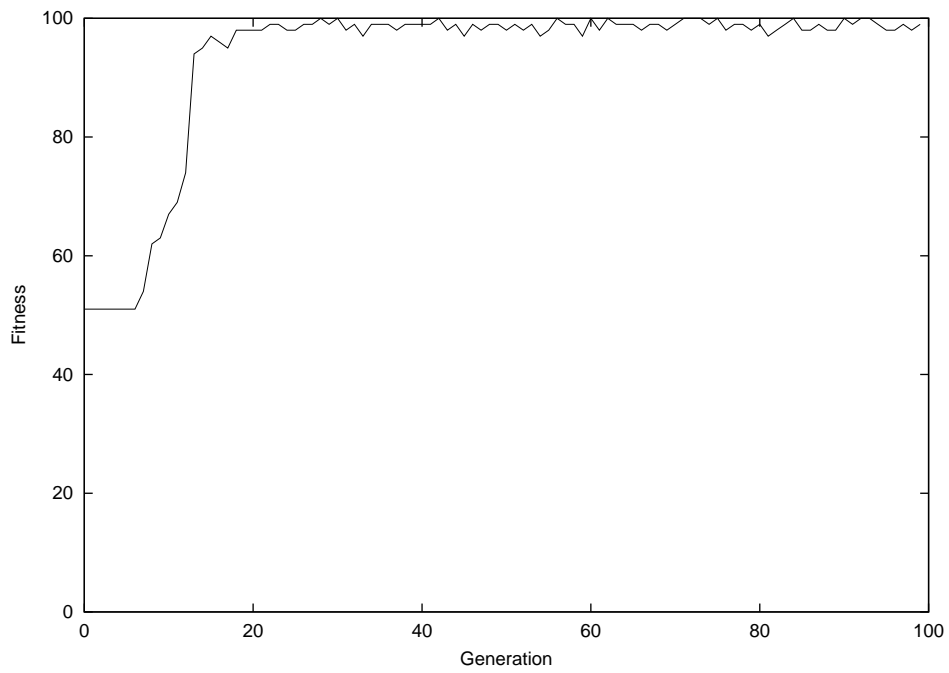


Figure 2.8: An example of a run ending in a particle-based rule

# Chapter 3

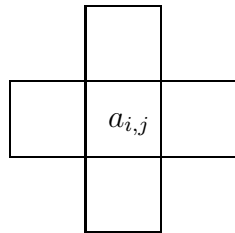
## Evolving multi dimensional CA

The two dimensional CA in this document are similar to the one dimensional CA discussed so far. Instead of a row of positions,  $C$  now consist of a grid of positions. The values are still only binary (0 or 1) and there still is only one transition rule for all the cells. The number of cells is still finite and therefore CA discussed here have a width, a height and borders.

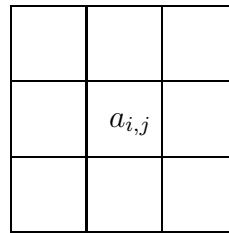
In a one dimensional CA the leftmost cell is connected to the rightmost cell. In the two dimensional CA this it is also common to link opposite borders. This means that every leftmost cell in a row is connected to the rightmost cell in the same row and every topmost cell in a column is connected to the bottommost cell in the same column. Note that such a CA forms a torus structure.

The big difference between one dimensional and two dimensional CA is the rule definition. The neighborhood of these rules is two dimensional, because there are not only neighbors left and right of a cell, but also up and down. That means that if  $r = 1$ ,  $s_{i,j}$  might consists of 5 positions, for instance the four directly adjacent to  $a_{i,j}$  plus  $a_{i,j}$  itself. This neighborhood is often called “the von Neumann neighborhood” after its inventor. The other well known neighborhood expands the von Neumann neighborhood with the four positions diagonally adjacent to  $a_{i,j}$  and is called “the Moore neighborhood” also after its inventor. Figure 3.1 shows these two neighborhoods.

A more formal definition of  $s_{i,j}$  for a two dimensional von Neumann neighborhood is given by  $s_{i,j} = \{a_{k,l} \mid (|k - i| + |l - j|) \leq r\}$ . Note that this defines a diamond shape of



von Neumann neighborhood



Moore neighborhood

Figure 3.1: Two often used and well known neighborhoods.

cells with a diameter of  $2r + 1$  and  $|s_{i,j}| = 2r^2 + 2r + 1$ . This can be generalized to a  $d$  dimensional von Neumann neighborhood with:

$$s_{c_1, c_2, \dots, c_d} = \{a_{g_1, g_2, \dots, g_d} \mid \sum_{i=1}^d |g_i - c_i| \leq r\}$$

Note that this only holds for infinite CA or finite CA with unlinked borders (see 3.2.1). If a CA has dimensions  $\{e_1, e_2, \dots, e_d\}$  and has linked borders the distance between two cells  $a_{c_1, c_2, \dots, c_d}$  and  $a_{g_1, g_2, \dots, g_d}$  is  $\sum_{i=1}^d \min(|g_i - c_i|, e_i - |g_i - c_i|)$ . Therefore a  $d$  dimensional von Neumann neighborhood with linked borders in a CA with dimensions  $\{e_1, e_2, \dots, e_d\}$  is defined as:

$$s_{c_1, c_2, \dots, c_d} = \{a_{g_1, g_2, \dots, g_d} \mid \sum_{i=1}^d \min(|g_i - c_i|, e_i - |g_i - c_i|) \leq r\}$$

The Moore neighborhood of a two dimensional CA can be defined as  $s_{i,j} = \{a_{k,l} \mid |k - i| \leq r, |l - j| \leq r\}$ . Note that this defines a square around a center cell  $a_{i,j}$  with a width and height of  $2r + 1$  and  $|s_{i,j}| = (2r + 1)^2 = 4r^2 + 4r + 1$ . This is generalized to  $d$  dimensional with:

$$s_{c_1, c_2, \dots, c_d} = \{a_{g_1, g_2, \dots, g_d} \mid |g_i - c_i| \leq r, 1 \leq i \leq d\}$$

Note that this too does not hold for finite CA with linked borders. The Moore neighborhood of a CA with dimensions  $\{e_1, e_2, \dots, e_d\}$  and linked borders is defined as:

$$s_{c_1, c_2, \dots, c_d} = \{a_{g_1, g_2, \dots, g_d} \mid \min(|g_i - c_i|, e_i - |g_i - c_i|) \leq r, 1 \leq i \leq d\}$$

Rules can be defined in the same rows of bits ( $R$ ) as defined in the one dimensional case, but the number of bits is higher because the neighborhoods are bigger. The number of cells in a neighborhood is defined as  $S(d, r)$  where  $d$  equals the number of dimensions in the CA and  $r$  is the radius of the neighborhood.  $S^N(d, r)$  defines the number of cells in a von Neumann neighborhood, while  $S^M(d, r)$  defines the number of cells in a Moore neighborhood.

In Moore neighborhood  $S^M(d, r) = (2r + 1)^d$  through normal geometry calculations, but  $S^N(d, r)$  is less trivial to calculate. Note that a one dimensional von Neumann neighborhood equals the neighborhood defined in section 1.1 and has  $2r + 1$  cells. Note that a two dimensional von Neumann neighborhood can be defined as a set of  $2r + 1$  one dimensional von Neumann neighborhoods with sizes  $1, 3, 5, \dots, 2r - 1, 2r + 1, 2r - 1, \dots, 5, 3, 1$ . That means:

$$\begin{aligned} S^N(1, r) &= 2r + 1 \\ S^N(2, r) &= 2 \left[ \sum_{i=0}^{r-1} 2i + 1 \right] + 2r + 1 \\ &= 4 \left[ \sum_{i=1}^{r-1} i \right] + 2r + 2r + 1 \\ &= 4 \frac{(r-1) \cdot r}{2} + 2r + 2r + 1 \\ &= 2(r^2 - r) + 2r + 2r + 1 \\ &= 2r^2 + 2r + 1 \end{aligned}$$

Note that  $S^N(d, r)$  can be defined as  $2r + 1$  von Neumann neighborhoods with  $d - 1$  dimensions with sizes  $S^N(d - 1, 0), S^N(d - 1, 1), \dots, S^N(d - 1, r - 1), S^N(d - 1, r), S^N(d - 1, r - 1), \dots, S^N(d - 1, 1), S^N(d - 1, 0)$ . Therefor:

$$S^N(d, r) = 2\left[\sum_{i=0}^{r-1} S^N(d - 1, i)\right] + S^N(d - 1, r)$$

From this follows that:

$$\begin{aligned} S^N(3, r) &= 2\left[\sum_{i=0}^{r-1} S^N(2, i)\right] + S^N(2, r) \\ &= 2\left[\sum_{i=0}^{r-1} (2i^2 + 2i + 1)\right] + 2r^2 + 2r + 1 \\ &= 2\left[\sum_{i=0}^{r-1} 2i^2\right] + 2\left[\sum_{i=0}^{r-1} 2r\right] + 2r + 2r^2 + 2r + 1 \\ &= 4\left[\sum_{i=1}^{r-1} i^2\right] + 4\left[\sum_{i=1}^{r-1} r\right] + 2r + 2r^2 + 2r + 1 \\ &= 4\frac{(r - 1) \cdot r \cdot (2(r - 1) + 1)}{6} + 4\frac{(r - 1) \cdot r}{2} + 2r + 2r^2 + 2r + 1 \\ &= \frac{2}{3}(2r^3 - 3r^2 + r) + 2(r^2 - r) + 2r + 2r^2 + 2r + 1 \\ &= \frac{4}{3}r^3 - 2r^2 + \frac{2}{3}r + 2r^2 + 2r^2 + 2r + 1 \\ &= \frac{4}{3}r^3 + 2r^2 + \frac{8}{3}r + 1 \end{aligned}$$

and

$$\begin{aligned} S^N(4, r) &= 2\left[\sum_{i=0}^{r-1} S^N(3, i)\right] + S^N(3, r) \\ &= 2\left[\sum_{i=0}^{r-1} \left(\frac{4}{3}i^3 + 2i^2 + \frac{8}{3}i + 1\right)\right] + \frac{4}{3}r^3 + 2r^2 + \frac{8}{3}r + 1 \\ &= \frac{8}{3}\left[\sum_{i=0}^{r-1} i^3\right] + \frac{12}{3}\left[\sum_{i=0}^{r-1} i^2\right] + \frac{16}{3}\left[\sum_{i=0}^{r-1} i\right] + 2r + \frac{4}{3}r^3 + 2r^2 + \frac{8}{3}r + 1 \\ &= \frac{8}{3}\frac{(r - 1)^2 \cdot r^2}{4} + \frac{12}{3}\frac{(r - 1) \cdot r \cdot (2(r - 1) + 1)}{6} + \frac{16}{3}\frac{(r - 1) \cdot r}{2} + 2r \\ &\quad + \frac{4}{3}r^3 + 2r^2 + \frac{8}{3}r + 1 \\ &= \frac{2}{3}(r^4 - 2r^3 + r^2) + \frac{2}{3}(2r^3 - 3r^2 + r) + \frac{8}{3}(r^2 - r) + 2r + \frac{4}{3}r^3 + 2r^2 + \frac{8}{3}r + 1 \\ &= \frac{2}{3}r^4 + \frac{4}{3}r^3 + \frac{10}{3}r^2 + \frac{8}{3}r + 1 \end{aligned}$$

In the same way  $S^N(r, 5), S^N(r, 6), \dots$  can be calculated, but the functions get ugly very fast. With the recursive function the values in Table 3.1 were calculated. Notice that

|             | $r$ |     |       |        |        |         |         |
|-------------|-----|-----|-------|--------|--------|---------|---------|
|             | 0   | 1   | 2     | 3      | 4      | 5       | 6       |
| $S^N(1, r)$ | 1   | 3   | 5     | 7      | 9      | 11      | 13      |
| $S^N(2, r)$ | 1   | 5   | 13    | 25     | 41     | 61      | 85      |
| $S^N(3, r)$ | 1   | 7   | 25    | 63     | 129    | 231     | 377     |
| $S^N(4, r)$ | 1   | 9   | 41    | 129    | 321    | 681     | 1289    |
| $S^N(5, r)$ | 1   | 11  | 61    | 231    | 681    | 1683    | 3653    |
| $S^N(6, r)$ | 1   | 13  | 85    | 377    | 1289   | 3653    | 8989    |
| $S^M(1, r)$ | 1   | 3   | 5     | 7      | 9      | 11      | 13      |
| $S^M(2, r)$ | 1   | 9   | 25    | 49     | 81     | 121     | 169     |
| $S^M(3, r)$ | 1   | 27  | 125   | 343    | 729    | 1331    | 2197    |
| $S^M(4, r)$ | 1   | 81  | 625   | 2401   | 6561   | 14641   | 28561   |
| $S^M(5, r)$ | 1   | 243 | 3125  | 16807  | 59049  | 161051  | 371293  |
| $S^M(6, r)$ | 1   | 729 | 15625 | 117649 | 531441 | 1771561 | 4826809 |

Table 3.1: The number of cells in neighborhoods in multi dimensional CA.  $S^N(d, r)$  stands for a  $d$  dimensional von Neumann neighborhood with a radius  $r$  and  $S^M(d, r)$  represents a  $d$  dimensional Moore neighborhood with radius  $r$ . Note that  $S^N(d, r)$  is a lot smaller and symmetric.

$S^N(d, r) = S^N(r, d)$ . Although this is surprising, it can be explained by rewriting the general function:

$$\begin{aligned}
S^N(d, r) &= 2\left[\sum_{i=0}^{r-1} S^N(d-1, i)\right] + S^N(d-1, r) \\
&= 2\left[\sum_{i=0}^{r-2} S^N(d-1, i)\right] + 2S^N(d-1, r-1) + S^N(d-1, r) \\
&= [2\left[\sum_{i=0}^{r-2} S^N(d-1, i)\right] + S^N(d-1, r-1)] + S^N(d-1, r-1) + S^N(d-1, r) \\
&= S^N(d, r-1) + S^N(d-1, r-1) + S^N(d-1, r)
\end{aligned}$$

Given that  $S^N(d, 0) = 1$  and  $S^N(0, r) = 1$  (else  $S^N(1, r)$  would not increment with two) this general relation automatically implies that  $S^N(d, r) = S^N(r, d)$ .

The number of bits in  $R$  needed to define a multi dimensional rule is defined as  $2^{S(d, r)}$ . That means that the length of  $R$  grows a lot faster than  $S(d, r)$  and that neighborhoods are easily too big for realistic computations. The experiments in this document will focus on two dimensional CA to make them easy to interpret, but can easily be scaled to multiple dimensions.

### 3.1 Majority Problem

To solve the Majority Problem with two dimensional CA the same genetic algorithm as in the one dimensional experiment [4] was used, so that it might be possible to compare the one dimensional with the two dimensional approach.

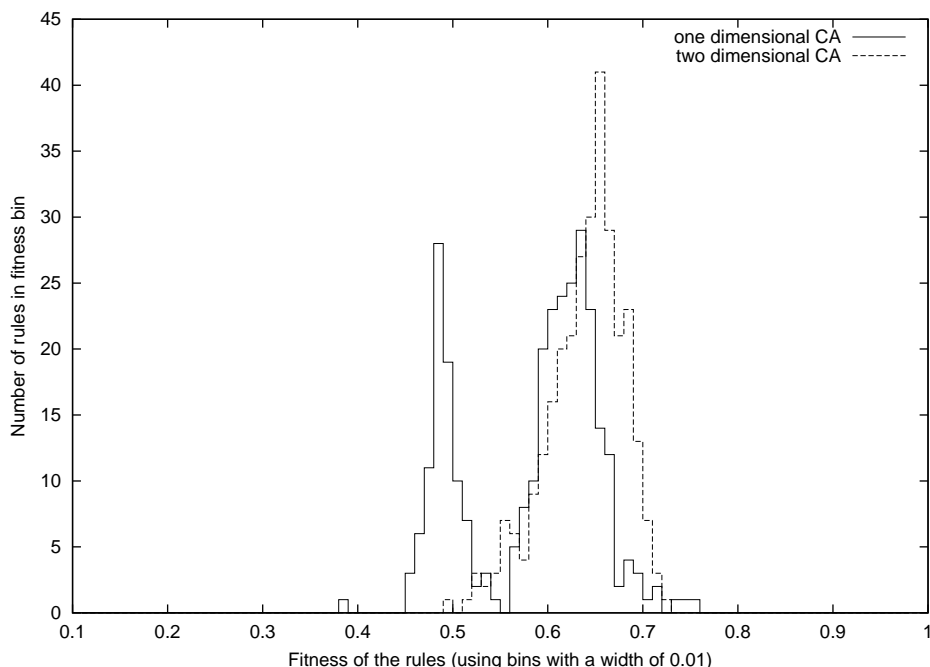


Figure 3.2: This figure displays the number of rules that have a certain fitness value in the two dimensional experiment and compares this to the one dimensional experiment. The fitness bins are 0.01 in width and for both algorithm  $F_{169,10^3}$  is calculated for 300 rules.

Preliminary experiments showed that it took much more time to evolve rules for the Moore neighborhood than was the case with the von Neumann neighborhood. The tests that were done with the Moore neighborhood also did not result in any encouraging results, which is consistent with [3]. That is why the von Neumann neighborhood was chosen for this experiment. Because this is a 5 bit neighborhood, the search space for CA rules is a lot smaller than in the one dimensional experiment. Instead of the  $2^7 = 128$  bits in the rule,  $R$  now consist of  $2^5 = 32$  bits. This means that the search space decreased from  $2^{128}$  to  $2^{32}$  and is now  $2^{(128-32)} = 2^{96}$  times smaller!

A two dimensional CA does not only have a width, but also a height. To make the two dimensional experiment comparable with the one dimensional version, a CA with width = 13 and height = 13 was used. This means that these CA have  $13 \times 13 = 169$  cells ( $N$ ) and are  $169 - 149 = 20$  cells larger than the one dimensional CA used in the original experiment. Note that increasing the size of a CA decreases the fitness of a rule in one dimensional CA as is shown in section 2.4 and it is reasonable to assume that this is no different for two dimensional CA.

In theory if ‘information’ were to ‘travel’ through the CA, it can do this only with a maximum step size equal to the radius of the neighborhood. Because the borders are linked, ‘information’ could be send in one direction and end up at the same position as it started from after  $i$  iterations where  $i = \min(\text{width}, \text{height})/r$ . In the two dimensional experiment this sums up to  $13/1 = 13$  iterations compared to the  $169/3 = 56.3$  for a comparable one dimensional CA. This is a good indication that  $I$  doesn’t need to be as high as it was for the one dimensional experiment and that will speed up the algorithm, therefore  $I$  was set to 50.

No other variable of the genetic algorithm as described section 2.4 was changed. The

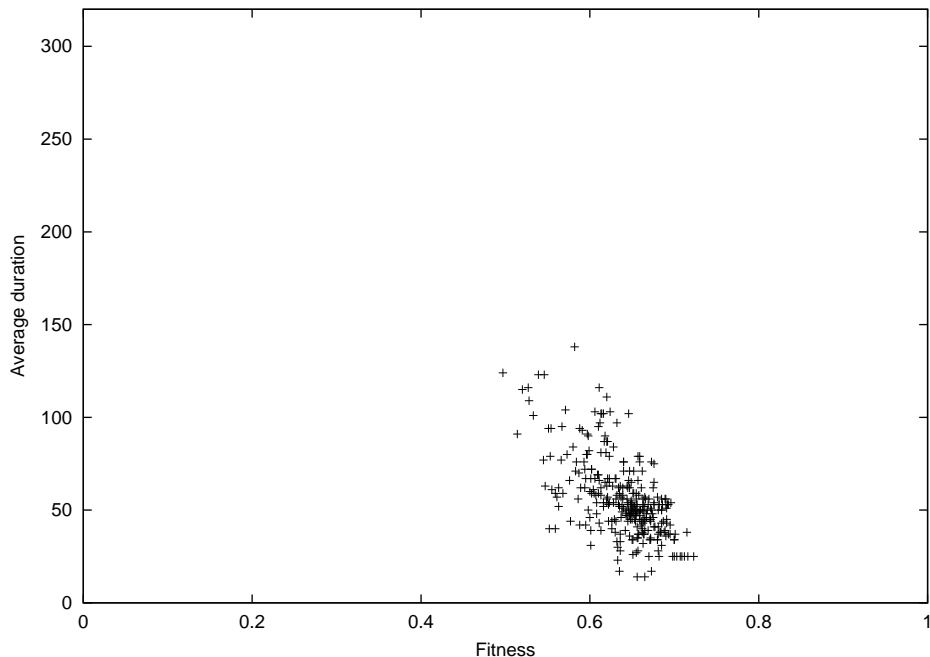


Figure 3.3: This figure displays the average duration of runs for the two dimensional algorithm ( $D_{N,M}$ ). For this algorithm  $N = 13 \times 13 = 169$  and  $M = 10^3 = 1000$ .

algorithm did 300 runs and tested all these rules by calculating  $F_{N,M}$ . For this  $F_{169,10^3}$  was used. The results of this test are shown in figure 3.2. The striking difference between this fitness distribution and the fitness distribution of the one dimensional rules is the absence of the peak around  $F_{N,M} \approx 0.5$ . In the two dimensional case almost all the evolved rules have a fitness above 0.58. A fitness around 0.66 seems to be average and the best rules have a fitness above 0.7. That is all very surprising taking into account that the Neumann neighborhood only consists of 5 cells.

To compare the duration (number of iterations) of the two algorithms, the maximum duration of the two dimensional algorithm was set to 320. This way there would be no difference in duration between runs that would not reach a correct end state in the different algorithms. Figure 3.3 shows the duration times as a function of the fitness for the two dimensional algorithm. These times are a lot different from the duration times in Figure 2.7.

In Figure 2.7 there are three groups, whereas in Figure 3.3 there is clearly only one group. What is also very striking is that the duration time seem to increase if the fitness increases in the one dimensional experiment, but it seems to decrease in the two dimensional experiment. This could mean that the two dimensional space is more suitable to solve the Majority Problem than the one dimensional space is.

This is enforced by figure 3.4 where it is clearly displayed that the two dimensional algorithm takes less computing time to process. Note that this is mainly due to the decreased maximum duration and the distance particles travel through the CA.

The best rule found in this experiment was tested with larger CA. Table 3.2 compares these results with the one dimensional experiment and shows how this rule does not outperform a one dimensional particle based rule, yet it does a lot better than the best

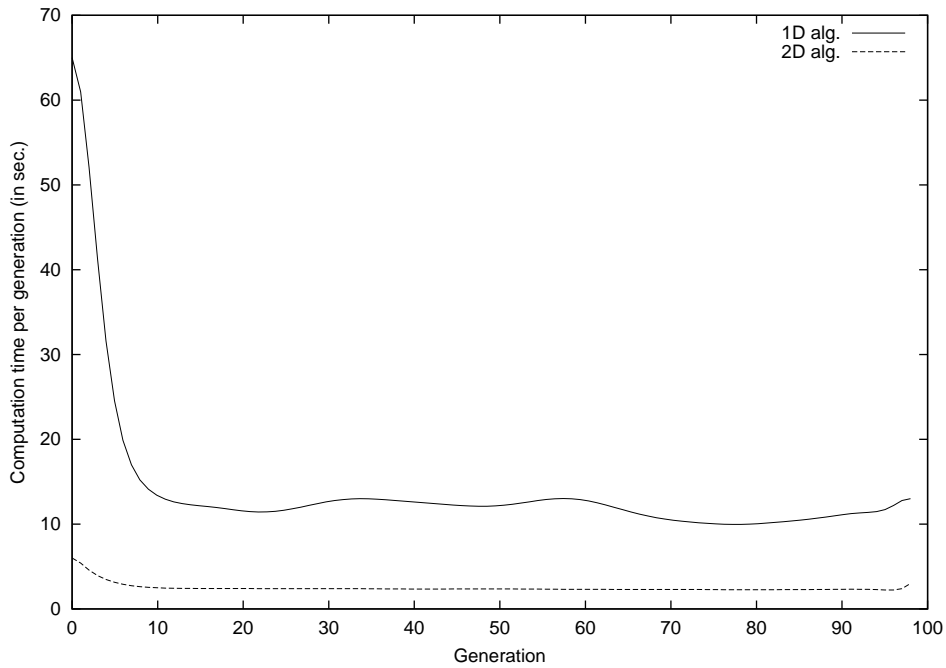


Figure 3.4: This figure displays the difference in computing time between the one dimensional algorithm and the two dimensional algorithm with a Neumann rule. The two tests were run on the same computer with similar loads. This difference is mainly due to the maximum duration and the distance particles need to travel.

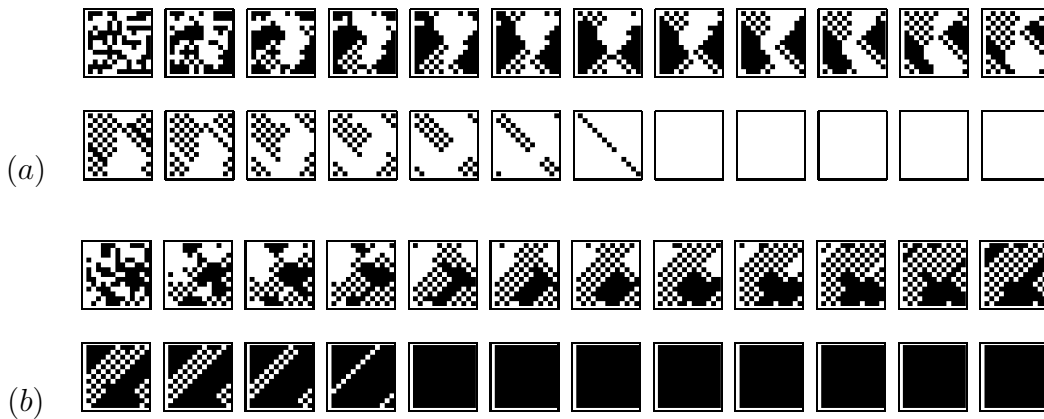


Figure 3.5: This figure shows two correct classifications of the Majority Problem by a two dimensional CA with both width and height equals to 13,  $\lambda = 84/169$  in (a) and  $\lambda = 85/169$  in (b). The transition rule was one of the best tested in the experiment and scored  $F_{169,10^3} = 0.715$ .

| <i>Rule name</i>       | <i>Fitness for different widths</i> |                |                |
|------------------------|-------------------------------------|----------------|----------------|
|                        | $F_{149,10^4}$                      | $F_{599,10^4}$ | $F_{999,10^4}$ |
| Best block-expand rule | 0.652                               | 0.515          | 0.503          |
| A particle-based rule  | 0.742                               | 0.718          | 0.701          |
| Two dimensional rule   | 0.702*                              | 0.665*         | 0.655*         |

Table 3.2: This table shows how the two dimensional rules compare to the one dimensional rules in terms of fitness. Note that the two dimensional approach is easily better than the block-expanding approach, but not as good as a particle-based rule. Also note how the two dimensional rule performs very consistent with larger CA sizes. (\* In order to compare the two dimensional rule with the one dimensional rules, the width and height of the two dimensional CA was set to  $\lceil \sqrt{n} \rceil$ . This corresponds to  $13 \times 13$ ,  $25 \times 25$  and  $33 \times 33$  respectively. Note that the CA for the two dimensional case are bigger.)

block expanding rule. It also seems that the two dimensional rule still performs well with bigger CA, just like the particle-based rule does. This is quite remarkable because the two dimensional rule only uses 5 cells instead of 7 in the one dimensional case.

The Majority Problem is a good example of a problem that forces cells in a CA to ‘communicate’ with another. The communication ‘particles’ can be seen in the one dimensional experiment, but are not easily spotted in the two dimensional experiment. That does not mean there are no ‘particles’ traveling in the two dimensional CA, because it might be very hard to identify these particles. In a two dimensional CA ‘particles’ are no longer restricted to travel in only one direction, but can travel to multiple directions at the same time. Traveling particles in two dimensional CA can therefore look like expanding areas with a distinct border. But there might be multiple particles traveling at the same time, meeting each other and thereby creating new particles. This is why communication between cells in a two dimensional CA is not very visible in the Majority Problem, although results show that this communication is present.

## 3.2 AND and XOR Problem

To show the communication between cells in a two dimensional CA a different experiment was conducted. A genetic algorithm was used to evolve rules for two dimensional CA that could solve the simple binary operators AND and XOR. These operators both have two input values and one output value which can only be determined if both input values are known. This is unlike the OR operator for example where the output value is always one if one or more of the input values is one, so if only one input value is known to be one then the value of the other input value is not needed. This may look very trivial, but it is very important in order to force the CA to combine the two values and thereby communicate.

### 3.2.1 Experiment details

To show the communications in a CA the information that needs to be combined must be initialized as far apart as possible. The following problem definition takes this into account:

| $a$ | $b$ | OR | AND | XOR |
|-----|-----|----|-----|-----|
| 0   | 0   | 0  | 0   | 0   |
| 0   | 1   | 1  | 0   | 1   |
| 1   | 0   | 1  | 0   | 1   |
| 1   | 1   | 1  | 1   | 0   |

Table 3.3: This figure shows the three main binary operators.

*Given a square CA with two ‘input cells’, one top left and one bottom right: find a rule that iterates the CA so that after  $I$  iterations the CA is in an ‘all one’ state if both the ‘input cells’ were one in the initial state and in an ‘all zero’ state otherwise.*

Small two dimensional CA were used with a width and a height of 5 cells and  $I$  was set to 10. The borders of the CA were unconnected to allow a larger virtual distance between the two corner cells. That means that the left most cell in a row was not connected to the rightmost cell in the same row and the topmost cell was not connected to the bottommost cell as was done with the Majority Problem experiment. Instead every cell in the border of the CA was connected to so called ‘zero-cells’. These ‘zero-cells’ stay zero whatever happens.

When using two input cells, there are four different initial states. These states are written as  $S_{(v_1, v_2)}$  where  $v_1$  and  $v_2$  are the two input values. All cells other than the two input cells were initialized with zero.

The fitness of a rule is defined as the total number of cells that have the correct values after  $I$  iterations. The number of ones in iteration  $t$  is written as  $O_{(v_1, v_2)}^t$ . The total fitness of the AND problem is defined as  $f = (N - O_{(0,0)}^I) + (N - O_{(0,1)}^I) + (N - O_{(1,0)}^I) + O_{(1,1)}^I$ . This makes the maximum fitness equal to  $4 \times 5 \times 5 = 100$ .

In this experiment another very simple genetic algorithm was used. A generation step starts by sorting the rules according to their fitness. Then it selects the top ten percent of the rules as ‘elite’ rules and copies them without changes to the next generation. Every ‘elite’ rule is then copied nine times or is used in single-point crossover to make the other 90 percent of the population. Both methods were tested and compared. The generated rules are mutated and also moved to the next generation. Mutation is done by flipping every bit in the rule with a probability  $p_m$ . The algorithm stops if it found a rule with  $f = 100$  or it reaches 1000 generations. In preliminary experiments a number of different values of  $p_m$  were tested. Setting  $p_m$  to a rather high value of 0.05 seemed to be the most effective choice.

The algorithm was run 100 runs with and without single-point crossover and using both the von Neumann and the Moore neighborhoods. The results are shown in table 3.4.

Although rules evolved with the von Neumann neighborhood were not able to solve the problem perfectly, it is already surprising that it found rules which work for 93 percent, for such a rule only misplaced 7 cells in the final state, all the other 93 cells have the correct value. This suggests that the information was combined, but the rule could not fill or empty the whole square using the same logic.

Figure 3.6 (a) shows how a von Neumann rule tries to classify the AND problem. It looks like the approach has three stages. In the first stage (step 1 to 3) the two information

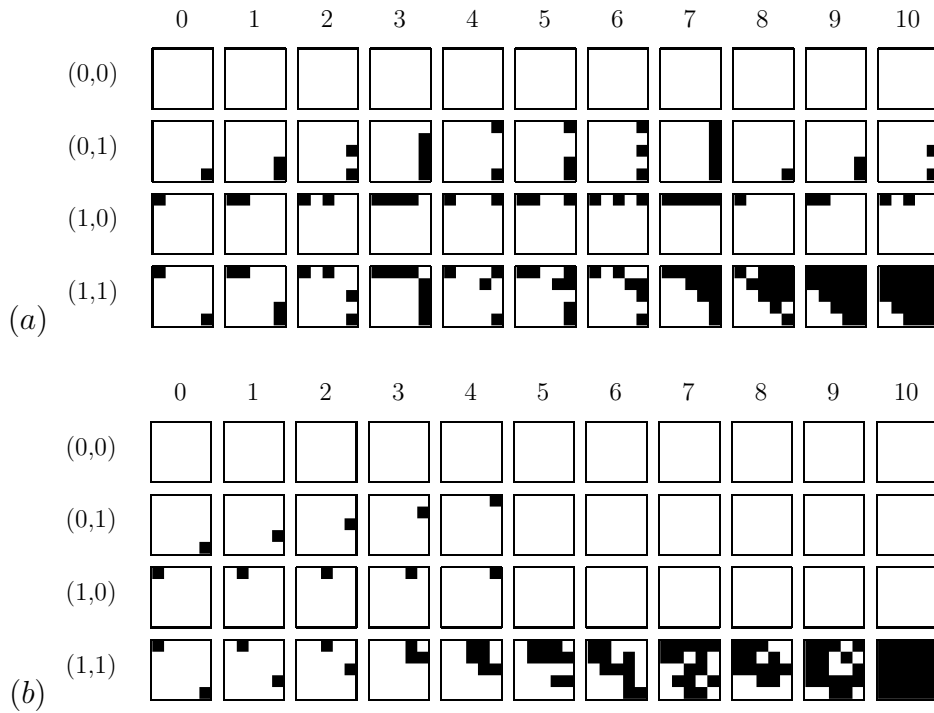


Figure 3.6: This figure displays the iterations of a CA solving the AND problem. Every row shows the iteration of the rule using a different initial state. Note that in the first column ( $t = 0$ ) the initial states are clearly visible and in the last column the coloring matches the output of an AND port. In (a) the von Neumann rule is used and in (b) Moore rule is used. Note that (a) is not a perfect solution although it does show communication between particles.

Table 3.4: Fitness values found in the AND problem.

| <i>Fitness</i> | <i>Number of runs</i> |                          |                       |                          |
|----------------|-----------------------|--------------------------|-----------------------|--------------------------|
|                | <i>Neumann</i>        |                          | <i>Moore</i>          |                          |
|                | <i>with crossover</i> | <i>without crossover</i> | <i>with crossover</i> | <i>without crossover</i> |
| 100            | 0                     | 0                        | 31                    | 21                       |
| 98-99          | 0                     | 0                        | 41                    | 54                       |
| 95-97          | 0                     | 0                        | 14                    | 25                       |
| 90-94          | 77                    | 93                       | 14                    | 0                        |
| 80-89          | 23                    | 7                        | 0                     | 0                        |
| 70-79          | 0                     | 0                        | 0                     | 0                        |
| < 70           | 0                     | 0                        | 0                     | 0                        |

Table 3.5: Fitness values found in the XOR problem.

| <i>Fitness</i> | <i>Number of runs</i> |                          |                       |                          |
|----------------|-----------------------|--------------------------|-----------------------|--------------------------|
|                | <i>Neumann</i>        |                          | <i>Moore</i>          |                          |
|                | <i>with crossover</i> | <i>without crossover</i> | <i>with crossover</i> | <i>without crossover</i> |
| 100            | 0                     | 0                        | 0                     | 1                        |
| 98-99          | 0                     | 0                        | 4                     | 4                        |
| 95-97          | 0                     | 0                        | 7                     | 6                        |
| 90-94          | 2                     | 1                        | 19                    | 21                       |
| 80-89          | 76                    | 96                       | 69                    | 66                       |
| 70-79          | 18                    | 3                        | 1                     | 2                        |
| < 70           | 4                     | 0                        | 0                     | 0                        |

particles travel to the topright corner, in step 4 the particles are combined (if present) and in step 5 to 10 the area is filled if needed. This approach just does not seem to be able to complete the task in 10 steps. If the rule would be given two extra steps it would probably be able to fill the bottomleft corner in the (1,1) case, but the rule does not seem to have the power to get rid of the particles in the (0,1) and (1,0) case.

The Moore neighborhood is clearly more powerful and was able to solve the problem perfectly. Figure 3.6 (b) shows how the Moore rule does this. The three stages are clearly visible here as well, only now the information particles are combined a step earlier (step 3) and the rule is able to move the particles without leaving some kind of trail behind.

It is surprising that using crossover in combination with a Neumann neighborhood does not outperform the same algorithm without the crossover. This may be due to the order of the bits in the transition rule and their meaning. This will be discussed in section 3.2.3.

### 3.2.2 The XOR Problem

The XOR Problem is similar to the AND problem. The same genetic algorithm and the same CA setup was used. The only difference was the fitness function. The XOR problem is defined as follows:

*Given a square CA with two ‘input cells’, one top left and one bottom right: find a rule that iterates the CA so that after  $I$  iterations the CA is in an ‘all one’ state if only one of the ‘input cells’ was one in the initial state and in an ‘all zero’ state otherwise.* This means that the total fitness of the XOR problem is defined as  $f = (N - O_{(0,0)}^I) + O_{(0,1)}^I + O_{(1,0)}^I + (N - O_{(1,1)}^I)$ .

The algorithm was run with  $p_m = 0.05$  for a maximum of 1000 generations for 100 runs with both Neumann and Moore neighborhoods with and without single-point crossover. The results are shown in table 3.5.

These results support earlier finding in suggesting that single-point crossover doesn’t really improve the performance when used in a two dimensional CA. The results show that the algorithm using only mutation has found ways to solve this rather difficult communicational problem. The Neumann neighborhood seemed unable to perform perfectly, yet it came rather close with three rules classifying the problem for 92 percent. The algorithm

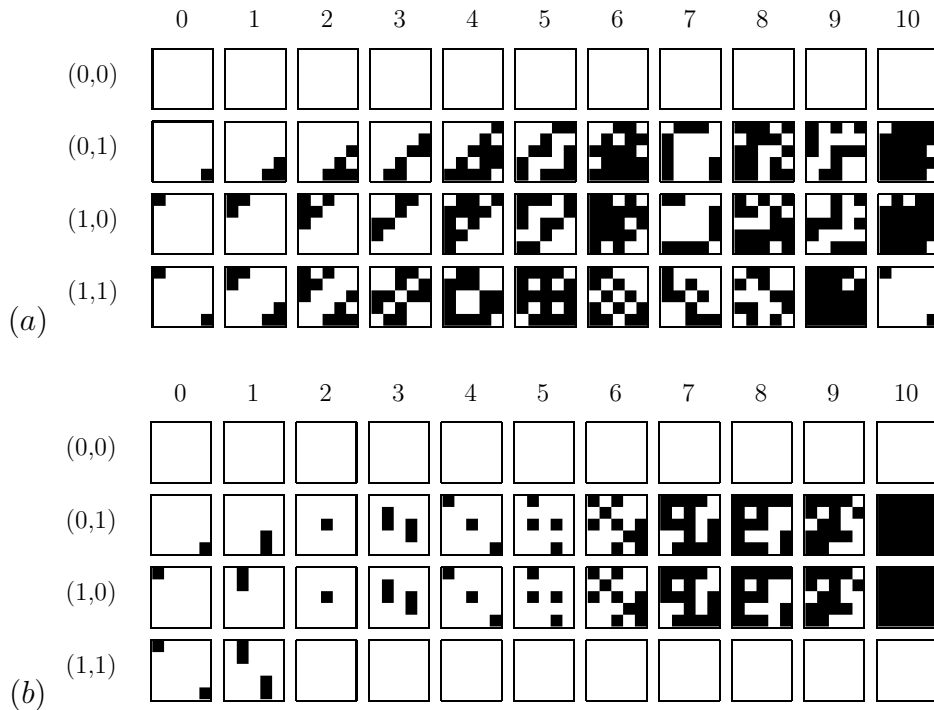


Figure 3.7: This figure displays the iterations of a CA solving the XOR problem using both a von Neumann and a Moore neighborhood. Every row shows the iteration of the rule using a different initial state. Note that in the first column ( $t = 0$ ) the initial states are clearly visible and in the last column the coloring matches the output of an XOR port. In (a) the von Neumann neighborhood is used and in (b) the Moore neighborhood is used.

found one transition rule using the Moore neighborhood that is able to solve the problem perfectly. These rules depicted in figure 3.7 show clear signs of “traveling particles” and are examples of how a local rule can trigger global behaviour. Even in figure 3.7 (b) it is clear that the particles are communicating even though the von Neumann neighborhood isn’t able to solve the problem perfectly.

Note that the information particles are combined as early as step 2 in the Moore neighborhood ((b)). This is possible because the particles can travel diagonally.

The AND and XOR problems were run on bigger CA with similar results. It seems that most rules found for the  $5 \times 5$  grid, also work on larger grids, because the process can be performed in the same 3 stages, it only takes more time. It was a lot harder to find Moore rules for a larger neighborhood though. This is probably due to the special way in which these rules try to fill the whole square at the end. With the von Neumann rules the only possible way to fill the whole square seems to be from one corner to the opposite, while with the Moore rules there are a lot more options. These options seem perfect for one size of a grid, but do not work well for other sizes.

### 3.2.3 Crossover and two dimensional mappings

As stated before in this document, the choice of GA was partly based on earlier findings with evolving CA [3, 4, 5]. These documents use single-point crossover together with mutation and they state no reason not to do so. In above experiments single-point crossover did not seem to contribute to better results, therefore it seemed fitting to elaborate on what happened in these results.

A CA rule is represented by a binary string so that a the GA can perform binary operations on a rule and evolve the rules using mutation and crossover. Every bit in the string represents a possible state in the CA and its value in the next iteration step. Single-point crossover combines two parents and tries to copying the best properties of the two parents by taking the left side of the string of one parent and combining it to the right side of the string of the other parent (as described in section 1.2.4). This works best if there exists some kind of local meaning in the bitstring so that there can at one time exist two parents with similar fitness that can be glued to form a better child.

In a two dimensional CA the cells in a neighborhood need to be numbered to give meaning to the corresponding bitstring. In experiments in this document all neighborhoods are numbered from left to right and then from top to bottom. At first it seems that this numbering is of no consequence to the outcome of the experiments, but when using single-point crossover it is important. This mapping for instance means that the first half of the bits in the string all represent states of the CA in which “the first” cell in the neighborhood is zero and in the other half of the bits this cell is one. This implies that single-point crossover will always copy the behaviour of one single parent concerning all the states of the CA where the first cell has a certain value. This could be explained as local meaning and hence single-point crossover would seem a good idea, yet the results in the experiments above all show something else.

In figure 3.8 the difference between using crossover together with mutation and using only mutation becomes clear. When using crossover, the algorithm sometimes gets stuck at around a fitness of 65, whereas the lowest fitness when using only mutation is about 78. The results suggest that using crossover speeds up the beginning of the evolution, but slows down the rest. Single point crossover also seems to increase the chance that the algorithm gets stuck.

It is unclear why this happens, although single-point crossover does have some limitations:

- If a problem has more than one optimum and the algorithm is closing in on more of them, then any form of crossover will create a lot of individuals somewhere in between the optima. This causes creative new possible answers and can be a good thing, but if that is not needed it slows down the algorithm.
- If a certain aspect of an individual is not defined in one position or area of the binary string, it is impossible to copy only that aspect from a parent to a child and a lot of other aspects of the parent will be copied with it. This happens because single-point crossover can only copy bits up until or from a certain position to the child.

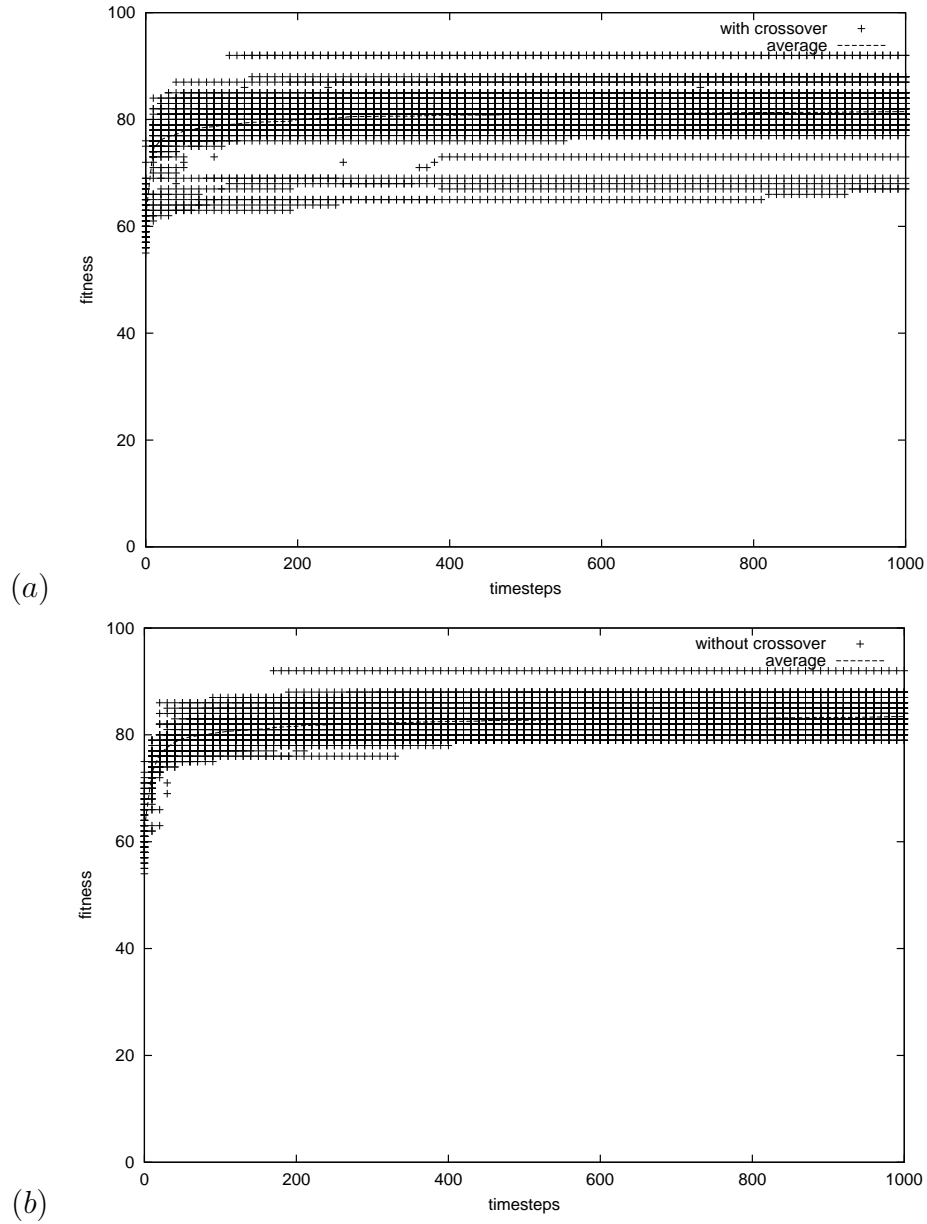


Figure 3.8: This figure shows the effect of using single-point crossover on the XOR problem with a von Neumann neighborhood. In (a) crossover is used together with mutation and in (b) only mutation is used. All 100 runs are plotted here together with the average. Note that using single-point crossover makes the algorithm less reliable than using only mutation.

Especially the last limitation seems very important in two dimensional CA. Behaviour is not defined by one cell in the neighborhood, but by the state of all those cells. If for instance the behaviour should be simply to move a particle from left to right then more than one state would be effected and this behaviour would be positioned throughout the entire binary string. Because the meaning in the rules is so scrambled, it is desirable to be able to combine the different parts of the string of one parent with parts of the other, but that is impossible in one step with single-point crossover.

There are other forms of crossover that do not have this limitation. Uniform crossover decides for every bit that is copied to a child which parent it will be copied from (section 1.2.4). Some short testing with this form of crossover did not result in better results, but this might be due to the different distribution of “the number of bits that are from one parent”. In single-point crossover this number is uniformly distributed, whereas in uniform crossover this number is normally distributed. This means that the chance that an individual is changed only a few bits is very small. Further reseach into two dimensional mappings seems needed as well as more tests with different types of crossover.

Some experiments were also conducted using “Gray code” mapping instead of the normal binary representation. Gray code is a code wherein the encoding of every adjacent integer only differs by exactly one bit. When the states of the CA are sorted in this way in theory the distance between two states is a lot smaller. The results did not support this theory and thus can not give a clear answer. It might be that although the states are better sorted, single-point crossover is still too limited or that the logic of traveling particles needs a totally different ordering all together and that using Gray code does not improve anything.

### 3.3 Evolving bitmaps

Now that it is shown that two dimensional CA can communicate, it is time to increase the challenge for the CA a bit. The aim of this experiment is to evolve rules for two dimensional CA that generate patterns (or bitmaps). This is a rather big challenge for a system based on local rules, but yields surprising results non the less.

The Bitmap Problem is defined as follows:

*Given an initial state and a specific desired end state: can the genetic algorithm find a rule that iterates from the initial state to the desired state in less than  $I$  iterations.*

Note it is not requested that the number of iteration between the initial and desired state is fixed. This number may be any number between 1 and  $I$ . This multiplies the search space of the problem by  $I$ , but that is not a problem in the computational sense, for all iterations have to be processed anyway. There is no harm in checking them while iterating. In preliminary experiments it was found that a fixed number of iterations sometimes makes it impossible for a CA to reach the desired state [7]. The CA is also not expected to stay at the desired state as previous experiments did expect. The neighborhood of a rule in the CA is very small and therefore it is already very difficult to make a rule stop altering the CA ones it has reached the desired state, but for a rule to go from the initial state to a desired state in the CA and then stay there seems a bit to challenging.



Figure 3.9: The bitmaps used in the pattern generation experiment.

Table 3.6: Number of successful rules found per bitmap.

| <i>Bitmap</i> | <i>Successful rules<br/>(out of a 100)</i> |
|---------------|--|
| “square”      | 80   |
| “hourglass”   | 77   |
| “heart”       | 35   |
| “smiley”      | 7  |
| “letter”      | 9  |

The CA used in this experiment is similar to the one used in the AND/XOR experiment (section 3.2). In preliminary experiments different sizes of CA were tried, but it was decided to concentrate on small square bitmap with a width and a height of 5 cells (as done in section 3.2). To make the problem harder and to stay in line with earlier experiments the CA has unconnected borders like in section 3.2. To make the problem even more challenging the von Neumann neighborhood was chosen instead of the Moore neighborhood and therefore the  $s_n$  consist of 5 cells ( $r = 1$ ) and a rule can be described with  $2^5 = 32$  bits. The search space therefore is  $2^{32} = 4294967296$ . A bitmap of 32 b/w pixels would have the same number of possibilities, therefore this experiment is very challenging to say the least.

After testing different initial states, the ‘single seed’ state was chosen and defined as the state in which all the positions in the CA are zero except the position ( $\lfloor width/2 \rfloor, \lfloor height/2 \rfloor$ ) which is one. The theory being that this ‘seed’ should ‘grow to be’ the desired state.

For the GA the same algorithm was used as in the AND and XOR experiments, because this experiment uses a Neumann neighborhood and the AND and XOR experiments suggested that the combination between the von Neumann neighborhood and single-point crossover was not a good idea, this experiment used only mutation. Like in section 3.2 mutation is performed by flipping every bit in the rule with a probability  $p_m$ . In this experiment  $p_m = 1/\{\text{number of bits in a rule}\} = 1/32 = 0.03125$ .

In trying to be as diverse as possible five totally different bitmaps were chosen, they are shown in figure 3.9. The algorithm was run 100 times for every bitmap for a maximum of 5000 generations and  $I = 10$ . The algorithm was able to find a rule for all the bitmaps, but some bitmaps seemed a bit more difficult than others. Table 3.6 shows the number of successful rules for every bitmap. Note that symmetrical bitmaps seem to be easier to generate than asymmetric ones, although they are also generated by the CA.

Although this experiment is fairly simple, it does show that a GA can be used to evolve transition rules for two dimensional CA that are able to generate many different pattern even with a simple von Neumann neighborhood. Figure 3.10 shows a few successful transition rules generated by the GA. Note that different transition rules can end up in the same desired state and have totally different iteration paths.

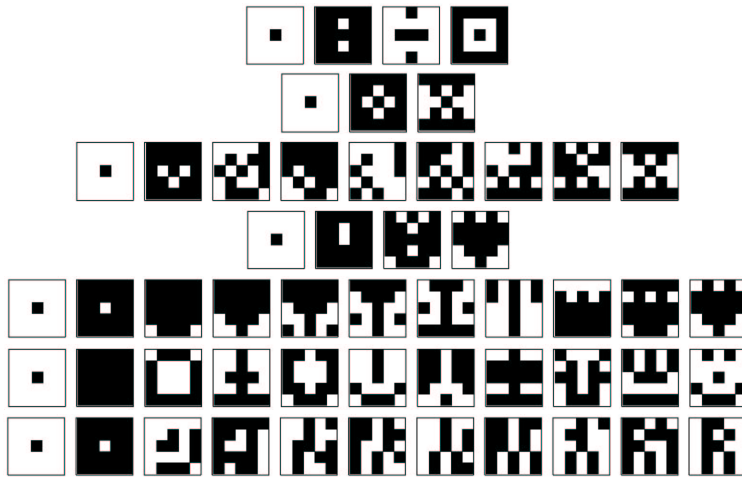


Figure 3.10: This figure shows some iteration paths of successful transition rules.

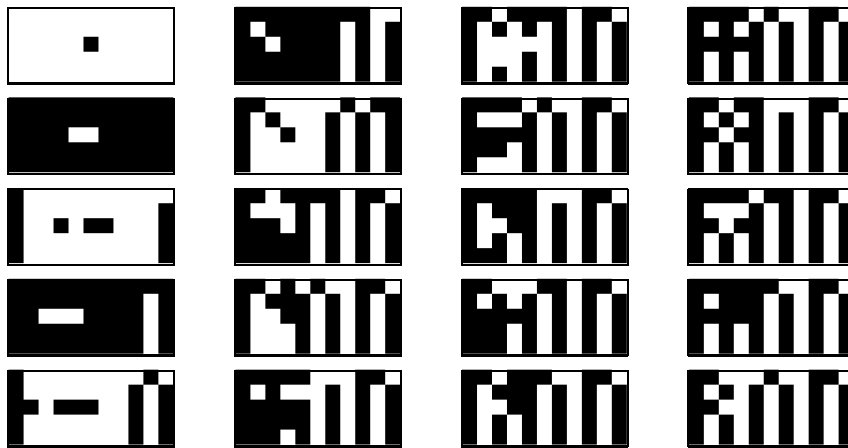


Figure 3.11: This figure shows the iteration of a two dimensional CA with a width of 11 and a height of 5. A rule was iterated to display a bitmap representing the name 'RON' with a von Neumann neighborhood with  $r = 1$ . States are ordered from top to bottom and then from left to right. The rule is able to generate the pattern with only three errors, being the three bottom cells of the 'O'. A  $5 \times 11$  bitmap has  $2^{55}$  different states, while there are 'only'  $2^{32}$  two dimensional von Neumann rules with  $r = 1$ . That means this is a surprising result and encourages further research.

When this approach is used in larger CA it becomes interesting to think about direct applications in the real world. One such application could be compression. Because a bitmap could be defined by the von Neumann rule that successfully generates it from a single seed state together with the duration this rule needs to iterate. With  $I = 10$  this means 32 bits for the rule and 4 for the duration and that means compression is possible for all bitmaps exceeding 36 cells. Note that it is impossible to be able to compress every possible bitmap defined by more than 36 pixels into less bits, the compression will always reduce the number of possible outcomes, but the hope is that the impossible outcomes are less likely to occur. Some small compression experiments were conducted with some success. Bitmaps with a width and height equals to 7 were generated within 10 steps using the von Neumann neighborhood. It seems that especially symmetrical features are easily compressed.

Ongoing experiments with even bigger CA suggest that they do not differ much from the small ones, although the restrictions on what can be generated from a single-seed state using only a von Neumann neighborhood seem to be bigger when size of the CA increases. Lossy image compression seems to be an option too, because even when the algorithm does not find the bitmap exactly, it seems to find a lot of features and the bitmap might still be recognizable. Also  $I$  can of course be increased to 16 (which is still 16 bits) and even much higher than that. Further research must find out if this is a valuable application or only works on small toy examples.

# Chapter 4

## Summary and Outlook

This document shows how two dimensional CA are able to solve the majority problem with similar results compared to one dimensional CA used in [4, 5]. Using the same GA as in [4, 5] a better average fitness was achieved suggesting that evolving two dimensional CA is easier and more reliable.

The document shows that two dimensional CA can show communicational behaviour in the form of the AND and XOR problems and that this behaviour can be evolved using a GA. The document also shows that a more generic behaviour can be evolved using a GA by showing how different patterns can be iterated from the same initial state.

These results all suggest that a multi dimensional CA is a very powerful tool and in combination with GAs they can be evolved to exhibit a specific desired behaviour. It is therefore not unthinkable that this combination can be used to solve all sorts of real world problems.

With answers come new questions. This research raises a lot of questions and a lot of parameters in the documented experiments are not explored yet. Below is a list of proposed further work on this subject:

- Try three dimensional CA. Does the improvement from one to two dimensional space imply that more dimensions is better?
- Investigate different mapping methods in combination with different crossover functions. Try to find out what really happens during the algorithm and why crossover did not work as well as expected in two dimensional experiments.
- Explore the bitmap problem on larger grids and maybe with the Moore neighborhood. Explore if this approach is able to compress large images.
- Investigate how bitmaps are iterated in the bitmap problem. Find out how many bitmaps can be iterated and what features are most easy and which are hard.
- Maybe try some kind of hybrid neighborhood which is a crossing between the von Neumann neighborhood and the Moore neighborhood to explore further possibilities of a two dimensional solution to the Majority Problem.

# Bibliography

- [1] David, A., Forest, B., Koza, H.: Discovery by genetic programming of a cellular automata rule that is better than any known rule for the majority classification problem. (1996)
- [2] Gacs, P., Kurdyumov, G. L., Levin, L. A.: One dimensional uniform arrays that wash out finite islands. *Problemy Peredachi Informatsii*. 12(1978) 92–98
- [3] Inverso, S., Kunkle, D., Merrigan, C.: Evolutionary Methods for 2-D Cellular Automata Computation. [www.cs.rit.edu/~drk4633/mypapers/gacaProj.pdf](http://www.cs.rit.edu/~drk4633/mypapers/gacaProj.pdf) (2002)
- [4] Mitchell, M., Crutchfield, J.P.: The Evolution of Emergent Computation. Proceedings of the National Academy of Sciences, SFI Technical Report 94-03-012
- [5] Mitchell, M., Crutchfield, J.P., Hraber, P.T.: Evolving cellular automata to perform computations: Mechanisms and impediments. *Physica D*, (1994), 75:361– 391
- [6] Li, W., Packard, N. H., Langton, C. G.: Transition phenomena in cellular automata rule space. *Physica D*, (1990), 45:77–94
- [7] Packard, N.H.: Adaptation towards the edge of chaos. In: Kelso, J.S., Mandell, A.J., Shlesinger, M.F. (eds.): *Dynamic Patterns in Complex Systems*, Singapore: World Scientific, (1988) 293–301
- [8] Wolfram, S.: Statistical mechanics of Cellular Automata. *Reviews of Modern Physics* volume 55 (1983)
- [9] Wolfram, S.: *Theory and Applications of Cellular Automata*. World Scientific, Singapore (1986)