Internal Report 2012–07 July 2012

Universiteit Leiden

Opleiding Informatica

Genetic Algorithms and Cellular Automata: unraveling the Bitmap Problem

Sjaak Wolff

MASTER'S THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Genetic Algorithms and Cellular Automata: unraveling the Bitmap Problem

Sjaak Wolff
sjaakwolff@gmail.com
Leiden Institute of Advanced Computer Science
Leiden University

July 30, 2012

Abstract

Using Genetic Algorithms to evolve Cellular Automata rules to solve a given problem is a well-known method. The Bitmap Problem however, with it's versatile and challenging characteristics, remains quite unknown. This thesis focuses on the Bitmap Problem and tries to expose its inner workings, possibilities and pitfalls. Multiple aspects of the Bitmap Problem like grid size, state set and updating method are being adjusted to measure their influence. In the final chapter of this thesis an algorithm is described which uses the Bitmap Problem as key ingredient for a data compression algorithm.

Contents

Chapter 1

Introduction

The Bitmap Problem brings together two subjects within the field of Natural Computing. It is a problem within the field of Cellular Automata (CA) [12, 15, 17] and Genetic Algorithms (GA) [1, 10] are a good way to find solutions to it. CA date back over half a century to the 1940s from the work of John von Neumann on self-replicating systems [14]. CA consist of a set of state machines, or cells, which are situated in an array or grid. Each cell resides in one of a finite number of states. The state of a cell can change to another state from the state set as time passes in discrete time steps. A local rule deterministically decides what the state of the cell becomes at the next time step considering the direct neighborhood of that cell. The CA that John von Neumann created, with help of his collegue Stanislaw Ulam, consisted of a 2-dimensional grid of cells each with 29 possible states and used the now well known 'von Neumann neigborhood' to iterate the system over time. The CA made endless copies of a certain initial pattern; the first self-replicating automata.

The Bitmap Problem is the challenge of finding a combination of local rules that iterates the CA from an initial configuration to a desired configuration within a given amount of steps. Altough the CA used for the Bitmap Problem can be of any grid size and state set, it proves to be a very difficult task to find successful rules. To discover its difficulties, possibilities and boundaries, the Bitmap Problem will first be solved for relatively small grid sizes and state sets. This is where GA come into play. There are many possible rules to consider when looking for a rule that will successfully solve a given Bitmap Problem. The very large search space of rules makes it impossible to find a successful rule by simply brute force or randomly trying out different rules. GA provide a smarter and more efficient way to search within these very large search spaces.

1.1 Cellular Automata

Cellular Automata (CA) form a lively and actively studied subject within the field of Natural Computing. Nowadays many variations to the original concept have been constructed. CA are being used in many types of simulations and observations because of the emergent global behaviour of the system based on simple local rules. The research in this thesis is largely based on the subject of CA and this section will provide the reader with the basic required knowledge on CA.

1.1.1 Definitions and example

A Cellular Automaton (CA) is a discrete and deterministic dynamical system, consisting of a collection of finite state machines called cells [12, 15, 17]. These cells are interconnected locally with each other forming a d-dimensional lattice, or grid. In the case of a one dimensional CA this would be an array, or row, of length n. Where n is the amount of cells in the CA.

$$A = \{a_1, a_2, ..., a_n\}$$

Where A is a one dimensional CA with n cells and the cell at position a_n is generally adjacent to the cell at position a_1, forming a ring-like shape.

The states in which the cells can reside in are defined by a finite state set

$$S = \{s_1, s_2, ..., s_p\}$$

p being the number of elements in the state set. Most CA in this thesis will use the simplest state set, which is the binary state set where $S = \{0, 1\}$.

The state of the cell at position a_1 at time t is written as a_1^t. The configuration c of the entire CA at time t is denoted as c^t and consists of the states in which the cells of the CA reside in at that particular point in time.

$$c^t = \{a_1^t, a_2^t, ..., a_n^t\}$$

The cells can change their internal state at discrete time steps. At each time step the states of the cells are evaluated and updated. this happens either synchronously, all at once, or asynchronously, one at a time. Synchronous updating is the most widely used updating method. Asynchronous updating will be more elaboratly discussed in section 1.1.4. What the state of a given cell at the next time step will be, depends on the current states of its neighborhood cells. Exactly which cells are to be considered the neighborhood cells differs. In our one dimensional case, the neighborhood N_i of cell a_i is defined as the local set of positions with a certain distance, or radius r, of a_i.

$$N_i = \{a_{i-r}, a_{i-r-1}, ..., a_i, ..., a_{i+r-1}, a_{i+r}\}$$

When taking the ring structure of the CA into account, this means that $N_1 = \{a_{148}, a_{149}, a_1, a_2, a_3\}$ for $r = 2$ and $n = 149$. Note that the size of the neighborhood in one dimensional CA always equals to $2r + 1$.

An update rule, or local transition function Φ, determines what the state of a given cell will be by considering the current state of its neighborhood.

$$\Phi : S^q \to S$$

Where q is the size of the neighborhood.

This means for a given cell a_i at time t, with its neighborhood N_i^t:

$$a_i^{t+1} = \Phi(N_i^t)$$

N_i^t being the state of the neighborhood of cell a_i at time t.

The local transition function Φ is applied to all cells at each discrete time step and thereby determines the global dynamics, or behavior, of the CA. At each time step, the configuration c^t is translated into c^{t+1}.

$$c^{t+1} = \{\Phi(N_1^t), \Phi(N_2^t), ..., \Phi(N_n^t)\}$$

Let C be the set of all possible CA configurations, $C = S^n$. The Global Transition function G is the function $G : C \to C$, so $G(c^t) = c^{t+1}$.

To summarize the above, a Cellular Automaton is a 4-tuple (d, S, N, Φ) where d is the dimension of the grid of cells, S is the state finite set, N the used the neighborhood setting and Φ is the local transition function. Two other variables could be added to more accurately describe the CA, being c^0 as the initial configuration of the CA and *upd*, the used update method, which is either synchronous or asynchronous.

To see how a Cellular Automaton actually works, we will consider a CA in one of its most simple forms; the Elementary Cellular Automata. Extensive research has been conducted to the Elementary Cellular Automata by Stephan Wolfram [17] and later on have been proven able to perform Universal Computation [6]. The Elementary Cellular Automaton consists of a one dimensional array of cells. This array is usually considered infinite in size, but in practical studies the array is given a size and the first and last cell are considered neighbors, making it a circular or ring like shape. The state set of the Elementary CA consists of two states $\{0, 1\}$ (binary). Generally visually represented as $Black$ or $White$ squares. The cells are updated synchronously at each time step with a local neighborhood of radius one, making the neighborhood three cells in total: the cell itself and the cells directly adjacent to the left and right. The amount of possible different neighborhoods, given a neighborhood size of three and a binary state set, is $2^3 = 8$. The transition function Φ maps each of these neighborhood configurations into a state from the state set. This will be the state of the cell at the next time step, see Figure 1.1.

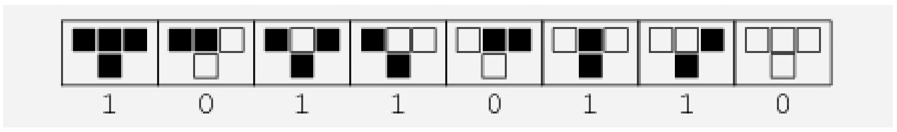


Figure 1.1: Picture of the 8 possible neighborhoods and their local transition function mappings in a Elementary Cellular Automata rule

Visualizations of one dimensional CA dynamics are often depicted as space-time diagrams. Horizontal rows are the consecutive configurations in time, where the top row is the initial configuration and subsequent rows represent the states further on in time. Figure 1.2 shows how such a space-time diagram is built step by step. The CA in the figure has an initial configuration of one black cell and uses the rule depicted in Figure 1.1.

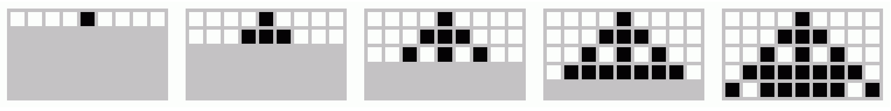


Figure 1.2: Picture showing how the rule depicted in Figure 1 influences the state of the cells of the CA in time, starting with an initial configuration of 1 black cell

1.1.2 Two-dimensional Cellular Automata

Instead of the row of cells we have seen in the one-dimensional CA, the two-dimensional CA consists of a grid, or lattice, of cells, somewhat like a checkerboard. The number of cells in the CA in this thesis is finite and therefore the grid has a certain width w and height h. The cells of the two dimensional CA require two coordinates to be addressed.

$$A = \begin{vmatrix} a_{1,1} & \cdots & a_{w,1} \\ \vdots & \ddots & \vdots \\ a_{1,h} & \cdots & a_{w,h} \end{vmatrix}$$

Where A is a two dimensional CA with width w and height h.

The borders can either be connected or unconnected. Connected borders means that the leftmost cell in a row is the neighbor to the rightmost cell of the same row. This is also the case for the bottom and top cells in a column, forming a torus topology. Unconnected borders does not have this feature and prevent information to 'flow' out of the borders to the other side of the grid. In order to still give the cells at the edges of the grid a full neighborhood, an imaginary border will be placed around the grid, having cells with a fixed value (in this thesis being '0' or 'white'). The setting of having connected or unconnected borders can have a huge impact on the dynamics of the CA over time.

Now that the cells of the two dimensional CA not only have neighboring cells to the left and right, but also above and beneath it, the neighborhood for the transition function has also been modified. Two commonly used neighbordhoods are the “von Neumann” and “Moore” neighborhood, named after their respective inventors [13, 14]. These neighborhoods can also be considered having a certain radius. Both neighbordhoods with a radius one are depicted in Figure 1.3.

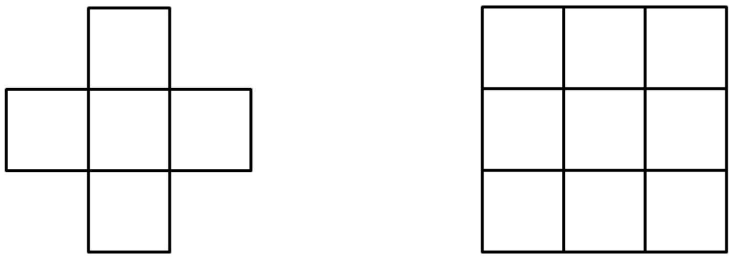


Figure 1.3: Two commonly used neighborhoods for two-dimensional Cellular Automata with a radius of one

As seen in the figure above, the neighborhoods shown consist of more cells than the Elementary CA described earlier. Increasing the size of the neighborhood also increases the amount of possible rules, or rulespace, of the CA. This will be discussed in more detail at section 1.1.3.

1.1.3 Rule encoding and rule spaces

Stephen Wolfram [17] introduced a naming scheme for the Elementary CA. Each Elementary Rule is specified by an eight-bit sequence, where every bit represents a mapping from a neighborhood configuration to the resulting state at the next time step. This eight-bit sequence can be interpreted as the binary representation of a decimal in the interval $[0, 255]$, the 'Wolfram number' of the Elementary Rule. For example, the rule depicted in Figure 1.1 is known as 'Rule 182', obtained by the binary expansion $181 = (10110110)_2$. All possible Elementary Rules can be represented by an 8 bit string and thus the amount of unique global transition functions is limited to $2^8 = 256$. This can be called the 'rulespace' of the Elementary CA. The encoding scheme is also applicable to more complex CA with larger neighborhoods and state sets. The size of the neighborhood N and the size of the state set S greatly influence the amount of unique rules that are constructable and will thereby also increase the length of the string representing the encoded rule. The size of the encoded rule is equal to the amount of neighborhoods which is given by the following formula:

$$|neighborhoods| = |S|^{|N|}$$

The rulespace size is then calculated by the fact that for each neighborhood there are $|S|$ possible outcomes, i.e.:

$$|rulespace| = |S|^{|S|^{|N|}}$$

The size of the rulespace increases dramatically by increasing the size either one of the sets S or N. Table 1.1 shows how the size of the rulespace increases by small increments of either one of the sets. The table shows for example, that the rulespace of the Elementary CA with neighborhood size $|N| = 3$ and state set size $|S| = 2$ equals $2^{2^3} = 2^8 = 256$. The rulespace for a two dimensional binary CA with the von Neumann neighborhood, like the well-known "Game of Life" [2, 5, 9], would be $2^{2^5} = 2^{32} = 4.294.967.296$.

		Neighborhood size $\|N\|$								
		1	2	3	4	5	6	7	8	9
$\|S\|$	1	1^1	1^1	1^1	1^1	1^1	1^1	1^1	1^1	1^1
	2	2^2	2^4	2^8	2^{16}	2^{32}	2^{64}	2^{128}	2^{256}	2^{512}
	3	3^3	3^9	3^{27}	3^{81}	3^{243}	3^{729}	3^{2187}	3^{6561}	3^{19683}
	4	4^4	4^{16}	4^{64}	4^{256}	4^{1024}	4^{4096}	4^{16384}	4^{65536}	4^{262144}

Table 1.1: Table that shows the influence on the size of the rulespace $|S|^{|S|^{|N|}}$ of both variables, size of state set S and size of neighborhood N.

1.1.4 Asynchronous updating methods

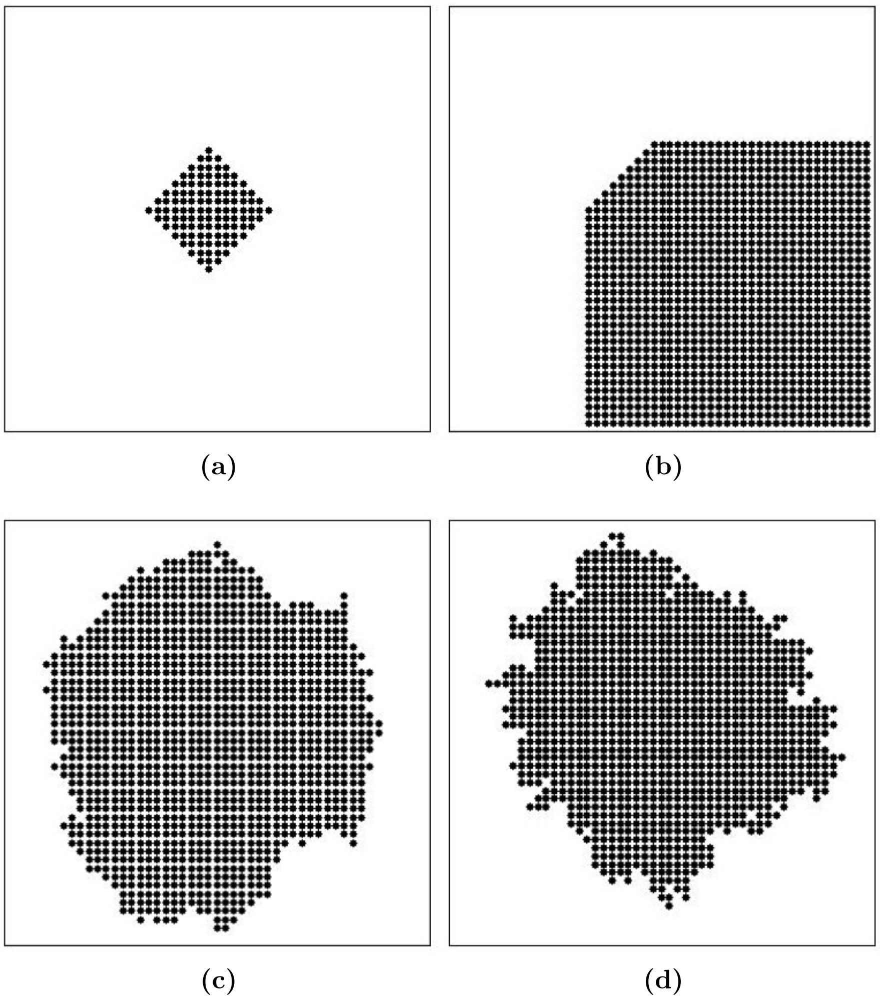
In contrary to the widely used synchronous updating method, where all cells are updated at the same time step, there are other ways of evaluating and updating cells in a CA. This means instead of evaluating all cells at once, evaluating the individual cells in the CA one at a time. This way, a cell's neighborhood could have changed multiple times, before the cell will be evaluated itself. The order in which the cells are to be evaluated needs to be specified in advance and the statistical properties of this order can have significant consequences for the dynamics and patterns generated by the CA.

There are several different algorithms to obtain such an updating order. However, only those that were used in this thesis will be described in this section. To read more on asynchronous updating, see [16]. Below follows a brief outline of each of the used asynchronous updating methods. Figure 1.4 shows the difference between the resulting configurations after using the same rule to iterate an initial configuration for a given amount of steps with synchronous and the various asynchronous updating methods.

Fixed directional, or *line-by-line sweep*, is the simplest form of asynchronous updating, where the cells of the CA are put in a predefined, fixed order to form a sequence in which the cells will be updated. This order will typically be from left to right and from top to bottom.

Fixed random sweep is quite similar to the simple line-by-line sweep method. In this case the sequence of cells is chosen randomly. The first cell of the sequence is chosen from all n cells in the CA, the second is chosen from the remaining $n - 1$ cells etc. In other words, the sequence is constructed by uniform distribution without replacement. The same sequence will then be used for all iterations, making it not that different from the line-by-line method, which is actually a special case of the fixed random sweep.

Random new sweep has its commonalities to *fixed random sweep*. This time however the sequence is not fixed and after each sweep through the grid, a new sequence is chosen by uniform distribution without replacement. The fixed random sweep would be a special case of random new sweep where by chance, the same sequence is chosen every time.


(a)

(b)

(c)

(d)

Figure 1.4: Pictures showing the different resulting configurations of the different updating methods using the same rule: a cell becomes black if at least one of it's von Neumann neighbors is black. The rule is iterated 8 steps for synchronous updating and 8 sweeps for the asynchronous updating methods. (a) Synchronous updating (b) Line by line sweep (c) Fixed random sweep (d) Random new sweep. Image concept borrowed from [16].

1.2 Genetic Algorithms

Genetic algorithms (GA) [1, 10] are part of the field of Evolutionary Computation, an interesting and useful subject in within the field of Natural Computing. A GA is used for solving search and optimization problems in high-dimensional search spaces. They are based on the paradigm of natural evolution; trying to mimic nature in evolving organisms to best suit the environment in a way that is known as 'survival of the fittest'. The research described in this thesis makes extensive use of GA and this section will give a short introduction to the field by providing an overview of their workings.

1.2.1 Evolutionary Loop

Population of individuals

A Genetic Algorithm contains a population, or 'pool', of one or more individuals. Each of these individuals represent a solution to problem which is to be solved. These representations of solutions are called 'genotypes', the solutions which they represent are called 'phenotypes'. An individual's genotype is somewhat comparable to a DNA string of living organisms; it is not the organism itself, but describes what it is, does and what it looks like. The genotype is typically represented by a vector of values $\vec{a}$, in our case this will be a bit string of length l: $\vec{a} = (a_1, ...a_l) \in \{0, 1\}^l$. The population of the genetic algorithm consists of λ individuals, each having their own genotype.

Fitness

The genetic algorithm starts by randomly initializing the λ individuals. Each individual in the population now represents a solution to the given problem. These random solutions will most likely be far from optimal, but some may be better than others. So now we have to be able to decide which individual is better or 'more fit' than another. This will be done by using a fitness function. This usually means mapping the genotype to the phenotype first and then calculating how good or bad the solution performs, for example by running some sort of simulator. This is usually the most expensive part of the algorithm, in terms of computation time. The fitness function then rates the solution and attaches this rating to the individual.

Exit criteria

After calculating the fitness of each individual, the exit criteria of the algorithm are checked. Reasons for the algorithm to stop could be that one of the individuals in the population has reached the maximum possible fitness rating, meaning that a perfect solution was found. Another reason would be that the maximum amount of time was spent or the maximum amount of generations has been reached. Other criteria could be thought of as well. As long as these criteria are not met, the algorithm will continue with the next step.

Selection

When all individuals have been given a fitness rating, a selection is made based on these fitness ratings, to decide which μ individuals will serve as 'parents' for the next generation of individuals. The individuals that were not chosen will be discarded and removed from the population. The selection can be done in multiple ways, some of which will be shortly described in section 1.2.2. The main differences between the various selection methods lie in the amount of 'luck' a 'bad' individual can have in still being chosen as a parent. E.g. very strict selection methods will always only choose the best individuals of the population, thereby denying the population to retain its diversity which in some cases is a necessity to find the optimal solution.

Recombination and mutation

After the selection has been made, the selected parents will be used to generate 'offspring'. Depending on the algorithm, this generating of offspring can mean multiple things: one method is that two parents are be combined using 'crossover' to create a new individual. Crossover will not be used in the GA in this thesis. Another method for creating new individuals, is simply copying the selected parents (multiple times), leaving them intact, to be the offspring. After the offspring is created, their genotypes will be slightly changed or 'mutated'. The mutation consists of changing one or more values in the string which forms te genotype. In this thesis, where binary strings are used as genotypes, all bits in the string have a certain probability $p_{mutation}$ to get 'flipped'. What the value of $p_{mutation}$ is exactly is defined seperatly for each experiment.

New generation(s)

After mutation has been applied, the resulting population consists of both the parents and the newly generated and mutated offspring. Here one can choose to dispose of the old individuals (the parents) and contintue the algorithm with only the new individuals. The notation for this is (μ, λ), also called a 'comma-strategy'. μ being the number of parents that were selected and used to create the offspring and λ represents the number of generated offspring each generation. The effect of disposing of the parents and continuing with only new indidivuals, is that possibly a old, good solution was mutated into a new, worse solution and that the good solution was disposed and lost forever. Keeping the old individuals in the population, notated by $(\mu + \lambda)$ and known as a 'plus-strategy', prevents this from happening. Both the 'plus-' and 'comma-strategy' have their pros and cons. Whereas the 'comma-strategy' will not easily focus on one solution, but allows the possibility of the population to decrease in fitness. The 'plus-strategy' on the other hand will not 'throw away' good solutions, while having the danger of focusing too much on a certain solution which turns out to be far from optimal and ending up in a 'local-optima'.

Evolutionary Loop

When the new generation is completed, the genetic algorithm continues by calculating the fitness for this new generation. After calculating the fitness the exit criteria are checked and if they are not yet met, the selection procedure will select the individuals which will be used to form the new offspring etcetera. This process is nicely displayed in figure 1.5.

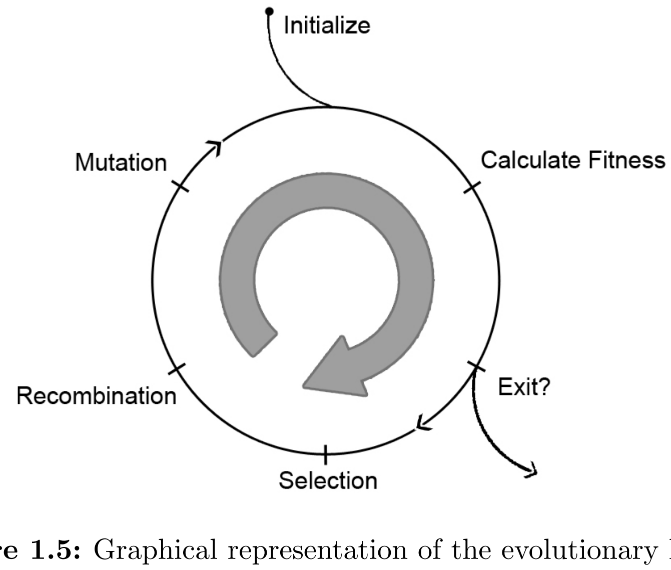


Figure 1.5: Graphical representation of the evolutionary loop in Genetic Algorithms.

1.2.2 Selection methods

Many different methods for selecting individuals based on their fitness ratings to create the new generation, have been constructed. The different selection methods all have their own characteristics. Where some aim to get a 'fairly good' solution as fast as possible using a high selection pressure, others are aimed at finding the optimal solution and the time it takes matters less. This subsection only describes the two selection methods that were used in this thesis, namely the Truncation Selection method and Tournament Selection. For a detailed explanation on various other selection methods, see [1].

Truncation selection

Truncation selection is a straightforward, easy to implement selection method. It just selects the best μ individuals, having the highest fitness ratings of the population. This selection method has the characteristic that it specializes on the best individuals in the population very fast, meaning that in a few generations all individuals are 'descendants' of the same small group and eventually same individual, greatly reducing the diversity the population. This is also the weakness of this method; in some cases the optimal solution (or 'global optimum' of the search space) to a given problem will not be found as the algorithm gets stuck in a 'good' solution (also called a 'local optimum' of the search space). This method is a typical example of a selection method with a high selection pressure.

Tournament selection

The other used selection method in this thesis is tournament selection. This method selects a certain number of individuals to take place in a 'tournament' with only one winner. The winner of this q individuals, q being the tournament size, is the individual with the highest fitness rating. This process repeats itself untill all μ parents have been selected from the population. The number of individuals selected has great influence on this method's behaviour. As it is obvious that using $q = 1$ makes this selection method completely random while using $q = n$ makes it always selecting the best individual. The higher q is set to, the higher the selective pressure becomes.

1.3 The Bitmap Problem

The Bitmap Problem is a challenging task in the field of CA, where a rule is to be found to iterate an initial configuration into a desired configuration within a limited amount of steps. The challenge lies in the fact that the CA has to generate the given desired configuration, or 'bitmap', based on local rules only. The Bitmap Problem has been defined as follows [3, 4]

Given an initial configuration and a specific desired configuration: find a rule that iterates from the initial configuration to the desired configuration in less then I iterations.

It is not required that the number of iterations is fixed, it can be any number between 1 and I. Also, the CA does not have to stay in the desired configuration, it only has to reach it within the limited amount of steps. Altough the Bitmap problem could exist in all CA dimensionalities, only two-dimensional CA have been used so far.

1.3.1 Previous results

The only experiments that has been done on the Bitmap Problem, prior to this paper, were conducted by R. Breukelaar [3, 4] and are quite limited, being more of explorative nature. A fairly simple Genetic Algorithm was used and only small CA sizes were tested. The used CA have a binary state space, the von Neumann neighborhood, unconnected borders and have a height and width of 5 cells. Figure 1.6 shows the different used desired configurations. All desired configurations for the Bitmap Problems used the same initial configuration of a single black cell in the middle of the CA, called a 'single seed' state. The maximum number of iterations for the desired state to be reached was set to $I = 10$.

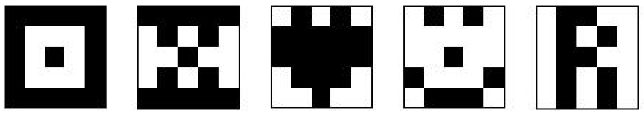


Figure 1.6: Picture of all the bitmaps used in the original experiments. From left to right named "Square", "Hourglass", "Heart", "Smiley" and "Letter".

The Genetic Algorithm used in the experiments to find successful rules for the Bitmap Problems had a population of 100 individuals using the 'plus strategy' $(\mu + \lambda)$ with $\mu = 10$ and $\lambda = 90$. Truncation selection was used to select the top 10% rules. No crossover was used to produce offspring, only Probabilistic Bit Flip Mutation was performed with a probability to flip every bit in the genotype with $p_m = 1/(rule\,length) = 1/32 = 0.03125$. The reason that no crossover was used is due to results of experiments in his dissertation, showing that using crossover together with the von Neumann neighborhood does not combine well. The maximum amount of generations was set to 5000. The definition of the used fitness function was unfortunatly not given. The results obtained from running this algorithm 100 times for every Bitmap Problem are shown in Table 1.2 [3, 4].

Bitmap	Successful rules (out of 100)
Square	80
Hourglass	77
Heart	35
Smiley	7
Letter	9

Table 1.2: Table showing the results obtained by R. Breukelaar, by running a Genetic Algorithm on different Bitmap Problems 100 times. Some Bitmap Problems seem to be harder to solve than others.

Chapter 2

The bitmap experiment

2.1 Copying the experiment

To be able to copy the previous experiments, a fitness function had to be defined first. The most obvious function would be:

"The more cells of the CA are correct in a certain timestep, as in the same as in the desired configuration, the better. If all the cells are in the same state as in the desired configuration, the solution was found."

A more formal definition is:

$$f(c^t) = \sum_{i=1}^{w}(\sum_{j=1}^{h}\psi(a_{i,j}^t, b_{i,j}^t)) \text{ and } \psi(a,b) = \{a = b \rightarrow \psi = 1, a \neq b \rightarrow \psi = 0\}$$

where $f(c^t)$ is the fitness score for a single configuration c at time t.
The fitness score of an individual of the population is given by:

$$\max(f(c^0), f(c^1), ..., f(c^I))$$

where I is the maximum number of iterations.

So the fitness function will have an individual from the population, which represents a CA rule, as input. This rule will then be used to iterate a CA with the given initial configuration and the the amount of 'overlap' between the current and desired configuration will be recorded at every iteration. After I iterations, the maximum score, as in the highest achieved overlap, will be returned as the individual's fitness rating. Using this fitness function, together with CA and GA settings as described in section 1.3.1, the achieved results are those shown in table 2.1.

Bitmap	Successful rules (out of 100)
Square	87
Hourglass	82
Heart	39
Smiley	13
Letter	9

Table 2.1: Table showing the results of trying to repeat the experiments conducted by R. Breukelaar. The results differ a little from the original results though. The difference in difficulty is still the same.

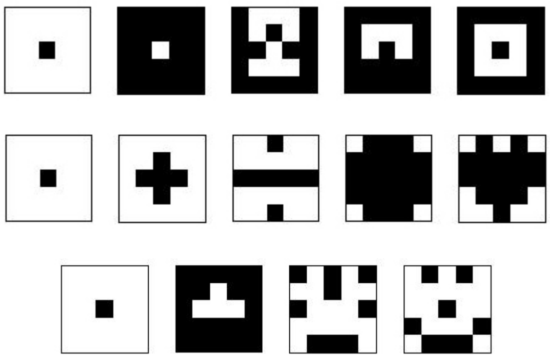


Figure 2.1: Some successful iteration paths of solved Bitmap Problems

The results from the copied experiment show a small increase in efficiency for all bitmaps. The reason for this could be that a different fitness function was used. However, it could also be coincidence as the results differ a little each run.

2.2 Expanding the experiment

The explorative experiments by Breukelaar will be used as a basis. They can be adjusted and extended in multiple ways and this section will describe what will be experimented on and why, with the ultimate goal to give a possible use for the Bitmap Problem in a real world application.

Starting with a statement Breukelaar wrote in his thesis, saying that 'to make the problem even harder', the CA would have unconnected borders instead of connected borders. A good first experiment would be to test this statement and see the actual difference between the two different CA settings. The difference will be measured in terms of the difficulty the GA with the same settings has on a given Bitmap Problem with the same initial and desired configurations, but with the different border settings. This experiment, together with the obtained results, is described in detail in section 3.2.

Another result that stands out from the initial experiments, is that there is a huge differences between the difficulty the GA has solving the different Bitmap Problems. This raises the obvious question of why these notable differences appear. One explanation that was already given by Breukelaar was that the 'easier' desired states had one thing in common: they all had symmetric characteristics to some extend. Where the easiest desired configuration, the 'square', is symmetric in both horizontal and vertical axes. The runner-up, the 'heart' configuration, is symmetric over the vertical axis. The 'smiley' configuration however, which is also symmetric over the vertical axis, has a much worse performance than the 'heart' configuration: what could explain this? Would it be possible to somehow predict in advance if the the GA will have a hard time finding rules for the given Bitmap Problem. Section 3.3 describes some experiments that try to discover if, given a initial and desired configuration, some prediction can be made on this.

Another property of the CA which greatly influences the dynamics of the CA, is the updating method which is used to iterate the system over time. It would be interesting to see how the different updating methods differ from each other in terms of efficiency. Section 3.4 will be devoted to the differences in performance between several asynchronous updating methods compared to the classical synchronous approach.

Consequently, experiments with bigger CA grid sizes have been done, to see how much influence the grid size has on the difficulty of a given problem. It is possible that as the amount of different possible configurations a grid can have increases exponentially, the genetic algorithm will also have much more difficulty in trying to find successful rules. Grid sizes will be increased from 5 by 5 cells to 7 by 7 and 10 by 10. Detailed experimental descriptions and their results are given in section 3.6.

The final component to change, compared to the initial experiments, is the state set of the cells. So far the experiments done used a CA with a binary state space. As seen in section 3.7, expanding the state set has the effect of dramatically increasing the rule space of a CA. This could cause major problems for the relatively simple genetic algorithms used. However, when rules can be found for Bitmap Problems with general state spaces, this would certainly emphasize the power of genetic algorithms and could also increase the possible applications for the Bitmap Problem.

Chapter 3

Experimental Results

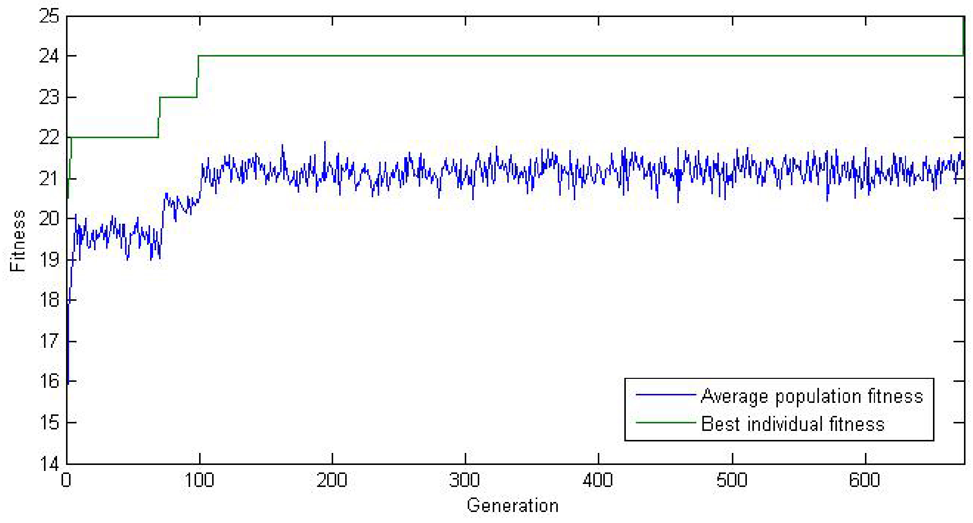
3.1 Genetic Algorithm settings

This chapter contains detailed descriptions, together with the results and conclusions of the different experiments described in the previous chapter (section 2.2). The Genetic Algorithm used in this section differs to some extend from what was used for copying the original experiments, described in section 2.1. This experiment will use a Genetic Algorithm that has a much smaller runtime, without losing too much of its ability to find solutions to the Bitmap Problem. Also, the Bitmap Problem settings are loosened a bit by increasing the setting of the maximum number of iterations I from 10 to 20. The reason for this is that the experiments took a fairly large amount of time while the settings did not seem well balanced, having a very high maximum generations setting opposed to a small population and high fitness pressure. Figure 3.1 shows what the adjustments in parameters produce in terms of Genetic Algorithm fitness convergence.

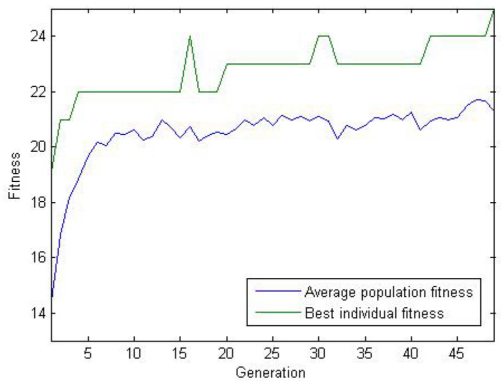
Bitmap	Successful rules Original settings	Successful rules Adjusted settings
Square	87	93
Cross	81	84
Heart	39	39
Smiley	13	8

Table 3.1: Table showing the difference between the different settings of a Genetic Algorithm.

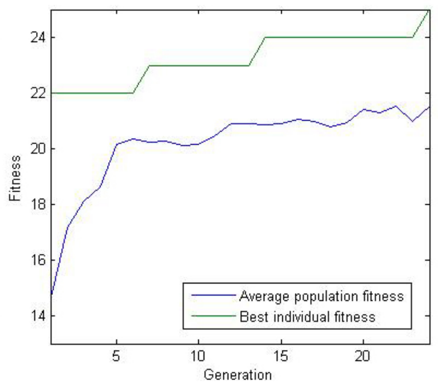
To summarize the Genetic Algorithm settings, population size $(\lambda+\mu) = 100$, mutation rate $p_{mutation} = 0,05$, selection method is Tournament selection, Tournament size $q = 10$, Parents size $\mu = 20$ and Offspring size $\lambda = 80$.



(a) Original experiment Genetic Algorithm settings



(b) Adjusted GA settings



(c) Adjusted GA and CA settings

Figure 3.1: Picture showing convergence graphs of different GA parameters of successful runs on solving the Bitmap Problem for the "Heart" desired configuration. (a) shows a typical run using the 'original' settings proposed by R. Breukelaar. (b) differs in that the mutation rate is increased by a small amount, also the selection method has changed from truncation selection to tournament selection. (c) Same GA settings as in b, but now the Bitmap Problem settings were changed by increasing the maximum iterations from 10 to 20.

3.2 Connected vs. Unconnected borders

The first experiment proposed in section 2.2, was one to test a statement made by R. Breukelaar, saying that unconnected borders would make the Bitmap Problem harder to solve. To test this, a GA with the settings described in section 3.1 was run on several Bitmap Problems to see the actual difference between the two CA settings. The used bitmaps are a bit different than in the original experiments and are shown in Figure 3.2 The results of running the GA 100 times for each bitmap, are shown in Table 3.2 for both connected and unconnected borders.

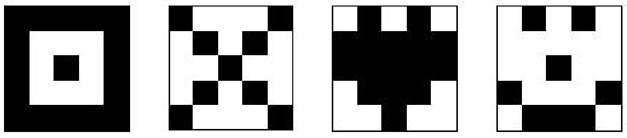


Figure 3.2: Picture of all the bitmaps used in the experiments in this section. From left to right named "Square", "Cross", "Heart", "Smiley".

Bitmap	Successful rules Unconnected borders	Successful rules Connected borders
Square	93	92
Cross	84	72
Heart	39	20
Smiley	8	9

Table 3.2: Table showing the results of running the same Genetic Algorithm on the same Bitmap Problem for a 100 times with both connected and unconnected borders.

The conclusions that can be made when looking at the results of both connected and unconnected borders, is that using unconnected borders is clearly more successful. This is in contrast with the statement from Breukelaar saying that unconnected borders would make the problem harder, which is an understandable assumption when looking from a 'communication' perspective, cells can reach each other faster and exchange information easier with less distance. The assumption however does not seem correct as the opposite is true. This difference could be explained by the fact that with the grid size of 5 by 5 and unconnected borders, 20 'extra' cells are added to the CA, which will never change value as explained in section 1.1.1. This means, at least while using these fairly small grid sizes for the Bitmap Problem, almost half of the cells in the grid have a fixed value (20 out of 45, 5 at every border). The fact that almost half of the grid is fixed, could result in a decrease of complexity, so that the Bitmap Problem can be solved more easily. This assumption would mean that when the size of the grid increases and therefore the ratio of fixed cells decreases, this property becomes less influential and eventually the CA using connected borders will be easier to solve. Anyway, it is clear that using either connected or unconnected borders has a major influence on the Bitmap Problem. For example, figure 3.3 shows two successfull iterations which could only occur within in one of the two settings.

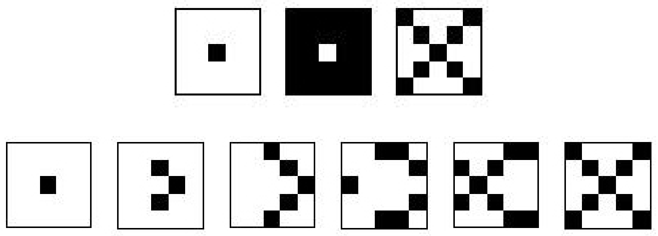


Figure 3.3: The image shows two successful iterations to the "cross" configuration. The top one will only work using unconnected borders. The bottom iteration path can only occur when the borders are connected.

3.3 Bitmap Fingerprints

Another question that was raised, concerns the differences between the amount of successful runs of the Genetic Algorithm on the several desired configurations. As already stated in section 2.2, the symmetry of the desired configurations only partly explained these differences. This section will describe a method which tries to create a 'fingerprint' of a given Bitmap Problem, using both the initial and desired configurations to determine the difficulty of the given Bitmap Problem.

The fitness function used in the earlier experiments, which only used the 'overlap' between a given (initial) configuration and a desired configuration to compute its fitness rating (as defined in section 2.1), did not really help in predicting the outcome of a given Bitmap Problem. For example see figure 3.4 which summarizes this in a single picture: there is no usable correlation between the fitness of the initial configuration c^0 (calculated by the function given in section 2.1) and the overall difficulty of that particular Bitmap Problem (defined as the number of solutions found by the GA in section 2.1. As the 'square' and 'heart' bitmaps have an almost equal overlap, but their difficulties are way off. Same goes for the 'cross' and 'smiley' bitmaps.

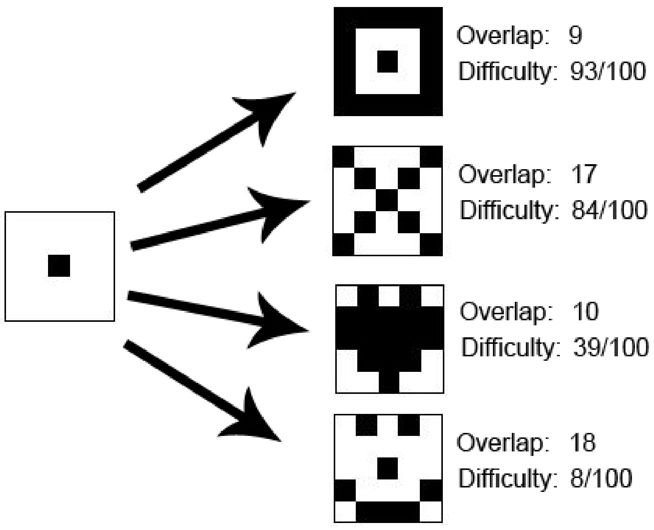


Figure 3.4: Picture showing the overlap the different bitmaps have with the initial configuration, the 'single seed' configuration. Difficulty is based on the experiment results from table 3.1. The higher the difficulty number, the easier the Bitmap Problem was to solve.

So instead of using cell overlap, the fingerprint method will try to describe the patterns that are present within the configuration as a numerical string. This string will contain information on what patterns are present in the configuration. It can then be compared to another configuration pattern string to search for similarities.

The steps required for creating the fingerprint are the following: First define a one dimensional array of zeroes with length $|S|^{|N|}$. $|S|$ being the number of elements in the state set S and $|N|$ being the size of the used neighborhood. As described in section 1.1.3, this number represents the amount of possible neighborhoods that can exist. Then start retracing all the cells of the two dimensional CA. For every cell $a_{i,j}$, look up its neighborhood $N_{i,j}$ and translate this into an integer value (as shown in Figures 3.5 and 3.6). Subsequently increase the value of the number at that position in the array by one.

When all cells have been retraced, the result is an array of length $|S|^{|N|}$ containing numbers with values between 0 and n. n being the number of cells in the CA. The resulting array of these steps for the 'Square' configuration is shown in table 3.3.

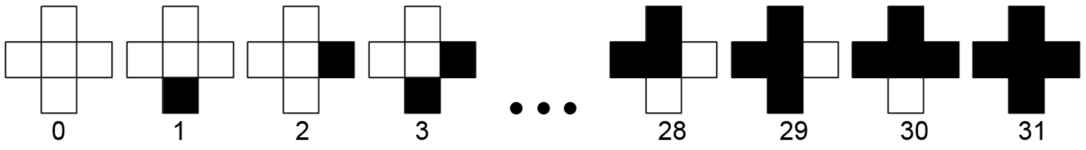


Figure 3.5: Picture that shows how the numbering of the von Neumann neighborhood works.

	0	0	0	1	1	0	0	0	0	1	2	0	0	0	0	3
Neighborhood	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

	0	2	1	0	0	0	0	3	1	0	0	0	0	3	3	4
Neighborhood	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Table 3.3: The fingerprint of the "Square" configuration

The array, or fingerprint, now contains some information on what kind of patterns are present in the given configuration. When comparing this fingerprint to another configuration's fingerprint, a fitness score can be derived. This is done by declaring a variable F with the value of the amount of cells in the CA grid and multiply this by two ($width * height * 2$). Then start at the value at position a_1 of the array of configuration A and compare it to the value of position b_1 of the

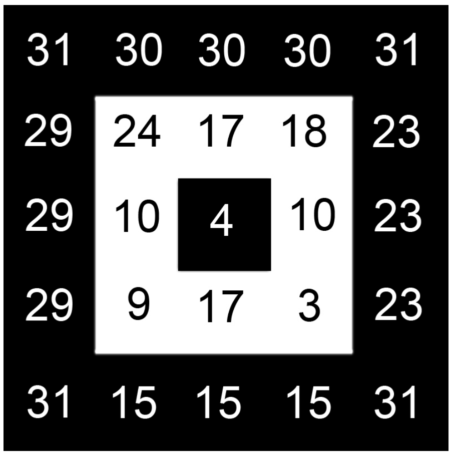


Figure 3.6: Picture of 'Square' bitmap configuration with each cell numbered according to it's neighborhood, using connnected borders.

array of configuration B: if they are not equal, deduct the value of variable F by the difference of the values of a_1 and b_1.. Repeat this for all elements a_i of the array until all i array positions are checked. The resulting value in variable F is the fitness of configuration A compared to configuration B. Table 3.4 shows the 'fingerprint fitness' of the bitmaps used in the experiments in this chapter.

Bitmap	Successful rules Connected borders	Overlap fitness compared to 'Single Seed'	Fingerprint fitness compared to 'Single Seed'
Square	92	9	2
Cross	72	17	10
Heart	20	10	4
Smiley	9	18	14

Table 3.4: Looking at the different fitnesses of the initial states compared to the desired states, the fingerprint method does not seem to clearly distinguish between the difficulties of the different Bitmap Problems.

From the results in table 3.4 can be concluded that the 'fingerprint method' is not sufficient to predict how difficult a given Bitmap Problem will be.The 'Square' bitmap has the lowest fingerprint fitness, but is actually the easiest Bitmap Problem to solve. Also the 'Cross' and 'Smiley' bitmaps are quite close together in terms of fingerprint fitness, but their level of difficulty is far apart. Although the fingerprint is not able to tell us wether or not a certain Bitmap Problem is solvable or not, it is still usable as an addition to the fitness function of the GA.

The given definition of the fingerprint fitness leaves open the possibility for different configurations to have the exact same fingerprint values. For example see figure 3.7, where a simple example shows that multiple configurations can have the same fingerprint. In the case of CA with connected borders and grid size of width n and height m, all configuration fingerprints already have at least $m * n$ matching fingerprints which do not match the exact configuration. This is easily explained, as the bitmap can be 'shifted' to the left or right and up or down, while maintaining the exact same patterns. However, it is not said to stop there as possibly more configurations have the same fingerprint.

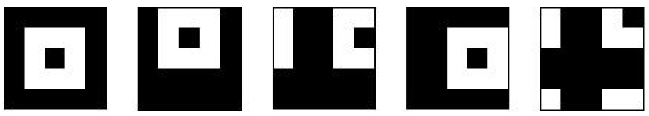


Figure 3.7: Different configurations can have the same fingerprint. This picture shows the 'Square' configuration and four other configurations which, when the CA uses connected borders, share the same fingerprint.

When completely replacing the overlap method for calculating the fitness in the Genetic Algorithm by the fingerprint method, it is forseeable that individuals will reach a perfect fitness score, without representing a solution for the given Bitmap Problem. Results of running the Genetic Algorithm with the settings described at the start of this chapter, but now using the fingerprint fitness as fitness function and using CA with connected borders, are shown in table 3.5. The experimental results could indicate that the "Heart" and "Smiley" configuration share their fingerprint with a higher number of other configurations than the "Square" and "Cross" configurations, as a lot of rules were found that iterated to matching fingerprints, while the actual bitmap was not reached.

Table 3.6 shows results for the same experiment, only this time using a CA with unconnected borders. As was described earlier, with connected borders, the patterns had the possibilty to 'shift' over the grid. Unconnected borders prevents this for most configurations, at least to some extend. So it is imaginable that the amount of successful runs decreases compared to using connected borders, but the ratio in which rules actually reach the desired configuration instead of only the desired fingerprint would increase. The reason for this is that there are less possible fingerprints for a given configuration. The results, shown in table 3.6, are a bit unexpected however. As for the 'Square', 'Cross' and 'Heart' bitmaps, all rules

Bitmap Connected borders	Successful rules Fingerprint fitness	Same rules translated to Overlap fitness
Square	76	40
Cross	78	32
Heart	63	0
Smiley	18	1

Table 3.5: Table containing the results of running the Genetic Algorithm 100 times for every bitmap with connected borders, using only the fingerprint fitness method. Afterwards, the successfully generated rules were tested with Overlap fitness.

that generated the right fingerprint also lead to the right desired configuration! Only the 'Smiley' bitmap seemed to have some rules that did lead to the right fingerprint, but not to the right configuration.

Bitmap Unconnected borders	Successful rules Fingerprint fitness	Same rules translated to Overlap fitness
Square	81	81
Cross	81	81
Heart	23	23
Smiley	9	4

Table 3.6: Table containing the results of running the Genetic Algorithm a 100 times for every bitmap using unconnected borders, using only the fingerprint fitness method. Afterwards, the successfully generated rules were tested with Overlap fitness.

Although previous results show that the fingerprint method on its own is not sufficient for use as fitness function, especially when using connected borders, it can still contribute in the fitness function to give additional feedback about the potential of a given individual. As it does seem to recognize some usable features of a rule, where the right pieces of the puzzle were generated, only the translation is incorrect. So a hybrid fitness function was constructed using both the 'overlap' and 'fingerprint' methods. While creating such a hybrid fitness function, choices had to be made in terms of which method has the most influence on the total result.

$$f = c \cdot overlap + (1 - c) \cdot fingerprint$$

Where f is the total fitness of an individual and c specifies the influence of the different fitness functions *overlap* and *fingerprint*, described in section 2.1 and 3.3 respectively. Using the setting of $c = 1$ results in using *overlap* fitness only and $c = 0$ would result in using *fingerprint* fitness only.

Different settings of c were tested and the results are shown in table 3.7 for connected and table 3.8 for unconnected borders.

Bitmap Connected borders	$c = 0.20$	$c = 0.33$	$c = 0.50$	$c = 0.67$
Square	83	86	93	93
Cross	69	80	81	79
Heart	12	17	22	24
Smiley	6	4	9	8

Table 3.7: Results of using a hybrid fitness function with different influence parameters. Higher influence of the 'overlap' part seems te have a positive effect on the results. In contrary to a higher influence of the 'fingerprint' part, which has a negative effect on the results. An equal influence of overlap and fingerprint seems to work best, even better than the original experiment results. The CA in this experiment uses connected borders.

Bitmap Unconnected borders	$c = 0.20$	$c = 0.33$	$c = 0.50$	$c = 0.67$
Square	89	92	90	91
Cross	91	94	96	95
Heart	40	35	44	47
Smiley	4	9	11	6

Table 3.8: Results of using a hybrid fitness function with different influence parameters. The CA in this experiment uses unconnected borders.

Looking at the results in tables 3.7 and 3.8 and comparing the results to those in table 3.2, it can be concluded that using a hybrid fitness function has a positive effect on the effiency of finding rules for the Bitmap Problem with the Genetic Algorithm using both connected and unconnected borders. As was expected the bitmaps with unconnected borders had the most advantage of the added fingerprint method.

3.4 Synchronous versus Asynchronous updating

Using a different updating method for the same rule and initial configuration, leads to entirely different iteration paths. Some configurations that might not even be reachable when using synchronous updating, could possibly be reached when using asynchronous updating. Of course this is no garantuee, but it is interesting nonetheless to see how the Bitmap Problem reacts on various asynchronous updating methods. It is not unthinkable that the differences in difficulty between bitmaps with symmetric and asymmetric properties will diminish or even disappear completely.

Only the updating methods **'line-by-line sweep', 'fixed random sweep' and 'random new sweep'** were used in the experiments in this section, to ensure all cells will get their evaluation at regular basis. Also these methods are more convenient to repeat when a successful rule is found. To stay in line with earlier experiments with synchronous updating, the maximum number of iterations I is increased from 20 to 500 $(20 * 25)$, because only one cell will be evaluated every iteration in asynchronous updating. This way every cell will have the same number of evaluations as in the experiments using synchronous updating. The experiments were done using the hybrid 'fingerprint-overlap' fitness function described in section 3.3.

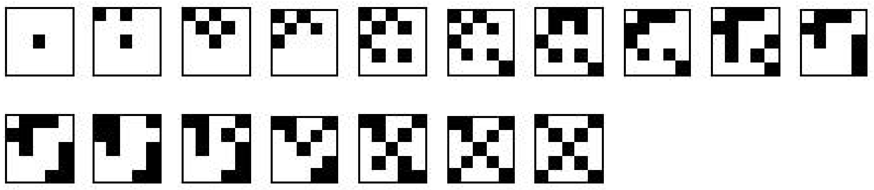


Figure 3.8: Successful iteration path to the 'Cross' configuration using the Line by Line sweep updating method. Each step in the picture is actually 5 iterations of the rule

Looking at the results shown in tables 3.9 and 3.10, it can be concluded that for most of the bitmaps the asynchronous updating methods do not perform better. The 'smiley' configuration is an exception though and seems to benefit from the asynchronous updating, especially using unconnected borders. There are however more observable changes that are worth noting. One if these is that the disparity of performance between the different bitmaps have been leveled somewhat. The performance difference between the square / cross and smiley configuration were really big when using synchronous updating. Using asynchronous updating brings

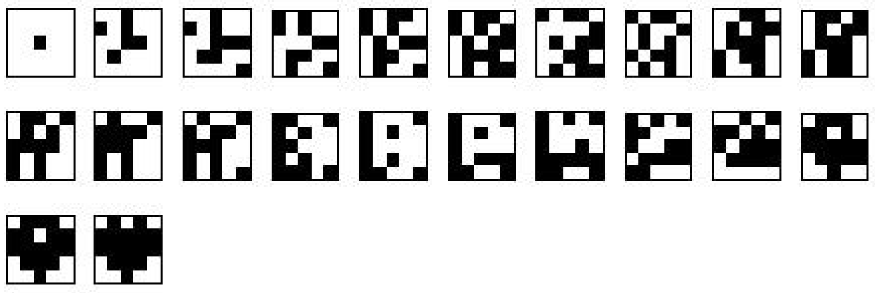


Figure 3.9: Successful iteration path to the 'Heart' configuration using the Random Fixed sweep updating method. Each step in the picture is actually 5 iterations of the rule.

Bitmap Connected borders	Line by line	Fixed Random	Random New
Square	19	32	34
Cross	29	11	7
Heart	13	18	15
Smiley	7	16	12

Table 3.9: Results of running the same GA as described in section 3.1, but now using a hybrid 'fingerprint-overlap' fitness function, an equal number of times on the same bitmaps. Only now, the CA uses another update method with another maximum number of iterations. This table shows the performance of the GA on the various bitmaps using connected borders.

Bitmap Unconnected borders	Line by line	Fixed Random	Random New
Square	43	29	45
Cross	32	22	42
Heart	28	38	37
Smiley	12	26	36

Table 3.10: Results of running the same GA as described in section 3.1, but now using a hybrid 'fingerprint-overlap' fitness function, an equal number of times on the same bitmaps. Only now, the CA uses another update method with another maximum number of iterations. This table shows the performance of the GA on the various bitmaps using unconnected borders.

their performance results much closer together. This could mean that the Bitmap Problem performance using asynchronous updating is less influenced by the difficulty of a given desired configuration. Figures 3.8 and 3.9 show successful iteration paths using asynchronous updating methods.

Another thing that is interesting to see, is the differences between the various asynchronous updating methods and their effect on the performance of the different bitmaps. Some bitmaps seem to have great benefit from an update method, where another bitmap suffers a decrease in performance. Looking at the Random New Sweep method, it has a terrible performance with the 'Cross' bitmap, the 'Smiley' bitmap however is quite good. It is hard to say exactly what the reasons are for these big differences. It is clear that using unconnected borders gives much more successful rules than using connected borders, which was also the case in synchronous updating.

3.5 Iteration path fitness

3.5.1 Synchronous iteration path

In an attempt to get more feedback on how a given Bitmap Problem is being solved, research has been conducted on how successful rules actually iterate to the desired state. This might generate information that could be used to improve the search for successful rules. As described in section 2.1, the fitness function iterates the given input rule for a maximum of I iterations. At each timestep the fitness score of the current configuration is being calculated and recorded. After the I^{th} iteration the best score over all iterations is returned as fitness score. Does an iteration path of a succesful rule exhibit predictable behaviour in terms of increasing fitness over time? To get a better understanding on what is going on when a rule is succesfully iterating to a desired configuration, we will look at the configuration of each timestep, calculate the configuration's fitness with respect to the desired state and see if there are any similarities between different successful iteration paths.

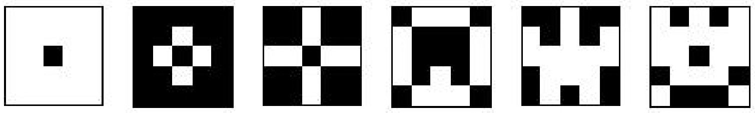


Figure 3.10: The iteration path of a rule solving the Bitmap Problem for the 'Smiley' configuration, using connected borders.

Figure 3.10 shows the iteration path of a rule solving the Bitmap Problem for the 'Smiley' configuration. Figure 3.11 is the associated fitness graph, where every timestep corresponds to an iteration step of the iteration path. There are possibly many rules, even while using a binary state space and the von Neumann neighborhood, which could successfully solve a particular Bitmap Problem. Looking at every possible rule and showing their iteration fitness graphs here would take up way too much time and space, but when looking at some samples, there are some characteristics that seem to be recurrent.

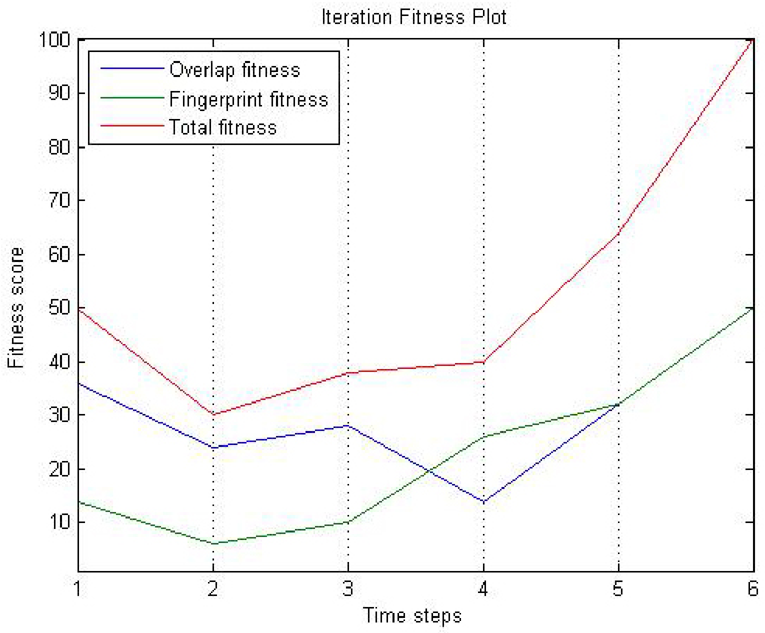


Figure 3.11: The iteration graph of the iterationpath depicted in figure 3.10. Where time step 1 is the initial 'single seed' configuration and time step 6 the desired 'Smiley' configuration.

The longer a rule needs to reach a desired configuration, the harder it is to predict its behaviour in terms of fitness over time. Because often, the longer iteration paths seem to be 'oscillating' rules, going from a large amount of black cells to large amount of white cells. This also greatly influences the fitness of the configuration at every timestep, which makes it difficult to see if it will reach the desired configuration at any given time. An example of what exactly is meant by 'oscillating' is shown in figures 3.12 and 3.13 where it takes a rule 16 iterations to successfully reach the desired state.

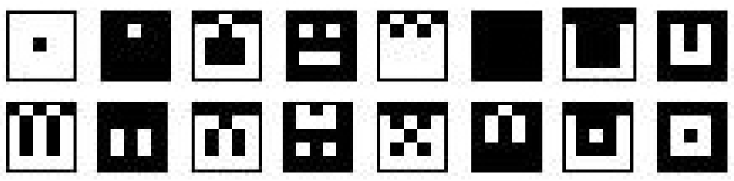


Figure 3.12: An example iteration path of an oscillating rule which successfully reaches the desired state.

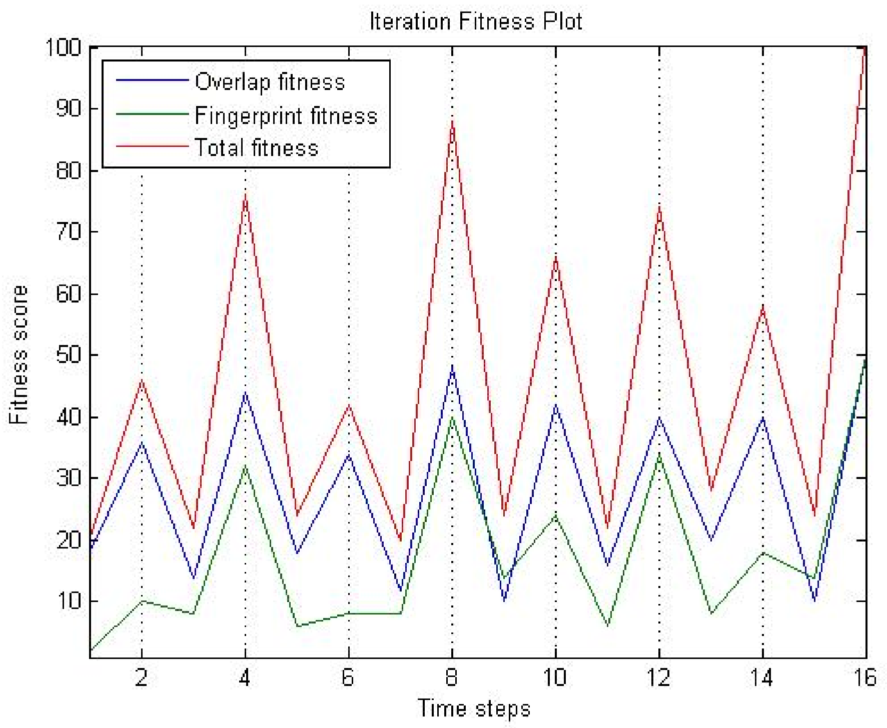


Figure 3.13: The fitness graph corresponding to the iteration path shown in figure 3.12. The spikes in the fitness graph make it hard for the Genetic Algorithm to figure out wether or not it has real potential.

The assumption that the longer a rule takes to reach a desired configuration, the harder it is to predict its behaviour and potential, would mean that decreasing the maximum amount of iterations would generate less unpredictable oscillating rules and therefore the amount of successful rules will not decrease equally proportionate. To check wether this assumption holds, a quick experiment was done using the 'Heart' desired configuration with various settings for the maximum amount of iterations. The remarkable results of this experiment are shown in figure 3.14 with its associated data table 3.11.

Max. Iterations	1	2	3	4	5	6	7	8	9	10	15	20	40
Successful rules	0	0	0	54	58	59	54	52	40	41	46	44	48

Table 3.11: Table containing the results of running the Genetic Algorithm 100 times for the 'Heart' bitmap using unconnected borders with different settings for the maximum allowed iterations.

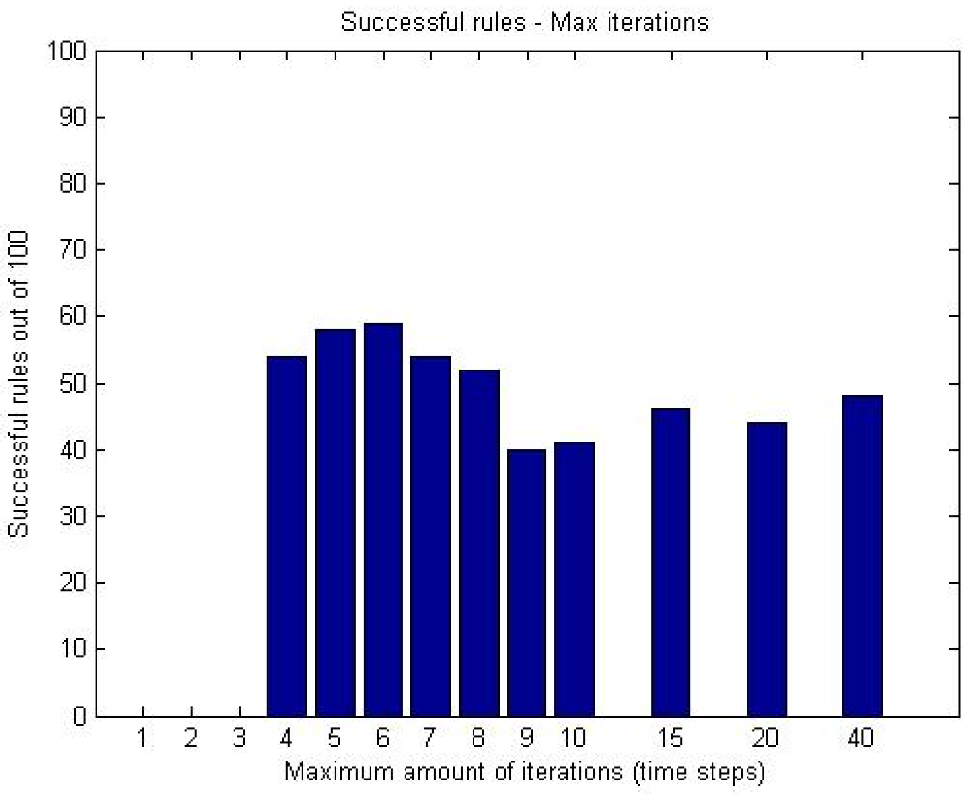


Figure 3.14: Graph showing the results of running the Genetic Algorithm 100 times for the 'Heart' bitmap using unconnected borders with different settings for the maximum allowed iterations

Looking at the results, the assumption seems to hold. Counterintuitively, a smaller amount of maximum iterations produces a higher efficiency of the Genetic Algorithm. Although it is quite clear that the total set of rules which solve a given Bitmap Problem is bigger when the maximum amount of iterations increases, as the set of rules with smaller iteration amounts are a subset of the set of rules with more iterations. A possible explanation for the increased efficiency of the Genetic Algorithm while the maximum amount of iterations is decreased, lies in the strategy of the Genetic Algorithm. The rules which have a large amount of iterations have a good chance of getting close to the desired configuration somewhere along the way. This results in that it will 'survive' the selection phase, while it has no true potential of ever reaching the desired state. The population of the Genetic Algorithm will converge towards this 'misleading' rule while it progresses in generations and will not be able to find a successful rule in the end. On the other hand, allowing only lower amounts of iterations, the rules will be more predictable (see figure 3.15) and the genetic algorithm has less problems with finding a successful rule, while the set of rules it is searching in is actually smaller. To see if not only the Bitmap Problem using the 'Heart' desired configuration had benefit from lowering the amount of iterations, the same experiment was repeated for the other bitmaps. The results are shown in table 3.12 and suggest that all the bitmaps seemed to have some benefit from lowering the maximum number of iterations by a certain amount. Not all bitmaps behave exactly the same though.

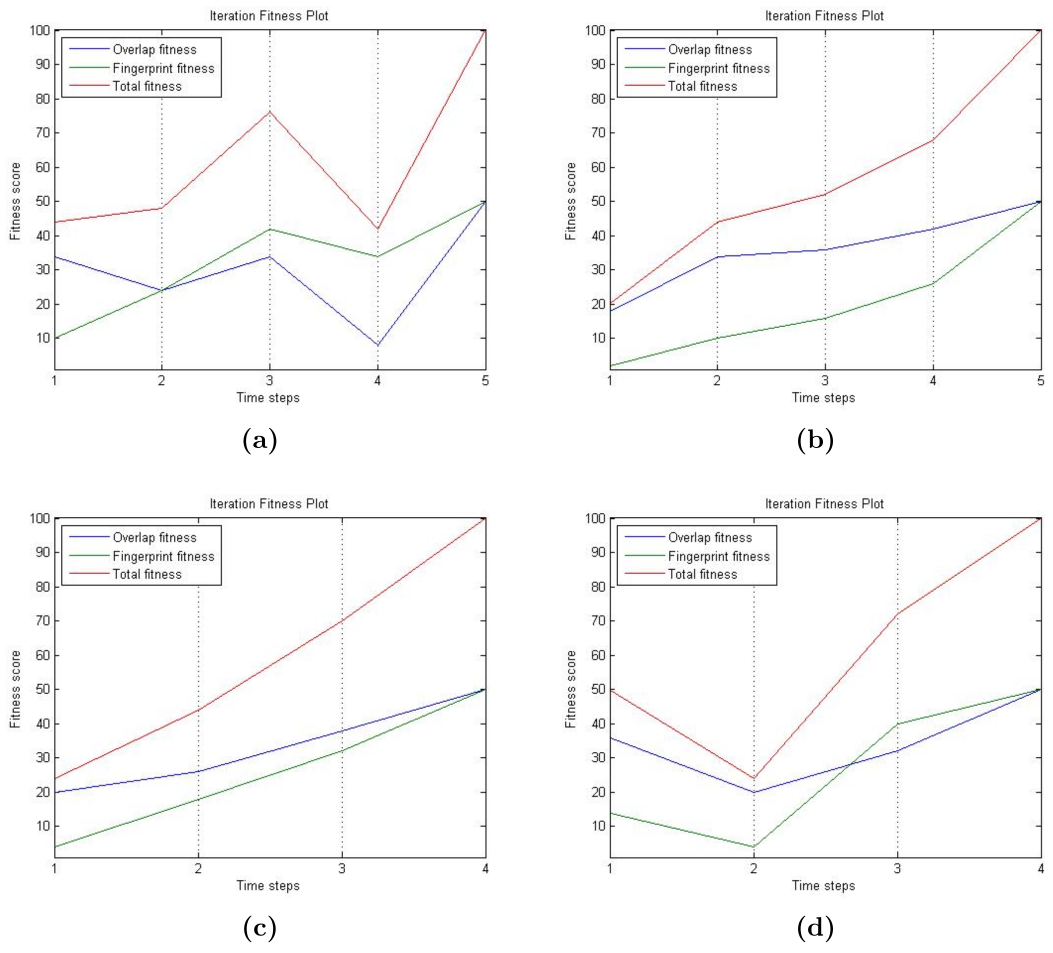


Figure 3.15: Picture showing typical fitness graphs of rules that were found by the Genetic Algorithm, having a maximum amount of iterations of 5. The rules, as opposed to rules with higher iteration amounts, are far easier to predict. (a) "Cross" graph (b) "Square" graph (c) "Heart" graph (d) "Smiley" graph

	$I = 5$	$I = 10$	$I = 20$	$I = 40$
Square	78	93	90	96
Cross	97	80	96	89
Heart	58	41	44	48
Smiley	11	14	11	13

Table 3.12: Results of lowering the amount of maximum allowed iterations to reach the desired state per bitmap. The table shows the successful runs of the Genetic Algorithm out of 100 runs.

3.5.2 Asynchronous iteration path

Using the asynchronous methods for cell updating seemed to produce significantly less successful oscillating rules than using synchronous updating. Although this is also due to when only updating one cell at a time, the change in configuration is not that big, there was still a noticeable difference when looking at the different fitness graphs. The graph in figure 3.16 represents the fitness graph of a successful oscillating rule iterating to the 'heart' desired configuration, using the asynchronous updating method 'Line by Line sweep'.

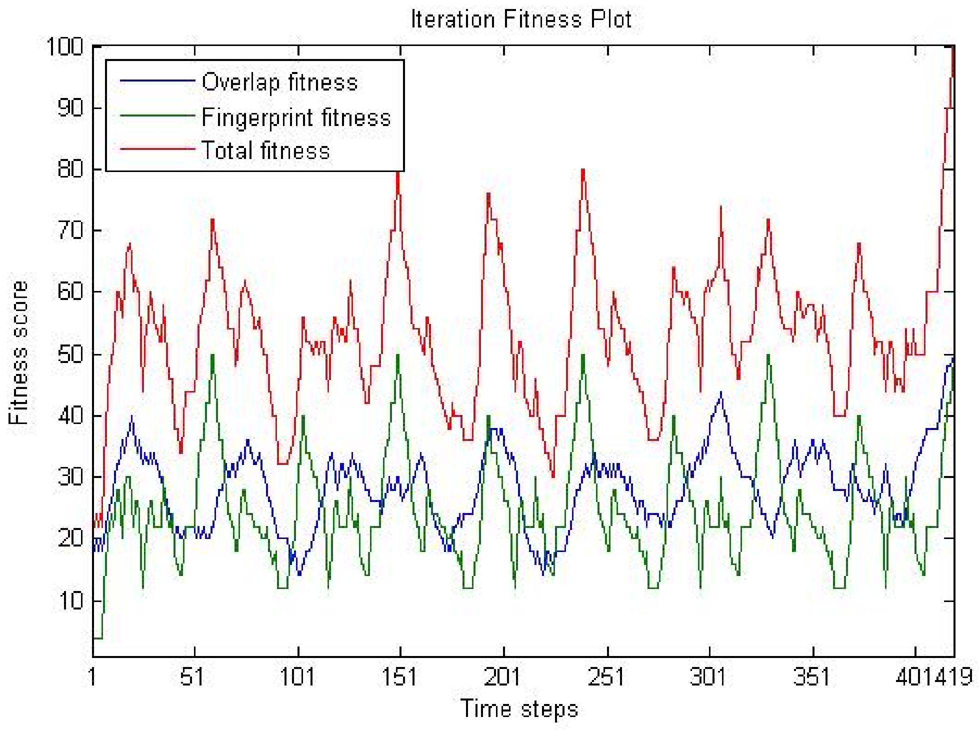


Figure 3.16: Fitness graph of an asynchronous oscillating rule successfully iterating to the desired configuration. The corresponding iteration path is shown in figure 3.17

The reason why less successful oscillating rules were recorded in comparison to the Bitmap Problems using synchronous updating, remains unclear. As the maximum iteration amount used of 500 compares to the setting of 20 maximum iterations in synchronous updating, which seemed to produce significantly more oscillating rules. When taking into account the results from section 3.4, which show us that the overall efficiency of the asynchronous updating methods is somewhat lower than using synchronous updating, together with the assumption *"Less oscillating rules*

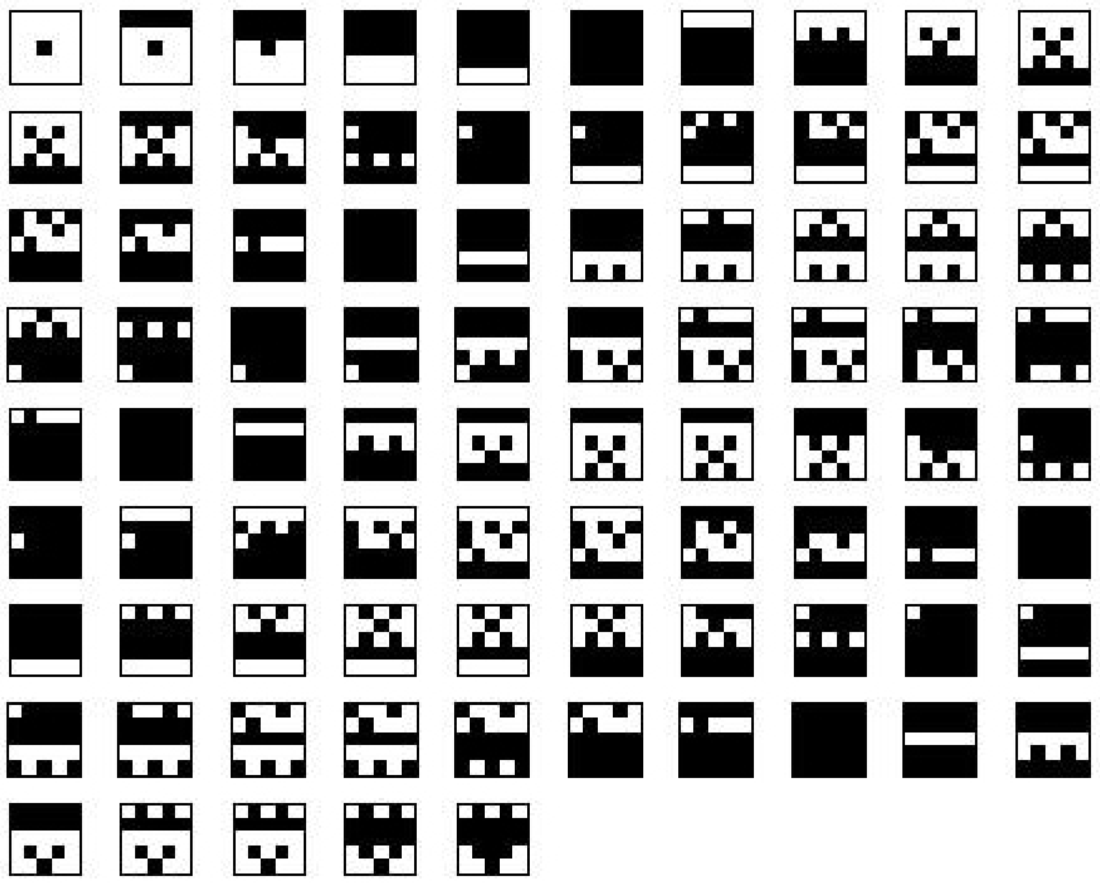


Figure 3.17: Iteration path of an oscillating rule successfully iterating to the desired 'heart' configuration using the asynchronous updating method 'Line by Line Sweep'. The rule takes 419 iterations to get there, so to save space the configurations are shown with an interval of 5 steps.

is better for the efficiency of the Genetic Algorithm" and the observation of less oscillating rules in asynchronous updating, it is not likely that the asynchronous updating method benefits from lowering the maximum of iterations as synchronous updating does.

The results shown in table 3.13 confirm the assumption that asynchronous updating methods would not react the same on lowering the maximum amount of iterations as synchronous updating. In contrary, it can be concluded that increasing the maximum amount of iterations improve the efficiency of asynchronous updating methods on the Bitmap Problem.

"Heart" bitmap	$I = 125$	$I = 250$	$I = 500$	$I = 1000$
Line by Line	21	28	28	40
Random fixed sweep	34	37	38	54
Random new sweep	31	27	37	46

Table 3.13: Results of lowering the amount of maximum allowed iterations to reach the desired state per bitmap. The table shows the successful runs of the Genetic Algorithm out of 100 runs.

3.6 Increasing the grid size

Using relatively small grid sizes of a height and width of 5 cells results in quite successful performance of the GA. This section describes the experiments to see if the same GA is still able to solve a given Bitmap Problem when the grid size increases. The in previous sections used von Neumann neighborhood consists of 5 cells and therefore, using a binary state set, has 32 (2^5) unique neighborhoods. 32 bits are needed to encode rules for these settings. This makes the amount of possible rules (2^{32}) much bigger than the amount of possible CA configurations (2^{25}) when the grid size is 5 by 5. It is interesting to see how well the GA still solves the Bitmap Problem when the amount of unique CA configurations succeed the amount of rules. Increasing the grid size will also expose if the difference between using connected or unconnected borders will still hold.

Different experiment setup

The other experiments in this thesis are using a rather small set of predefined desired configurations (sometimes referred to as bitmaps) while always the same initial configuration (single seed) is used. There are two reasons to abandon this setup of experiments in this section. The first reason being that simply scaling the bitmaps to larger grid sizes may cause an unfair comparison between the smaller and bigger grid sizes, as it is not said that the difficulty will scale equally per bitmap. The second reason is that when the grid size increases, the single seed initial configuration does need perform well anymore. This is something that was already noted by Breukelaar in his conclusions and preliminary experiments confirm this statement. So to ensure that comparing the performance of the GA on the Bitmap Problem with increased grid sizes is done as fair and accuratly as possible, this experiment will be done using randomly generated initial and desired configurations. The GA gets twenty tries to solve a Bitmap Problem with randomly generated (using a normal distribution) initial and desired configuration pair. After these twenty tries a new initial and desired configuration pair is generated. This process is repeated 100 times for the CA grid sizes of 5 by 5, 7 by 7 and 10 by 10. The results of this experiment are shown in table 3.14.

The results show that the fairly simple GA has great difficulty solving the random Bitmap Problems for higher grid sizes. The difference between connected and unconnected borders is quite big for the 5 by 5 sizes. To see if the difference still holds with bigger grid sizes, the same experiment is repeated using the Moore neighborhood instead of the von Neumann neighborhood. It is noteworthy, that by going from the von Neumann to the Moore neighborhood, there is a huge in-

	Connected borders	Unconnected borders
5 by 5	30	61
7 by 7	0	0
10 by 10	0	0

Table 3.14: Results from the experiments with increased grid sizes using the von Neumann neighborhood

crease of rule space. The amount of bits needed to encode the rule increase from 32 to 512 bits. For this reason, the mutation probability in the GA has been adjusted accordingly: from 0.05 ($1/32 \cdot 1.6$) to 0.003125 ($1/512 \cdot 1.6$). To keep it completely fair, the maximum amount of generations of the GA should be increased as well. However, as time is limited, the rest of the GA settings is kept the same and thus maximum amount of generations will be kept at 100.

	Connected borders	Unconnected borders
5 by 5	100	100
7 by 7	71	54
10 by 10	0	0

Table 3.15: Results from the experiments with increased grid sizes using the Moore neighborhood

This time the results (table 3.15) show that the 7 by 7 grid size Bitmap Problems could be solved as well. The difference between connected and unconnected was surprisingly reversed! Altough is hard to say if this only due to the increased grid size, or that the change in neighborhood topology had its influence as well. From these results can be concluded that the Bitmap Problems with increased grid sizes are much harder to solve than the originals of 5 by 5 cells. The GA that was used however is quite limited and future research on using more advanced search algorithms, like Evolutionary Strategies, might acquire better results.

3.7 General state space Bitmap Problem

Expanding the state set of a CA by only one element already drastically increases both the size of the rulespace (table 1.1) and the possible unique configurations of the CA. This makes it very interesting to see if a GA is still able to find rules within this massive search space that will successfully solve Bitmap Problems with more than two states in the state set. With one extra state in the state set, going from the binary state set to $S = \{0, 1, 2\}$, four different bitmaps were created to see if they could be solved. The bitmaps are shown in figure 3.18 below.

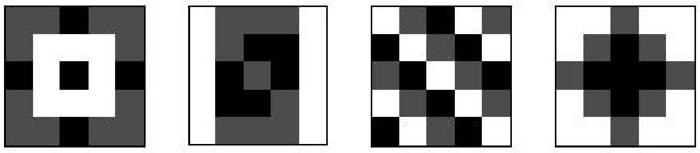


Figure 3.18: The three-state bitmaps used in the experiments in this section, 0 is represented as *white*, 1 as *grey* and 2 as *black*. For reference named; from left to right 'Square^3', 'Letter S', 'Pattern' and 'Star'.

Because of the huge increment of search space, the parameters of the genetic algorithm were modified. Still using the 'plus-strategy' $(\mu + \lambda)$, the population size was raised to be 400 with parentsize $\mu = 80$ and offspring size $\lambda = 320$. Tournament selection was used with a tournament size $q = 40$. The mutation rate changed to $p_m = 0.005$, this because of the increase of the state set, the length of the genotypes expanded to $3^5 = 243$. Also the way mutation is performed had to be adjusted, as 'bit flipping' was no longer possible. When a position in the genotype array is chosen to be mutated, the state at that position is changed randomly into a state from the state set S, other than it already was. The maximum amount of generations is 100. Also the maximum amount of iterations that a rule was iterated was raised, this time using 32 as maximum, instead of 20 in earlier experiments. The algorithm was run a 100 times for every bitmap, the results are shown in table 3.16.

The results are quite impressive considering the enormous search space. Looking at a sample of the found solutions to the 'Square^3' and 'Star' bitmaps, the amount of iterations needed to go from the initial to the desired configuration was always very small. Every solution needed three steps, but these include the initial and desired configuration, so actually only 2 steps were taken; from initial to configuration

Bitmap	Successful rules (out of 100)
Square[3]	44
Letter S	6
Pattern	1
Star	77

Table 3.16: Results from the experiments with an extended state set.

2, and from 2 to the desired configuration. The iteration paths for the bitmaps 'Letter S' and 'Pattern' showed much more variation. Nonetheless, it seems to be that the setting of 32 maximum iterations was way too high. A successful iteration path for each bitmap in this experiment is shown in figure 3.19.

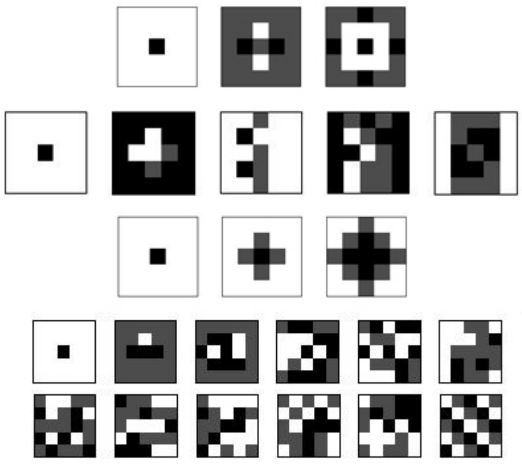


Figure 3.19: Successful iteration paths for each of the bitmaps used in the experiment in this section.

3.8 Enhanced Bitmap Problem

The following experiment that is described could actually be used within the variation of the compression algorithm described in section 4.2 and could be seen as a 'Bitmap Problem to the second degree'. Where an initial configuration does not only lead to one, but two desired configurations. Altering the definition slightly:

Given an initial configuration and two different desired configurations: find a rule that iterates from the initial configuration to both desired configurations, in any order, in less then I iterations.

The difference lies in that not only a rule has to be found that iterates the initial to the desired configuration, but while it iterates to the desired configuration it also has to 'touch' another configuration on the way. Some preliminary experiments were conducted on this enhanced Bitmap Problem. Although it seemed much harder to find successful rules, they were found nonetheless. An example of a successfully solved problem of this definition is shown in figure 3.20.

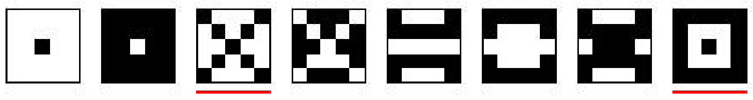


Figure 3.20: Successful iteration path from the single seed configuration to the 'Square' configuration, reaching the 'Cross' configuration on the way. The successful rule was found using a Genetic Algorithm and uses a CA grid with unconnected borders.

The fitness function used to solve this enhanced Bitmap Problem uses the hybrid fitness function described in section 3.3 with an equal influence of the different fitness score methods. Just like when solving the original Bitmap Problem, a rule is iterated for a maximum amount of timesteps. But instead of checking the fitness of the current configuration of the CA at every timestep for only one desired configuration, it is now checked against both desired configurations. When *I* timesteps have been iterated and checked, the sum of the highest fitness scores that were recorded for both seperate desired states form the fitness of the rule as a whole. It is both beautifull and remarkable to see that the Genetic Algorithm is able of actually solving two Bitmap Problems in parallel within only one iteration path.

Figure 3.21 shows another successfully solved enhanced Bitmap Problem. This time the two bitmaps that were proven to be more difficult were used, the 'heart' and 'smiley' bitmaps. Looking at the fitness graphs for the independent configurations, they look as any other successful rule iterating to the desired configuration. So it is clear there is an overlap between rules solving the Bitmap Problem for different configurations, which definitely has its potential.

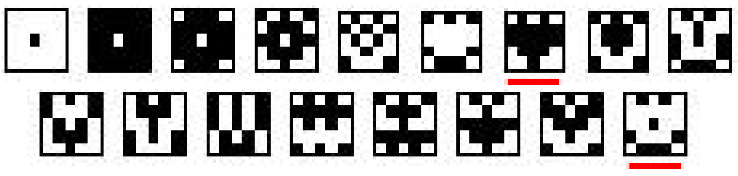


Figure 3.21: Successful iteration path from the single seed configuration to the 'Smiley' configuration, reaching the 'Heart' configuration on the way. The successful rule was found using a Genetic Algorithm and uses a CA grid with unconnected borders.

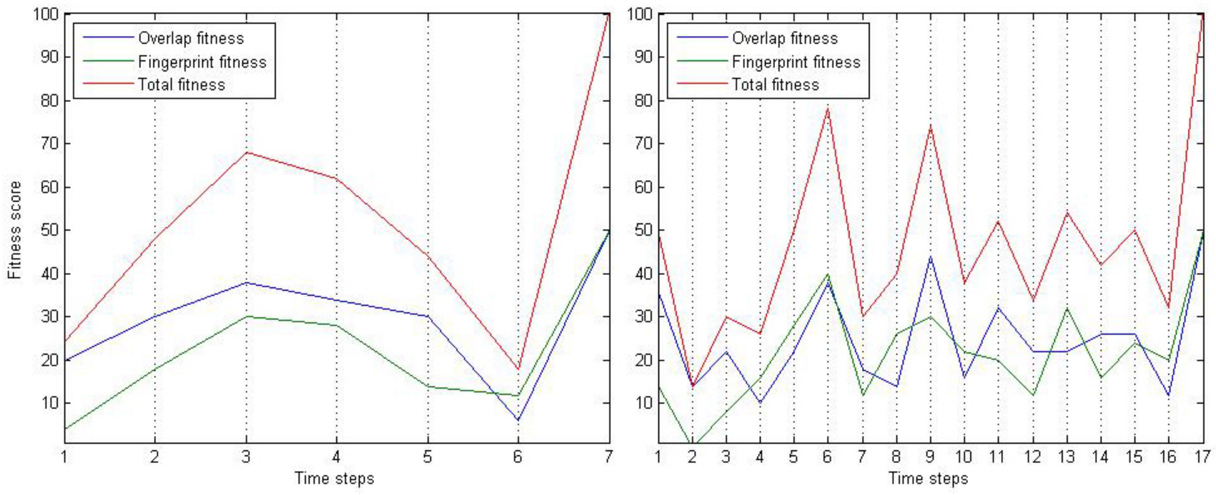


Figure 3.22: The left graph represents the fitness graph of the iteration path in figure 3.21 leading to the 'Heart' configuration. The right graph represents the fitness graph for the 'Smiley' configuration.

To put this in perspective, it is straightforward to see what configurations are in the set of what a rule can generate: just do the iteration and all unique configurations that are reached are in the set. The other way around is much harder already: does a given configuration belong to the set of configurations that a rule can generate? Which is actually the Bitmap Problem. It is even harder to ask: are these N configurations within the set of configurations that can be generated by a rule? The fairly simple Genetic Algorithm solves these problems, what shows how powerful evolutionary algorithms are.

Chapter 4

Summary and Outlook

This thesis further elaborates the work of R. Breukelaar on the 'Bitmap Problem' which he proposed in his master thesis [4]. The various settings of the original Bitmap Problem have been tested extensively to discover what exactly their influence is on the difficulty of the problem. Also a new additions to the Bitmap Problem have been introduced within this thesis, namely the 'fingerprint' method and the Enhanced Bitmap Problem. Furthermore, a real world application was suggested at the end of the thesis.

The setting of the border, being either connected or unconnected, has been proven to be in the favor of unconnected borders, especially for smaller grid sizes which show significant better results on the tested Bitmap Problems. Changing the updating method that is used within the CA shows great differences in performance as well, not only between synchronous and asynchronous updating in general, but also between the various asynchronous updating methods large differences can be observed.

The experiments with increased CA grid sizes did not use the same bitmaps as for most of the other experiments in this thesis. As simply scaling the previously used bitmaps did not seem adequate, the initial and desired configurations for these experiments were randomly generated using normal distribution. The GA then tried to solve each of the randomly generated initial/desired configuration pairs with a maximum of 20 tries. This process repeated 100 times to gather some empirical data. The results are somewhat disappointing and show a substantial decrease in performance every small increment in grid size.

Last of settings that was tested on influence was the size of the state set. Experiments with a state set increased from two to three elements show that despite of the tremendous increase in search space, the GA is still a viable method for finding successful rules.

Bitmap 'fingerprints' were introduced to help explain the differences in difficulty between the several bitmaps. Although the fingerprint method did not help in this particular issue, it proved to be usefull in enhancing the fitness function for the GA. Observing the fitness evolution of a successful iteration path in detail helped to further improve the performance of the GA in a somewhat counterintuitive way.

The results that where found in this research raises the question on what the optimal settings for the GA and CA are to achieve a highest as possible result on any given Bitmap Problem. For example the finding ideal setting for maximum iterations a Bitmap Problem can use to reach the desired state, which is probably bound to the size of the grid. The fingerprint method proposed a new way of approaching and describing the Bitmap Problem. While constructing this method, various other features were tried to extract from it, other than a fitness score alone. Altough so far without succes, there is a chance that this approach can offer more that was described in this thesis. Also on the side of the searching algorithm there are multiple improvements to be made, more advanced search methods could lead to much better results. Of course it would be very interesting to see, as the knowledge on the Bitmap Problem matures, if a compression algorithm based on the Bitmap Problem would actually work in real-world test cases.

4.1 Data Compression Algorithm

In the explorative research of Breukelaar, it was already proposed that the Bitmap Problem could be used for lossy image compression. This section will describe a conceptual compression algorithm based on the Bitmap Problem for binary data in general. To explain how the algorithm works, we will go through the process step-wise using examples to clarify the required steps.

The data that is to be compressed, lets say $data_{original}$, is a binary string of a given length l: $data_{original} = \{0,1\}^l$. The first step is to cut this string in pieces of length x and use them to form a two dimensional grid of x columns and y rows, as shown in figure 4.1.

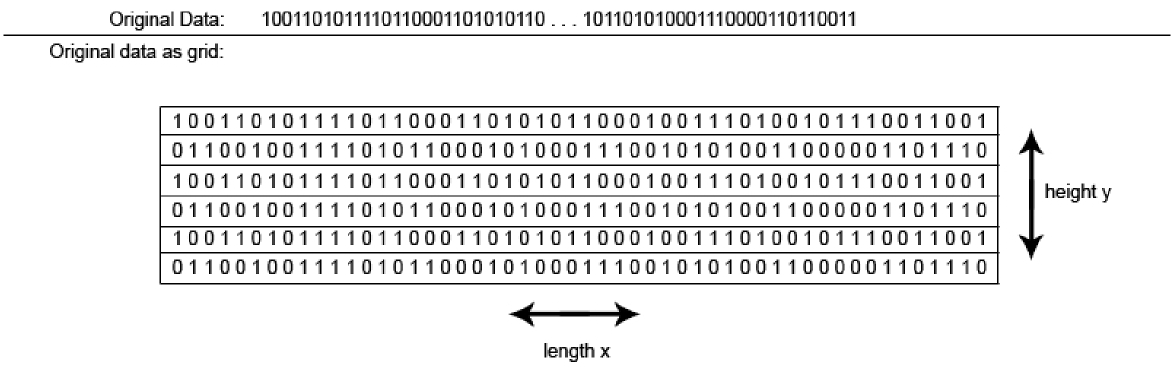
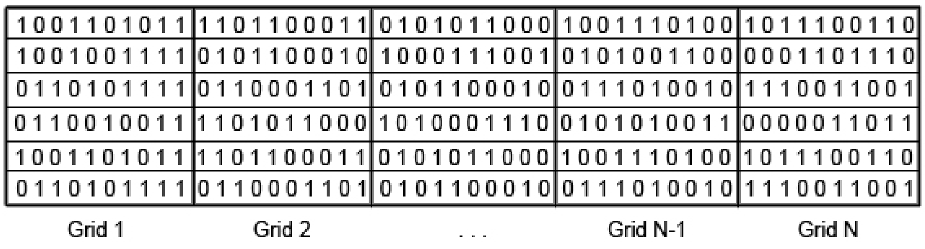


Figure 4.1: Visualization of creating a big grid by cutting the string in pieces of equal length.

The next step is to divide this grid into 'subgrids' of equal size as in figure 4.2. Each of these subgrids will be forming the basis for both an initial and desired configuration for an independent Bitmap Problem.

The subgrids form a 'chain' of Bitmap Problems where each desired configuration is also the initial configuration for our next Bitmap Problem. So the next step in the algorithm is to solve all of these single Bitmap Problems of the chain, starting at the last one and working its way up to the front. One of the key features of the algorithm is that the information representing the solution to the Bitmap Problem, being the found rule and the number of iterations needed to get to desired configuration, is to be encoded within the chain of Bitmap Problems itself.



Grid 1	Grid 2	...	Grid N-1	Grid N
1001101011	1101100011	0101011000	1001110100	1011100110
1001001111	0101100010	1000111001	0101001100	0001101110
0110101111	0110001101	0101100010	0111010010	1110011001
0110010011	1101011000	1010001110	0101010011	0000011011
1001101011	1101100011	0101011000	1001110100	1011100110
0110101111	0110001101	0101100010	0111010010	1110011001

Figure 4.2: Divide the large grid into subgrids of equal size

So when the initial data, $data_{original}$, has been divided into N subgrids, $subgrids = \{sg_1, sg_2, ..., sg_{N-1}, sg_N\}$, the process will start by using sg_N as desired configuration and sg_{N-1} as initial configuration for the first Bitmap Problem of the chain. When decompressing the data, the algorithm has to know when the final subgrid is reached. Therefore a special 'end flag' is concatenated to sg_N making it recognizable later on. The size of the subgrid is not important for now. After adding this 'end flag' to sg_N, the height and width of sg_{N-1} no longer matches those of sg_N. To fix this, a 'standard init flag' is added to sg_{N-1} to make them equal in size again. Both the 'end'- and 'init' flag are defined in advance. Now that sg_N and sg_{N-1} have been prepared, they will form the first Bitmap Problem which is to be solved by a Genetic Algorithm (or equivalent method).

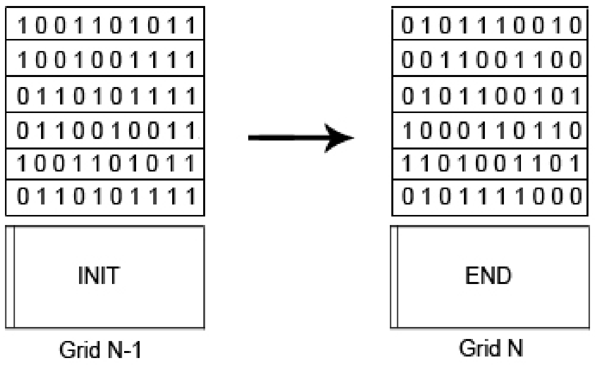


Figure 4.3: Special 'flags' have been concatenated to the subgrids, making them the first Bitmap Problem of the chain.

When a successful rule is found, the encoded rulestring (as described in section 1.1.3), together with the used updating method and the amount of iterations needed to reach the desired configuration, will replace the 'initial flag' in sg_{N-1}. sg_{N-1} now contains all the information needed to reconstruct the configuration of subgrid sg_N.

Thereafter sg_{N-2} with the standard initial flag will be used as initial configuration and sg_{N-1}, with the previously described information appended to it, acts as desired configuration. Together sg_{N-2} and sg_{N-1} form the next Bitmap Problem of the chain. This process repeats itself until all N subgrids, and therefore all initial data, has been encoded within this chain. In theory, the only information needed to unravel this chain and reconstruct the original data again, is the information on the used flags (we have to know how the initial flag looks and how to recognize the end of the chain) and the first initial configuration containing the rule and number of iterations needed to get starting.

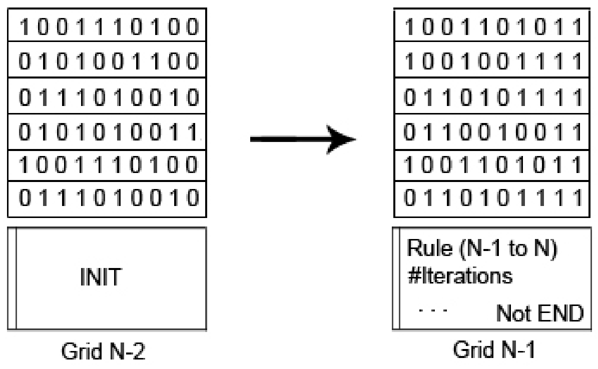


Figure 4.4: The 'init flag' has been appended to subgrid N-2 and subgrid N-1 has the required information to get to subgrid N appended to it, making them the next Bitmap Problem of the chain.

Logically there are some remarks to be made to this algorithm and it might not even work at all for several reasons, at least not for every given binary string. However, it is just a concept and is meant to be explanatory.

4.2 Algorithm variation

There are several variations to think of, one of them will be shortly discussed in this subsection. The ’original’ concept suggest one starting configuration accompanied by one rule and other required information to get started. Another way to go would be to, like in the ’original’ concept, start by dividing the original data in subgrids. In this variation however, the subgrids will not function as both initial and desired configuration in a chain, but only as a desired configuration. Having only one initial configuration, a Genetic Algorithm or similar will find rules and iteration numbers to get to the various data parts. The compressed data will then consist of the initial configuration and a rule and iteration amounts for every subgrid of the original data. See figure 4.5 for a schematic representation.

This means that, in contrary to the original concept, the compressed data will increase in size when the data which is to be compressed increases. An important thing to note in this case, is that the size of data that describes how to get to a certain subgrid, consisting the rule and amount iterations, should be smaller than the subgrid it points to. The ratio between these sizes will define the factor of compression of the algorithm as a whole.

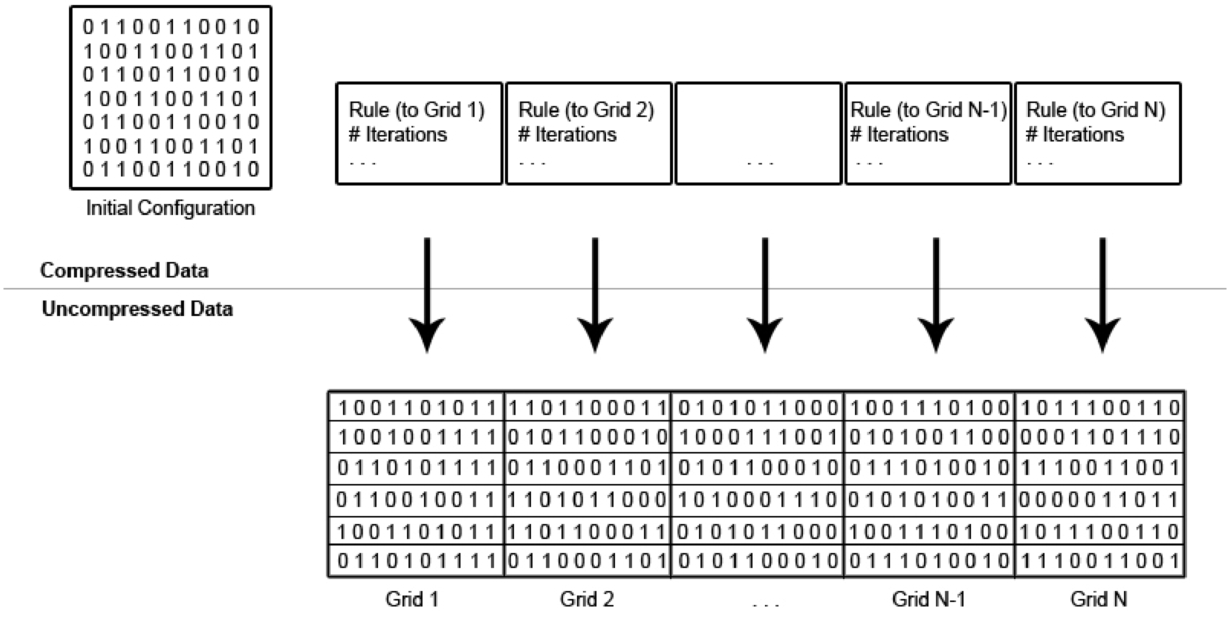


Figure 4.5: A visual representation of a variation on the original concept of the compression algorithm. This variation removes the 'chain' of the created Bitmap Problems and uses the same initial configuration for all created desired configurations, saving the found rules and iteration values to reconstruct the original data.

Bibliography

[1] B*ä*ck, T. Fogel, D.B., Michalewicz, Z. editors, Handbook of Evolutionary Computation. Oxford University Press and Institute of Physics Publishing, Bristol/New York, 1997.

[2] Berlekamp, E.R., Conway, J.H., Guy, R.K., Winning ways for your Mathematical Plays Vol.2 Academic, 1982

[3] Breukelaar, R., Interaction and Evolutionary Algorithms, 2010.

[4] Breukelaar, R., Evolving Transition Rules for Cellular Automata with multiple dimensions, Master Thesis Leiden University, 2004.

[5] Chapman P, ”Life Universal Computer”, http://www.igblan.free-online.co.uk/igblan/ca/ 2002

[6] Cook, M., *Universality in Elementary Cellular Automata*, 2004

[7] Das R., Crutchfield J.P., Mitchell M., *“Evolving Globally Synchronized Cellular Automata”*, Proceedings of the 6th International Conference on Genetic Algorithms, pp. 336-343, 1995

[8] Gacs, P., Kurdyumov, G.L., Levin, L. A., One dimensional unifrom arrays that wash out finite islands. Problemy Peredachi Informatsii. 1978

[9] Gardner M, *“Mathematical games – the fantastic combinations of John Conway's new solitaire game ‘Life’”* Scientific American 223 pp. 120-123

[10] Goldberg, D.E., Genetic Algorithms in Search, Optimization and Machine Learning. Addison-Wesley, 1989.

[11] Idilli, G., Evolving Cellular Automata Synchronization

[12] Kari, J., Theory of cellular automata: A survey, 2004

[13] Moore, E.F., Machine Models of Self-Reproduction, 1966

[14] von Neumann, J., Burks, A.W., Theory of Self-Reproducing Automata, 1966

[15] Schiff, Joel L., Cellular Automata A Discrete View of the World, 2008

[16] Schönfisch, B., de Roos, A., Synchronous and asynchronous updating in cellular automata, 1998

[17] Wolfram, S., *A New Kind of Science*. Wolfram Media Inc. 2002

[18] Wolz D., de Oliveira P.P.B., *"very effective evolutionary techniques for searching cellular automata rule spaces"*, Journal of Cellular Automata; 3, pp. 121 – 142, 2008