Interaction and evolutionary algorithms

Breukelaar, R.

Interaction and Evolutionary Algorithms

Proefschrift

ter verkrijging van

de graad van Doctor aan de Universiteit Leiden,

op gezag van de Rector Magnificus prof. mr. P.F. van der Heijden,

volgens besluit van het College voor Promoties

te verdedigen op dinsdag 21 december 2010

klokke 11:15 uur

door

Ron Breukelaar

geboren te Winterswijk, Nederland

in 1978.

Samenstelling promotiecommissie:

promotors:	Prof. Dr. T.H.W. Bäck	Universiteit Leiden
	Prof. Dr. J.N. Kok	Universiteit Leiden
overige leden:	Prof. Dr. T. Bartz-Beielstein	Universität Köln
	Prof. Dr. F. Arbab	Universiteit Leiden
	Prof. Dr. B. Katzy	Universiteit Leiden

Interaction and Evolutionary Algorithms
by Ron Breukelaar
Dissertation University Leiden

This work is part of the research programme of the Foundation for Fundamental Research on Matter (FOM), which is part of the Netherlands Organisation for Scientific Research (NWO). FOM Project: An evolutionary approach to many-parameter physics, project nr.: 03TF78-2, werkgroep FOM-L-24

ISBN 978-94-9109-804 8

Contents

Chapter 1

Introduction

Evolution and Interaction are two processes in Computer Science that are used in many algorithms to create, shape, find and optimize solutions to real world problems. Evolution has been very successfully applied as a powerful tool to solve complex search problems in fields ranging from physics, chemistry and biology all the way to commercial application such as aircraft fuselage design and civil engineering grading plans. Defining interaction is a big part of algorithm design. Not only defining the inputs and outputs of an algorithm but for a complex algorithm the interactions inside of an algorithm are as important. This thesis will concentrate on where Evolution overlaps Interaction. It will show how evolution can be used to evolve interaction, how the interaction inside an evolutionary algorithm impacts its performance and how an evolutionary algorithm can interact with humans. By touching on these three forms of overlap this thesis tries to give insight into the world of evolution and interaction. This chapter will give a brief introduction on each of these three overlaps.

1.1 Evolving Interaction

With all due respect to the people that believe our earth is no older than 6,000 years, the general consensus is clear: Evolution is real. The overwhelming evidence points to a common ancestor with apes about 5-8 million years ago. Evolution gradually changed us from a tree climbing leaf eater to a car driving hamburger lover. It made us walk upright, loose most of our

hair and it might even have made us aware of ourselves. Evolution has performed miracles, but not only for our own species. Any natural history museum has a whole collection of extinct species that appeared on this earth in the past for no (apparent) other reason than that they evolved from slightly different (often more primitive looking) species. Dinosaurs, birds, fish, frogs, flies, flowers, trees, even the bacteria and single celled organisms that keep you alive by digesting your food and protecting your skin; they all seem to have come into being through the process of evolution. Even though our understanding of our world has been increasing almost exponentially the past 60,000 years, science will probably never fully explain why evolution exists, but it seems to be a pretty good method to sustain life throughout changing environmental conditions. In a way evolution is nature's search algorithm for improving life's chances.

The parallel between evolution in nature and search algorithms in computer science at first may seem like a stretch, but that is mainly due to the 'unnatural' characteristics of a computer. Many sciences have been inspired by nature and computer science in no exception to that. The field that studies computer science (and algorithms in particular) inspired by nature is aptly called 'Natural Computation'. It studies for instance the intricate way the neurons in the brain work and how abstract computer generated simulations of brain cells (Neural Networks) can learn and solve problems that seemed outside of the realm of computers before. It studies how simple cells live and crystals grow and how to build models to simulate, predict and apply these phenomena in other fields. Natural Computation also studies the process of evolution and in particular its application in search algorithms. The group of algorithms in computer science inspired by evolution in nature is called 'Evolutionary Algorithms'.

Evolutionary Algorithms work by simulating an abstract form of natural evolution to find a better solution to a hard problem. In nature evolution works with a rigorous selection process. If an animal is sick or malformed it will have more trouble staying alive and will have a smaller chance to have offspring. This means that on average fitter individuals have more offspring. Through the use of DNA the characteristics of the parent individuals are given to the offspring, giving this offspring a similar chance to survive and have offspring of its own. Because small mutations are introduced during the copying process of DNA, new offspring will have a slightly higher or slightly lower chance of survival compared to their parents. The slightly 'fitter' ones will on average generate more offspring in the end and this loop continues generating ever fitter individuals that are better able to survive and produce offspring.

Evolutionary Algorithms work the same way, but instead of life forms an EA

is evolving answers and instead of being eaten or starvation as a selection procedure an EA uses a computer problem as an evaluator. The answers are in this case the 'individuals' and the selection procedure is called a 'fitness function', but the rest works pretty much the same. An EA has a 'pool' of 'individuals' which each have a certain 'fitness' calculated by the 'fitness function'. A selection step selects the best individuals and they generate new offspring in the pool. This new offspring looks a lot like their parents but is slightly 'mutated', which means that in the next iteration of the algorithm the total fitness of the individuals has probably increased. Which in terms of the computer algorithm means: it found a better solution. (A more in depth introduction to EA will be given in Chapter 2)

In nature interaction is what makes life possible. From the macrobiotic scale of mammals and plants all the way to the microbiotic scale of single celled organisms: without interaction the 'task' of finding food, staying alive and generating offspring would be totally impossible. The same way that appearance and function of a single individual evolves in nature, so also evolves the interaction between an individual and its environment over time. The way an individual sees things, feels things, conveys messages to other individuals are mostly all encoded inside DNA and evolved alongside other traits. This is true for the evolution of sensors like eyes, nose and ears for instance, but this is also true for interactions between different individuals of the same species. A good example of evolved interactions in a species is the ant.

An ant colony can seem chaotic and crowded and we don't think of individual ants as being highly intelligent, yet somehow a colony of ants is able to find shortest paths to food, coordinate attacks against enemies, nurture thousands of babies, build bridges, air vents, flotillas and intricate tunnel systems. One ant might not seem very smart, but the ant colony as a whole could easily 'outsmart' your average pet. The reason for this lies in the evolved communication between the ants. Ants excrete pheromones as a way of sending messages to other ants when they find food, or get attacked for instance. The intricate rule set for which pheromone means what message is different between different species of ant. This message system evolved with the ant species to work best for the specific environment the ant species is living in. This is very visible in the case of an ant colony, but these evolved interactions are present in all forms of life including humans, apes, bees, but also in flowers and trees, even in single celled organisms. Interaction and cooperation seems to be a good idea if you want to stay alive.

A Cellular Automaton is an abstraction of this interaction between single celled organisms. In its most basic form it describes a ring of cells in the

form of an array of binary values. Each cell is in a certain state (either 0 or 1) and is connected to neighboring cells (one left and one right). Time is simulated with iterations which are applied synchronously to all cells at once. At each iteration every cell looks at the states of its neighborhood and decides what its next state is going to be according to a transition rule. Usually every cell will have the same transition rule and therefore in effect the same behavior. This gives a simple yet powerful framework to simulate interaction which allows CA to simulate complex physical, chemical and biological system and are known to exhibit communication.

Chapter 4 will investigate the evolution of interaction by evolving the transition rules inside a Cellular Automaton. By inverse designing the behavior of CA we demonstrate how problems that need interaction between cells to be solved, are solved using a generic evolutionary approach. Demonstrating not only that interaction evolved inside of a local individual can exhibit behavior on a global scale, but also how this approach can be applied to real world applications.

1.2 Interaction inside Evolution

An Evolutionary Algorithm could also be viewed as an iterative process of interacting individuals. Generating offspring is often done using multiple individuals and combining their traits is an interaction for instance, while selection could be viewed as one big interaction between all the individuals that results in finding the fittest one. The benefit of looking at evolution as having interaction between individuals in a population is that some hard to understand phenomena observed in EA become understandable.

A good example of a non trivial interaction inside an Evolutionary Algorithm is self-adaptation. With self-adaptation some parameters for the EA that are normally fixed or are changing using some mathematical function, now change using the evolution itself. For instance the parameters with which an individual is mutated (mutation amount / speed) can be part of the individual's description. The idea being that when a certain way of mutating an individual is more appropriate at a certain stage of the evolution, the offspring generated using that mutation will on average have a better fitness. Which means that selection will probably select individuals with better mutation settings which then propagate towards the offspring of these individuals and so on. This works for mutation parameters, but also for more complicated global parameters like selection and offspring generation. In all cases the algorithm becomes more flexible and can handle a lot more different problems using the same settings, but some unwanted

behavior resulting from using this approach is harder to understand and fix.

Chapter 5 begins with describing how self-adaptation was implemented on one of the experiments described in Chapter 4. And although this was successful, there were some unexpected side effects that were not easy to explain. The chapter then tries to describe the EA in terms of interacting parts and concentrates on why self-adaptation in this EA did not work as expected. After reproducing the same effect on a much simpler experiment the chapter concludes that self-adaptation in this flavor of EA has a general problem that is important to be aware of. This counter intuitive behavior of the EA is only counter intuitive from the point of the global behavior, but becomes understandable if the individual interactions of its parts are examined. Apart from pointing out this particular problem, it demonstrated the power and usefulness of describing evolution in terms of interactions.

1.3 Interacting with Evolution

Computers are a big part of almost every person's work and life nowadays, yet despite our best efforts and intentions computers are not like humans. This means that there is a clear disconnect in the communication between computers and humans. We don't understand each other. The reason we use computers at all is for the simple fact that they are faster and more precise than humans. This means that by using a computer we can solve problems that are much larger and more complex than anything we have solved in the past. We can simulate the physical world, iterate through mathematical equations and visualize virtual output in great detail. The computer has opened up a world of possibilities that we are only just starting to utilize.

The main problem in using the computer effectively is interaction. The more complex the task is that a user wants the computer to perform, the more complicated the interface becomes between the human and the computer. The field that studies this interaction is amply called Human Computer Interaction. It studies ways to efficiently use a computer screen, a mouse, task bar, windows, buttons, sliders, images and text to name a few. It studies how humans like to work, what is intuitive and what is not.

When computers are running complex algorithms that take parameters and input from the user while they are running, we are taking about a subset of HCI: Human Algorithm Interaction. Instead of concentrating on what the user finds intuitive, this concentrates more on how a certain algorithm

can be manipulated by a user and what effect this has on the algorithm in question. The benefit of human input to algorithms is especially apparent when an algorithm can make use of the experience of a human specialist. A civil engineer for instance might have acquired knowledge over the years that is very hard to put into a computer algorithm. Not only is it hard to define one algorithm that takes care of all exceptions the engineer has encountered in his work, but the engineer will also have trouble relaying all those exceptions without having a situation to remind him. Human Algorithm Interaction gives a user the ability to solve a complex problem with a relatively simple algorithm and let a human steer that algorithm using his expert knowledge of the problem.

A lot of complex problems have complex specific algorithms to solve the problem. Generating such a specific algorithm usually means that a lot of knowledge of the problem needs to be put into the algorithm and the algorithm will then only work on that specific problem. With the speed of computers and the amount of data exponentially increasing over time it is no surprise that the problems we want to solve with our computers are becoming more and more complex as well. This means it is getting harder to understand exactly what algorithm is needed to solve the problem and for a lot of problems there is even no known algorithm that can solve them. Evolutionary Algorithms have been successfully used in exactly these cases where it is hard to translate the specific information about a problem into a specific algorithm.

The powerful thing about Evolutionary Algorithms is that they don't need any problem specific information to find better solutions to hard problems. That means they are a good answer to problems that are hard to solve using conventional algorithms and almost the only answer to problems that have no known algorithm to solve them. It also makes them a good candidate for use in Human Algorithm Interaction. They don't need much knowledge to start from and have a clear path of search that can be shown at any time during the algorithm. The human basically becomes part of the algorithm which adds the human expert's knowledge to the search process without having to translate this knowledge into an algorithm itself. Chapter 6 will investigate this interaction between humans and EA and in particular study the effect human input has on different flavors of EA.

By showing three different ways of combining Interaction with Evolution this thesis demonstrates the power and flexibility of Evolutionary Algorithms, while at the same time introducing some new and interesting findings in the fields of Inverse Design of Cellular Automata, Self-Adaptation in Genetic Algorithms and Human Algorithm Interaction.

1.4 Overview of this Thesis

This Section will give an overview of the thesis chapter by chapter.

Chapter 2 will give a brief introduction on Evolutionary Algorithms, on how they work and how they are employed in this thesis.

Chapter 3 will introduce Cellular Automata in general and binary synchronous Cellular Automata in particular.

Chapter 4 discusses the inverse design of Cellular Automata using a Genetic Algorithm. Cellular automata are used in many fields to generate a global behavior with local rules. Finding the rules that display a desired behavior can be a hard task especially in real world problems. This chapter proposes an improved approach to generate these transition rules for multi dimensional cellular automata using a genetic algorithm, thus giving a generic way to evolve global behavior with local rules, thereby mimicking nature. Three different problems are solved using multi dimensional topologies of cellular automata to show robustness, flexibility and potential. The results suggest that using multiple dimensions makes it easier to evolve desired behavior and that combining genetic algorithms with multi dimensional cellular automata is a very powerful way to evolve very diverse behavior and has great potential for real world problems.

Chapter 5 will describe the findings on using self-adaptation in Genetic Algorithms. Self-adaptation is used a lot in Evolutionary Strategies and with great success, yet for some reason it is not the mutation adaptation of choice for Genetic Algorithms. This chapter describes how a self-adaptive mutation rate was used in a Genetic Algorithms to inverse design behavioral rules for a Cellular Automaton. The unique characteristics of this search space gave rise to some interesting convergence behavior that might have implications for using self-adaptive mutation rates in other Genetic Algorithm applications and might clarify why self-adaptation in Genetic Algorithms is less successful than in Evolutionary Strategies.

Chapter 6 will discuss Evolution Strategies within the context of interactive optimization. Different modes of interaction will be classified and compared. A focus will be on the suitability of the approach in cases, where the selection of individuals is done by a human user based on subjective evaluation. We compare the convergence dynamics of different approaches and discuss typical patterns of user interactions observed in empirical studies. The discussion of empirical results will be based on a survey conducted via the world wide web. A color (pattern) redesign problem from literature will be adopted and extended. The simplicity of the chosen problems allowed

us to let a larger number of people participate in our study. The amount of data collected makes it possible to add empirical support to our hypothesis about the performance and behavior of different Interactive Evolution Strategies and to figure out high-performing instantiations of the approach. The behavior of the user was also compared to a deterministic selection of the best individual by the computer. This allowed us to figure out how much the convergence speed is affected by noise and to estimate the potential for accelerating the algorithm by means of advanced user interaction schemes.

1.5 Overview of Publications

Below is a list of all the publications used in this thesis by chapter.

Chapter 4 is based on multiple publications including:
Ron Breukelaar and Thomas Bäck, Evolving Transition Rules for Multi Dimensional Cellular Automata, *proceedings of Sixth International Conference on Cellular Automata for Research and Industry*, ACRI 2004, Peter M.A. Sloot, Bastien Chopard and Alfons G. Hoekstra (editors), Springer-Verlag, LNCS 3305, pg. 182–190 (2004)

Thomas Bäck, Ron Breukelaar and Lars Willmes, Problem Solving by Evolution: One of Nature's Unconventional Programming Paradigms, *pre-proceedings of Unconventional Programming Paradigms workshop*, UPP 2004, Jean-Pierre Banâtre, Pascal Fradet, Jean-Louis Giavitto and Olivier Michel (editors), Springer-Verlag, pg. 8–13 (2005).

Ron Breukelaar and Thomas Bäck, Using a Genetic Algorithm to Evolve Behavior in Multi Dimensional Cellular Automata, *proceedings of Genetic and Evolutionary Computation Conference*, GECCO 2005, Hans-Georg Beyer et al. (editors), ACM 1-59593-010-8/05/0006, pg. 107–114 (2005).

Thomas Bäck, Ron Breukelaar and Lars Willmes, Inverse Design of Cellular Automata by Genetic Algorithms: an Unconventional Programming Paradigm, *Unconventional Programming Paradigms: International Workshop UPP 2004, Revised Selected and Invited Papers*, Jean-Pierre Banâtre et al. (editors), Springer-Verlag, LNCS 3566, pg. 161–172 (2005).

Ron Breukelaar and Thomas Bäck, Using a Genetic Algorithm to Evolve Behavior in Cellular Automata, *proceedings of Computation: 4th International Conference*, UC 2005, Sevilla, Spain, October 3 - 7, 2005., Cristian S. Calude, Michael J. Dinneen, Gheorghe Paun, Mario J. Péréz-Jiménez and

Grzegorz Rozenberg (editors), Springer-Verlag, LNCS Volume 3699, pg. 1–10 (2005).

Chapter 5 is based on:
Ron Breukelaar and Thomas Bäck, Self-adaptive Mutation Rates in Genetic Algorithm for Inverse Design of Cellular Automata, *proceedings of Genetic and Evolutionary Computation Conference*, GECCO 2008, pg. 1101–1102 (2008).

Chapter 6 is based on:
Ron Breukelaar, Michael Emmerich and Thomas Bäck, On Interactive Evolution Strategies, *proceeding of Applications of Evolutionary Computing*, EvoWorkshop2006: EvoINTERACT, Franz Rothlauf et al. (editors), Springer-Verlag, LNCS Volume 3907, pg. 530–541 (2006).

Chapter 2

Evolutionary Algorithms

Evolutionary Algorithms is the name for the algorithms in the field of Evolutionary Computation which is a subfield of Natural Computing and already exists more than 40 years. It was born from the idea to use principles of natural evolution as a paradigm for solving search and optimization problems in high-dimensional combinatorial or continuous search spaces. The most widely known instances are genetic algorithms [17, 18, 22], genetic programming [26, 27], evolution strategies [33, 34, 38, 39], and evolutionary programming [15, 14]. A detailed introduction to all these algorithms can be found e.g. in the Handbook of Evolutionary Computation [6], but this chapter will give a short intro to each of them and will then go into some depth on the algorithms used for this thesis.

Today the Evolutionary Computation field is very active. It involves fundamental research as well as a variety of applications in areas ranging from data analysis and machine learning to business processes, logistics and scheduling, technical engineering, and others. Across all these fields, evolutionary algorithms have convinced practitioners by the results obtained on hard problems that they are very powerful algorithms for such applications. The general working principle of all instances of evolutionary algorithms is based on a program loop that involves simplified implementations of the operators mutation, recombination, selection, and fitness evaluation on a set of candidate solutions (often called a population of individuals) for a given problem. Next this chapter will define this evolution loop and all its parts in a generic EA and it will show the differences between the different flavors of EA in terms of data structure and general workings.

2.1 Individuals

Every Evolutionary Algorithm (EA) works by maintaining a group of one or more individuals as its 'population' (sometimes also called 'pool'). Each of these individuals is defined as the representation of a solution to the problem that needs to be solved. We call the solution a 'phenotype' and the representation of this solution a 'genotype'. In some algorithms the phenotype of an individual can be identical to the genotype, but that usually depends on which class of algorithm is used.

One individual's genotype is usually denoted with a vector of values $\vec{a}$, while the phenotype of the individual is denoted with $\vec{x}$. A population of individuals is usually denoted with $P = \{\vec{a_1}, ..., \vec{a_\lambda}\}$, where λ is the size of the population. The state of an individual at time t is then denoted with $\vec{a}(t)$, so that the state of a whole population at t can be denoted with $P(t) = \{\vec{a_1}(t), ..., \vec{a_\lambda}(t)\}$.

As mentioned before there are four main classes of EA: Genetic Algorithms, Genetic Programming, Evolutionary Strategies and Evolutionary Programming. The main distinguishing trait between these classes is the way they represent their individual's genotype and phenotype. Most other differences are directly or indirectly related to these difference in representation. What follows is a very brief and incomplete overview of some of these characteristics and differences.

Genetic Algorithms (GA) usually represent an individual with a bit string $\vec{a} = (a_1, ..., a_l) \in \{0, 1\}^l$. The philosophy being that "if nature does it, it must be right". Nature represents their individuals using DNA of which a bit string is an abstraction. Although the genotype is a binary representation, the phenotype can have any kind of representation as long as there is a way to 'map' the genotype to the phenotype in order to evaluate an individual, this mapping is then denoted as $\vec{x} = \Upsilon(\vec{a})$.

Genetic Programming (GP) traditionally represents its individuals with a tree structure. In which each node of the tree represents an equation that needs to be performed on the result of each of its leaves, making it possible to define and evolve mathematical equations quite efficiently. There are different representations possible, most more complex than this one, but in almost all cases the genotype is a direct representation of the phenotype.

Evolutionary Strategies (ES) usually represent their individuals with a real valued array $\vec{a} = (a_1, ..., a_l) \in \mathbb{R}^l$. This makes it easier to solve real world problems where rounding a parameter due to a mapping can have a

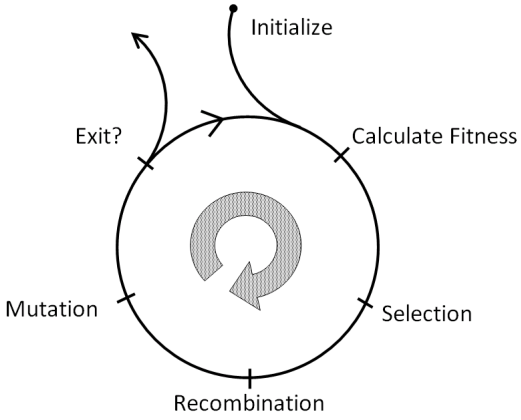

Figure 2.1: This figure shows the different steps in an evolutionary loop.

big impact on the accuracy of the results. The more flexible representation opens up possibilities for more advanced mutation and recombination operators using the direction or size of previous mutation steps. Some operators store additional values into the individual which are then considered part of the genotype but not part of the phenotype. Except for these additional values the phenotype is most of the time identical to the genotype.

Evolutionary Programming (EP) traditionally represents its individuals with a finite state machine. It looks a little bit like Genetic Programming in terms of its phenotype representing a program, while it uses what looks like an Evolutionary Strategy approach towards changing its values. Here the genotype (the values) is not identical to the phenotype (the state machine using these values). Evolutionary Programming looks a lot like Evolutionary Strategies nowadays.

In this thesis only the Genetic Algorithm and the Evolutionary Strategy will be used.

2.2 Evolutionary Loop

The evolutionary loop starts by **initializing the individuals** in the population. This can be done randomly using some uniform or Gaussian distribution or it can be started from a fixed location. Usually the initial size (here denoted by n) of the population is also the size throughout the loop

(it never changes), but there are some algorithms that break that rule (as there are algorithms that break any rule in this brief intro).

Next the **fitness** of every individual in the population is calculated using a fitness function $\Phi(\vec{x_i})$. Note that for some algorithms this means that the genotype needs to be mapped to the phenotype first in which case $\Phi(\vec{x_i}) = \Phi(\Upsilon\vec{a_i})$. The fitness value $f_i \in \mathbb{R}$ of the individual is usually represented with a value between 0 and 1 and is attached to the individual for later use.

Then the a **selection** is made. There are many different ways of doing selection and some will be discussed in more detail in Section 2.3, but the general aim of this step is to select the 'best' μ individuals based on their fitness value as parents to a new generation of λ children. The main distinction between the different selection methods is the amount of 'luck' a worse individual is allowed to have. The philosophy being that searching for the best individual sometimes means there is a need to diversify and not concentrate too much on what is currently the best individual.

Right after the fitness is calculated the **exit criteria** are checked. Most of the exit criteria are based on fitness or duration of the algorithm, but anything is possible here. The general question that is answered: "When does the algorithm need to stop?". For example the two exit criteria used in this thesis are: 'Stop if the maximum fitness in the population reached the optimum.' and 'Stop if the algorithm reached the maximum number of generations.' The maximum number of generations is a parameter that is defined for each experiment separately.

The **recombination** step almost overlaps but follows the selection step, because recombination makes its own selection to choose which individuals are going to make offspring. Depending on the algorithm and the representation, recombination can mean anything from just plain copying one individual to calculating the intermediate weighted location in a multi dimensional space using multiple parents. There are many different ways of doing this and Section 2.4 introduces the ones that are used in this thesis.

Then **mutation** is applied to all newly generated individuals. The way an individual is mutated depends heavily on how it is represented which traditionally depends on the class of algorithm that is used (as described in Section 2.1). Section 2.5 shows the different ways mutation is used in this thesis.

After the mutation step the resulting population of individuals can consist of both old and new individuals. One way to notate the specific selection type

of an algorithm is with (μ, λ) or $(\mu + \lambda)$ where μ stands for the number of parents that are selected to produce offspring and λ stands for the number of offspring that is generated each iteration (or 'generation'). When a ',' is used in the notation every generation only uses newly created individuals, but when a '+' is used the parents of the offspring get copied into the next generation as well. Using a 'comma strategy' makes for an algorithm that will not easily focus too much on one solution, while a 'plus strategy' makes for an algorithm that does not easily 'throw away' a good solution.

The loop is closed by taking the resulting population that comes out of the mutation step and going back into the fitness evaluation. The new individuals will then get their fitness values which will be checked against the exit criteria. If the exit criteria is not yet met, the selection step will again select the 'best' and so on. The only way to successfully stop the loop is if one of the exit criteria triggers. After that the result is usually one or more individuals with the best fitness in the population.

2.3 Selection

There are many different ways to select individuals based on their fitness value. The most popular include Probabilistic Fitness (or Roulette Wheel) Selection, Truncation Selection, Probabilistic Rank Selection and Tournament Selection. Each has a different probability distribution for the chance that an individual is going to be selected based on its fitness or rank in the population. What follows is a brief description of each of these selection operators. For a detailed explanation on what each of these (and other) selection methods are doing, see [6].

Truncation Selection is the easiest selection method there is. It selects the μ individuals with the highest fitness, always. The drawback of this algorithm is that it specializes on good individuals very quickly. That means that the population has difficulties staying diverse, which means for some problems that the best solution (the 'global optimum') will never be found and the algorithm gets stuck on only a 'pretty good' solution (a 'local optimum'). This method has what is called a 'high selection pressure' because the selection is deterministic and relies totally on the fitness value of the individual relative to the entire population. This is however the most common selection method in most Evolutionary Algorithms not just because of its simplicity. Its deterministic nature makes it easier to combine with smart mutation and recombination operators that often have trouble with a more stochastic selection method.

(μ, λ)**- and** $(\mu + \lambda)$**-Selection** is basically identical to Truncation Selection and usually refers to its use in an ES. The two different forms are often called a 'comma strategy' and a 'plus strategy' respectively and they refer to the way the parents are treated in the evolutionary loop. In a comma strategy μ parents are selected from n individuals, they recombine to generate λ children and then the parents die and only the children go to the next generation. In that case the population size $n = \lambda$. While in a 'plus strategy' the μ parents also generate λ children, but then both the children and the parents go to the next generation. Note that then $n = \mu + \lambda$. Also note that usually only children are mutated in the evolutionary process, which leaves the parents in the next generation unaltered.

Probabilistic Fitness Selection or **Roulette Wheel Selection** is one of the probabilistic selection methods. It select a subset (could be all) of the population and gives each a 'pie-piece' of a virtual roulette wheel. The size of this 'pie-piece' is relative to the fitness of the individual. Then the wheel is 'spun' to select individuals. The probability of individual i being selected is then given by:

$$p_i = \frac{f_i}{\sum_{j=1}^{n} f_j}$$

The main problem with this approach is that the selection probability is very dependent on the fitness function and how a good fitness relates to a bad fitness in the population. In most fitness functions the relative fitness increase becomes smaller when the algorithm gets closer to the optimum. This makes that the relative chance to select a better individual decreased over time and that generates stagnation. For some fitness functions subtracting a constant value c from each fitness value to change the relative selection pressure improves performance:

$$p_i = \frac{f_i - c}{\sum_{j=1}^{n} f_j - c}$$

Yet this makes the value of c another problem specific setting that only improves the performance in some rare cases for certain parts of the search space.

Probabilistic Rank Selection is another classic probabilistic selection method. It works very similar to Probabilistic Fitness Selection except that

now the probability of selection depends on the rank in the sorted list of all individuals. The fittest individual gets the highest chance to be selected, then the second fittest and so on. Usually only a subset of the population gets a chance to be selected and the chances are fixed for each rank.

There are many different ways to distribute the selection probability over the ranks, but linear distribution is most commonly used. Given that $P(\vec{a})$ is ordered so that $\vec{a_1}$ has the highest fitness (p_1) and $\vec{a_n}$ has the lowest (p_n), then:

$$p_i = \frac{(n-i)\cdot p_1 + (i-1)\cdot p_n}{n-1} \quad \text{while} \quad \sum_{i=1}^{n} p_i = 1 \text{ and } p_i \geq 0 \quad \forall i \in \{1, ..., n\}$$

and seeing that the average probability is $1/n$ this implies that:

$$p_1 = \frac{1}{n} + c \quad \text{and} \quad p_n = \frac{1}{n} - c \quad \text{while} \quad 0 \leq c \leq \frac{1}{n}$$

Rank based selection methods are a lot less dependent on the fitness distribution in the population than Probabilistic Fitness based selection methods. This plus the ability to change the selection pressure with relative ease, makes this selection operator a valid alternative to the more common Truncation and Tournament selection methods.

Tournament Selection is a very flexible selection method that is very close to nature. To select an individual, q ('tournament size') individuals are chosen to take part in a 'tournament' with only one winner. The winner of that tournament is the individual with the highest fitness values of the individuals in the tournament and is selected as a parent. This process is repeated μ times to select all the parents for the next generation.

Note that the q individuals for each tournament are pulled from the entire population, which means that (unlike the other selection methods mentioned here) an individual can be selected multiple times in the same generation. In which case the individual would be copied and used as a parent multiple times. Also note that Tournament Selection has a flexible 'selection pressure' through changing q. When $q = 1$ the selection method is a complete random selection without any selection pressure, while with $q = n$ it would only select the best individual μ times.

The probability for an individual to be selected in Tournament Selection can be defined as:

$$p_i = \frac{1}{n^q} \cdot \Big((n-i+1)^q - (n-i)^q \Big)$$

(see [5] for a proof)

Note that in this thesis only Truncation Selection (both (μ, λ) and $(\mu + \lambda)$) and Tournament Selection are used.

2.4 Recombination

Like for selection, for recombination there are many different approaches, but unlike for selection the representation of the individual has a great impact on which recombination operator can be used. Because recombination is performed in the genotype, calculating the average between two points ('Intermediate Recombination') is not possible on an individual represented by a binary array. Some popular examples of recombination are:

Copy does not really recombine anything, but does generate offspring and therefore belongs in this step. Note that there are different ways to select a single parent multiple times which can have a big impact on the algorithm's behavior. In this thesis we only use the copy operator to copy a parent exactly $\frac{\lambda}{\mu}$ times, but basing this on rank or fitness or even making it probabilistic is not uncommon.

Crossover is only used on binary representations. It generates the binary array of the offspring by combining the binary arrays of randomly chosen parents. This is usually done on only 2 parents, but in theory can be done on more than 2. There are a few different ways to combine two binary arrays:

- With 'Uniform Crossover' every single bit has an equal probability to come from either parent.
- With 'Single Point Crossover' a split point is chosen at random and all bits until that point are copied from one parent and all other bits from the other. This only works with 2 parents.
- With 'Multi Point Crossover' multiple split points are chosen at random and the source of the bits changes to the next parent with every split point along the bit array.

Note that because parents are usually chosen randomly one parent can be chosen multiple times. Not only for multiple offspring, but even for the same

offspring. This means that if all parents chosen for the offspring are one and the same, that parent is 'copied' without changes.

Intermediate Recombination is mostly used in Evolutionary Strategies. It combines multiple parents by calculating the average values for each value in the real valued arrays of the individuals. In some popular algorithms a weighted average is used instead, weighted on the rank if the parents in the population.

In this thesis only Copy and Crossover recombination are used.

2.5 Mutation

Mutation also comes in many different flavors and is probably the most problem specific operator of the entire evolutionary loop. It determines how fast and how far an evolution is able to 'jump' from one solution to the next and (like recombination) is very representation dependent. Some well known mutation operators:

Probabilistic Bit Flip flips each bit in a binary array with the same probability p_m (called 'mutation rate'). Given $\vec{a} = \{a_1, ..., a_n\} \in \{0, 1\}^n$ this operator can be defined as:

$$a_i(t+1) = \begin{cases} a_i(t) & \text{, if } U_i > p_m \\ 1 - a_i(t) & \text{, otherwise} \end{cases}$$

where U_i is a real value between 0 and 1 randomly sampled from a uniform distribution for each bit in the bit string. This mutation is mainly used on Genetic Algorithms and Genetic Programming, because it only works on binary strings.

Self-adaptive Probabilistic Bit Flip works the same way as Probabilistic Bit Flip, but it employs self-adaptation on the mutation rate p_m. This means it adds a separate mutation rate p_{m_i} to the representation of each individual. Then this mutation rate evolves with the individual using their own mutation operator and each individual uses their own mutation rate to be mutated. Given $\vec{a_i} = \{a_1, ..., a_n\} \in \{0, 1\}^n$ and each individual having a separate p_{m_i} this operator can be defined as:

$$a_j(t+1) = \begin{cases} a_j(t) & \text{, if } U_j > p_{m_i}(t) \\ 1 - a_j(t) & \text{, otherwise} \end{cases}$$

$$p_{m\,i}(t+1) = \Big(1 + \frac{1 - p_{m\,i}(t)}{p_{m\,i}(t)} \cdot \exp(-\gamma \cdot N(0,1))\Big)^{-1}$$

where U_i is a real value between 0 and 1 randomly sampled from a uniform distribution for each bit in the bit string, $N(0,1)$ is a real value randomly sampled from a Gaussian (or normal) distribution with mean 0 and standard deviation 1 and γ is a constant to control the speed at which the mutation rate $p_{m\,i}$ mutates.

Like Probabilistic Bit Flip, this mutation operator applies to binary strings and is only really used on Genetic Algorithms.

Gaussian Mutation mutates a real valued individual using a random Gaussian distributed step size. The Gaussian (or 'normal') distribution is a commonly used probability distribution that has very useful symmetrical properties that are exceptionally well suited for random mutation of real values. The distribution is given by:

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \;\; e^{-\frac{(x-\eta)^2}{2\sigma^2}}$$

where η is the mean of the distribution and σ^2 the 'variance'. The Gaussian distribution is symmetrical in the mean and has the added benefit that the standard deviation σ can be input straight into the distribution. This makes the distribution very compatible with step size mutation of real values where the distance a child mutates from a parent needs to be 'balanced' in order to prevent a bias towards bigger or smaller step sizes.

Gaussian Mutation is used in ES where an individual is represented by an array of l real values: $\vec{x_i} = \{x_1, ..., x_l\} \in \mathbb{R}^l$. (Note that $\vec{x} = \vec{a}$ in an ES) A simple Gaussian Mutation would then mutate each value of $\vec{x_i}$ with a certain 'step size' σ by sampling the distribution above to generate random differences to each of the values in $\vec{a_i}$. Given that sampling the Gaussian distribution is denoted with $N(\eta, \sigma)$, mutating $\vec{x}$ looks like:

$$x_i(t+1) = x_i(t) + N(0,\sigma)_i \quad \forall i \in \{1, ..., l\}$$

or for the entire pool:

$$\vec{x_i}(t+1) = \vec{x_i}(t) + \vec{N}(0,\sigma) \quad \forall i \in \{1, ..., n\}$$

Note that this is the simplest form of Gaussian Mutation. There are a lot of ways to improve the performance by manipulating the step size σ over time, or through self adaptation (see below).

Self-adaptive Gaussian Mutation has many different incarnations in Evolutionary Strategies. The basic principle is the same as in Self-adaptive Probabilistic Bit Flip in that the mutation rate for an individual is stored as part of the representation of that individual and mutated with that individual. The mutation operator for Gaussian Mutation can be a lot more complex than a simple Bit Flip operator though and each of the different ways of mutating the object variables can be part of the mutation operators that are evolved with the individual. There is the version where each object variable of an individual has its own mutation rate for instance, this helps an algorithm learn to move in one dimension, but stay where it is in another. There is a version in which the direction of the mutation is a vector that is mutated with the individual as well and there is even a very popular version in which the entire 'covariance matrix' is adapted to evolved a direction in the search space. All different degrees of complexity to tackle different order of magnitude of complexity in search spaces.

Until now there has not been one mutation operator that works every time. A good mutation operator is usually very problem specific, but the more problem specific information is added to the operator, the smaller the chance becomes that the algorithm will find something unexpected. That is why in the field of Evolutionary Computation a simple operator that does the trick is more often than not the best one.

Chapter 3

Cellular Automata

In the 1940s John von Neumann studied the problem of self-replicating systems at the Los Alamos National Laboratory, when his colleague Stanislaw Ulam suggested that instead of using actual parts to make a robot that could build itself, he would use a virtual model not unlike the model Ulam was using to simulate crystal growth. The resulting research generated the first so-called "Cellular Automata". It was two dimensional using a small neighborhood size in which each cell's only neighbors were its four direct neighbors in each direction and itself. This neighborhood has since been called the "von Neumann neighborhood". Within the CA a certain pattern would make endless copies of itself making it the first self-replicating automata.

Thirty years later in the 1970s a CA called "The Game of Life" got a lot of attention. This much simpler automata constructed by John Conway is able to generate and maintain a large variety of moving and looping patterns. Instead of the 29 states that each cell could have in von Neumann's CA, The Game of Life only has two states in each cell, but it uses the same small neighborhood as Neumann used and is also two dimensional. The patterns in this CA seem to move and merge, some even generate other patterns. The patterns seem to be 'alive'.

In 1983 Stephen Wolfram started investigating CA more closely and concentrated on an even simpler class of CA he called 'Elementary Cellular Automata'. These one dimensional CA have a neighborhood size of only 3 cells and only two states per cell. Wolfram showed that even in an automata this simple there exists a high level of complexity in terms of behavior. So

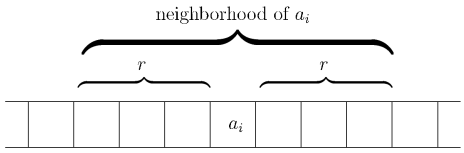

Figure 3.1: This figure shows the shape of a one dimensional neighborhood of cell a with radius $r = 3$.

complex even that he claimed that one of the possible rules for this CA ('rule 110') was 'Turing Complete'. A claim later proven by Matthew Cook around 2000 which means that it can be adapted to simulate the logic of any computer algorithm given a large enough CA and enough time.

3.1 One Dimensional Cellular Automata

According to Wolfram [41] Cellular Automata (CA) are mathematical idealizations 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 a bit 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

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

where C is a CA of width n and a_n is adjacent to a_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).

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

Due to the ring structure of the CA 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 chapter. In the synchronous approach at every time step (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

$$\forall i \in \{1, \ldots, n\} : a_i^{t+1} = \Theta(s_i^t)$$

This means that given $C^t = \{a_1^t, ..., a_n^t\}$:

$$C^{t+1} = \{\Theta(s_1^t), ..., \Theta(s_N^t)\}$$

Because $a_n \in \{0,1\}$ the number of possible states of s_i equals $2^{|s|} = 2^{2r+1}$. The transition rule Θ can be defined as the resulting state of a_i for each and every possible state of s_i. Because there can be 2^{2r+1} different possible states of s_i the transition rule Θ is defined by a binary string with 2^{2r+1} bits. The bits in the transition rule are ordered so that the state of the cell with the lowest index in s_i ('the leftmost cell in the neighborhood') corresponds to the most significant bit in the index of the bit in the transition rule.

Because the transition rule Θ is 2^{2r+1} bits there are $2^{2^{2r+1}}$ different transition rules for a one dimensional CA. For a CA with $r = 3$ this will already be $2^{2^7} \approx 3.4 \times 10^{28}$. That is a lot of different behaviors for a simple automaton.

3.2 Two Dimensional Cellular Automata

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 w, a height h and borders. Also the cell a now has two coordinates and the CA C looks like:

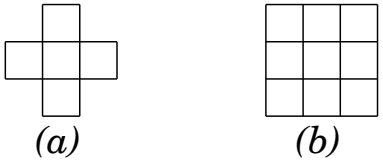

Figure 3.2: Two often used and well known two dimensional neighborhoods. *(a)* the von Neumann neighborhood and *(b)* the Moore neighborhood.

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

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 $a_{1,j}$ is connected to the rightmost cell $a_{w,j}$ in the same row and every topmost cell $a_{i,1}$ is connected to the bottommost cell $a_{i,h}$ 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 the rule is two dimensional, because there are not only neighbors left and right of a cell, but also up and down. That means that if $r = 1$, $s_{i,j}$ mights consists of 5 positions, for instance the four directly adjacent to $a_{i,j}$ plus $a_{i,j}$ itself.

$$s_{i,j} = \{a_{i,j-1}, a_{i-1,j}, a_{i,j}, a_{i+1,j}, a_{i,j+1}\}$$

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 $a_{i,j}$:

$$s_{i,j} = \{a_{i-1,j-1}, a_{i,j-1}, a_{i+1,j-1}, a_{i-1,j}, a_{i,j}, a_{i+1,j}, a_{i-1,j+1}, a_{i,j+1}, a_{i+1,j+1}\}$$

This neighborhood is called the 'Moore neighborhood' also after its inventor. Figure 3.2 shows these two neighborhoods.

3.3 Multi Dimensional Neighborhoods

A more formal definition of the neighborhood $s_{i,j}$ 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 cells with a diameter of $2r+1$ (r cells on both sides and one in the center) and that the total number of cells in s can be defined by $|s_{i,j}| = 2r^2 + 2r + 1$. This can be generalized to a d dimensional von Neumann neighborhood with:

$$s_{k_1,k_2,...,k_d} = \{a_{l_1,l_2,...,l_d} \mid \sum_{i=1}^{d} |k_i - l_i| \leq r\}$$

Note that this only holds for infinite CA or finite CA with unlinked borders, yet if a CA is using linked borders the distance between two cells needs to take that into account. If a CA has dimensions $\{e_1, e_2, ..., e_d\}$ and has linked borders then

The distance between $a_{k_1,k_2,...,k_d}$ and $a_{l_1,l_2,...,l_d}$ is:

$$\sum_{i=1}^{d} \min(|k_i - l_i|, e_i - |k_i - l_i|)$$

Therefore a d dimensional von Neumann neighborhood with linked borders in a CA with dimensions $\{e_1, e_2, ..., e_d\}$ is defined as:

$$s_{k_1,k_2,...,k_d} = \{a_{l_1,l_2,...,l_d} \mid \sum_{i=1}^{d} \min(|k_i - l_i|, e_i - |k_i - l_i|) \leq r\}$$

The Moore neighborhood of a two dimensional CA can be defined in a similar way 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 $r2+1$ (again r to both sides and one in the center) and $|s_{i,j}| = (2r+1)^2 = 4r^2 + 4r + 1$. This can be generalized to d dimensions with:

$$s_{k_1,k_2,...,k_d} = \{a_{l_1,l_2,...,l_d} \mid |k_i - l_i| \leq r \text{ for } 1 \leq i \leq d\}$$

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

$$s_{k_1,k_2,...,k_d} = \{a_{l_1,l_2,...,l_d} \mid \min(|k_i - l_i|, e_i - |k_i - l_i|) \leq r \text{ for } 1 \leq i \leq d\}$$

3.4 Neighborhood Size

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 the number of cells $S^M(d, r) = (2r + 1)^d$ being a simple hypercube, but for the multi dimensional von Neumann neighborhood $S^N(d, r)$ is less trivial to calculate. Note that a one dimensional von Neumann neighborhood equals a normal one dimensional neighborhood and has $2r + 1$ cells:

$$S^N(1, r) = 2r + 1$$

Then note that a two dimensional von Neumann neighborhood can be defined as a set of $r2 + 1$ one dimensional von Neumann neighborhoods with sizes $\{1, 3, 5, ..., 2r - 1, 2r + 1, 2r - 1, ..., 5, 3, 1\}$, basically forming a diamond shape. This can be put in a simple equation calculating two stepping pyramids and then subtracting one of the biggest bases, fitting these pyramids together then gives a diamond shape. This gives:

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

The three dimensional von Neumann neighborhood is a little bit harder to visualize but can be defined as $2r+1$ slices, each a two dimensional von Neumann neighborhoods with sizes:

$$\{S^N(2,0), S^N(2,1), ..., S^N(2,r-1), S^N(2,r), S^N(2,r-1), ..., S^N(2,1), S^N(2,0)\}$$

Putting that in a summation defines:

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

Note how a pattern has emerged in which a n dimensional von Neumann neighborhood can be defined by $2r+1$ neighborhoods that have $n-1$ dimensions. This can be done for any number of dimensions, creating a generic

recursive definition of a multi dimensional von Neumann neighborhood. For the neighborhood size this means that:

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

Using this equation the neighborhood sizes for the four, five and six dimensional neighborhoods can be calculated using some tedious calculus that is partially skipped here, but can be found in Appendix A. These calculations reveal:

$$\begin{aligned} S^N(4,r) &= 2\Big[\sum_{i=0}^{r} S^N(3,i)\Big] - S^N(3,r) \\ &\cdots \\ &= \frac{2}{3}r^4 + \frac{4}{3}r^3 + \frac{10}{3}r^2 + \frac{8}{3}r + 1 \end{aligned}$$

$$\begin{aligned} S^N(5,r) &= 2\Big[\sum_{i=0}^{r} S^N(4,i)\Big] - S^N(4,r) \\ &\cdots \\ &= \frac{4}{15}r^5 + \frac{2}{3}r^4 + \frac{8}{3}r^3 + \frac{10}{3}r^2 + \frac{46}{15}r + 1 \end{aligned}$$

$$\begin{aligned} S^N(6,r) &= 2\Big[\sum_{i=0}^{r} S^N(5,i)\Big] - S^N(5,r) \\ &\cdots \\ &= \frac{4}{45}r^6 + \frac{4}{15}r^5 + \frac{14}{9}r^4 + \frac{8}{3}r^3 + \frac{196}{45}r^2 + \frac{138}{45}r + 1 \end{aligned}$$

Putting the resulting sizes in a table generates Table 3.1. Note how S^M is growing a lot faster than S^N. Looking at the equations this can be explained by defining the order of magnitude of both the von Neumann and Moore neighborhoods. The von Neumann neighborhood grows each dimension by adding an additional exponent to the equation making $S^N(d,r) = O(r^d)$. The Moore neighborhood is also clearly exponential, but the base of the exponent is a lot bigger. Making $S^M(d,r) = O((2r+1)^d)$ which grows a lot faster than the von Neumann neighborhood.

Also note how the von Neumann neighborhood seems to be symmetrical. Note how Table 3.1 shows that $S^N(a,b)$ seems to be identical to $S^N(b,a)$.

	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.

This follows directly from the recursive definition of S^N as follows:

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

In the case where $r = 1$ each dimension only has two cells directly adjacent to the center cell, so $S^N(d,1) = 2r+1$. This is symmetrical to the one dimensional neighborhood where $S^N(1,r) = 2r+1$. Given the above symmetry this implies that the row $S^N(d,2)$ also needs to be symmetrical to the column $S^N(2,r)$. The same holds for all the other rows and columns so that:

$$S^N(a,b) = S^N(b,a)$$

Chapter 4

Inverse Design of Cellular Automata

Cellular Automata are used in many fields to generate a global behavior with local rules. Finding the rules that display a desired behavior can be a hard task especially in real world problems. This paper proposes a generic approach to generate these transition rules for Cellular Automata using a Genetic Algorithm, thus giving a way to evolve global behavior with local rules, thereby mimicking nature. Five different problems are solved using different topologies of cellular automata and different Genetic Algorithm parameter settings to show robustness, flexibility and potential.

4.1 Introduction

Cellular Automata have been used as the engine for simulations in fields ranging from biology, physics and mathematics all the way to real world application including airflow simulations, weather modeling and volcanic flow predictions. In most cases the reason for choosing a CA for the simulations lies in the power of defining simple local rules which exhibit complex global behavior. Defining the behavior of flowing magma for instance is not a trivial task, but it becomes a lot easier when the global problem is transformed into a local problem by directing the speed and direction of a single particle of magma based on its surrounding neighbor particles (as reported by Barca

D. et al. [9]). The resulting sum of interactions between all the particles of magma then represents a surprisingly accurate model of the magma flow as a whole.

Designing a local rule that has the right global behavior for a certain simulation is not always easy. There is a danger to oversimplify the interactions that are needed inside a CA which could result in a biased global behavior resulting in flawed conclusions when using the resulting model. That is why CA are mostly used in problems that can be easily abstracted to have a simple well understood local rule and a complex global behavior. Using CA in other problem areas where the local interactions are not as well understood is often impossible. This chapter will outline an approach to utilize the power of CA in problem areas where a local rule is unknown or non trivial, but the wanted global behavior is well understood by utilizing the power of evolution.

In nature this kind of global behavior through local rules is all around us. Think about the collective drive of an ant colony for instance, or a bee hive, a wolf pack, even the synchronized hatching of millions of turtles can be contributed to local rules within the biological clock of an unborn turtle with a great impact on the global behavior of a species. It is generally excepted that these behavioral traits in animals are somehow encoded in the genes of the individuals and are therefore subject to the evolutionary processes that also form limbs, eyes and feathers. Many of these traits have been studied from the point of 'benefit to a species' and most have a well defined path of evolution through time.

The need for species to develop a language for instance can be contributed to the benefit this gives in collectively undertaking a task such as alerting others to valuable resources, attacking a larger animal or defending against a common threat. Ants use pheromones to accomplish these tasks, bees aerial patterns and dances, while wolves have a much more complex system of howls and smells. Even complex hierarchical structures are used to assign different tasks within a species to more efficiently survive and reproduce. Are ants, bees and wolves conscious of this far reaching and complex global behavior of their own species? Probably not. But more importantly: they don't have to be.

Through biological research into these intricate social interactions between animals, complex local rules have been found that seem to account for most (if not all) of the global behavior of a colony, hive and pack. Some of these local rules are even more complex than the global behaviors they generate, yet they all seem to have evolved from the need of having these global benefits as a species. It seems nature has been successful in reverse engineering

complex sets of local rules to accomplish a set global goal.

This chapter will discuss an approach that tries to mimic nature by evolving local rules of simple Cellular Automata to reverse engineer local rules that exhibit a desired global behavior. Not only does this show that CA can be utilized in areas where a local rule is unknown yet suspected to work or needed, but the approach also gives new insights into the evolution of interaction. This chapter will introduce a range of experiments supporting both areas of interest.

The chapter begins with describing a well known interaction problem in the field of Cellular Automata called the 'Majority Problem'. A brief overview of previous work on this problem will be discussed including some work done to inverse design this problem using a Genetic Algorithm (GA). It will describe how these experiments were reproduced and improved upon by examining the parameter space of the GA. Next the chapter will examine the interaction resulting from these experiments, introduce distance metrics on a CA and show how the same experiment was used to examine different topologies with vastly different resulting interactions. These results inspired the use of multi dimensional CA with again surprising results.

Then, in an effort to more clearly visualize the evolution of this global interaction, three different new problems are defined. The 'AND' and 'XOR' problems evolve AND and XOR logic on a small 2D CA structure, while the 'Checkerboard Problem' defines a synchronization problem that implements a checkerboard pattern that can be scaled up through different dimensionality and sizes. Both these problems very clearly show interaction between cells and propagation of information evolved from nothing more than the need to solve the problem. This research suggests that this process works very similar to the process in nature and is a very minimalistic yet powerful way to study the emergence and evolution of interaction.

At the end of the chapter a fifth experiment is described in which the limits of inverse designing CA are explored by evolving transition rules that generate specific patterns in different sized CA. Multiple bitmaps are evolved from a 'single seed' initial state showing that the approach is flexible and can handle diverse tasks, while at the same time highlighting some limitations and insight into which tasks are easy and which are hard for a CA.

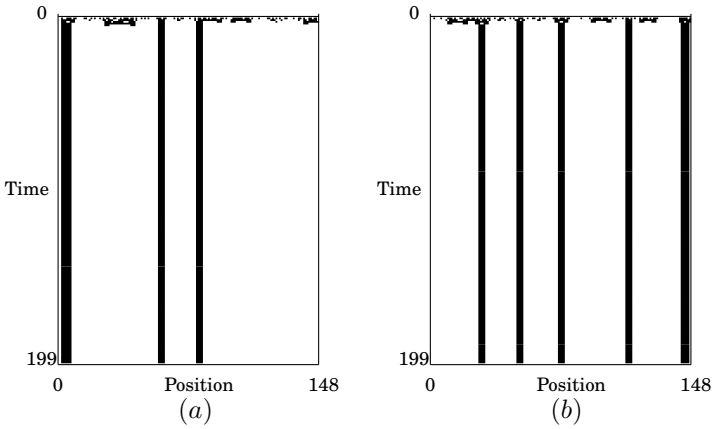

Figure 4.1: These are examples of majority problem classification by the 'Majority Rule'. The pictures show how the rule gets stuck on 'thick lines' in the time plot. Time t proceeds from top to bottom and every row corresponds to C^t. Note that there is not much change going on after $t = 5$.

4.2 Majority Problem

One of the best known global problems that is (partly) solvable with local rules is the Majority Problem. The Majority Problem can be defined as follows:

Given a set $A = \{a_1, ..., a_n\}$ *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 at first glace does not look like 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 has to be solved within the framework of a CA it becomes a lot more difficult. This is because the transition rule in a CA does not let a position 'look' past its neighborhood. That means that each cell only 'knows' what is going on in it's own neighborhood. The only way that a CA is able to solve the Majority Problem is if all the cells work together using some form of communication.

The Majority Problem in this thesis runs on a one dimensional synchronous CA and, as used by other authors [29, 30], has exactly 149 cells. Every cell in the CA can only have two states (0 or 1) and each cell has exactly the same transition rule. The CA is iterated by applying this rule on every cell

at the same time (synchronous) over and over again until the state of the CA (denoted with C^t) does not change anymore or the maximum number of iteration has been reached. The maximum number of iterations is set to 320, as used by the other authors [29, 30].

A transition rule is applied to a cell by determining the current state of the neighborhood of this cell at time t, using the rule to calculate the resulting state the cell will need to become (0 or 1) and then changing the cell to that state at time $t + 1$. The neighborhood of a cell a_i in the Majority Problem is usually defined as the three cells to the left of a_i ($a_{i-3}, a_{i-2}, a_{i-1}$), plus ($a_i$) itself and the three cells to the right of a_i ($a_{i+1}, a_{i+2}, a_{i+3}$). This is called the one dimensional neighborhood with radius $r = 3$. The CA used for this problem is usually 'linked', meaning that the first cell of the CA a_1 is linked to the last cell in the CA a_n, making the neighborhood wrap around this link as well.

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

Find a transition rule that, given an initial state of a CA where N is 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.

Evaluating the quality of a transition rule for this problem is done by iterating M randomly generated initial states, running the CA as described above and then calculating the relative number of correct classifications resulting from that run. The fitness of a transition rule is denoted with $F_{N,M}$ where N is the width of the CA and M the previously mention number of randomly generated initial states. The quality of the transition rule is then defined as the average number of correctly classified initial states resulting in an 'all zero' state if the initial state had more 0's than 1's and resulting in an 'all ones' state otherwise.

There are different distributions in the number of ones that can be used in the initial states. The default is a binomial distribution (its fitness denoted with $F^B_{N,M}$) where every cell in the CA has an equal (50%) chance of being initiated with either a 1 or a 0 for every initial state. The alternative to this distribution is a uniform distribution (its fitness denoted with $F^U_{N,M}$) in which all fractions of 0's versus 1's are equally represented in the distribution of initial states. The uniform distribution has some benefits towards evolution that will be explain later in this chapter, while the binomial distribution is used to rate transition rules historically and throughout this chapter.

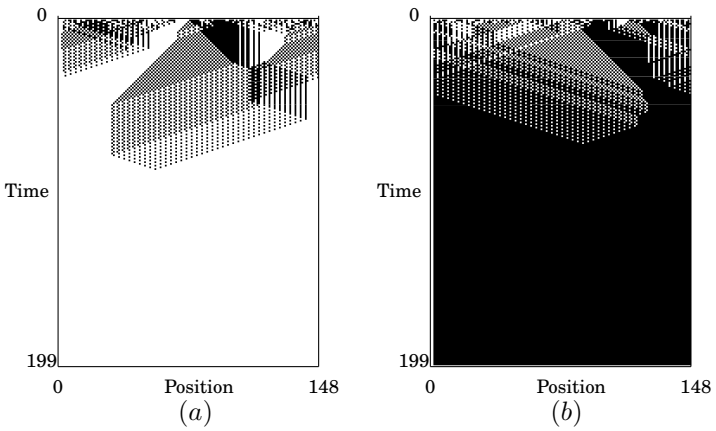

Figure 4.2: These are examples of Majority Problem classification by the rule found by David, Forrest and Koza. [1]. Both are correct classifications (a) with 74 ones in the initial state, (b) with 75. Note how different 'particles' of information travel in straight lines and meet to form new particles.

At first glace, the most obvious transition rule is what has been called the 'Majority Rule'. This is the rule where the output value is 1 if the number of ones in the neighborhood is more than the number of zeros, and a zero otherwise. This is basically the definition of the Majority Problem solved literally in the local neighborhood. Surprisingly (or not) this does not at all solve the problem for the global CA (as is shown in Figure 4.1). The majority rule gets stuck in states where on the border of a thick line in the time plot the cells can't 'agree' on the global answer. When for instance 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 probably only sees ones on its right and therefore decides to stay one. This way the information fails to propagate through the CA and classification fails.

Researchers in the field of cellular automata have published many different rules to solve this problem, one such rule is the GKL rule so named after its inventors Gacs, Kurdyumov and Levin [16]. 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 L. Davis found a better one in 1995 which did 81.8%. In the same year R. 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]. Figure 4.2 shows a plot of this rule in action.

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. It is already a big accomplishment for a CA to get 70% of all random initial states correct, for this shows there is some kind of communication going on; some kind of emerging interaction.

4.3 Inverse Design of the Majority Problem

First it is important to point out that this research is not aimed at finding better performing transition rules on the Majority Problem. Even though such a feat would be of considerable importance to the field of Cellular Automata and Evolutionary Algorithms in general, it was never the aim of this research. There is so much already known about the intricacies of the Majority Problem that asking a Genetic Algorithm to come up with that knowledge on its own is a tall order. It is even in line with this research to admit that using a Genetic Algorithm for the inverse design of the Majority Problem is not the best and most efficient approach to solve that problem. For that goal making use of Genetic Programming like David, Forrest and Koza successfully employed in 1996 [1] seems like a much more likely candidate.

However, in all successful transition rules discussed in Section 4.2 the generation of the rule was either entirely or in part guided by human knowledge of the problem. The GLK rule was designed by its authors [16], as was the Davis' and Das' rule. The rule by David, Forrest and Koza [1] was generated by a Genetic Programming approach, but the input to that algorithm consisted of parts of already known good approaches which in turn were generated by humans. The research in this chapter is not aimed at finding better rules, but instead wants to understand more about the process of inverse engineering a solution to a complex problem requiring interaction. To introduce knowledge about the problem would only make the origin of such emerging interaction harder to verify.

The research into inverse design of transition rules in a Cellular Automaton using a Genetic Algorithm was inspired by research conducted by M. Mitchell, J. P. Crutchfield and P. T. Hraber. In [29, 30] they show that using a simple GA to evolve transition rules for the majority problem (explained in Section 4.2) can already give surprisingly good results without adding any problem specific knowledge to the algorithm. About half of the rules that were found in this research performed better than the most trivial rule and

about 7 rules out of 300 rules that were found seemed to use some primitive form of communication that worked for more than 70% of the classifications. This is not better than rules that are made by hand, but it does show how a GA can evolve global behavior based on local rules from scratch using only the power of evolution.

This research uses their findings as a starting point to more thoroughly examine how interaction can be evolved using a Genetic Algorithm. Without ever telling the CA explicitly how to communicate we try to solve problems that can only be tackled using some form of communication. The resulting behavior of the CA's show very clear interaction protocols that have emerged solely from the use of evolution.

4.4 The Genetic Algorithm

Because CA define their behavior in the form of a binary transition rule, they are well suited to be evolved with a genetic algorithm. As introduced above, M. Mitchell, J. P. Crutchfield and P. T. Hraber have shown [29, 30] that using a simple GA to evolve transition rules for the majority problem (explained in Section 4.2) can evolve interaction in a CA. To be able to compare our results we will start by using an identical setup to this research and from there test the robustness of the approach by running a set of experiments with different settings.

The GA in this chapter uses tournament selection as defined in Section 2.3 in short and in [6] more extensively. This selection involves running 'tournaments' on the population in order to determine the next generation. Every tournament q individuals are selected at random from generation t and the one with the highest fitness is then copied to generation $t + 1$. For a population of λ individuals this process is repeated λ times and added to generation $t + 1$. In the initial experiments $q = 10$.

After selection is complete, recombination is applied. Recombination is done by using single-point crossover on a subset of the population with a crossover-rate denoted with c. In most experiments $c = 0.9$. Then the resulting individuals are mutated using probabilistic bit flip mutation. This works by flipping every bit in the individual with a probability p_m. If not mentioned otherwise $p_m = \frac{2}{l}$ where l is the number of bits in the individual.

All the individuals in the pool are initialized at random with a uniform distribution over the number of ones in an individual. This means that the number of individuals with a certain number of ones will be roughly equal

to the number of individuals with a different number. This prevents the algorithm from specializing in a particular area of the search space at the beginning of the algorithm. The evolution ends after D generations and the best individual of the last generation is considered to be the answer. For most experiments $D = 100$.

The GA is expected to behave differently with different settings of q, c, p_m and D. This research tries to find one single approach that works in different experiments with only minimal changes. The Majority Problem was used to determine some good settings, which were then used in the other experiments.

4.5 1D Experiment

The algorithm was run for 900 runs. Note that this is three times as many runs as was calculated in the original experiment by M. Mitchell, J. P. Crutchfield and P. T. Hraber [29]. Afterwards $F^U_{149,10^3}$ 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 the fitness of a rule during evolution is calculated with 100 initial states and is therefore only a rough estimate.

In Figure 4.5 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 $F^U_{149,10^3} \approx 0.5$ 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 $F^U_{149,10^3 \approx 0.63}$ and corresponds to the 'block expanding' algorithms (as shown in Figure 4.3 and the 'particle based' rules are situated where $F^U_{149,10^3} > 0.71$ (as shown in Figure 4.4).

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 [29] 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 $F^U_{149,10^3} > 0.7$ 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 [30], but it is not unthinkable that other rules have a totally different approach.

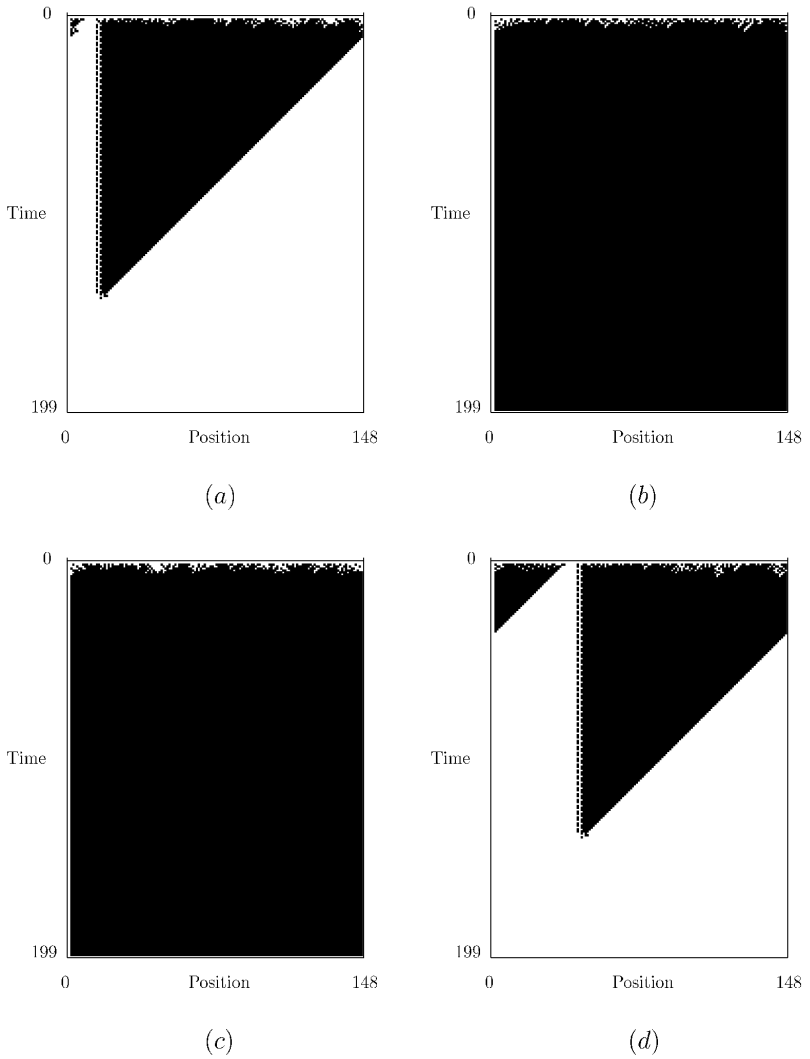

Figure 4.3: These are examples of majority problem classification by a typical block expanding rule with $N = 149$ and $F^{U}_{149,10^4} \approx 0.65$. Both (a) and (b) are correct classifications (a) with 74 ones in the initial state, (b) with 75. Note that in (a) there emerges a block of zeros right 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.

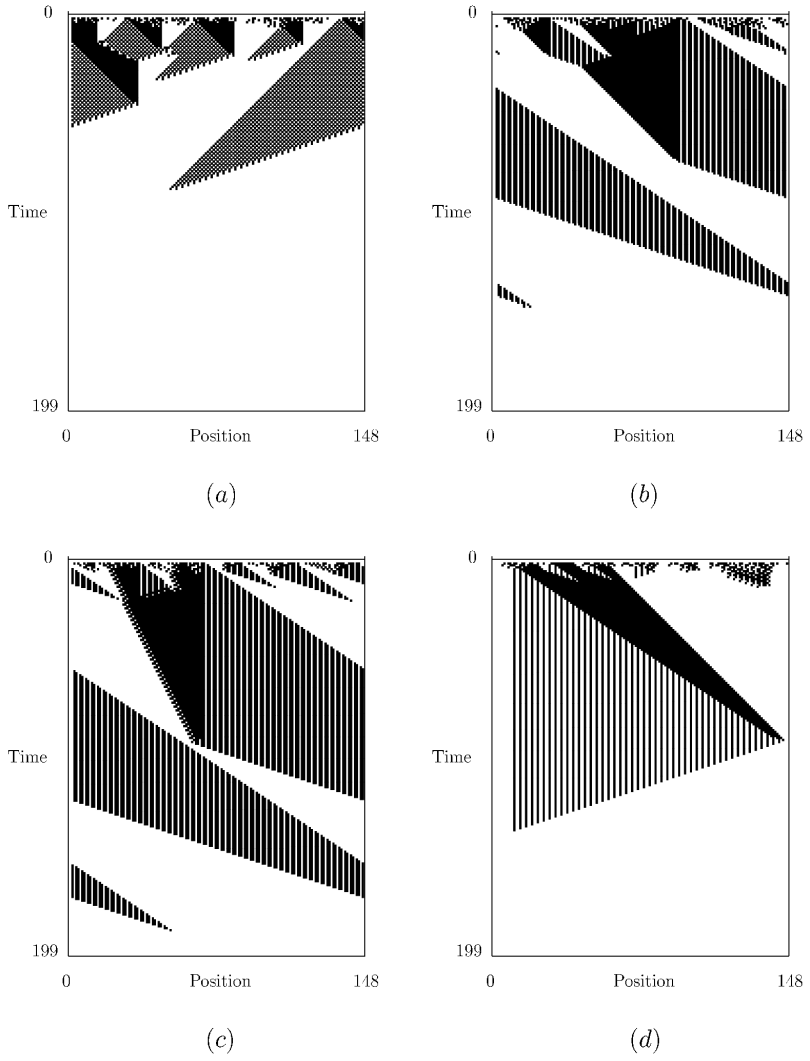

Figure 4.4: This figure displays four correct classification of the majority problem by four different particle based rules. (a) and (c) both have $F^U_{149,10^4} \approx 0.76$, (b) has $F^U_{149,10^4} \approx 0.75$ and (d) has $F^U_{149,10^4} \approx 0.73$ with $N = 149$.

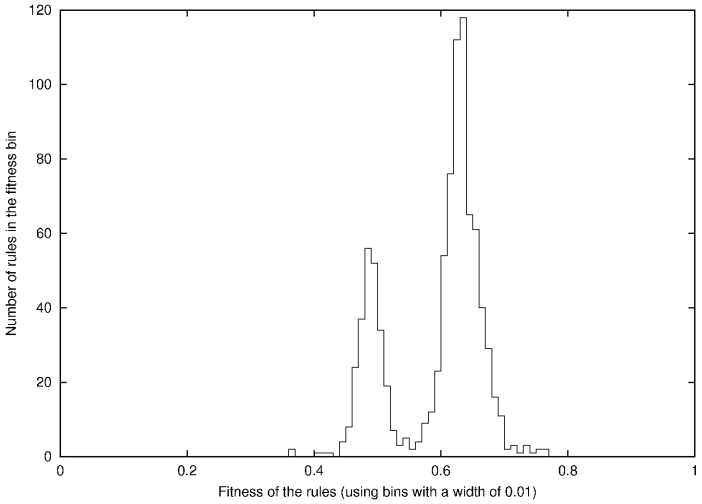

Figure 4.5: This figure displays the frequency with which rules have a certain fitness value in the one dimensional experiment. The fitness bins are 0.01 in width and 900 rules are displayed.

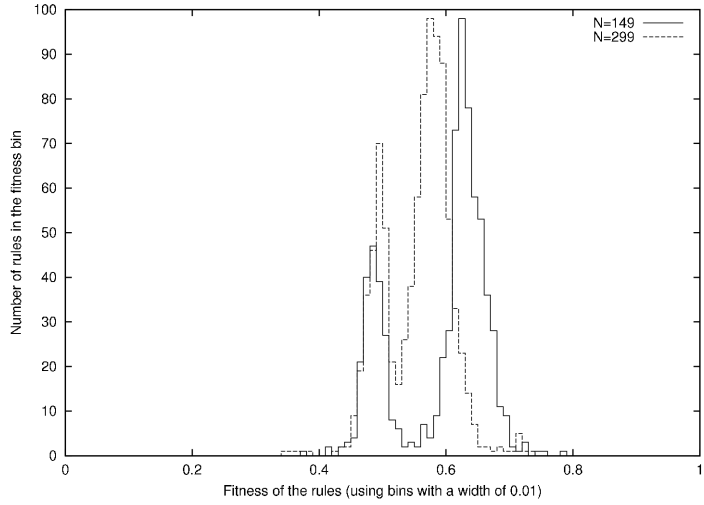

Figure 4.6: This figure shows the effect of increasing the width of the CA (N) from 149 to 299 in the one dimensional experiment. The fitness bins are 0.01 in width and 900 rules are displayed.

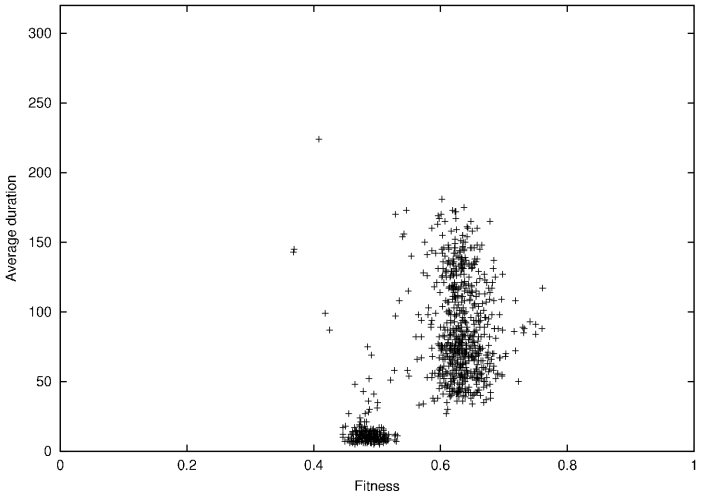

Figure 4.7: This figure shows the average duration ($D_{N,M}$) of runs for the one dimensional algorithm. For this algorithm $N = 149$ and $M = 10^3$.

$F_{n,10^4}$ was calculated for different n of the CA. As stated in [29] '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 4.6 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 by M. Mitchell, J. P. Crutchfield and P. T. Hraber [29, 30].

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 4.7 shows the average duration of these rules against their fitness ($F^U_{149,10^3}$).

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 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.

4.6 Different Parameters in GA

The Genetic Algorithm defined in Section 4.4 has been proven to work in inversely designing CA for the Majority Problem. This Section will describe experiments that examine the parameters of this algorithm to find improvements in terms of efficiency, performance and robustness. The knowledge gathered in this process will help us understand how to use this algorithm for other experiments.

The algorithm as proposed in Section 4.4 was used to evolve transition rules for the Majority Problem. Initial parameters for the GA were the same as introduced in Section 4.4 namely: population size $\lambda = 100$, tournament size $q = 10$, crossover rate $c = 0.9$, mutation rate $p_m = \frac{2}{l} = \frac{2}{128} = 0.015625$ and maximum GA generations $D = 100$.

Preliminary experiments as well as experiments by Packard et al. [28] and Mitchell et al. [29, 30] suggest that it is very difficult to evolve good transition rules with a GA while using a binomial distribution over the number of ones in the initial states. The solution for this is using a uniform distribution while evolving the rules. This distribution generates more 'easy' initial states with a large difference between the number of ones and the number of zeros, thus making it easier to train the desired behavior. The fitness using initial states with this uniform distribution over the number of ones is denoted with $F^U_{N,M}$.

This distribution has a drawback though. Because rules are selected using a different fitness function than the one used to test them in the end, it seems possible that the rules will specialize in a behavior that would seem pretty good for the uniform distribution, but very bad for the binomial distribution. To counter this effect a "gliding distribution" is introduced. This distribution is different for every generation of the genetic algorithm. It "glides" gently from a uniform distribution in generation 0 to a binomial distribution in generation D. This is achieved by generating $\lfloor M\frac{g}{D} \rfloor$ initial states with a binomial distribution and $\lceil M(1-\frac{g}{D}) \rceil$ initial states with a uniform distribution, where g is the current generation. This distribution has the benefits of the uniform distribution in the beginning of the algorithm

$F^B_{149,10^4}$	q					
	2	3	5	10	20	50
0.0 - 0.5	23	0	1	0	2	3
0.5 - 0.55	37	0	0	0	2	2
0.55 - 0.6	14	6	4	1	2	4
0.6 - 0.65	25	79	69	48	52	47
0.65 - 0.7	1	16	42	50	39	40
0.7 - 0.75	0	0	0	1	3	4
0.75 - 0.8	0	0	0	0	0	0
0.8 - 1.0	0	0	0	0	0	0

Table 4.1: This table shows the fitness distribution using different values for the tournament size q. Other settings are the same as the initial values proposed in Section 4.4.

without the drawbacks at the end. This distribution is denoted with $F^G_{N,M,g}$. Note that $F^G_{N,M,0} = F^U_{N,M}$ and $F^G_{N,M,D} = F^B_{N,M}$.

Different parameter settings were tested on the GA. Experiments with different values for the mutation rate m didn't show any real improvement and it was concluded that $m = \frac{2}{128} = 0.015625$ was best. Also changing the number of generations D did not seem to yield improvements immediately although in theory a larger D should increase the chance of good results. Because of the time restrictions and historical compatibility with [29, 30] we decided to use $D = 100$.

Exploring different tournament sizes q values however seemed to give very different results. Experiments with $q = \{2, 3, 5, 10, 20, 50\}$ were conducted. Each setting was run a 100 times. Results are shown in Table 4.1. Note that these results imply that a high selection pressure is needed to gain good results. Settings $q = \{2, 3, 5\}$ don't seem to be very good in generating rules that exceed the 0.7 barrier, $q = 10$ is better, but $q = \{20, 50\}$ generate both very good results. Because $q = 50$ seems to produce more 'bad' rules with $F < 0.6$, so it was decided to use $q = 20$ in the future.

Different crossover rates also seemed to change the results. Using the new tournament size $q = 20$ four different values for c were tried: 0.6, 0.8, 0.9 and 0.95. Table 4.2 shows the results. Note that the best results are achieved using $c = 0.6$, but the difference is minimal. This together with the findings for the different mutation rates m implies that the algorithm is robust under different mutation settings and might be usable for different problems without changing these settings.

$F^B_{149,10^4}$	c			
	0.6	0.8	0.9	0.95
0.0 - 0.5	0	0	2	2
0.5 - 0.55	2	1	2	0
0.55 - 0.6	1	1	2	2
0.6 - 0.65	54	52	52	56
0.65 - 0.7	37	42	39	36
0.7 - 0.75	3	4	3	3
0.75 - 0.8	3	0	0	1
0.8 - 1.0	0	0	0	0

Table 4.2: This table shows the fitness distribution using different values for the crossover rates c, $q = 20$ and other settings are the same as the initial values proposed in Section 4.4.

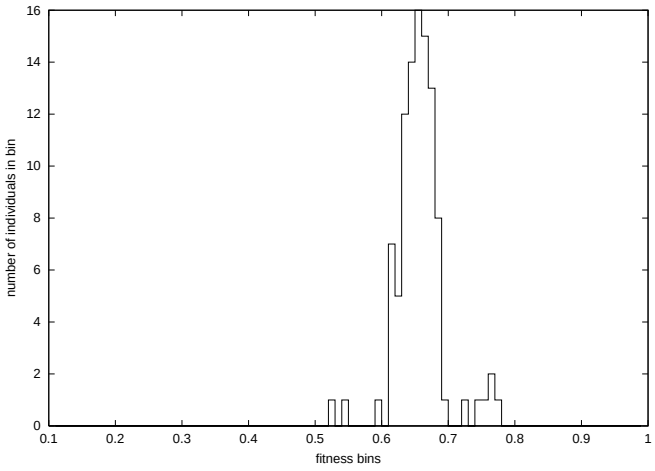

Figure 4.8: This figure displays the number of transition rules found for different fitness values. The CA used is one dimensional and has 149 cells, a radius $r = 3$, a maximum duration $I = 320$ and a "gliding distribution" was used for the initial states. Setting for the GA: $q = 20$, $c = 0.6$, $m = 2/128$, $D = 100$.

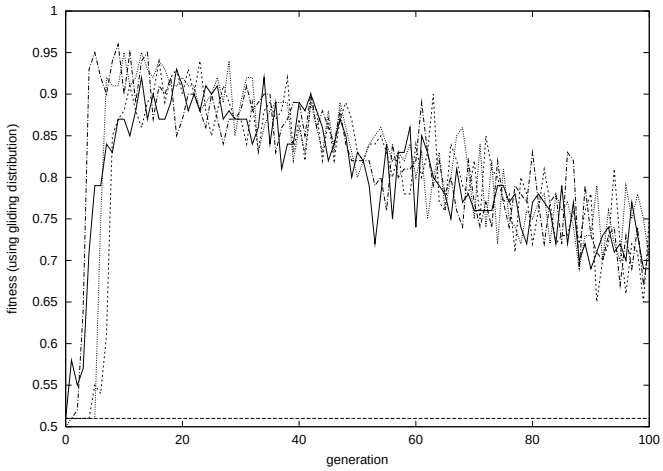

Figure 4.9: This figure shows $F^G_{149,100,g}$ for five runs of the algorithm. Settings are all as proposed in Section 4.4 except for q which is set to 20. Note how the gliding distribution suppresses the fitness and increases the noise.

All these experiments seem to suggest that although there is some performance gains from changing a couple of parameters to the algorithm, this will not result in finding new results with higher fitness values. What it does do however is show increases in the reliability with which good results are generated by the GA. For the benefit of the experiments in the remainder of the chapter the following conclusions were drawn:

The 'gliding distribution' in the initial states of the CA did not show any real improvement as apposed to the 'uniform distribution'. For all other experiments the standard 'uniform distribution' will be used.

Changing the tournament size q of the tournament selection had a big impact on the reliability of generating good results. The results where $q = 20$ seem to outperform all the others, so that is the value used in the rest of the experiments.

Changing the crossover rate had some impact as well in that the runs where $c = 0.6$ seem to outperform all the other runs. Therefore $c = 0.6$ for the remaining experiments in this chapter.

4.7 Changing the Topology

A normal one dimensional CA (as described in Chapter 3) has a very simple defined neighborhood: "all the cells within a certain radius r of the center cell". Although this seems to be the most logical neighborhood, CA can have many different shapes each with a different behavior.

Every iteration step in a synchronous CA the state of every cell is updated using the information in the cells of the neighborhood of that cell. That means the way the information moves ('travels') through the CA is defined by the shape of the neighborhood. Note that because the standard one dimensional neighborhood is symmetrical, if information travels from cell a_i to cell a_j it could also travel from cell a_j to cell a_i. The number of iterations it takes for information to travel from cell a_i to cell a_j can be called the "distance between a_i and a_j". If the shape of the neighborhood is different, the distance between cells in the CA will be changed too. Not only will this change the behavior of the CA, but it might also change the possible final states of the CA. That means that two neighborhood shapes might perform different on the same problem.

Intuitively, to solve a global problem with local rules information from every cell in the CA has to travel to every other cell in the neighborhood to be combined in a way that suits the problem. Because combining information results in new information and the space in a CA is fixed, there will be information loss. Therefore combining the information as fast as possible seems to be a good way to counter this information loss and solve the problem in the best possible way. This then would mean that the distance between two cells in the CA needs to be minimized so that the information can be combined faster.

To examine the rate at which information is combined in a CA two metrics are defined. The 'maximum distance' between two cells and the 'average distance' in a CA. Because every cell has the same neighborhood shape every cell has the same maximum distance to a cell and the same average distance to all the cells in the CA. Note that these metrics are very dependent on the shape of the neighborhood and the size and topology of the CA.

Given that $s(a_i) = \{a_{j_1}, a_{j_2}, ..., a_{j_k}\}$ is the set of all cells that are 'neighbors' of a_i, a 'path' can be defined an array of 'walks' from one neighbor to another:

$$P(a_i, a_j, w) = \left\{ \{p_1, ..., p_w\} \middle| \; p_1 = a_i,\; p_w = a_j,\; p_{k+1} \in s(p_k),\; \forall k \in \{1, ..., w-1\} \right\}$$

Name	Physical distance							Max. distance	Avg. distance
Normal	-3	-2	-1	0	1	2	3	25	12.79
Exponents of 2	-4	-2	-1	0	1	2	4	19	9.97
Exponents of 3	-9	-3	-1	0	1	3	9	10	5.48
Exponents of 5	-25	-5	-1	0	1	5	25	6	3.84

Table 4.3: This table shows the physical layout of the four different neighborhoods used in this experiment. The maximum and average distance are executed on a CA with 149 cells.

Note that in this definition w is the length of all paths in the set of paths $P(a_1, a_j, w)$. With that we can now define the 'distance' between two cells $d(a_i, a_j)$ as the length of the shortest existing path between those cells:

$$d(a_i, a_j) = \min\Big\{w\Big|P(a_i, a_j, w) \neq \emptyset\Big\} - 1$$

The 'maximum distance' d_{max} in a CA can now be defined as maximum distance between any two cells in the CA. This measure gives an indication on how many iterations a CA would have to preform to propagate data from one cell to all other cells in the CA. Because the neighborhood of each cell in the CA is identical, the set of distances from each cell to the rest of the CA is also identical. This means that d_{max} can be defined as:

$$d_{max} = \max\Big\{d(a_1, a_i)\Big|1 \leq i \leq n\Big\}$$

where n is the size of the CA.

Similarly the 'average distance' d_{avg} can be defined as the average distance from one cell to all other cells in the CA.

$$d_{avg} = \frac{1}{n}\sum_{i=1}^{n} d(a_1, a_i)$$

An experiment was conducted to compare different neighborhood shapes in combination with the previously introduced Majority Problem (see Section 4.2) and measure what the impact of the different neighborhoods is on the performance. In order to test the performance of a neighborhood shape a GA was used to search for a good transition rule that solves the Majority Problem.

In this experiment four different neighborhoods were tested. The first one was the standard one dimensional neighborhood including cells 1, 2 and 3

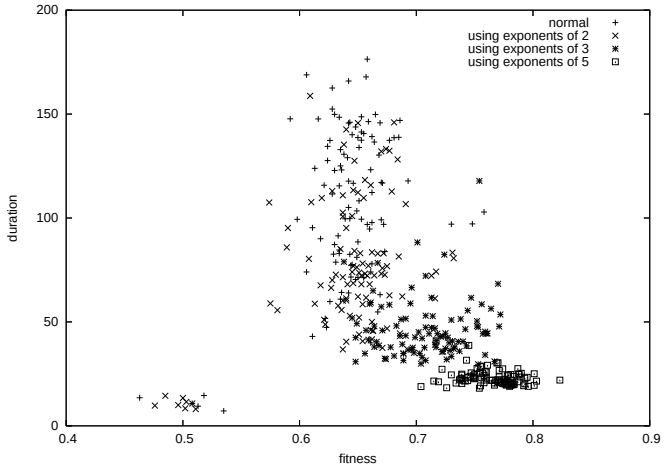

Figure 4.10: This figure shows all the rules found with the four different neighborhoods. Note how the fitness goes up and the duration goes down if the neighborhood is wider. The fitness is calculated using $F_{149,10^3}$.

on both sides of the center cell. The other three were chosen to minimize the distance between cells in the neighborhood and include the cells with a physical distance of an integer to the power of 0, 1 and 2. The three integers chosen for this are 2, 3 and 5 (all prime). Table 4.3 gives an overview of the layout of the four neighborhoods.

For every neighborhood 100 runs are calculated with the GA as described in Section 4.4. Unsurprisingly the results of the normal neighborhood matched that of results in Section 4.5 and are only a little bit better than the results in [29, 30]. The rest of the results are a lot less trivial though. The neighborhood with "exponent of 2" is performing slightly better than the normal neighborhood, whereas "exponents of 3" performs a lot better with more than half of the rules topping 0.7 and about 10% over 0.75. And "exponents of 5" is even better with all the rules found being above 0.7 and some even topping 0.8 with the best at 0.813 coming very close to the best rule found on the Majority Problem. Furthermore the average number of iterations that a rule needs to classify an initial state gets smaller the higher the exponent is.

Figure 4.10 shows the best rule from all the runs. Note how the fitness goes up and the duration goes down if the neighborhood is wider. These results support the claim that the shape of the neighborhood is very important for the performance of the CA and that decreasing the distance between cells in

the CA increases the performance and decreases the duration. Both these characteristics seem to be a direct result from decreasing the distance between cells in the CA.

4.8 Multi Dimensional CA

Inspired by the finding that the topology of the neighborhood in a CA has a great impact on the performance of that CA in the Majority Problem, an experiment was conducted to test the effect of dimensionality on the results. Both two dimensional and three dimensional CA's were used to evolve transition rules on the Majority Problem. This Section first gives an introduction into the differences between one dimensional and multi dimensional CA and continues with some results on experiments for inversely designing CA for these topologies.

This chapter will not give an extensive introduction to multidimensional CA, for that please read Chapter 3. The most simple two dimensional CA can be viewed as a grid of positions $a(i, j)$ ($i \in \{1, ..., w\}, j \in \{1, ..., h\}$) instead of a row in the one dimensional case. The borders of this CA are connected in such a way that every first cell in a row $a(1, j)$ is connected to the last cell $a(w, j)$ and every first cell in a column $a(i, 1)$ is connected the last cell in that column $a(i, h)$. This topology is also known as a 'torus' or 'donut' shape.

There are two neighborhoods that are often used in this two dimensional space, namely the von Neumann neighborhood and the Moore neighborhood (as introduced in Section 3, both named after their inventors.

These neighborhoods can be extended to have a larger radius and more dimensions if defined in terms of distance: Every cell in a neighborhood has a path to the center cell that is equal or less than r steps to 'adjacent' cells. In a CA with d dimensions and their sizes $e_1, e_2, ..., e_d$, cells $a(i_1, i_2, ..., i_d)$ and $b(j_1, j_2, ..., j_d)$ are 'adjacent' in a von Neumann neighborhood if

$$\sum_{k=1}^{d} \min(|i_k - j_k|, e_k - |i_k - j_k|) = 1$$

In a Moore neighborhood cells are 'adjacent' if

$$\min(|i_k - j_k|, e_k - |i_k - j_k|) \leq 1 \text{ for } 1 \leq k \leq d \text{ and } a \neq b$$

Note that a one dimensional von Neumann neighborhood is equal to a one dimensional Moore neighborhood.

Transition rules are defined in the same way as in the one dimensional CA where every bit in the index of the bitstring represents one input cell in the neighborhood. The cells in the neighborhood are numbered from 1 to n in an iterative way over the d dimensions. This means cells are numbered from left to right (dimension 1), then top to bottom (dimension 2), then front to back (dimension 3) and then from start to finish in the fourth dimension and so on. Note that this means that the center cell independent of the dimensionality of the CA is always numbered $\frac{n+1}{2}$.

The number of cells in these neighborhoods grows very fast if r or d is increased. Table 3.1 in Chapter 3 clearly shows that the Moore neighborhood grows a lot faster than the von Neumann neighborhood. In this chapter we will use a few different combinations and explore their differences.

Rules are defined with the same rows of bits (R) as defined in the one dimensional case. For a von Neumann neighborhood a rule can be defined with $2^5 = 32$ bits and a rule for a Moore neighborhood needs $2^9 = 512$ bits. This makes the Moore rule more powerful, for it has a bigger search space. Yet, this also means that searching in that space might take more time and finding anything might be a lot more difficult. In [24] the authors discourage the use of the Moore neighborhood, yet in Section 4.10 and Section 4.12 results clearly show successes using the Moore neighborhood, regardless of the larger search space.

In a one dimensional CA the leftmost cell is connected to the rightmost cell. In the two dimensional CA this is mimicked by linking every cell on the left edge of the CA to the cell on the right edge of the CA in the same row and link every cell at the top edge of the CA to the cell on the bottom edge in the same column.

Preliminary experiments showed that it took much more time to evolve rules for the Moore neighborhood than for the von Neumann neighborhood. The tests that were done with the Moore neighborhood also did not result in any encouraging results, this being in line with [24]. That is why the von Neumann neighborhood was chosen for this experiment. Because this neighborhood consists of five cells, the search space for CA rules is a lot smaller than in the one dimensional experiment where 7 cells were used. Instead of the $2^7 = 128$ bits in the rule, R now consists of $2^5 = 32$ bits, thus drastically decreasing the search space from 2^{128} to 2^{32} possible rules, which is now $2^{(128-32)} = 2^{96}$ times smaller!

Using a smaller search space makes that the transition rules are a lot less

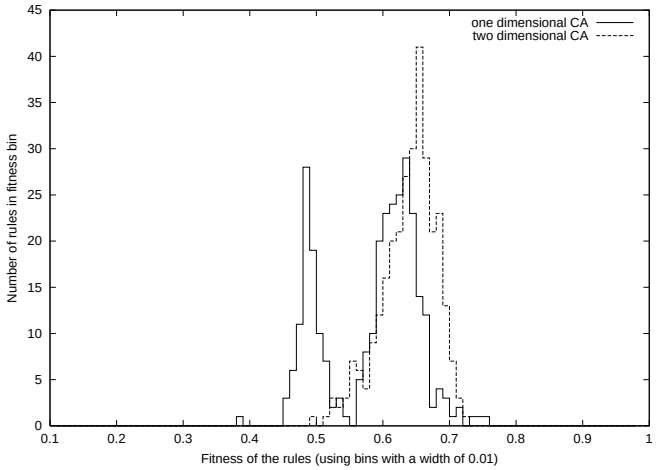

Figure 4.11: This figure displays the number of rules that have a certain fitness value in the two dimensional experiment and compares this to the one dimensional experiment. The fitness bins are 0.01 in width and for both algorithms $F_{169,10^3}$ is calculated for 300 rules.

complex. This has pros and cons. On the one hand this means that it can be expected that the global behavior of the rules in the CA is a lot simpler, meaning less pattern states and less different communication possibilities. On the other side this also means that the GA will search in a smaller search space and will probably have a higher chance of getting close to an optimum.

For this experiment we used a CA with width $= 13$ and height $= 13$. 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 before.

Using a larger CA to test the rules also makes the task harder for a CA to get to an 'all ones' or 'all zeros' state. If the CA can not get into these states in time, the classification is counted as a failure. But in order to get as close as possible to a fair comparison the two dimensional CA needs to be a square and at least the same size as the one dimensional CA, which makes 13×13 the closest option.

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 travel 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

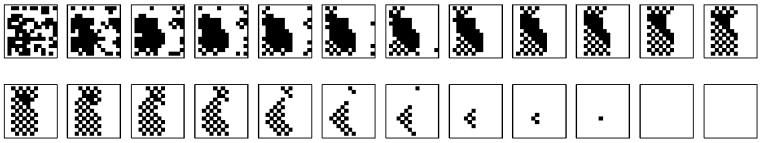

Figure 4.12: This figure shows a correct classification of the Majority Problem by a two dimensional CA with both width and height equal to 13 and $\lambda = 84/169$. The transition rule was one of the best tested in the experiment and scored $F_{169,10^3} = 0.715$.

to $13/1 = 13$ iterations compared to the $149/3 = 49.7$ for a comparable one dimensional CA. This is a good indication that the maximum number of iterations I does not 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.

This algorithm was run 300 times and each winning rule was tested by calculating $F_{N,M}$ using $F_{169,10^3}$. These results are plotted against results of our own one dimensional experiments in Figure 4.11. The striking difference between this distribution of fitness and the distribution of fitness in the one dimensional experiment is the absence of the peak around $F_{N,M} \approx 0.5$ in the two dimensional results. In the new results almost all the evolved rules have a fitness above 0.58. The average fitness is approximately 0.66 and the best rules have a fitness above 0.7. That is all very surprising taking into account that the experiment used the smaller von Neumann neighborhood and a bigger CA.

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 traveling 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.

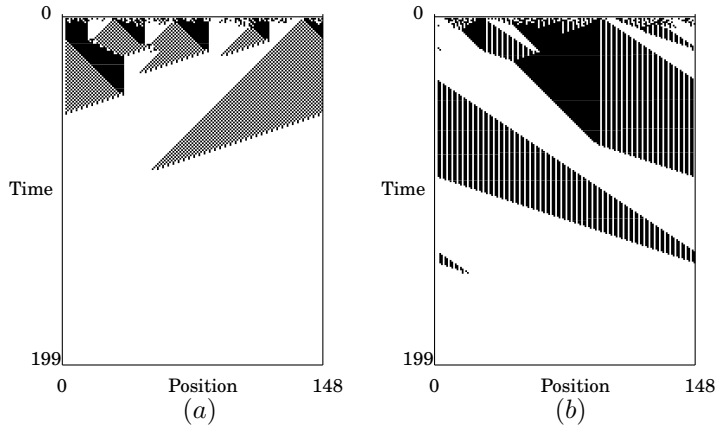

Figure 4.13: This figure shows two one dimensional CA correctly classifying the Majority Problem using transition rules evolved with the GA. Note the triangular structure in the space time plot depicting the moving and combining 'particles' in the two different transition rules.

4.9 Looking for Interaction

Up until now the only proof that interaction is evolved with the use of a Genetic Algorithm is the resulting behavior of the CA on the Majority Problem and because the Majority Problem is considered to need some form of interaction between cells, the evolution of that interaction seems implied, but not proved. This Section takes a closer look at the resulting behavior inside the CA and tries to find where this interaction takes place.

Figure 4.13 shows two runs of two different transition rules in a one dimensional CA. Both use a neighborhood with $r = 3$ and are synchronous. Although they seem to have slightly different ways to solve the problem, the general principle seems to be based on making different patterns in the 1D space and then propagating these patterns to the left and right through time at different speeds. Some patterns dissolve over time and when two patterns meet they seem to generate new patterns.

These patterns could be viewed as 'states', because when a cell is part of a certain pattern it stays part of that pattern until the edge of the pattern passes of the cell. Note that this does not mean that the cell stays the same value while being part of a pattern. Some patterns shift inside of their own area. The edges of the patterns seem to have many different shapes and move at different speeds through the CA. These speeds then govern the

timing for the different edges to meet and generate new patterns. The level of 'blackness' of a certain pattern seems to say something about the message it is trying to propagate to the rest of the CA. The 'blacker' the pattern, the more the pattern seems to 'convince' the rest of the CA that it should also become black. This approach seems very close to the approaches used in the best known rules for solving the Majority Problem [1, 16].

So where does the interaction take place in these CA? Of course every cell in the CA is interacting with every cell in its neighborhood and reacts to that interaction, but that is basically the definition of the CA. Not every interaction between a cell and its neighborhood contains the same amount of information though. The interactions inside these CA when trying to solve the Majority Problem can be divided into two categories:

- 'Static interactions': interactions between cells to maintain a pattern (or state) in the CA.
- 'Moving interactions': interactions between cells to propagate a pattern (or state) change through the CA.

Static interaction are basically there to maintain a certain state inside a cell. This can of course be very complex especially if there are many different states used by the approach, while at the same time there can only be two states in a cell. These static interactions could be viewed as 'forming a base' to use for the more complicated moving interactions that solve the problem. In a way the moving interactions are like 'messages' or 'particles' that move through the CA and spread the word to individual cells. Interaction is visible where two particles collide and form other particles. These collisions of particles define the behavior of the CA and can be viewed as the 'evolved intelligence' in these experiments.

Going from one dimensional to two dimensional CA changes the way states and interactions are used to solve the Majority Problem. Instead of only two directions there are now at least four directions information can travel through the CA. That said, the interactions between cells are surprisingly similar. Figure 4.14 shows a time plot of a two dimensional 50×50 CA correctly classifying the Majority Problem on an initial state with more 0's than 1's. The figure shows the plots of every fifth iteration. (on $t = \{1, 6, 11, ..., 71\}$) Note that the rule used in this run was evolved using much smaller CA (13×13), showing that the approach is robust for different CA sizes.

The static interactions are immediately apparent in the form of patterns generated in the shape of areas in the CA. Because the von Neumann neighborhood only has 5 cells, the number of different patterns that can be gen-

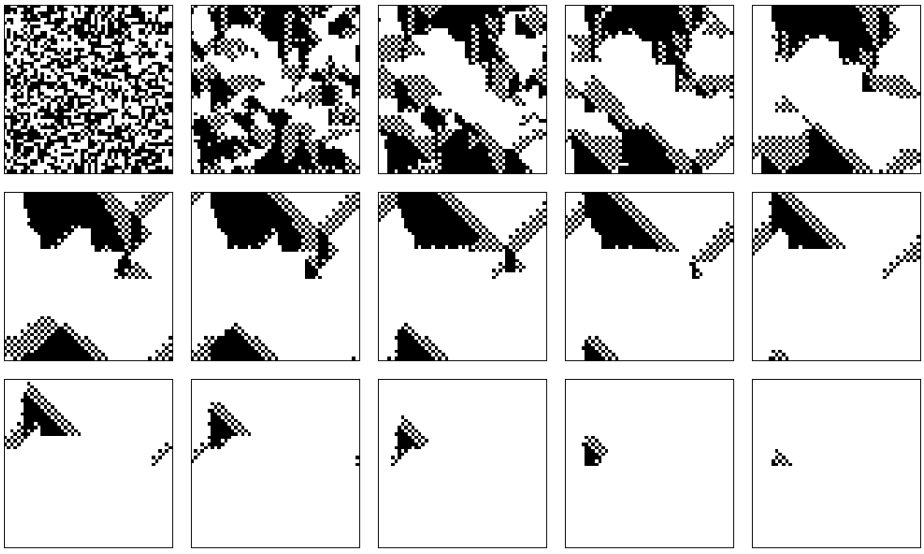

Figure 4.14: This figure shows the time plot of a 50×50 CA using the von Neumann neighborhood correctly classifying the Majority Problem with a transition rule that was evolved using a 13×13 CA showing the robustness of the approach. The plots were generated using a 5 iteration interval (plots at $t = \{1, 6, 11, ..., 76\}$). Right after the last plot in this figure the CA came in an 'all zero state'.

erated is less than in the one dimensional $r = 3$ neighborhood, but the additional directions that the information can travel seems to make up for that. The moving interactions in the two dimensional CA are no longer 'particles' in the form of a few cells, but rather boundaries of areas of a certain pattern. These boundaries seem to move and cascade into each other, sometimes creating new patterned areas in the process. Some try to expand, some collapse on top of themselves, but all seem to interact in a hard to understand, deceiving chaos solving this seemingly trivial Majority Problem.

The behavior shown in Figure 4.14 is defined by a transition rule with 32 bits, which can be considered a very simple automaton. The problem it is trying to solve also seems (maybe deceptively) simple and the workings of the CA itself are close to trivial and well understood, but the actual interactions and 'the way it all works together' is a lot harder to fully understand. The interactions are there, they are visible, they work and have emerged solely from the use of an evolutionary process, but they are so complex that it is hard to fully identify what kind of intelligence has actually emerged from the EA.

4.10 AND / XOR problem

In order to study the evolution of interaction more closely, a new experiment was conducted. Instead of trying to show the power of the evolutionary approach by defining a hard problem, this experiment was instead directed towards visualizing an evolved interaction by defining a very basic problem while at the same time making sure interaction was mandatory to solve the problem.

A genetic algorithm was used to evolve rules for two dimensional CA to make those CA behave like the binary functions 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 distinction may look trivial, but it is important in order to force the CA to combine the two values and thereby communicate and that is exactly what this experiment aims to do.

The AND Problem

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:

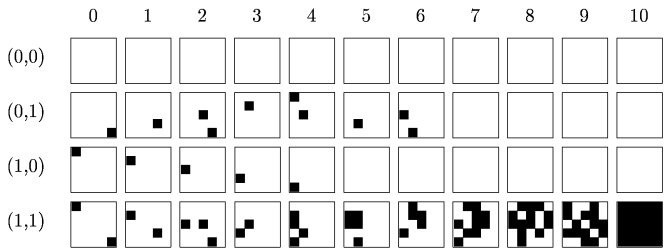

Figure 4.15: This figure displays the iterations of a CA solving the AND problem. Every row shows the iteration of the rule using a different initial state. Note that in the first column ($t = 0$) the initial states are clearly visible and in the last column the coloring matches the output of an AND port.

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 the maximum number of iterations I was set to 10. The borders of the CA were unconnected to allow a larger virtual distance between the two corner cells. This means that the leftmost 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 (in contrast to the connections made in the Majority Problem experiment). Instead, every cell on the border of the CA was connected to so called 'zero-cells'. These 'zero-cells' stay zero whatever happens and are never recalculated.

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 binary input values. All cells other than the two input cells are 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 $O^t_{(v_1,v_2)}$. The total fitness of the AND problem is defined as $f = (N - O^I_{(0,0)}) + (N - O^I_{(0,1)}) + (N - O^I_{(1,0)}) + O^I_{(1,1)}$. This makes the maximum fitness equal to $4 \times 5 \times 5 = 100$.

In this experiment a slightly different variation of the same Genetic Algorithm described in Section 4.4 was used. It uses Truncation Selection instead of Tournament Selection by first sorting the rules according to their

Table 4.4: Fitness values found in the AND problem.

	Number of runs			
	Neumann		*Moore*	
Fitness	*with crossover*	*without crossover*	*with crossover*	*without crossover*
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

fitness and then selecting the top 10% of the rules as 'elite' rules, copying 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% of the population. Both copying and cross-over were tested and compared. The newly generated rules are mutated and also used in the next generation. (When using cross-over, 90% of the rules were effected: $c = 0.9$) The algorithm stops if it finds a rule with $f = 100$ (the highest possible fitness) or it reaches 1000 generations ($D = 1000$).

Like in previous experiments Probabilistic Bit Flip Mutation was used. This is done by flipping every bit in the rule with a probability m. In preliminary experiments a number of different values of m were tested. Setting m to a rather high value of 0.05 turned out to be the most effective choice, which seems to confirm our insight (and is also in line with Bäck [5]) that with increasing selection pressure higher mutation rates than the usual $\frac{1}{l}$ (l being the length of the binary string) are performing better.

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 4.4.

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

The Moore neighborhood is clearly more powerful and was able to solve the problem perfectly. The rules that are able to do this clearly show communicational behavior in the form of 'traveling' information and processing this information at points where information 'particles' meet.

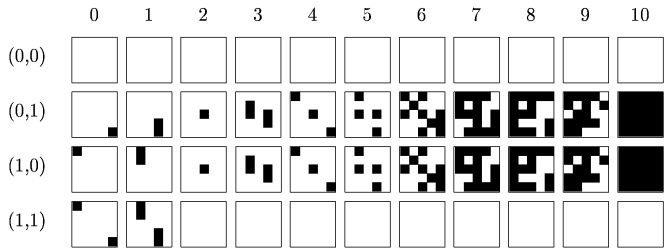

Figure 4.16: This figure displays the iterations of a CA solving the XOR problem. Every row shows the iteration of the rule using a different initial state. Note that in the first column ($t = 0$) the initial states are clearly visible and in the last column the coloring matches the output of an XOR port.

It is also 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 is worth exploring in future work. Maybe using other forms of crossover might give better results in combination with multi dimensional CA.

The XOR Problem

The XOR Problem is not much different from the AND problem. The same genetic algorithm and the same CA setup was used. The only difference is 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 $f = (N - O^I_{(0,0)}) + O^I_{(0,1)} + O^I_{(1,0)} + (N - O^I_{(1,1)})$.

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 4.5.

These results support earlier findings in suggesting that single-point crossover does not 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 von Neumann neighborhood seems unable to perform for 100%, yet it came

Table 4.5: Fitness values found in the XOR problem.

	Number of runs			
	Neumann		*Moore*	
Fitness	*with crossover*	*without crossover*	*with crossover*	*without crossover*
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

rather close with one rule classifying the problem for 92%. The algorithm found one transition rule using the Moore neighborhood that is able to solve the problem for the full 100%. This rule depicted in Figure 4.16 shows clear signs of 'traveling particles' and is another example of how a local rule can trigger global behavior using interaction.

Figures 4.15 and 4.16 clearly show interaction in the form of 'particles' moving from the two initial corners to the center where they combine to form the behavior of the two logical ports. It seems that the hardest part of the problem is not to combine the data in the beginning, but to propagate the result reliably throughout the CA. Figure 4.16 shows very clearly how at $t = 2$ the XOR problem is already solved in the center cell of the CA while all the other cells are in a zero state. In the AND problem that decision is a little bit less obvious, but the way the particles travel is very visible in $(0, 1)$ and $(1, 0)$. In both cases the propagation of the particle does not meet another particle and just disappears when the other side of the CA is reached. Only in the $(1, 1)$ row there is an interaction that results in a formation that grows and fills the entire CA.

Both the AND and XOR experiments show very clearly evolution of interaction on a very small scale. The way these experiments are able to show the actual pieces of information travel through a CA and interact is new and a lot clearer than can ever be observed in the Majority Problem.

Evolving interaction with nothing more than a problem definition that needs global information proves once more that evolution is very capable of inversely designing complex protocols and languages without any input from 'outside'. The evolution of language and communication in nature seems to be no more (and no less) miraculous than the diversity and complexity of its species. It seems reasonable to think they are part of one and the same evolutionary process.

4.11 Checkerboard Problem

To verify that the approach of inverse engineering Cellular Automata using a Genetic Algorithm works for many different problems, another experiment was set up. The aim of this experiment was to evolve a transition rule that lets a CA generate a checkerboard pattern. The idea being that cells inside the CA only know the states their neighbors are in and will have to decide which part of the checkerboard pattern they need to be based only on that limited information.

The nice thing about this problem is that it is independent of the number of dimension of the CA. The checkerboard pattern can be defined as *"a pattern in which every direct neighbor of every cell has the opposite value of that cell."* This means that fixing the value of one cell in the CA implies the values of every other cell in the CA. Which means that there are only two opposite checkerboard patterns possible, independent of dimensionality or size. The experiment aims to evolve a behavior in the cells that can successfully 'negotiate and come to an agreement' on which of the two patterns needs to be displayed.

The checkerboard problem can be defined as follows:

Find a transition rule that, given an initial state of a CA, iterates this CA to a stable 'checkerboard pattern' within I iterations.

In a one dimensional CA a checkerboard pattern could look like:

$$\{0, 1, 0, 1, ..., 1, 0, 1\}$$

Note that the first cell and the last cell are linked and should therefore also have different values. The problem is more intuitive in a two dimensional CA where cells are not only connected horizontally, but also vertically and the desired state therefore resembles a checkerboard. The problem can even be imagined in three dimensional CA where the end result should resemble a stack of checkerboards where every odd board in the stack is turned 90 degrees. In theory this problem is extendable to higher dimensional spaces, we will test our approach with $d = \{1, 2, 3\}$.

Just like in the Majority Problem the Checkerboard Problem used multiple initial states to determine the fitness of a transition rule. The fitness of a transition rule is measured by the relative number of directly adjacent cells in the end state that have an inverted value.

The same GA was used as in the Majority Problem, even the parameters have the same values (that is: optimal values as used in the last experiments). That means: $q = 20$, $c = 0.6$, $p_m = \frac{2}{l}$ and $D = 100$. Note that p_m

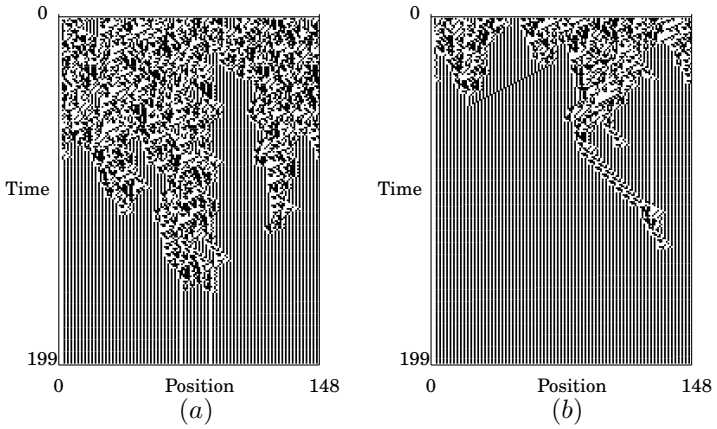

Figure 4.17: This figure shows two one dimensional CA iteration for the checkerboard problem. Note that (a) does not result in a perfect pattern, whereas (b) does.

is different for two dimensional CA, because S is a lot smaller. The number of cells in a dimension of the CA needs to be even, else a perfect checkerboard pattern will be impossible. CA with 150, 12^2 and 6^3 cells were used for $d = 1, 2$ and 3 respectively.

The algorithm was run 100 times for all three topologies also used in the Majority Problem: the one dimensional CA with $r = 3$, the two dimensional CA with a von Neumann neighborhood with $r = 1$ (5 cells) and the three dimensional CA also with a von Neumann neighborhood with $r = 1$ (7 cells). For all three runs the same parameters were used as in the last experiments on the Majority Problem. Note that m has different values for different topologies, because this variable is dependent on the number of cells in a neighborhood S and was always set to $\frac{2}{2^S}$. This means that for the one and three dimensional CA $m = \frac{2}{2^7} = \frac{2}{128} = 0.015625$ and for the two dimensional CA $m = \frac{2}{2^5} = \frac{2}{32} = 0.625$.

In the one dimensional experiment all the runs resulted in transition rules with $F^B_{150,10^3} > 0.95$ and the best rule has a fitness of 0.999. The two dimensional experiment had similar results with about 80% of the runs with $F^B_{12^2,10^3} > 0.95$ and the best rule with a fitness $F^B_{12^2,10^3} = 0.994$. Note that these results are achieved with a lot smaller neighborhood. In the three dimensional experiment however all the runs (except two) evolved rules with fitness values $F^B_{6^3,10^3} > 0.996$. Some of the rules even registered a perfect fitness for all the 10^4 random initial states.

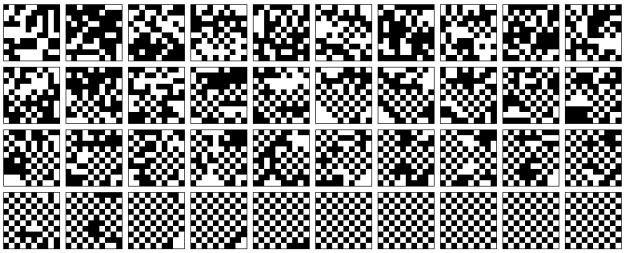

Figure 4.18: This figure shows a correct two dimensional CA iteration for the checkerboard problem. It starts top left with a random initialization of a 10×10 CA, iterates from left to right, top to bottom and ends up with a perfect checkerboard pattern in the end state.

The Genetic Algorithm did not seem to have any problems evolving the Checkerboard Pattern in a CA. The plot in Figure 4.19 shows complex interactions that evolved to solve this problem. It shows how a 50×50 CA was randomly initialized and run with a von Neumann transition rule that was evolved on a 13×13 CA, supporting earlier findings (in Section 4.9) that imply that the rules generated with this approach are robust with respect to the size of the CA. It shows how initially the rule generates local checkerboard patterns that expand and then touch to form the two opposing ideas on how to build a pattern. The boundaries of these patterns are then moving and merging to become smaller and smaller and in the end disappear to leave a perfect checker board pattern.

The static particles are very clear in this one: *"If I already have a checkerboard pattern in my neighborhood, do nothing."* The moving particles are then of course the moving boundaries that are like 'rubber bands' collapsing on themselfs. Looking at different rules reveals that there are different ways for these rubber bands to collapse, one more efficient than the other, but all of them seem to work by keeping one end still and moving the corners of the other end in a certain direction. The interactions to make this happen are maybe a little bit more apparent here than in the Majority Problem, but a lot less apparent than in the AND and XOR Problems.

Yet again it is shown how our approach is able to evolve a complex local interaction model to solve a global problem using nothing but the global problem definition for an Evolutionary Algorithm.

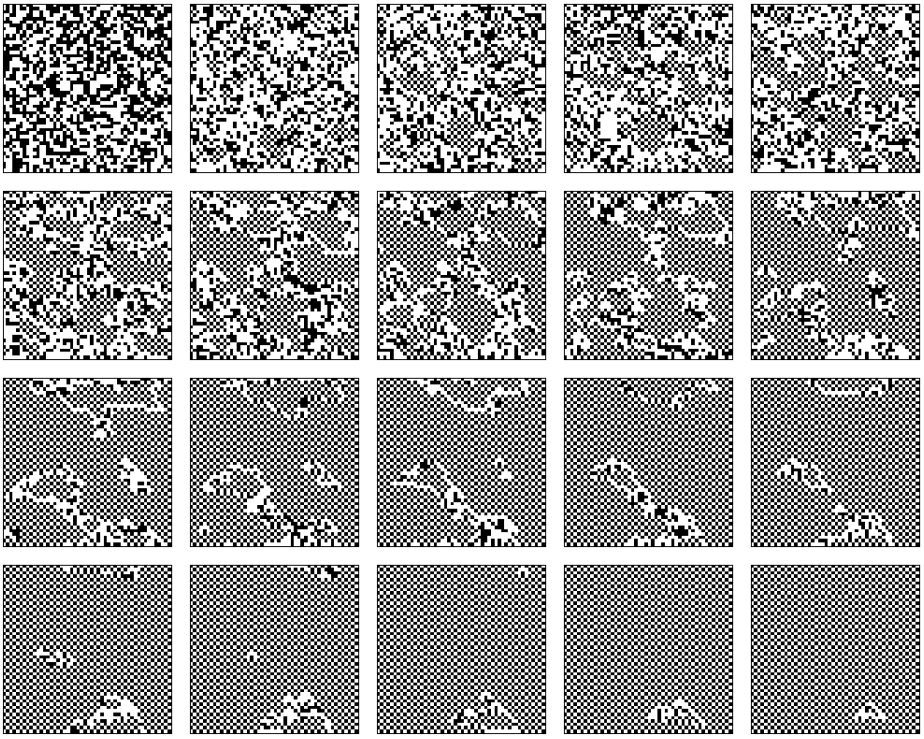

Figure 4.19: This figure shows the time plot of a 50×50 CA using the von Neumann neighborhood generating a Checkerboard Pattern with a transition rule that was evolved using a 13×13 CA showing again the robustness of the approach. The plots were generated using a 5 iteration interval (plots at $t = \{1, 6, 11, ..., 101\}$). Right after the last plot in this figure the CA came in an 'all zero state'.

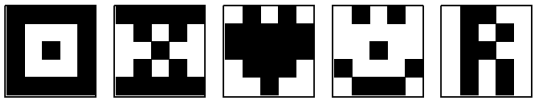

Figure 4.20: The bitmaps used in the pattern generation experiment.

4.12 Bitmap Problem

The problems defined in this chapter so far have all had one thing in common: they all had a very simple global problem definition. And although the problems were all very hard to solve from the local point of view of a single cell, it might be argued that the simplicity of the global problem impacted the performance of our approach. In order to find the limitations of using a GA to inverse design transition rules for a CA a new experiment was conducted.

The aim of this experiment is to evolve rules for two dimensional CA that generate patterns (or bitmaps).

The Bitmap Problem is defined as follows: *Given an initial state and a specific desired end state: find a rule that iterates from the initial state to the desired state in less than I iterations.* Note that this does not require the number of iterations between the initial and the desired state to be fixed.

The CA used in this experiment is not very different from the one used in the AND/XOR experiment (Section 4.10). In preliminary experiments we tried different sizes of CA, but decided to concentrate on small square bitmaps with a width and a height of 5 cells (as done in Section 4.10). To make the problem harder and to stay in line with earlier experiments the CA have unconnected borders like in Section 4.10. The von Neumann neighborhood was chosen instead of the Moore neighborhood and therefore s_n consist of 5 cells ($r = 1$) and a rule can be described with $2^5 = 32$ bits. The searchspace therefore is $2^{32} = 4,294,967,296$.

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 $(\lfloor \text{width}/2 \rfloor, \lfloor \text{height}/2 \rfloor)$ which is one. For the GA we used the same algorithm as we used 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 4.10 mutation is performed by flipping every bit in the rule with a probability p_m. After some preliminary experiments a

Table 4.6: Number of successful rules found per bitmap.

Bitmap	*Successful rules (out of a 100)*
'square'	80
'hourglass'	77
'heart'	35
'smiley'	7
'letter'	9

mutation rate of $2/l = 2/32$ seemed too high and the mutation rate was set to $p_m = 1/32 = 0.03125$.

In trying to be as diverse as possible five totally different bitmaps were chosen, they are shown in Figure 4.20. The algorithm was run 100 times for every bitmap for a maximum of 5000 generations. The algorithm was able to find a rule for all the bitmaps, but some bitmaps seemed a bit more difficult than others. Table 4.6 shows the number of successful rules for every bitmap. Note that symmetrical bitmaps seem to be easier to generate then asymmetric ones.

Although this experiment is fairly simple, it does show that a GA can be used to evolve transition rules in two dimensional CA that are able to generate patterns even with a simple von Neumann neighborhood. Ongoing experiments with bigger CA suggest that they don't differ much from these 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.

4.13 Conclusion

This chapter shows how Genetic Algorithms can be used to evolve transition rules for Cellular Automata. It shows how these Cellular Automata clearly exhibit interaction and how this implies that the interaction is evolved using nothing but the problem definition. A generic approach is introduced to inversely design local rules with the aim to find global behavior. This approach is shown to be robust in terms of algorithm parameters and flexible in terms of Cellular Automata topology.

The approach was applied on the Majority Problem in different topologies. Different offsets between neighbors in a one dimensional neighborhood were tested. Results show how spreading out the neighborhood in an exponential way seems to improve the performance of the algorithm. Two new distance

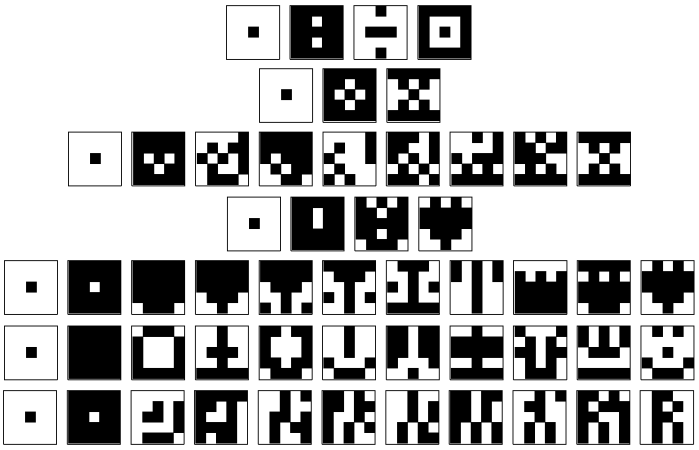

Figure 4.21: This figure shows some iteration paths of successful transition rules.

metrics were introduced: 'maximum distance' and 'average distance'. Results suggest that these metric between cells can be used to understand the performance differences between the experiments.

Different dimension CA were tested using the Majority Problem. The results show three dimensional CA outperforming two dimensional and two dimensional outperforming one dimensional CA. Even though the CA size had to be increased to run the Majority Problem on a multi dimensional CA and the transition rules in the two dimensional experiment were $\frac{1}{4}$ the length of that of the one dimensional version, the problem seemed to be easier to solve for multi dimensional CA.

The interactions within the CA were investigated and although it is clear that interactions are there it is hard to visualize these interactions in the Majority Problem especially in the multi dimensional CA. That is why the more constrained AND and XOR problems were introduced. Experiments were conducted to evolve transition rules that mimic the behavior of logical AND and XOR ports. Results show how the use of both the von Neumann and the Moore neighborhood successfully resulted in rules that solved these problems. The Moore neighborhood outperformed the von Neumann neighborhood and was able to generate a few rules that solved the problem of filling the CA with the right answer for a full 100%. The resulting time plots show a very clear 'particle' movement and interaction. Proving that very distinct interaction on a local level was evolved using the only the global desired behavior.

Solving the Checkerboard Problem was another example of the flexibility of the approach and resulted in some nice examples of evolved interactions on the one and two dimensional plots. Most successful transition rules seem to use a 'rubber band particle' to decide which checkerboard pattern is going to be used globally. This rubber band then tries to shrink and reduce the area inside of it until it evaporates. Rules generated using a small 13×13 CA also work on a larger 50×50 CA proving that the approach is capable of evolving generic rules invariant in the size of the CA.

The Bitmap Problem tested the limits of the approach. Transition rules to generate multiple different bitmaps were evolved. Symmetric bitmaps seem easier to evolve than non symmetrical ones, but for all 5×5 bitmaps successful rules were found. This shows that the approach can solve many different problems and mimic many different behaviors.

Not only has this chapter researched a generic way of inversely designing transition rules for Cellular Automata using a Genetic Algorithm, it has also shed some light on the Evolution of Interaction. Not only showing that it is possible, but also how it is possible and which criteria have an effect on the performance of the evolution. These findings might be valuable to research into the evolution of language in humans and animals, while at the same time having applications in the field of Grid Computing, Image Processing, various Biology research areas and of course in the fields of Cellular Automata and Evolutionary Algorithms itself.

Chapter 5

Self-adaptive Mutation Rates in Genetic Algorithm

Self-adaptation of mutation rates has been used with great success in Evolutionary Strategies in both research and real world applications. For Genetic Algorithms however, it is not (yet) the mutation rate adaptation of choice. This chapter describes how a self-adaptive mutation rate was used in a Genetic Algorithms to inversely design behavioral rules for a Cellular Automaton. The unique characteristics of this search space gave rise to some interesting convergence behavior that might have implications for using self-adaptive mutation rates in other Genetic Algorithm applications and might clarify why self-adaptation in Genetic Algorithms is less successful than in Evolutionary Strategies.

5.1 Introduction

Self-adaptation of mutation rates has been used with great success in Evolutionary Strategies in both research and real world applications. For Genetic Algorithms however it is not the mutation rate adaptation of choice. There are a few studies ([4, 3, 12] among others) on how self-adaptation could be employed in Genetic Algorithms, yet there still seems to be a lot we can learn about this combination.

This chapter investigates the impact of self adaptation on the inverse de-

sign of Cellular Automata rules. It described a comparison of fixed mutation rates with self-adaptation and tries to see if self-adaptation of mutation rates in GA can be used in a noisy fitness function like the Majority Problem. This problem in particular, and inverse design of behavior rules in general, are known to be very noisy and seem to have a very "rugged" fitness landscape. Both these characteristics make the Majority Problem a prime candidate to test the limits of applying self adaptation of mutation rates in GA, while at the same time exploring the use of self adaptation on inverse design of CA for the first time.

5.2 Majority Problem

Section 4.2 gives an extensive introduction into the Majority Problem, so this chapter will not attempt the same. Instead this section give a brief overview of the problem.

The Majority Problem can be defined as follows:

Given a set $A = \{a_1, ..., a_n\}$ *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?'.*

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

Find a transition 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* $\eta < 0.5$ *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 the state in which every cell is one.*

Evaluating a transition rule for this problem is done by iterating M randomly generated initial states and calculating the relative number of correct classification. The fitness of a transition rule is denoted with $F_{N,M}$ where N is the width of the CA. The fitness can be calculated with different distributions over the number of ones in the initial state, but the default is a binomial distribution (denoted with $F^B_{N,M}$) where every cell in the CA has a 50% chance of being initiated with a one for every initial state.

A uniform distribution over the number of ones is used while evolving the rules in the GA. This distribution generates more 'easy' initial states with a large difference between the number of ones and the number of zeros, thus making it easier to train the desired behavior. The fitness using initial states with this uniform distribution over the number of ones is denoted

with $F^U_{N,M}$. Experiments described in this chapter all use this distribution.

5.3 Genetic Algorithm

The genetic algorithm that is used in this article is based on algorithms used in Mitchell et al. [29, 30] and Breukelaar et al. [11, 7].

The genetic algorithm for all experiments is a $(10, 100)$ strategy where 100 stands for the number of individuals in the pool and 10 for the number of parents that is selected from that pool. Parents are selected by choosing the ten fittest individuals from the population. Every parent is then copied ten times every generation (making 100 children) and all individuals are mutated using a mutation operator.

Unlike previous experiments, no crossover was used in these experiments. Not only have earlier results by Breukelaar et al. [11, 7] (see Chapter 4) shown that the impact of crossover on the Majority Problem is minimal, but investigating the effect of self-adaptation is also a lot easier without taking into account crossover.

The algorithm uses a comma strategy. This is also different from previous experiments, which used a plus strategy. Preliminary experiments seem to suggest that self-adaptation of mutation rates in GA do not work very well in a plus strategy. This probably has to do with the success rate in the algorithm and the fact that the mutation rate only changes if the individual is mutated. In a plus (or elitist) strategy the parents are not mutated and copied to the next generation without any changes. If the success rate in the algorithm goes down, these elite individuals will drown out any form of diversity while at the same time keeping the mutation rate identical and probably too high. Using a plus strategy with self adaptation is probably possible if this effect can be countered somehow and this is worth investigating in the future, but by choosing a comma strategy this problem is evaded. Every surviving individual is mutated every generation, which results in a dynamic mutation rate that evolves at the same time as the individuals object values, thus forcing the algorithm to select the best mutation rate at different stages in the optimization.

5.4 Self Adaptation

In this chapter a self-adaptive method will be used as first proposed by Bäck et al. [5] which suggests to add a floating point value to every individual to represent their individual mutation rates. The mutation rate is mutated every generation using the following formula:

$$p_m(t+1) = \left(1 + \frac{1 - p_m(t)}{p_m(t)} \cdot \exp(-\gamma \cdot N(0,1))\right)^{-1}$$

Where $p_m(t)$ is the parents mutation rate at generation t, $p_m(t+1)$ the new mutation rate and γ is a constant to impact the convergence speed of the mutation rate (usually set to 0.22). $N(0,1)$ represents a random sample taken from a normal distribution with mean 0 and standard deviation 1. The individual bit string will then be mutated using a probabilistic bit flip operator using the new mutation rate $p_m(t+1)$.

If the mutation rate p_m ever drops below $1/l$, the rate will be adjusted to $1/l$ exactly, the idea being that a mutation rate below $1/l$ does not change the population and therefore does not make sense in a comma strategy. The impact of this and other aspects of the self-adaptive mutation rate will be discussed at the end of Section 5.5.

The initial value for the mutation rate in all experiments in this chapter will be $p_m = 0.5$. This is probably way too high, but the self-adaptive nature of the algorithm should have no problem adapting this rate to a more optimal value, which will both show the power of self adaptation and its successful implementation into the inverse design of cellular automata.

5.5 Experiments

In order to put the self-adaptive mutation rate results into context three different fixed mutation rates were also run in this experiment: $p_m = 2/l, p_m = 4/l$ and $p_m = 8/l$, where n is the length of the bit string. Because all the bit strings are 128 bits long, this gives $p = [0.015625, 0.03125, 0.0625]$ for the three fixed mutation rates.

The Majority Problem was run on a one dimensional CA with 149 cells. The cells were initialized with 100 initial states uniformly distributed in the number of ones (there were the same number of initial states for each number of cells set). The CA was updated synchronously and was run until no cell would change or 320 steps were reached.

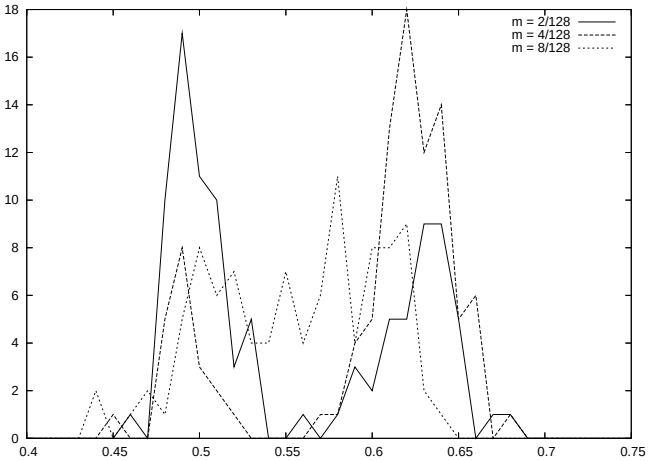

Figure 5.1: A comparison of fitness distributions $F^B_{149,1000}$ of three fixed mutation rates after 100 generations, 100 runs each.

First the algorithm was run 100 times for all three of the fixed mutation rates to be able to compare the performance. The GA was run for 100 generations every time. The resulting best individuals of the runs were then measured using the initial states with the normal distribution. These new fitnesses of the individual runs were sorted and then put in groups of width 0.01 in the fitness space. This then gives a frequency graph of the occurrence of a certain fitness value. Figure 5.1 represents the three experiments and their distribution on fitness after 100 generations.

Note how in figure 5.1 the setting with mutation rate $p_m = 4/l$ seems to have outperformed the other two, for it has the highest number of fitness values between 0.6 and 0.7. The setting with mutation rate $p_m = 2/l$ seems stuck on a local optimum around 0.5, and the setting with mutation rate $p_m = 8/l$ is all over the place with the fitness value.

In order to find out if the setting with mutation rate $p_m = 2/l$ really gets stuck, that experiment was extended to a length of 200 generations per run. Figure 5.2 shows that after 200 generations the mutation rate of $p_m = 2/l$ has about the same fitness distribution as the run with $p_m = 4/l$ had after 100 generations. Both runs were still improving at the end and could possibly improve more beyond the 200 generations. Note that this is the average fitness over 100 runs, it says very little about the quality of the individual solutions or how close they come to a global optimum. What can

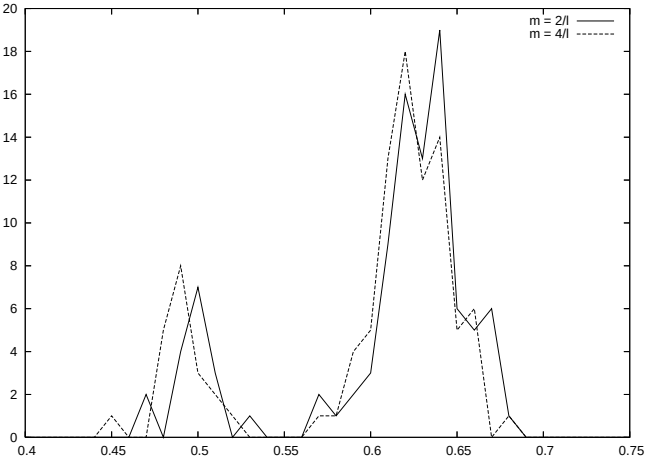

Figure 5.2: A comparison of fitness distributions $F^B_{149,1000}$ of two fixed mutation rates ($p = 4/l$ and $p = 2/l$) after 100 and 200 generations respectively, 100 runs each. Note that the fitness distribution of $p = 4/l$ after 100 generations is almost identical to the fitness distribution of $p = 2/l$ after 200 generations.

be learned from this is that a mutation rate of $p_m = 4/l$ seems to be the best, especially at the beginning of the run. A lower rate will get there eventually, but half the rate takes twice the time.

5.6 Self-Adaptive Experiment

Next the self-adaptive mutation rate was used. The rest of the algorithm was kept totally identical in order to compare the self-adaptive method to the fixed rates. After some preliminary results it was decided to let all the self-adaptive algorithms run for 200 generations to see not only the short term, but also the longer term effects. Figure 5.3 shows the average fitness of all the 100 runs over the generations of the self-adaptive strategy compared to the fixed strategy with $p_m = 2/l$.

The big difference between the self-adaptive mutation runs and runs with a fixed rate can be found in the beginning. The self-adaptive method does not settle on a fitness of 0.5 straight away, while the fixed mutation rate almost immediately jumps to a fitness of 0.5 and stays there. This can be

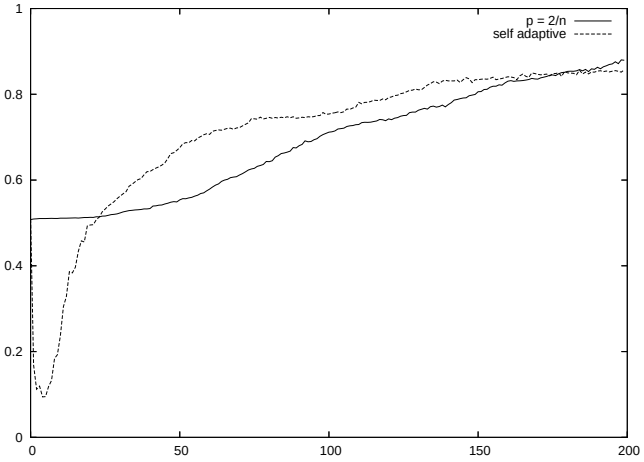

Figure 5.3: This figure shows the average fitness of the self-adaptive mutation versus the slowest fixed mutation rates tested ($p_m = 2/l$). Notice how the self-adaptive method does not start at 0.5 in the same way as the fixed mutation does.

contributed to the initialization of the CA in combination with the low mutation rate of the fixed mutation. Initializing rules with a uniform distribution in the number of ones makes for at least one rule with all ones and at least one rule with all zeros. Both these rules have a trivial behavior that makes the entire CA either ones or zeros in the first step. Because this classifies on average half of the initial states correctly, the fitness is 0.5. The self-adaptive method starts with a much higher mutation rate of 0.5 however and this kills these trivial rules straight away, bringing the average fitness down for the first twenty generations.

Yet even though the self-adaptive method does not start out at a fitness of 0.5, it catches up very quickly. It climbs very fast in the beginning and then stagnates after about 140 generations. In comparison the lowest fixed rate that was tested keeps on going until it eventually catches up with the self-adaptive strategy around generation 175. And more importantly, the fixed mutation rate does not show any sign of stagnation within these 200 generations and promises to continue upwards at least another 50 generations or so.

In order to understand what is going on figure 5.4 shows the average mutation rate of the self-adaptive method compared to the three fixed mutation

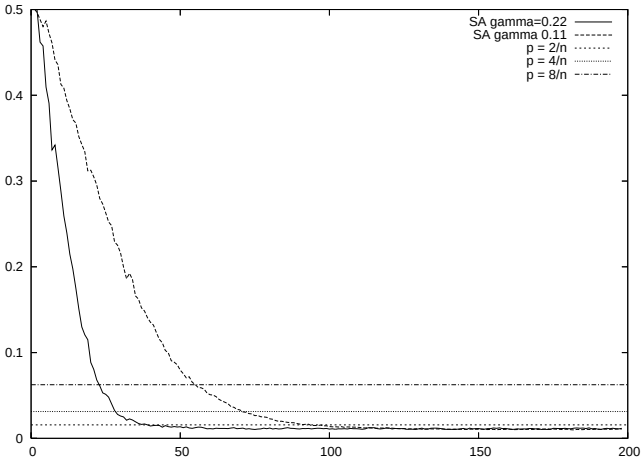

Figure 5.4: Shows the average mutation rate of the self-adaptive methods compared to the three fixed mutation rates. Notice how the self-adaptive mutation rates dives fast and constant towards its minimum value $1/l$.

rates. And the important thing to notice here is that not only the mutation rate goes down quickly and constantly, but it hits the minimum of $p = 1/l$ as early as generation 50. This is surprising given the fact that in the first 100 generations a mutation rate of $p = 4/l$ and even $p = 8/l$ clearly outperforms $p = 2/l$. Mutation using those settings should have been better during this part of the algorithm and self-adaptive mutation should have evolved into something close to this mutation rate. There clearly is another force at work here pulling the mutation rate down.

Some small side-experiments showed that running the self-adaptive algorithm without the $1/l$ minimum keeps on decreasing the mutation rate to far below $1/l$. This is surprising because intuitively a mutation rate of $p = 1/l$ would always outperform a mutation rate lower than $1/l$, but that does not seem to be the case here.

5.7 Battling Convergence

In order to counter the rapid convergence, it was decided to lower the γ value in the self-adaptive mutation rule. A lower gamma would translate into a slower rate of change in the mutation rate and to be sure to provoke

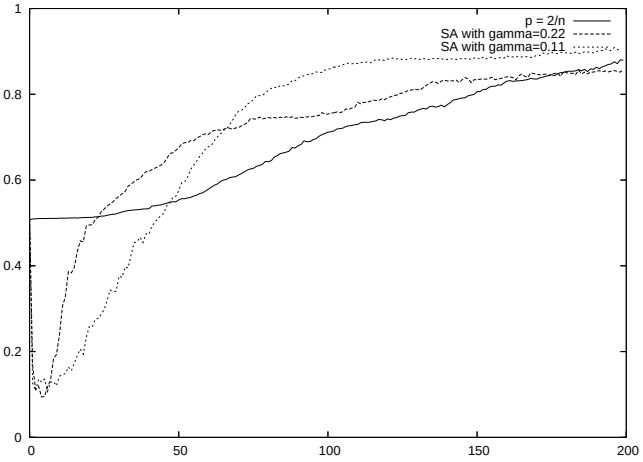

Figure 5.5: Shows the average fitness of the two self-adaptive mutation experiments with different γ settings versus the slowest fixed mutation rates tested ($p = 2/l$). Notice how the self-adaptive method does not start at 0.5 in the same as the fixed mutation does.

a response from the algorithm, it was decided to half the γ value to 0.11. A new experiment was run with 100 runs, 200 generations, 100 initial states and a self-adaptive mutation rule with $\gamma = 0.11$.

This clearly improved the performance of the algorithm and it outperformed the fixed mutation rate. Figure 5.5 shows this new experiment compared to the other two experiments that were run with 200 generations. It shows clearly how the self-adaptive method with $\gamma = 0.11$ takes a little longer to catch up with the fixed mutation rate, but when it does it shoots past the self-adaptive method with $\gamma = 0.22$. This implies that the self-adaptive method with $\gamma = 0.22$ converges too fast and gets stuck on a local optimum. But as mentioned earlier there is no clear reason why the algorithm should do this, other than an (until now) unknown force pulling on the mutation rate.

Although this experiment with a different γ-setting improved on the former experiment, the new results also stagnate after 100 generations. Figure 5.4 shows the average mutation of the five different mutation rates throughout the runs. Note that while both runs with $\gamma = 0.22$ and $\gamma = 0.11$ seem to adapt their mutation rates, they both adapt the mutation rate to an irrational low value at a fitness that is known to work a lot better with a higher fixed

mutation rate. Now there could always be a local optimum that a GA gets stuck on, but these graphs represent averages and one would expect the distribution of local optima to be identical for both self-adaptive and fixed mutation.

This seems to prove that self adaptation of mutation rates to inversely design Cellular Automata works, but that there is a reason the adaptation prematurely converges to a rate that seems a lot lower than expected. It almost seems to 'give up' too soon. The rest of this chapter will investigate in more detail what force is at work here.

5.8 Noise

One possible explanation for the early convergence is noise. The Majority Problem fitness function is calculated by "measuring" how often a rule classifies something correctly. This is done on random initialized CA states and that means that the fitness function could get "unlucky" and classify a rule slightly better or worse than it can actually perform. This then in turn would lower the selection pressure and the chance of selecting wrong mutation rate changes would increase. At least that is the working hypothesis.

To test this hypothesis we conducted an experiment in which we took the original self-adapting method with $\gamma = 0.22$ and increased the number of initial states used in the fitness function from 100 to 250 to dampen noise. Furthermore the number of generations for the GA was increased to 500, so that stagnation would be better visible.

Figure 5.6 shows the results of this experiment. The effect on the convergence was very limited. It hit the same stagnation at almost the same time and it can be clearly seen that this stagnation is there to stay. The algorithm no longer improves, not even after 500 generations. Although this does not conclusively prove that noise is not a problem here it does show that noise is not main reason behind this stagnation.

If it is not the noise that is causing this convergence and changing the γ setting only delays the same behavior, there has to be another reason why self-adaptation has such a big problem with this search space. The next Sections speculate in some more detail what might be happening to self-adaptation for it to converge prematurely in the case of a "rugged" combinatorial fitness landscape like the Majority Problem.

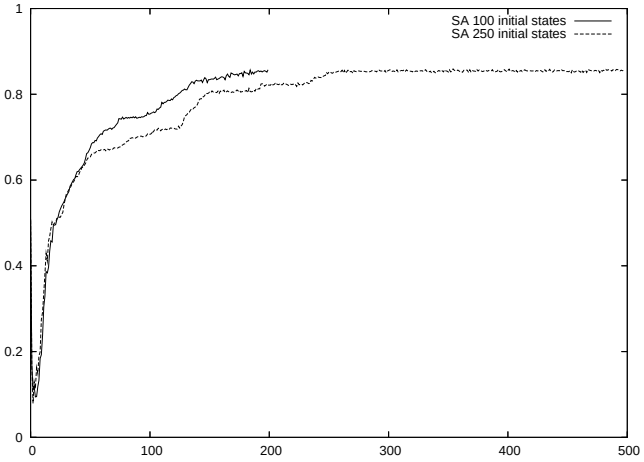

Figure 5.6: This figure shows the average fitness of the two self-adaptive mutation experiments with different noise levels. The noise level was decreased by increasing the number of initial states that was used by the Majority Problem. Note that the impact of reducing the noise has almost no effect.

5.9 Unseen Forces

All the experiments in this chapter have been using a comma strategy in which every individual in the population is mutated every generation, yet that does not mean all the parents in the population are destroyed every generation. Even though every individual is mutated, there is always the possibility that none of the bits in an individual are flipped.

Let p_{clone} be the chance that an individual is not changed by mutation. Then:

$$p_{clone} = (1 - p)^n$$

where p is the mutation rate and n is the bit string length. This means that for $p = 1/l$ (minimum allowed in experiments) and $n = 128$ (true for all experiments):

$$p_{clone} = (1 - \frac{1}{128})^{128} \approx 0.3664$$

Every parent in the experiments above was copied 10 times. This means that if the mutation rate is $p = 1/l$ for an individual and that individual is chosen to be a parent, then on average 3 to 4 copies of this parent will reside unaltered in the next generation. This makes the comma strategy effectively some kind of "probabilistic plus strategy".

With fixed mutation rates this effect is not a very big deal, for it will in worse case slow down the algorithm, but when self-adaptive mutation is employed an interesting effect emerges.

An important aspect of the self-adaptive mutation rule is that the mutation rate should have the same probability to change up, as it has to change down. This ensures that the search for an optimal mutation rate is not biased in any one direction. The experiments in this chapter suggest that the self-adaptive mutation used, seems to have such a bias. Surprisingly the "probabilistic plus strategy" as explained above could create such a bias.

As mentioned before every parent in the experiments was copied ten times and right before the bit flip mutation is employed, the mutation rates are themselves mutated. If the mutation rate of a child is decreased its p_{clone} increases and it is more likely to stay identical than a child of which the mutation rate is increased. This simple fact implies that the average mutation rate of the children that are unchanged is lower than the mutation rate of their parent and the average mutation rate of the changed children is higher than that of their parent.

At the beginning of the algorithm there are still a lot of improvements to be made and they are relatively easy to find. The success rate of the algorithm is high. In a "rugged" landscape however, the success rate of a Genetic Algorithm might go down drastically. In a comma strategy the fitness of the next generation is allowed to be lower than that of the previous one, but if p_{clone} is high enough, clones will be generated such that the fitness will not easily go down. This means that the harder it becomes to find an improvement, the more appealing it will be to stay in the same location. The cloned individuals will get a comparatively high fitness and will be chosen over most mutated individuals. Because the mutation rates of the clones are very likely to be lower than their parents, the mutation rate will decrease and only make it harder to leave the comfort of the "couch". The algorithm will grind to a premature halt.

Due to the similarities with the addictive properties of TV and the comfort giving properties of a couch, this effect will from here on be called the 'Couch Potato Effect'. It is very likely that an effect as described above is the cause for the stagnation seen in the experiments in this article. The Majority

Problem is known to have a "rugged" landscape and the "success rate" might be low enough to cause this kind of premature convergence.

5.10 MAXONE Problem

To make sure that the premature stagnation is caused by the cloning of individuals and not by other reasons for premature stagnation as described by Rudolph, G. [36] and Salomon, R. [37], a new and simpler experiment was conducted. The previous research mentioned above mainly concentrated on reasons of stagnation due to local optima. The 'Couch Potato Effect' as described in Section 5.9 implies that stagnation could occur even if there would be only one optimum.

One of the simplest binary fitness functions is called the MAXONE Problem. It can be defined as follows:

Given a bit string of a fixed size n, find the bit string that has the largest number of ones.

The answer to this problem is of course trivial (the bit string consisting of all ones) and since no Cellular Automata or inverse design is used in this experiment, any Genetic Algorithm should not have much trouble finding that exact answer. This Section does not try to show the power or versatility of GA, but rather examine whether the "Couch Potato" effect is real and how big this effect is on a simple problem like the MAXONE Problem.

The same Genetic Algorithm as described in Section 5.3 was used with the same self adaptive mutation, but in order to simplify the analysis of the results, a $(1, 5)$-strategy was chosen. This means that for each generation only one parent is selected, which is then copied five times while the parent itself is discarded. All five children are mutated in exactly the same way as described in Section 5.3.

The aim of this experiment was to test whether the effect of the self adaptive mutation on GA can be analyzed in more detail using a simpler approach to see if the 'couch potato' effect exists in a simple experiment and if so, how big it is. In order to get a baseline for the optimal mutation rate, this experiment was also run using different fixed mutation rates. The mutation rates used are: $\{0.1/l, 0.2/l, 0.5/l, 1/l, 2/l, 4/l, 8/l\}$. Note that a mutation rate of $0.1/l$ seems very low, but preliminary results indicated that this value was needed in the baseline.

The experiment was run 1000 times, each run with a maximum of 1000 generations. The bit string size l was set to 128 for all experiment. Fig-

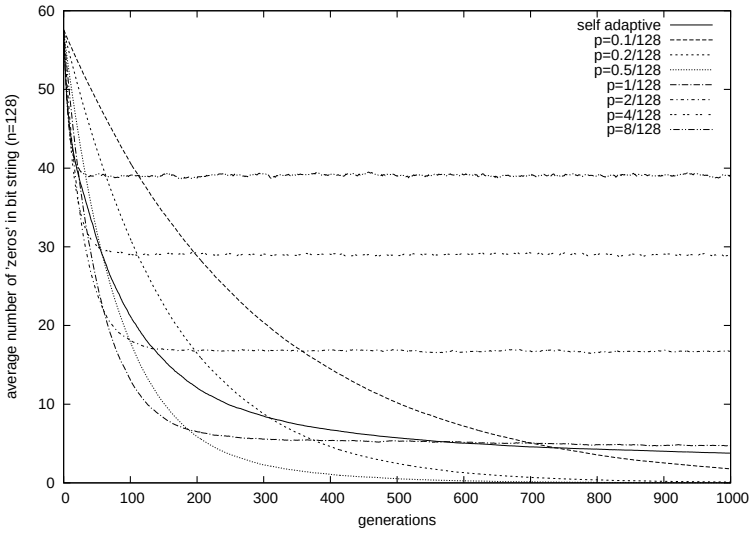

Figure 5.7: This figure shows the average fitness of runs for different fixed mutation rates $p = \{\frac{8}{128}, \frac{4}{128}, \frac{2}{128}, \frac{1}{128}, \frac{0.5}{128}, \frac{0.2}{128}, \frac{0.1}{128}\}$ and self adaptive mutation rate. Each line is the average of 1000 runs of the GA as described in Section 5.10. Note how although self-adaptation of the mutation rates seems to work, its performance is worse than half the fixed mutation rates.

ure 5.7 shows the average number of 'zeros' in the bit strings over all 1000 runs.

Note how mutation rates of $8/l$, $4/l$, $2/l$ and even $1/l$ seem to stagnate on various locations in the search space, while $0.5/l$ and $0.2/l$ both seem to find the optimum all the time (the one quicker than the other) and $0.1/l$ seems too slow to find the optimum in the maximum 1000 generations. The reason that the high mutation rates stagnate is probably due to the comma strategy. The parents are not used in the next generation and this means that a high mutation rate can throw away valuable steps into the right direction. The higher the mutation rate, the bigger the chance that information is thrown away that way. This is also the reason that the best performing mutation rate seems to be $0.5/l$ and that this is below the minimum suggested by [4].

Also note that the self adaptive runs started out on a good fitness value going parallel to $8/l$, but then they flatten out fairly quickly, stagnating to become even worse then $0.1/l$. What is going on here? Could this be the same effect as viewed in the inverse design task earlier this chapter? Could

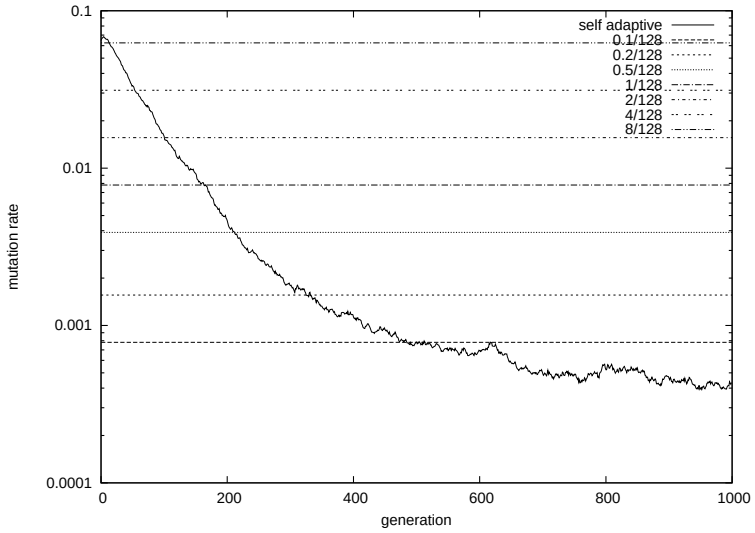

Figure 5.8: This figure shows how the average self adaptive mutation rate compares to the fixed mutation rates on a logarithmic scale. The mutation rate is an average of 1000 runs as described in Section 5.10 and concerns the same experiment as displayed in Figure 5.7. Note how the self adaptive mutation rate decreases very quickly and continues to go down even after it went below $\frac{0.1}{128}$. This is surprising because Figure 5.7 shows how $\frac{0.5}{128}$ clearly outperforms lower mutation rates.

this be the “Couch Potato Effect”?

Figure 5.8 shows the average mutation rate of the self adaptive mutation runs plotted against the fixed mutation rates. Note how the self adaptive mutation rates goes down rapidly right past all the fixed rates and ends up at about half that of the lowest fixed rate used. That is unexpected to say the least. Self adaptive mutation rates should be trying to find the best rate during the run. Instead it finds a reason to stagnate on a rate that is 10 times lower and still falling. What is the reason for this?

The nice thing about the MAXONE problem is that probabilities and optimal mutation rates can be calculated exactly. These exact calculations can then be compared to the results generated from the experiment to figure out what is going on and what effect is causing self-adaptation to perform poorly. The next Section tries to do exactly that.

5.11 Calculating Progress

Bäck shows in [5] a detailed analysis of the MAXONE Problem (under the name 'the Counting Ones Problem') in which the exact probabilities of success, stagnation and failure are calculated. It concentrates on a theoretical analysis of the convergence velocity and reports on empirical evidence that supports these results. This thesis will use this analysis as a starting point to try and explain why self adaptation in GA seems to converge prematurely. It differs from the research in [5] in that instead of concentrating on the convergence velocity of the population as a whole, in Section 5.12 it concentrates on the 'chance to be selected' of a single individual taking into account the fitness of the other individuals in the pool. This difference in approach allows us to shed some light on what it takes for an individual to survive in a population and how this could negatively effect self adaptation of mutation rates.

In order to calculate the exact chance of successful mutation of a bit string with length n in the MAXONE Problem, the bit string needs to be defined a bit more abstract. Let the bit string B be defined as a set of n_0 zeros and n_1 ones, then $n_0 + n_1 = n$ and the fitness of B is defined as exactly n_1 (or $\frac{n_1}{n}$ to make the fitness fall between 0 and 1). Then a successful mutation can be defined as a mutation after which the new number of ones n_1' is bigger than n_1.

The problem with calculating the exact chance for a successful mutation is that there are many different ways to make a mutation successful. If one zero changes into a one and that is the only change in the string then the mutation is successful, but if three zeros turn into ones and two or less ones turn into zeros, the mutation is also successful. In order to help calculate the chance for a successful mutation an intermediate chance is defined as the chance that exactly i bits flip in k bits of the bit string:

$$p_{exactly}(i,k) = \binom{k}{i} (p_m)^i \cdot (1-p_m)^{k-i}$$

where p_m is the mutation rate of the probabilistic bit flip mutations operator.

Now a successful mutation can be defined as a mutation in which *'more zeros were mutated into ones than ones into zeros'*. Which is the same as saying *'the exact number of bits flipped in n_0 bits of the bit string is higher than the exact number of bits flipped in n_1 bits of the bit string'*. The chance for this last definition to happen can be defined in the following summation:

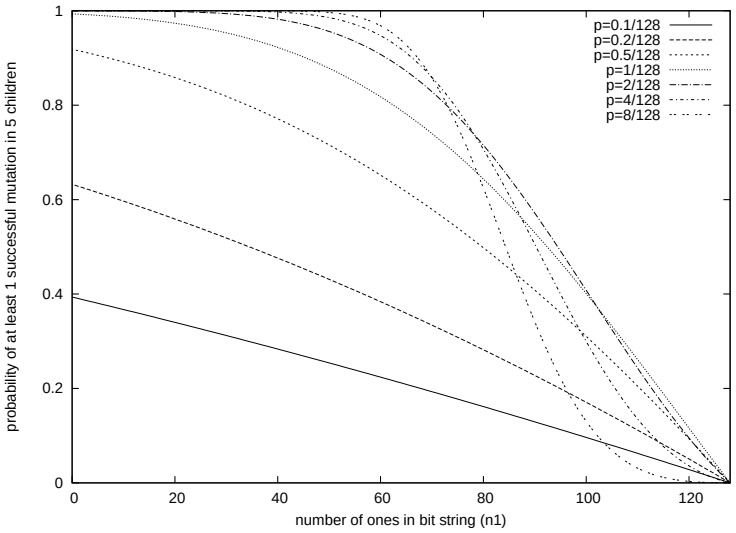

Figure 5.9: This figure shows the probability of at least one successful mutation given a population size $\lambda = 5$ and a bit string $l = 128$ for all different number of ones n_1 and different mutation rates $p = \{\frac{8}{128}, \frac{4}{128}, \frac{2}{128}, \frac{1}{128}, \frac{0.5}{128}, \frac{0.2}{128}, \frac{0.1}{128}\}$. Note how $p = \{\frac{8}{128}, \frac{4}{128}, \frac{2}{128}, \frac{1}{128}\}$ each have the highest success probability for different intervals of n_1.

$$p_{success} = \sum_{i=1}^{n_0} \Big[p_{exactly}(i, n_0) \cdot \sum_{j=0}^{\min(i-1, n_1)} p_{exactly}(j, n_1) \Big]$$

or to write this as a function of the number of ones n_1 (because $n_0 + n_1 = n$):

$$p_{success} = \sum_{i=1}^{n-n_1} \Big[p_{exactly}(i, n - n_1) \cdot \sum_{j=0}^{\min(i-1, n_1)} p_{exactly}(j, n_1) \Big]$$

In exactly the same way the chance of an unsuccessful mutation can be defined as *'the exact number of bits flipped in n_0 bits of the bit string is lower than the exact number of bits flipped in n_1 bits of the bit string'*. Then keeping in mind that $n_1 = n - n_0$ this gives:

$$p_{unsuccess} = \sum_{i=0}^{n-n_1} \left[p_{exactly}(i, n-n_1) \cdot \sum_{j=i+1}^{n_1} p_{exactly}(j, n_1) \right]$$

Then there is the possibility that the mutation of the bit string does not change the number of ones at all. This happens when *'the exact number of bits flipped in n_0 bits is identical to the exact number flipped in n_1 bits*. Or to use the notation of the previous chances:

$$p_{same} = \sum_{i=0}^{\min(n-n_1, n_1)} \left[p_{exactly}(i, n-n_1) \cdot p_{exactly}(i, n_1) \right]$$

Note that $p_{success} + p_{unsuccess} + p_{same} = 1$. Also note that the chance that no bits are flipped (p_{clone}) is part of p_{same}.

These are the chances for different outcomes of the mutation for only one individual. In order to say something about the success rate of the Genetic Algorithm, these chances need to be applied on the entire pool of individuals. There are three different types of generations for a GA:

- It was 'successful': The generation generates at least one individual with a better fitness than its parent.
- It was 'unsuccessful': All children in this generation have worse fitness than their parent.
- It is the 'same': At least one individual in this generation has the same fitness as its parent and non are successful.

The chances for these three different types are:

$$p_{gen\_success} = 1 - (1 - p_{success})^{\lambda}$$

$$p_{gen\_unsuccess} = (p_{unsuccess})^{\lambda}$$

$$p_{gen\_same} = 1 - p_{gen\_success} - p_{gen\_unsuccess}$$

where λ is the number of children created. Note that p_{gen_same} can not easily be expressed in $p_{success}$ and p_{same} because these chances are dependent and the *'chance that non are successful* **given that** *at least one has the same fitness'* is not that easy to calculate (although possible) and would in the end be identical to the simpler definition above.

The chance for a successful generation depends heavily on the number of bits that are already correct (n_1). Figure 5.9 shows the success rate $p_{gen\_success}$

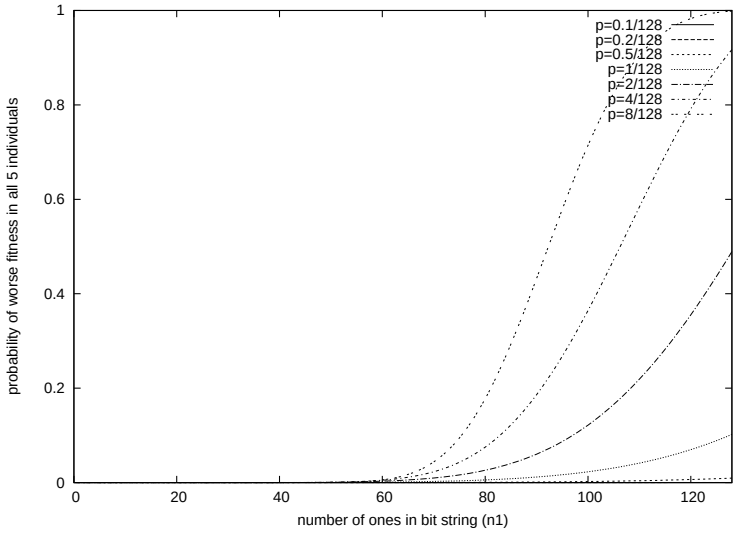

Figure 5.10: This figure shows the probability of having only unsuccessful mutations given a population size $\lambda = 5$ and a bit string $l = 128$ for all different number of ones n_1 and different mutation rates $p = \{\frac{8}{128}, \frac{4}{128}, \frac{2}{128}, \frac{1}{128}, \frac{0.5}{128}, \frac{0.2}{128}, \frac{0.1}{128}\}$. Note how this probability is almost 0 for $p = \{\frac{0.5}{128}, \frac{0.2}{128}, \frac{0.1}{128}\}$ and increased with higher mutation rates and n_1.

for all different number of 'ones' (n_1) and different mutation rates (p). Note how if the number of 'ones' is low, higher mutation rates have more success, but when the number of 'ones' is high, lower mutation rates have more success.

Figure 5.10 shows the same set of mutation rates for all number of 'ones' (n_1), but this time the chance for an unsuccessful generation $p_{gen\_unsuccess}$ is plotted. Note how low mutation rates don't seem to have much chance for an unsuccessful generation, while higher mutation rates only seem to have unsuccessful generation when the number of 'ones' outnumber the number of 'zeros' ($n_1 > n_0$).

Because the GA is using a comma (or 'non elitist') strategy the success of the algorithm is not only dependent on how often the mutation generates favorable results, but it also depends on how often that algorithm throws that result away the next generation. So in order to calculate the success rate of the entire algorithm both $p_{gen\_success}$ and $p_{gen\_unsuccess}$ need to be taken into account. For this the 'progress rate' P is defined as:

$$P = p_{gen\_success} - p_{gen\_unsuccess}$$

Note that this rate has values ranging from -1 to 1, where positive numbers imply the algorithm will get closer to the optimum and negative numbers imply the population will drift away from the optimum. This also means that where $P = 0$ the algorithm will have trouble making any progress and will probably get stuck.

Also note that unlike the convergence speed investigated by Baeck in [5] this 'progress rate' does not take into account the magnitude of the changes made by the mutation. This simplification can be justified by the assumption that the stagnation effect that this section is investigating occurs when the mutation rate p_m falls below $1/l$ and the chance of a change in fitness of two or more is getting unlikely. By making the assumption that the size of the fitness change is less important than the direction of the change, the interaction through selection between the individuals in the population can be studied more easily.

Figure 5.11 shows this rate for different mutation rates for all different values of n_1. Note that this plots the calculated progress of the GA as described earlier, a different algorithm might of course have a different behavior. The plot shows a lot of interesting information though. Most important of all: it predicts what the best mutation rate is for a given number of 'ones' (n_1) by showing which mutation rate has the highest progress rate at a given n_1. Surprisingly (although expected after the experiments), when n_1 becomes

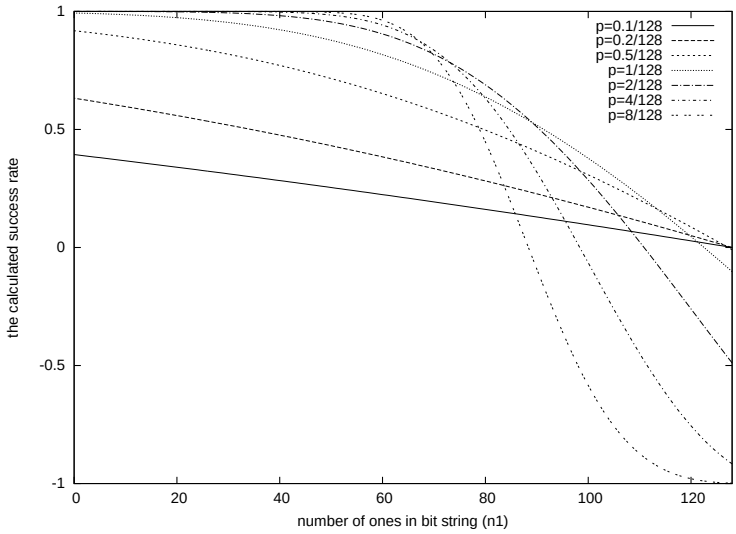

Figure 5.11: This figure displays the 'success rate' ($P = p_{gen\_success} - p_{gen\_unsuccess}$) for $\lambda = 5$ and $l = 128$ for all different values of n_1 and mutation rates $p = \{\frac{8}{128}, \frac{4}{128}, \frac{2}{128}, \frac{1}{128}, \frac{0.5}{128}, \frac{0.2}{128}, \frac{0.1}{128}\}$. Note how in contrast to Figure 5.9 here $p = \frac{0.5}{128}$ starts to outperform all other mutation rates for $n_1 > 115$.

high, the best mutation rate ($0.5/l$) is lower than the classical minimum mutation rate $1/l$.

The intersections in the plot represent the places in the search space where the performance of a lower mutation rate becomes better than the higher one. Where the $8/l$ line meets the $4/l$ line for instance (around $n_1 = 67$) the mutation rate $p = 4/l$ starts to outperform $p = 8/l$. Until that point $p = 8/l$ was the best performing mutation rate. Then around $n_1 = 73$ it starts to become better to use $p = 2/l$ and around $n_1 = 88$ $p = 1/l$ seems best. That lasts up to $n_1 = 113$ at which point $p = 0.5/l$ becomes the best progress rate and stays best until the optimum is reached. Figure 5.7 shows these progress differences between different mutation rates in the form of the 'rate of descent'. The steeper the line goes down, the bigger its progress. Note that exactly at the values of n_1 where the progress rates intersect in Figure 5.11 the steepness of the corresponding lines is identical in Figure 5.7. The calculated progress rate seems to confirm the experimental results.

Figure 5.11 also shows at which n_1 the progress rates becomes negative. This only happens to $p = \{8/l, 4/l, 2/l, 1/l\}$ and not to $p = \{0.5/l, 0.2/l, 0.1/l\}$. The mutation rate $p = 8/l$ yields a negative progress rate at $n_1 = 88$, $p = 4/l$ at $n_1 = 98$, $p = 2/l$ at $n_1 = 110$ and $p = 1/l$ at $n_1 = 122$. Or to express that in number of zeros: $n_0 = \{40, 30, 18, 6\}$. Note how these locations in the search space correspond exactly with where the different mutation rates stagnate in Figure 5.7. The progress rate calculation seems to confirm that these mutation rates are too high to get to the optimum reliably. The $(1, 5)$ strategy used in this experiment needs a lower mutation rate to keep the algorithm from throwing away good mutations. This corresponds with the 'Hidden Elitist Strategy' discussed in Section 5.9.

The calculation of the progress rate seems to confirm what the experiments in Section 5.10 and earlier in this chapter suggested: self adaptation in a genetic algorithm does not adapt to the best mutation rate for the algorithm. The mutation rate with the highest progress rate at different points in the search space corresponds with the performance of the different mutation rates in the experimental results, while the stagnations in the same results corresponds perfectly with locations where progress rate P is 0. Both approaches have the same conclusion: no mutation rates below $p = 0.5/l$ are optimal or needed for convergence. Why then does self adaptation go all the way down to values like $0.05/l$? The next Section aims to answer this question by investigating the individual's chance of survival and the way that impact mutation rates.

5.12 Calculating Survival

Now that it is clear that self adaptation adapts to the wrong value, it might be time to look at exactly what value it is asked to adapt to. Up until now the 'success rate' of the algorithm has been defined as *'the speed at which an algorithm gets closer to the optimum'* (or 'convergence speed'), 'success' from the point of the algorithm. Next this Section will try to define 'success' from the point of an individual. This can be better defined as *'the chance to be selected'*. All the individuals ever 'want' to do is increase the chance to survive and create offspring. At first glace the two definitions of success are identical or at least closely related. If the individual wants to be selected it needs to perform best in its generation, which means its offspring will perform similarly, but will need to outperform its siblings. All seems to point to the standard self-adaptive methodology that would only promote 'speed' in the algorithm, but there is a difference as will become clear later in this Section.

A genetic algorithm is very complex iterative loop in which many situations can occur, that is why the following calculations make two assumption:

- Two individuals that both have been successfully mutated have an equal opportunity to be selected.
- Also two individuals that both have been unsuccessfully mutated have an equal opportunity to be selected.
- The entire pool of individuals except the one we are calculating chances for, uses the same mutation rate.

Note that in the experiment the fitness of both individuals dictates the order in which they are selected. Because this Section is mostly investigating what happens with low mutation rates with a small pool size, these assumptions don't really impact the conclusions. Also note that if two individuals both do not change their fitness, they do have an equal opportunity to be selected, also in the experiment.

The state of the other individuals in the population is very important to the chance for an individual to be selected. This is the only time there is interaction between individuals of the same generation (apart from crossover). An individual might already have a fitness calculated before selection occurs, but selection puts this fitness in perspective. In a way, the fitness of an individual is totally unimportant to its survival, it is the 'relative fitness' that will determine the individual's fate.

There are many different states the pool of individuals can be in, but the assumptions made at the beginning of this Section simplify these down to a couple. To be able to calculate the chance to be selected, two mutation rates are defined: p_m^{ind} is the mutation rate of the individual we are calculating our chance for and p is the mutation rate used for all the other individuals in the pool. This makes it possible to say something about the direction self-adaptation would like to go in respect to a certain mutation rate in the rest of the pool. In the experiment every individual of course has their own mutation rate and simplifying this with just one rate will have a slight impact on the results, but given that p could be viewed as the 'average mutation of the rest of the pool', the impact should be minimal.

Using both p^{ind} and p different chances for successful mutation, identical fitness and unsuccessful mutation can be calculated similar to Section 5.11. The chances using p^{ind} will be denoted with $p^{ind}_{success}$, p^{ind}_{same} and $p^{ind}_{unsuccess}$ respectively, whereas the chances using p will be denoted in the normal way.

The chance for an individual to be selected can be split up into three different independent chances:

- The chance that the individual was successfully mutated and is selected ($p^{ind}_{success\_select}$).
- The chance that the individual has the same fitness as its parent, but is still selected ($p^{ind}_{same\_select}$).
- The chance that the individual was unsuccessfully mutated, but is still selected ($p^{ind}_{unsuccess\_select}$).

The chance that the individual was successfully mutated and is selected depends on the number of other individuals that were also successfully mutated. The following summation walks each different number of other successful individuals and takes into account the permutations possible in each case:

$$p^{ind}_{success\_select} = p^{ind}_{success} \cdot \sum_{i=0}^{\lambda-1} \left[\frac{1}{i+1} \binom{\lambda-1}{i} \cdot (p_{success})^i \cdot (1-p_{success})^{\lambda-1-i} \right]$$

Note how the division by $i+1$ takes care of the 'equal chance' to be selected from the group of successful individual.

A similar equation is needed to calculate the chance that the individual is selected while it has the same fitness as its parents denoted with $p^{ind}_{same\_select}$. The chances used are a bit different though. Instead of only caring about

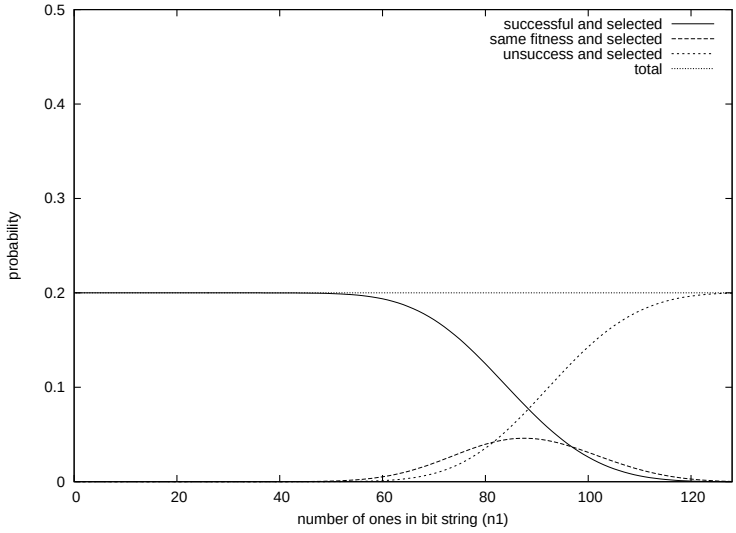

Figure 5.12: This figure shows the different probabilities of an individual to be selected in a population of $\lambda = 5$ for all different number of ones n_1 if all individuals get mutated with $p = \frac{8}{128}$. Note how even though for bigger n_1 the probability for selection on success goes down an the probability for selection when unsuccessful goes up, the total probability stays identical to $\frac{1}{\lambda}$, which makes sense because all individuals are mutated with the same mutation rate.

'success', this time no individual should have been mutated successfully. Otherwise that individual would have been selected instead of the one we are calculating this chance for, therefore p_{same} and $p_{unsuccess}$ are used to calculated the chance:

$$p^{ind}_{same\_select} = p^{ind}_{same} \cdot \sum_{i=0}^{\lambda-1} \Big[\frac{1}{i+1} \binom{\lambda-1}{i} \cdot (p_{same})^i \cdot (p_{unsuccess})^{\lambda-1-i} \Big]$$

In order to be selected after an unsuccessful mutation, all other individuals in the pool also need to have had unsuccessful mutations. Given the assumption that all unsuccessful mutation have an equal chance of being selected, the following simpler equation follows:

$$p^{ind}_{unsuccess\_select} = \frac{p^{ind}_{unsuccess} \cdot (p_{unsuccess})^{\lambda-1}}{\lambda}$$

Now the total chance for an individual to be selected is of course nothing more than the summation of the three chances above:

$$p^{ind}_{select} = p^{ind}_{success\_select} + p^{ind}_{same\_select} + p^{ind}_{unsuccess\_select}$$

Figure 5.12 shows the three chances and total plotted for the case where $p = p^{ind} = \frac{8}{l}$. Note that in this case the total chance of being selected is exactly identical to 0.2. This makes sense because the other individuals in the population have exactly the same mutation rate and therefore the same chance to be selected independent of n_1. Since $\lambda = 5$ this chance is 0.2. Also note where $p^{ind}_{success}$ intersects $p^{ind}_{unsuccess}$. This point corresponds exactly again with the location in the search space where $p = \frac{8}{l}$ stagnates in the experiments.

Now to find out at what point changing the mutation rate makes sense, figure 5.13 shows the three chances and the total plotted for the case where $p = \frac{8}{l}$ but $p^{ind} = \frac{4}{l}$. Note this plot is interesting for a lot of reasons, but mostly because it shows exactly where it is beneficial for an individual to go to a lower mutation rate and how much more beneficial this is. At $n_1 = 67$ p^{ind}_{select} crosses the 0.2 line and stays above that line for the rest of the graph. This means that if $n_1 > 67$ it is more beneficial for an individual to use $p = \frac{4}{l}$ than $p = \frac{8}{l}$. This is exactly the value that was found in Section 5.11 and seems to contradict the existence of a 'Couch Potato Effect'.

But when lower mutation rates are plotted, something unexpected happens: a lower mutation rate **always** seems to be better! Figure 5.14 shows what

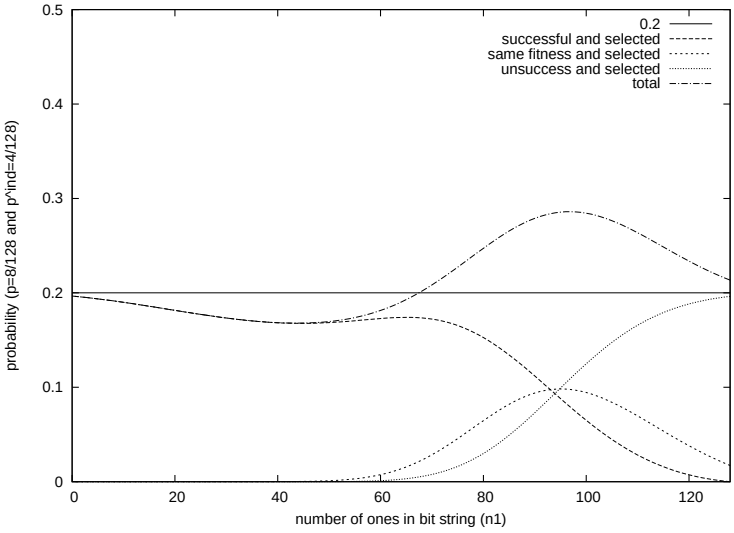

Figure 5.13: This figure shows the individual probabilities of an individual to be selected in a population of $\lambda = 5$ for all different number of ones n_1 if the individual gets mutated with $p = \frac{4}{128}$ and all other individuals get mutated with $p = \frac{8}{128}$. Note how for lower values of n_1 the probability of selection is lower than $\frac{1}{\lambda} = 0.2$, but for $n_1 > 67$ the probability of selection becomes higher than average (> 0.2) mainly due to the probability to be selected with an unsuccessful mutation.

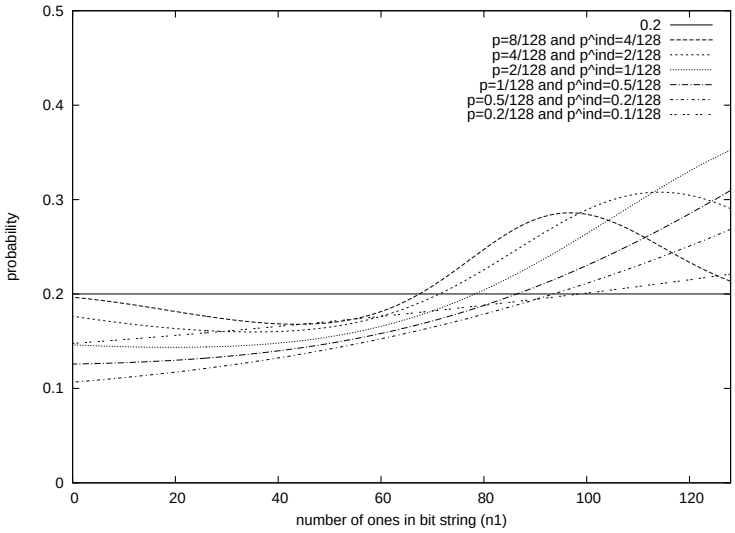

Figure 5.14: This figure shows the probability for an individual to be selected when $\lambda = 5$, $l = 128$, the individual is mutated with p^{ind} and all other individuals with p for all values of n_1 and mutation rates $(p, p^{ind}) = \{(\frac{8}{128}, \frac{4}{128}), (\frac{4}{128}, \frac{2}{128}), (\frac{2}{128}, \frac{1}{128}), (\frac{1}{128}, \frac{0.5}{128}), (\frac{0.5}{128}, \frac{0.2}{128}), (\frac{0.2}{128}, \frac{0.1}{128})\}$. Note how for each of these situations the lower mutation rate outperforms the higher mutation rate for bigger values of n_1. This is even true for the case where $p^{ind} = \frac{0.1}{128}$ and $p = \frac{0.2}{128}$ which implies that in a population with a mutation rate $p = \frac{0.2}{128}$ and $n_1 > 100$ lowering an individual's mutation rate gives that individual a higher chance to survive, even though previous results have shown that this decreases the convergence speed.

happens in each of the used mutation rates when the individual has the next lower mutation rate. So for $p = 8/l$ it shows $p^{ind} = 4/l$, for $p = 4/l$ it shows $p^{ind} = 2/l$ and so on until for $p = 0.2/l$ it shows $p^{ind} = 0.1/l$. Note that for each subsequent smaller mutation rate, the next smaller step seems to generate a higher selection chance around the area that mutation rate would be employed. This plot basically says: 'when $n_1 > 100$ a lower mutation rate is always better!' This is not at all what Section 5.11 predicted should happen to optimize the progress. It seems we have found our culprit for the premature slowdown of the GA. Somehow it is 'survival'.

To find the reason for this counter intuitive result figure 5.15 plots the three probabilities and total for the case where $p = 0.5/l$ and $p^{ind} = 0.1/l$. As reported above the chance for an individual to survive goes up dramatically for cases where $n_1 > 100$ when the mutation rate of that individual is much lower than the mutation rate of the individuals in the pool, but the figure shows a surprising origin of this chance. It is the chance to have 'the same fitness as its parent and being selected' p^{ind}_{same} that goes up drastically. Throughout the search space, but especially closer to the optimum, p^{ind}_{same} is the main reason that $0.1/l$ outperforms $0.5/l$.

With a mutation rate as low as $0.1/l$ the chance of 'mutating two or more bits in the bit string and getting the same fitness' is a lot smaller than the 'chance of changing no bits' (which automatically gives the same fitness). This means that p^{ind}_{same} mostly consist of 'the chance to stay in the same location and get selected', or 'selected clones'. As the plots show it is easier to clone individuals if the mutation rate is lower, which means that from the perspective of an individual, in order to have the highest chance to survive it needs to try and stop moving, stay where it is, and at least it will not loose the fitness its parent gave to it.

5.13 Conclusions

This chapter shows how self-adaptive mutation rates have been successfully applied to the inverse design of transition rules for Cellular Automata. The mutation rates adapted from an initial very high rate to a level that was usable to run the Genetic Algorithm.

Self-adaptive mutation rates as it was applied here however, did have some problems with the complexities of the search space of the Majority Problem. In particular a premature convergence was measured that did not seem to be the effect of any speed setting of the algorithm or noise level of the fitness function.

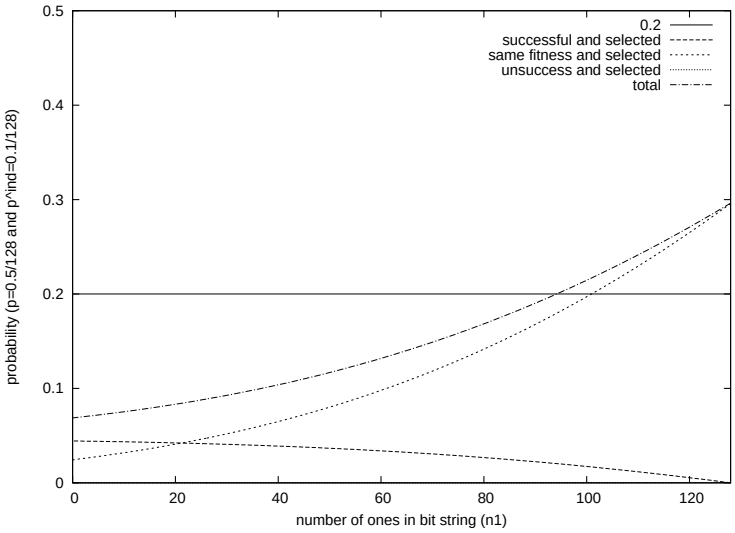

Figure 5.15: This figure shows the different probabilities for an individual to be selected when $\lambda = 5$, $l = 128$, the individual is mutated with $p^{ind} = \frac{0.1}{128}$ and all other individuals with $p = \frac{0.5}{128}$ for all values of n_1. Note how this clearly shows that for $n_1 > 100$ the probability for selection is above average which implies that lowering the mutation rate in a population with $p = \frac{0.5}{128}$ is increasing the probability of survival for the individual. Also note how for $n_1 > 100$ the biggest part of this probability comes from the probability to be selected when the individual's fitness is unchanged. This implies that self adaptation will keep on decreasing the mutation rate and the GA will stagnate.

A second experiment was introduced to try and explain the effect measured in the first. The MAXONE Problem was used to show how even in a much simpler experiment the same premature convergence is possible. This stagnation was compared to fixed mutation rate runs and shown to be caused by a premature drop in the mutation rate by the self adaptation.

The progress rate for the MAXONE Problem was calculated for different fixed mutation rates and fit the experiments exactly. The self-adaptive mutation rate dropped way below the calculated optimal mutation rate proving self-adaptation to stagnate prematurely without any local optima.

Next the chance of survival for a single individual was calculated and that showed a totally different picture. The calculations showed that, if the success rate is below a certain point, it is much more beneficial for an individual to stay unchanged. This result proves the existence of a stagnating force in self adaptation in GA and shows once more how complex the interactions between individuals in an Evolutionary Algorithm can be.

This effect of 'survival of the laziest' deserves the name 'Couch Potato Effect'. It is not unthinkable (although not proved) that this effect caused the premature convergence in the inverse design experiment earlier in the chapter and it is very likely that a similar force will have a substantial effect everywhere self-adaptation is used in genetic algorithms. All it needs is a comma (or 'non-elitist') strategy, a low success rate and a chance to 'stay put'. This research implies that a similar effect is likely in any application of self-adaptive mutation rates in Genetic Algorithms.

Further research is needed to understand how this relates to studies of premature convergence in Evolutionary Strategies in difficult topologies. Looking at a way to map the binary search space used in GA to a simplified real valued search space with the same dimensionality in an ES could help here. It might also be valuable to investigate the difference between stagnation of local optima and stagnation on high dimensional rough functions. What are the similarities and what are the differences?

As of yet our research has not yielded any counter measure for the Couch Potato Effect and further research into this area seems warranted. Possible counteractions could include: increasing the population size (which should make the effect smaller), using a plus strategy (which might pose a problem for self-adaptation when selecting parents of previous generations) or maybe even change the selection method to never include the 'Couch Potatoes' (which might destroy self-adaptation all together). All approaches that need further study, but non of which seem immediate answers to this easily underestimated problem of the global behavior of an evolutionary loop based on the local interactions of its individuals.

Chapter 6

On Interactive Evolutionary Strategies

In this chapter we discuss evolution strategies in the context of interactive optimization. Different ways of interaction will be compared and classified. A focus will be on the suitability of the approach in cases, where the selection of individuals is done by a human expert based on some subjective criteria. First of all, this chapter will study the behavior of the step-size adaptation mechanism, which might be seen as the most distinguishing feature of evolution strategies as compared to other evolutionary algorithms. Moreover, we compare the convergence dynamics of different approaches, and discuss typical patterns of user interactions as observed in empirical studies.

The discussion of empirical results will be based on a survey conducted via the world wide web. Color (pattern) redesign problems were chosen as test case. The simplicity of the chosen problems allowed us to let a large number of people participate in our study. The amount of data collected made it possible to add statistical support to our hypothesis about the performance and behavior of different interactive evolution strategies, and last but not least helped us to figure out high-performing instantiations of the approach.

6.1 Introduction

The research field of human-algorithm interaction (HAI) puts forward the involvement of human beings in the algorithmic solution processes. In contrast to human computer interaction the focus of this technology is on computational processes that are assisted by users. In contrast to interactive software like text processing systems or drawing software, the main structure of the solution process for the higher level task is still governed the algorithm, The user has to assist the algorithm at some stages, that call for decisions based on subjective preferences, or that require the insights of experts in a problem, the formalization of which is often very difficult.

On a very global level we propose to distinguish between *reactive* or *proactive* interaction, i.e. user feedback requested by the algorithm, or optional interventions by the users into an autonomously running algorithm. An example for reactive feedback could be the request of an optimization algorithm on the subjective evaluation of solutions by means of the user. Contrarily, an example for proactive feedback would be given, if the user halts an optimization algorithm that is in a phase of stagnation, changes some parameters, and lets it continue with the changed settings. A boundary case for proactive feedback, would be, if the user simply decides to finish an algorithm and pushes some 'stop' button that terminates it.

Among the few algorithmic classes that already integrates the user in the computational process, interactive evolutionary algorithms are one of the most well known. Applications range from arts [2] and music [23], to industrial engineering applications [31, 13], mixture optimization [21], and prototyping in product design [8]. An excellent overview on applications of IEA was given by Banzhaf [8] and more recently by Takagi et al. [40].

In this chapter we will mainly focus on the discussion of interactive variants of evolution strategies (ES) [39, 34]. ES are instantiations of evolutionary algorithms that are mainly used for the purpose of parameter optimization. In particular they feature self-adaptive parameters of the stochastic distribution used in the mutation. This allows to minimize the effort of the user when working with these evolutionary algorithms, as for many other EA the choice of the adequate parameters can cause a significant problem for the unexperienced user. Moreover, the self-adaptation makes it possible to automatically scale the behavior of the variation operator between a more exploratory coarse sampling or a finer sampling, which is needed to achieve a high approximation accuracy to the optimum in the end.

Evolution strategies have been already successfully applied for interactive optimization in parametric design. In particular the pioneering work of

Herdy [20, 21] in this field should be mentioned here, who applied interactive variants of the evolution strategy to various problems ranging from the design of color mixtures to the search for coffee mixtures that meet a desired taste.

However, we believe that there are still many open questions with regard to interactive evolution strategies. For example the step-size adaptation deserves further attention, and the 'typical' behavior of the user. Moreover, it is an interesting question how a theory of interactive evolution strategies might look like. In this chapter, we intend to provide contributions to these questions. In particular we discuss new methods of how to conduct research in interactive evolution strategies, analyze the user behavior in the selection process and study the feasibility of the self-adaptive step size adaptation within this context.

We will base our discussion on a representative problem for interactive evolution. The problem we have chosen is the *re-design of RGB colors* by means of subjective evolution. The problem can be easily increased by using color patterns instead of single patterns. Moreover, it can be easily explained to people participating in experimental studies, and thus can be readily used for collecting statistical data in math experiments. In the studies presented in this chapter, a survey conducted via the world wide web served us to gain a larger amount of data, which allowed us to better support the hypothesis about the algorithms' behavior and performance.

As it can be concluded from this introduction, the contribution of this chapter is not only to be seen as a presentation of new empirical results, but also to a fair extend as a discussion of a general research methodology in the field of interactive evolutionary algorithms.

The structure of this chapter is as follows: After a short introduction to evolution strategies (Section 6.2) we will present a survey on interactive evolution strategies (Section 6.3). We continue with a discussion of self-adaptive features 6.4 in evolution strategies and discuss their role in interactive evolution strategies. Finally, in Section 6.5, we report on first statistical studies of self-adaptive IES on color (pattern) redesign problems. The chapter concludes (Section 6.7) with a summary of first results and an outline of some open questions for future research.

6.2 Evolution Strategies

The main loop of the evolution strategy is displayed in algorithm 1. Firstly, the algorithm initializes a population (multi-set) P_0 of μ individuals (objec-

tive function vectors and mutation parameters) randomly (e.g. uniformly distributed within the parameter space). P_0 forms a starting populations, and within the subsequent *generational loop* a series of new populations $(P_t)_{i=1,2,...}$ is generated by means of a stochastic process:

Algorithm 1 (μ, κ, λ)-Evolution Strategy

1: $t \leftarrow 0$
2: $P_0 = \mathbf{init}_\mu()$
3: **while terminate**$() = false$ **do**
4: $\quad Q_t \leftarrow \text{generate}_{\mu\rightarrow\lambda}(P_t)$ /* Generate λ new variations based on P_t */
5: $\quad P_{t+1} \leftarrow \text{select}^{\kappa}_{\lambda\rightarrow\mu}(P_t \cup Q_t)$ /* Select μ 'best' individuals */
6: $\quad t \leftarrow t + 1$
7: **end while**
8: **return** P_t

First λ random variations of individuals in P_t are generated by means of a variation operator, the details of which will be described later. The new variants form the population Q_t, the so-called offspring population. Then, among all individuals in P_t and Q_t the best μ individuals are selected by means of a selection criterion. The parameter κ stands for the maximum number of generations a parent of a previous generation can be selected as a parent for this generation. Note that this means that if $\kappa = \infty$ the algorithm is a pure 'plus strategy' and all μ parents can be selected for the next generation, while if $\kappa < \infty$ only those individuals are taken into account that have been generated in an iteration t_0 with $t_0 > t - \kappa$ and if $\kappa = 0$ the algorithm is a pure 'comma strategy'. The selection process is usually governed by an objective function $f : \mathbb{I} \rightarrow \mathbb{R}$, i.e. the μ best solutions with regard to this function are selected. However, it is not always necessary to have an objective function, and it suffices to establish a ranking on the merged population or just to have some criterion that can extract the μ best solutions from all other solutions.

Note, that the stochastic process defined by the series $(P_1)_{i=1,2,...}$ has the Markov property, if we consider the selection criterion to be fixed. This means that the stochastic distribution of P_{t+1} is determined by the population P_t.

The variation-selection process is meant to drive the populations into regions of better solutions as t increases. However, there is no criterion that can be used to determine whether the best region has been found (except in cases with a pre-defined goal or bound on the objective space). Hence the process is usually terminated in case of stagnation or if the user decides to stop it, because of his/her time constraints.

An operator that deserves some further attention in the evolution strategy is the variation operator that is used to generate offspring. Let us first repeat that individuals within the evolution strategy consist of a vector of decision variables $\mathbf{x} = (x_1, \ldots, x_{n_x}) \in \mathbb{R}^{n_x}$ and a vector of parameters of the mutation distributions (often referred to as step-size vector) $\mathbf{s} = (s_1, \ldots, s_{n_s}) \in \mathbb{R}^{n_s}$.

A variation of this vector is generated via a mutation of the step-size vector and the subsequent mutation of the vector of decision variables using the new mutation parameters. As an example let us discuss the so called 3-point mutation operator first introduced by Rechenberg [33, 34], that works with a single step-size, i.e. $n_s = 1$ is described in algorithm 2.

Algorithm 2 Generate λ offspring via 3-point mutation

$Q = \emptyset$
for $i \in 1 \ldots \lambda$ **do**
  choose $(\mathbf{x}, \mathbf{s})$ randomly out of P_t
  $u \leftarrow \text{uniform}(0, 1)$ // uniformly distributed random number between 0 and 1
  $$s_1' \leftarrow \begin{cases} s_1\alpha & \text{if } u < \frac{1}{3} \\ s_1/\alpha & \text{if } u > \frac{2}{3} \\ s_1 \text{ otherwise} \end{cases}$$
  for $j \in \{1, \ldots, n_x\}$ **do**
    $x_j' = x_j + s_1 \text{normal}(0, 1)$
    /* normal(0,1) generates standard normal distributed random number */
  end for
end for
$Q = Q \cup \{(\mathbf{x}', \mathbf{s}')\}$

In order to generate a new individual, first the step-size of the given individual is multiplied by a constant factor, the value of which can be 1, α or $1/\alpha$ depending on a random number. Then this step-size of the new individual is used to obtain the decision variables of the new individual. These are obtained by adding an offset to the corresponding value of the original individual. The value of this offset is determined by a random number that is gaussian distributed with mean value 0 and standard deviation s_1. The idea behind this mutation operator is that decision variable vectors that are generated with a favorable step-size are more likely to be part of the next generation, and thus also the information about the step size that was used to generate them is transferred to that generation. The process of mutative step-size adaptation was investigated in detail by Beyer et al. [10]. Due to his findings, simple adaptation rules like the 2-point or 3-point mutatation

for the step sizes serve well, if the population is small and only a few iterations of the algorithm can be afforded. For a higher number of iterations, say $t_{max} >> 100$, more sophisticated adaptation mechanisms should be considered for the parameters of the mutation. State of the art techniques include the individual step size adaptation by Schwefel [39] and the covariance matrix adaptation (CMA) by Hansen and Ostermeier [19]. Note, that in order to allow for a mutative step-size adaptation, a surplus of offspring individuals needs to be generated in each generation. The recommended ratio of $\mu/\lambda \simeq 1/7$ leads to a good average performance of evolution strategy in many cases [39].

6.3 Interactive Evolution Strategies

There are many possibilities to integrate user interaction in the evolution strategy. In general, we can distinguish between reactive and proactive feedback. Reactive feedback is feedback requested by the algorithm, e.g.

- the user might be asked for evaluation (grading) of offspring individuals
- the user is asked for selecting individuals
- the user is asked for generating variants

In contrast to this, proactive feedback denotes an optional intervention by the user, e.g.:

- he/she might change the step-size parameter actively, after watching the search process stagnate)
- he/she might insert a new individual into the population or actively change the parameters of an individuals (manual mutation)

In this chapter we are more interesting in strategies with reactive feedback and the only proactive feedback will be given, when the user decides to stop the search process.

Probably the most simple form of reactive feedback that might be given by the user is to simple select the best individual(s) out of a population, as suggested by Herdy [20]. A more complicated scheme of subjective evaluation would be a grading procedure, where the user has to provide a grade to each individual. Note, that the information the user has to provide in each iteration, consists of λ numbers in case of grading, and μ numbers, namely the numbers of the best solutions, in case of selecting the best variant. As $\lambda \gg \mu$, the latter seems to be a favorable choice.

A selection procedure following this strategy is described by the simple algorithm:

- The algorithm presents the λ new solutions from Q_t and the μ solutions from P_t that have not exceeded a maximal age of κ generations to the user.
- The user decides which one of them are the best μ solutions and these form the new population.

A theoretical analysis of such kind of processes involves different kind of complications. First of all we need to find an adequate model. A model that is frequently used for the analysis of evolution strategies is that of a Markov chain. A Markov chain can be viewed as an autonomous stochastic automaton $(S, Pr : S \times S \to [0,1])$, where S denotes a state space, and Pr denotes a function that assigns a probability to each state transition, such that $\forall s \in S : \sum_{s' \in S} Pr(s, s') = 1$. By setting $S = \mathbb{I}^\mu$ evolutionary algorithms on a finite search space can be modeled as Markov chains. This allows to obtain results about the limit behavior and average behavior on some test problems.

It was suggested by Rudolph [35], to extend this model to a stochastic mealy automaton with deterministic output, in order to model interactive evolution strategies. Such an automaton would be denoted with $(S, X, Pr : S \times S \times X \to [0,1])$, where X denotes a set of input symbols. Now, the probability function $Pr(S, X, Pr : S \times S)$ assigns a probability value to each state, input pair $(s, x) \in S, X$. Accordingly, the function $Pr : S \times X \to S$ must obey $\sum_{(s,x) \in S \times X} Pr(s, x, s') = 1$.

An interesting observation is now, that given a stream of inputs, the behavior of this strategy breaks down to a Markov chain, whereby the input stream becomes part of the deterministic formulation of Pr. This, for example, provides us with a means to analyze the best case behavior of a strategy for some target function f, by minimizing the mean convergence time t_{conv} over all possible user inputs:

$$E(t^*_{conv}(f)) = \min_{\mathbf{w} \in X^*} E(t_{conv}(\mathbf{w}, f)) \tag{6.1}$$

This convergence time of an 'ideal user' could be compared to the real behavior of a user in order to assess the performance of the user interaction and judge if this is the weak point of the algorithm. If so, further measures to support the user interaction might be considered, like user modeling, a better presentation of the variants or even the request of further information from the user.

However, even for quite simple variant of interactive evolution strategies, the computation of an ideal user behavior might be a challenging task. From a more abstract point of view, the input of the user in an interactive evolution strategy with subjective selection is a set of disjoint indices $\{i_1, \ldots, i_\mu\} \subset \{1, \ldots, \mu + \lambda\}$. Accordingly, he/she has $\binom{\lambda+\mu}{\mu}$ possibilities of choice, the value of which reduces to λ in case of a 'comma' strategy with $\mu = 1$. Hence, in case of t time steps there are already λ^t possible input streams that need to be considered. Hence, without any simplifying assumptions the determination of the ideal user behavior will be intractable, even in cases where $E(t_{conv}(\mathbf{w}, f))$ can be obtained, the computation of which might in itself also cause severe computational effort.

In summary, it seems that only in a few, very simple cases it will be possible to get meaningful results from a convergence theory of interactive evolution strategies and empirical results will likely play an important part in the dynamic convergence theory of these algorithms, even if we assume the 'ideal user'.

6.4 Self-adaptation and Interaction

One of the main research questions addressed in this chapter is, whether self-adaptive mechanisms of the ES can be utilized also for the interactive ES. With regard to this, there are some important differences between the standard ES and the interactive ES.

First of all, for the standard ES in continuous spaces, the precision of an optimum approximation can, in principle, get arbitrarily close. In applications of the interactive evolution strategy, the subjective nature of the objective function usually prevents an arbitrarily close approximation of some solution. The reason for this is that in many cases the user will not be able to measure arbitrarily small differences in quality. For example, when comparing two colors, a human user will perceive two colors as equal if their distance is below a *just noticeable difference* (JND). The concept of JNDs is quite frequently discussed in the field of psycho-physics, a subbranch of cognitive psychology [25]. It is notable, that the JND depends on the intensity and complexity of the stimulus presented to the user. Moreover, it has been found that the lower the difference between two stimuli and the more complex the stimuli are, the more time it takes for the user to decide upon the similarity of two patterns. We will come back to this result, when we discuss the empirical results of our experiments.

Another difference between the standard ES and the ES with subjective se-

lection criterion is that the user's attention level will decrease after a while. For a theory of attention we refer to the work of [32]. Hence, the number of experiments is usually very limited and very fast step-size adaptation mechanisms have to be found, and only a few parameters of the mutation distribution can be adapted.

Moreover, the amount of interaction should be minimized, e.g. by choosing a simple selection scheme. This might prevent the use of step-size adaptation strategies that demand for numerical values of the fitness function value. A performance measure would be based on the number of selections made by the user, rather than on the number of function evaluations, and even the time spend on the selection needs to be considered.

Taking all the above into consideration, the experiments discussed below are designed to be short in duration and minimal in the amount and complexity of interactions needed by the user. Yet in all experiments the user is the only selection operator in the algorithm.

6.5 A color redesign test-case

To study the effect of a human as fitness function and selection mechanism, a small experiment was constructed. An evolutionary algorithm was implemented in the form of a JAVA applet for the simple problem of finding the RGB values of a certain color or combination of colors (see figure 6.1). In this experiment the user selects one color or color pattern out of several alternatives that is closest to a given target color. This selection is then used as a parent for the next generation of alternatives that the user can choose from. When the user thinks the algorithm will not improve the results any more he can choose to stop the search by clicking the 'Done' button. All data collected in this applet is then send to a database and can be used for this research.

A one dimensional experiment was run using squares with only one color and a two dimensional experiment was done by having a left and a right color in the same square. (see figure 6.1) Comparing these two experiments might give insight into the scalability of this type of experiment.

The whole LIACS faculty at the University of Leiden was asked to help with this experiment by running this applet in a web browser. About 200 runs were collected this way and the findings in the chapter are based on these runs. This experiment was then also run using an ideal user in the form of a computer program that would always select the color with the smallest Euclidian distance to the target color. Having data on both a deterministic

selection compared to a human selection mechanism gives some insight into how ideal a normal user is and how an evolutionary algorithm reacts on this difference.

Two different algorithms were used. One with a fixed step size and one with an self adapting step size. Three different step sizes were used in the fixed algorithm: 10, 20 and 40. Note that RGB values can range from 0 to 255, that makes the relative step sizes approximately 0.039, 0.078 and 0.156. For the self adaptation the Rechenberg algorithm with three groups is used with $\alpha = 1.3$. That means that every generation three different groups of offspring is generated, one with a step size equal to the parent, one with a step size 1.3 times larger and one with a step size 1.3 times smaller. In the one dimensional algorithms a population size of 9 was used, in the two dimensional this was increased to 16 so that the user might have more alternatives to choose from with the harder problem.

When the applet is started an algorithm is randomly selected for that run. The user will only know what algorithm was selected after the experiment so that the user will not be influenced by that knowledge. The random generation was done in such a way that 50% of the runs were done with the Rechenberg algorithm and 16.667% of the runs for everyone of the three fixed step size algorithm.

The target color was fixed in all the experiments to make them comparable and was chosen to be $[R = 220, G = 140, B = 50]$ so that every component of the color was not 0 or 255 and not equal or close to any other component. This happened to be a brownish version of orange.

6.6 Results

First results we obtained from our internet application are displayed in figures 6.2 and 6.3. Next, we will discuss these results one by one.

Figure 6.2 shows the average function values for the different strategies. Note that not all runs had the same length. Some people put more effort into finding a better result than others. This is all part of the effect of using humans instead of a computer. This does influence the quality of the average value and conclusions based on this data should take this into account.

The one dimensional plot in figure 6.2 shows that self adaptation seems to outperform all the other algorithms, with human selection as well as with

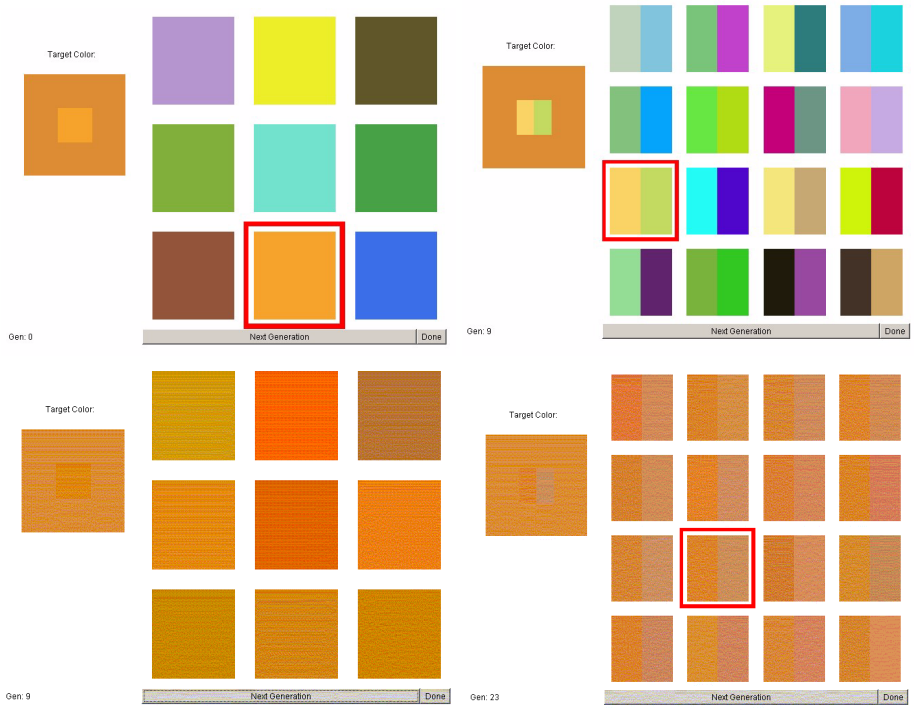

Figure 6.1: Subjective selection dialogue with user: The upper figures show the initial color patterns (single color (l) and two-color test case (r)) and the lower figures show color patterns at a later stage of the evolution. The bigger box on the left hand side displays the respective target color, and in its center a small box is placed displaying the selected color. Once the user presses the NEXT bottom a selection gets confirmed and a new population will be generated and displayed. If the user is finally satisfied with the result he/she presses the Done button, in order to stop the process.

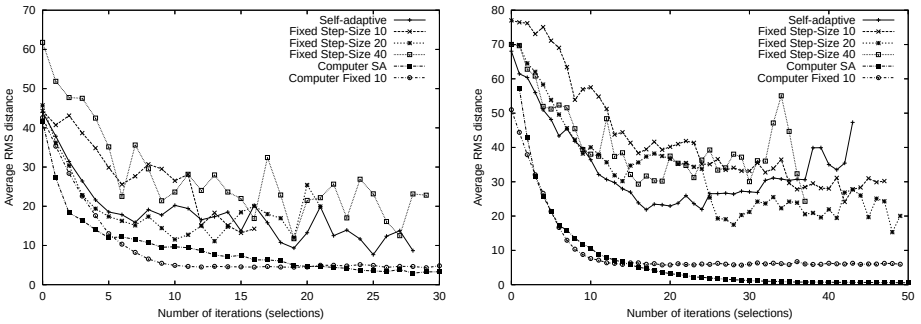

Figure 6.2: The convergence behavior of different ES obtained in the online experiment. The left figure displays results for a single RGB color, and the right figure shows the results for two different RGB colors. Each line is an average of all the runs with these settings.

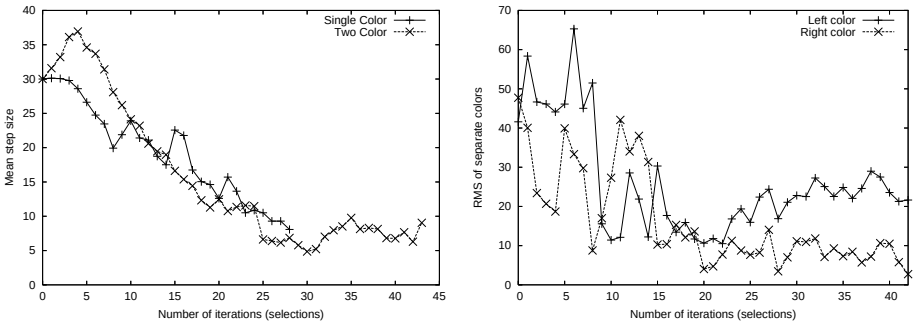

Figure 6.3: The left plot show step-size adaptation in interactive evolution strategy for single color and two-color example. The right plot displays the history of a typical run of the two color problem. The distance of the two colors to the target color is displayed over the number of iterations. Note the divergence of the left color after 20 iterations.

the computer selection. The fixed algorithm with step size 20 seems to converge a lot faster in the beginning though and stagnate after 10 iterations. This effect is also very plain with the computer selection, there the fixed step size outperforms self adaptation until generation 10 where self adaptation closes the gap and overtakes. The computer selection was also run on the three different fixed step sized, but step size 10 was by far the best in both the one dimensional and the two dimensional case.

The two dimensional plot is a bit different though. Here a fixed step size of 20 seems to be the better choice. The self adaptive runs start out the best here but after 20 iterations the step size seems to increase for convergence stagnates. This is a weird effect that only seems to occur with the self-adaptation in the two dimensional case if human selection is used. In the case of the computer selection this effect is totally absent and self-adaptation outperforms all the fixed step sizes.

In an effort to explain this effect figure 6.3 (right) shows the error of both colors separately in a typical run of self adaptation using human selection. Note how both errors seem to converge nicely up until generation 20. From that point onward only the right color seems to converge to an optimum whereas the left color is actually moving away from the optimum. This suggests that this shows how a human might try to use a certain strategy to optimize two dimensions. In this case it seems the user tried to optimize the right color first and worry about the left color later. Although this seems a feasible strategy, the self-adaptive algorithm has some problems with that. The total error goes up, the step size is stagnating and even increasing a bit (as figure 6.3 (left) shows).

What figure 6.3 also shows is that the step sizes are decreasing at the same rate in the one dimensional problem as they do in the two dimensional problem. It seems though that in the two dimensional problem the step size starts off a bit higher.

The fact that the computer selection outperforms the human selection is not very surprising. The human fitness and selection method seems to be very non-deterministic and could be viewed as a noisy fitness function. But there seems to be more to it than just adding noise to the fitness function. The results suggest humans use strategies that are based on some outside knowledge or even feelings that influences self adaptation in an unfavorable way.

6.7 Conclusion

This chapter contributed to the analysis of interactive Evolution Strategies (IEA) with subjective selection criteria. The approach has been related to the context of human algorithm interaction. Differences between interactive evolutions to non-interactive ones were pointed out, including a discussion of different ways to analyze these methods.

A main focus of our research was to find out if the feature of self-adaptive step sizes, that is essential to evolution strategies, works also within the context of IEA. A color (pattern) re-design example was utilized for empirical research and by means of an online experiment a significant amount of data was gathered.

The results clearly indicate the benefit of the step-size adaptation. Strategies that work with step-size adaptation turned out to be more robust and diminish the risk to choose a completely wrong step size. Yet the results also show that human selection can not be treated as just another noisy function.

An interesting insight we got from the results was that for more complex target definitions (2 color example) the user starts to use a strategy, e.g. to first optimize the first color and then the second. Such kind of user behavior has rarely been addressed in the context of interactive evolutionary algorithms and deserves further attention.

Chapter 7

Summary and Conclusion

This chapter gives an overview of all the important conclusions in the thesis chapter by chapter. For a more in depth discussion on these conclusions see the chapters themselves.

Chapter 4 shows how Genetic Algorithms can be used to evolve transition rules for Cellular Automata. It shows how these Cellular Automata clearly exhibit interaction and how this implies that the interaction is evolved using nothing but the problem definition. A generic approach is introduced to inversely design local rules with the aim to find global behavior. This approach is shown to be robust in terms of algorithm parameters and flexible in terms of Cellular Automata topology.

The approach was applied on the Majority Problem in different topologies. Different offsets between neighbors in a one dimensional neighborhood were tested. Results show how spreading out the neighborhood in an exponential way seems to improve the performance of the algorithm. Two new distance metrics were introduced: 'maximum distance' and 'average distance'. Results suggest that these metric between cells can be used to understand the performance differences between the experiments.

Different dimension CA were tested using the Majority Problem. The results show three dimensional CA outperforming two dimensional and two dimensional outperforming one dimensional CA. Even though the CA size had to be increased to run the Majority Problem on a multi dimensional CA and the transition rules in the two dimensional experiment were $\frac{1}{4}$ the length of that of the one dimensional version, the problem seemed to be

easier to solve for multi dimensional CA.

The interactions within the CA were investigated and although it is clear that interactions are there it is hard to visualize these interactions in the Majority Problem especially in the multi dimensional CA. That is why the more constrained AND and XOR problems were introduced. Experiments were conducted to evolve transition rules that mimic the behavior of logical AND and XOR ports. Results show how the use of both the von Neumann and the Moore neighborhood successfully resulted in rules that solved these problems. The Moore neighborhood outperformed the von Neumann neighborhood and was able to generate a few rules that solved the problem of filling the CA with the right answer for a full 100%. The resulting time plots show a very clear 'particle' movement and interaction. Proving that very distinct interaction on a local level was evolved using the only the global desired behavior.

Solving the Checkerboard Problem was another example of the flexibility of the approach and resulted in some nice examples of evolved interactions on the one and two dimensional plots. Most successful transition rules seem to use a 'rubber band particle' to decide which checkerboard pattern is going to be used globally. This rubber band then tries to shrink and reduce the area inside of it until it evaporates. Rules generated using a small 13×13 CA also work on a larger 50×50 CA proving that the approach is capable of evolving generic rules invariant in the size of the CA.

The Bitmap Problem tested the limits of the approach. Transition rules to generate multiple different bitmaps were evolved. Symmetric bitmaps seem easier to evolve than non symmetrical ones, but for all 5×5 bitmaps successful rules were found. This shows that the approach can solve many different problems and mimic many different behaviors.

Not only has this chapter researched a generic way of inversely designing transition rules for Cellular Automata using a Genetic Algorithm, it has also shed some light on the Evolution of Interaction. Not only showing that it is possible, but also how it is possible and which criteria have an effect on the performance of the evolution. These findings might be valuable to research into the evolution of language in humans and animals, while at the same time having applications in the field of Grid Computing, Image Processing, various Biology research areas and of course in the fields of Cellular Automata and Evolutionary Algorithms itself.

Chapter 5 shows how self-adaptive mutation rates have been successfully applied to the inverse design of transition rules for Cellular Automata. The mutation rates adapted from an initial very high rate to a level that was usable to run the Genetic Algorithm.

Self-adaptive mutation rates as it was applied here however, did have some problems with the complexities of the search space of the Majority Problem. In particular a premature convergence was measured that did not seem to be the effect of any speed setting of the algorithm or noise level of the fitness function.

A second experiment was introduced to try and explain the effect measured in the first. The MAXONE Problem was used to show how even in a much simpler experiment the same premature convergence is possible. This stagnation was compared to fixed mutation rate runs and shown to be caused by a premature drop in the mutation rate by the self adaptation.

The progress rate for the MAXONE Problem was calculated for different fixed mutation rates and fit the experiments exactly. The self-adaptive mutation rate dropped way below the calculated optimal mutation rate proving self-adaptation to stagnate prematurely without any local optima.

Next the chance of survival for a single individual was calculated and that showed a totally different picture. The calculations showed how that, if the success rate is below a certain point, it is much more beneficial for an individual to stay unchanged. This result proves the existence of a stagnating force in self adaptation in GA and shows once more how complex the interactions between individuals in an Evolutionary Algorithm can be.

This effect of 'survival of the laziest' deserves the name 'Couch Potato Effect'. It is not unthinkable (although not proved) that this effect caused the premature convergence in the inverse design experiment earlier in the chapter and it is very likely that a similar force will have a substantial effect everywhere self-adaptation is used in genetic algorithms. All it needs is a comma (or 'non-elitist') strategy, a low success rate and a chance to 'stay put'. This research implies that a similar effect is likely in any application of self-adaptive mutation rates in Genetic Algorithms.

Further research is needed to understand how this relates to studies of premature convergence in Evolutionary Strategies in difficult topologies. Looking at a way to map the binary search space used in GA to a simplified real valued search space with the same dimensionality in an ES could help here. It might also be valuable to investigate the difference between stagnation of local optima and stagnation on high dimensional rough functions. What are the similarities and what are the differences?

As of yet our research has not yielded any counter measure for the Couch Potato Effect and further research into this area seems warranted. Possible counteractions could include: increasing the population size (which should make the effect smaller), using a plus strategy (which might pose a prob-

lem for self-adaptation when selecting parents of previous generations) or maybe even change the selection method to never include the 'Couch Potatoes' (which might destroy self-adaptation all together). All approaches that need further study, but non of which seem immediate answers to this easily underestimated problem of the global behavior of an evolutionary loop based on the local interactions of its individuals.

Chapter 6 not only gave a brief overview and definition of the field of Interactive Evolution Strategies, but it also contributed to this field by applying self-adaptive mutation rates to a simple IES. A color (pattern) re-design example was utilized for empirical research and by means of an online experiment a significant amount of data was gathered.

It clearly showed the benefits of step-size adaptation in that it seems more robust and diminishes the risk to choose the wrong step-size. Yet the results also show that human selection can not be treated as just another noisy function.

More importantly the results showed that employing a human as a selection function can be a very powerful way to incorporate expert knowledge into an evolutionary process, but it can also be very unpredictable. In the two color experiment for instance, many users started to match one color first before thinking about matching the other color. This had unpredicted effects on the self-adaptation and these need further attention.

Interacting with an Evolutionary Algorithm is a very powerful way to incorporate the knowledge of an expert into the search capabilities of the computer, but it is not an effortless fusion. Many evolutionary processes and operators designed for Evolutionary Algorithms are designed with predictable selection and fitness evaluation in mind. A human is not that predictable and certainly not deterministic in nature. Generating more robust algorithms and interfaces to close the gap between the human and the machine seems a worthwhile effort and IEA looks to be at the forefront of that work. The robustness and ease of use of self-adaptation seem prime reasons to consider it a good candidate for real world applications in the future.

By showing three different ways of combining Interaction with Evolution this thesis has demonstrated the power and flexibility of Evolutionary Algorithms, while at the same time introducing some new and interesting findings in the fields of Inverse Design of Cellular Automata, Self-Adaptation in Genetic Algorithms and Human Algorithm Interaction.

Appendix A

Generalizing Multi Dimensional Cellular Automata

In the introduction chapter to Cellular Automata (Chapter 3) a generalization to multi dimensional CA is given in which the von Neumann and Moore neighborhoods are defined for different radii r and dimensions d. Some calculations in that chapter to calculate the size of these neighborhoods were omitted for readability. This appendix lists these calculations.

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

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

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

$$
\begin{aligned}
S^N(5,r) &= 2\Big[\sum_{i=0}^{r} S^N(4,i)\Big] - S^N(4,r) \\
&= 2\Big[\sum_{i=0}^{r} \frac{2}{3}i^4 + \frac{4}{3}i^3 + \frac{10}{3}i^2 + \frac{8}{3}i + 1\Big] - \frac{2}{3}r^4 - \frac{4}{3}r^3 - \frac{10}{3}r^2 - \frac{8}{3}r - 1 \\
&= 2\Big[\sum_{i=1}^{r} \frac{2}{3}i^4\Big] + 2\Big[\sum_{i=1}^{r} \frac{4}{3}i^3\Big] + 2\Big[\sum_{i=1}^{r} \frac{10}{3}i^2\Big] + 2\Big[\sum_{i=1}^{r} \frac{8}{3}i\Big] + 2r + 2 \\
&\qquad -\frac{2}{3}r^4 - \frac{4}{3}r^3 - \frac{10}{3}r^2 - \frac{8}{3}r - 1 \\
&= \frac{4}{3}\Big[\sum_{i=1}^{r} i^4\Big] + \frac{8}{3}\Big[\sum_{i=1}^{r} i^3\Big] + \frac{20}{3}\Big[\sum_{i=1}^{r} i^2\Big] + \frac{16}{3}\Big[\sum_{i=1}^{r} i\Big] \\
&\qquad - \frac{2}{3}r^4 - \frac{4}{3}r^3 - \frac{10}{3}r^2 - \frac{2}{3}r + 1 \\
&= \frac{4}{3}\Big[\frac{1}{5}r^5 + \frac{1}{2}r^4 + \frac{1}{3}r^3 - \frac{1}{30}r\Big] + \frac{8}{3}\Big[\frac{1}{4}r^4 + \frac{1}{2}r^3 + \frac{1}{4}r^2\Big] \\
&\qquad + \frac{20}{3}\Big[\frac{1}{3}r^3 + \frac{1}{2}r^2 + \frac{1}{6}r\Big] + \frac{16}{3}\Big[\frac{1}{2}r^2 + \frac{1}{2}r\Big] \\
&\qquad - \frac{2}{3}r^4 - \frac{4}{3}r^3 - \frac{10}{3}r^2 - \frac{2}{3}r + 1 \\
&= \Big[\frac{4}{15}r^5 + \frac{2}{3}r^4 + \frac{4}{9}r^3 - \frac{4}{90}r\Big] + \Big[\frac{2}{3}r^4 + \frac{4}{3}r^3 + \frac{2}{3}r^2\Big] \\
&\qquad + \Big[\frac{20}{9}r^3 + \frac{10}{3}r^2 + \frac{10}{9}r\Big] + \Big[\frac{8}{3}r^2 + \frac{8}{3}r\Big] \\
&\qquad - \frac{2}{3}r^4 - \frac{4}{3}r^3 - \frac{10}{3}r^2 - \frac{2}{3}r + 1 \\
&= \frac{4}{15}r^5 + \Big[\frac{2}{3} + \frac{2}{3} - \frac{2}{3}\Big]r^4 + \Big[\frac{4}{9} + \frac{4}{3} + \frac{20}{9} - \frac{4}{3}\Big]r^3 \\
&\qquad + \Big[\frac{2}{3} + \frac{10}{3} + \frac{8}{3} - \frac{10}{3}\Big]r^2 + \Big[-\frac{4}{90} + \frac{10}{9} + \frac{8}{3} - \frac{2}{3}\Big]r + 1 \\
&= \frac{4}{15}r^5 + \frac{2}{3}r^4 + \frac{8}{3}r^3 + \frac{10}{3}r^2 + \frac{46}{15}r + 1
\end{aligned}
$$

$$
\begin{aligned}
S^N(6,r) &= 2\Big[\sum_{i=0}^{r} S^N(5,i)\Big] - S^N(5,r) \\
&= 2\Big[\sum_{i=0}^{r} \frac{4}{15}i^5 + \frac{2}{3}i^4 + \frac{8}{3}i^3 + \frac{10}{3}i^2 + \frac{46}{15}i + 1\Big] \\
&\quad - \frac{4}{15}r^5 - \frac{2}{3}r^4 - \frac{8}{3}r^3 - \frac{10}{3}r^2 - \frac{46}{15}r - 1 \\
&= 2\Big[\sum_{i=1}^{r} \frac{4}{15}i^5\Big] + 2\Big[\sum_{i=1}^{r} \frac{2}{3}i^4\Big] + 2\Big[\sum_{i=1}^{r} \frac{8}{3}i^3\Big] + 2\Big[\sum_{i=1}^{r} \frac{10}{3}i^2\Big] + 2\Big[\sum_{i=1}^{r} \frac{46}{15}i\Big] \\
&\quad + 2r + 2 - \frac{4}{15}r^5 - \frac{2}{3}r^4 - \frac{8}{3}r^3 - \frac{10}{3}r^2 - \frac{46}{15}r - 1 \\
&= \frac{8}{15}\Big[\sum_{i=1}^{r} i^5\Big] + \frac{4}{3}\Big[\sum_{i=1}^{r} i^4\Big] + \frac{16}{3}\Big[\sum_{i=1}^{r} i^3\Big] + \frac{20}{3}\Big[\sum_{i=1}^{r} i^2\Big] + \frac{92}{15}\Big[\sum_{i=1}^{r} i\Big] \\
&\quad - \frac{4}{15}r^5 - \frac{2}{3}r^4 - \frac{8}{3}r^3 - \frac{10}{3}r^2 - \frac{16}{15}r + 1 \\
&= \frac{8}{15}\Big[\frac{1}{6}r^6 + \frac{1}{2}r^5 + \frac{5}{12}r^4 - \frac{1}{12}r^2\Big] + \frac{4}{3}\Big[\frac{1}{5}r^5 + \frac{1}{2}r^4 + \frac{1}{3}r^3 - \frac{1}{30}r\Big] \\
&\quad + \frac{16}{3}\Big[\frac{1}{4}r^4 + \frac{1}{2}r^3 + \frac{1}{4}r^2\Big] + \frac{20}{3}\Big[\frac{1}{3}r^3 + \frac{1}{2}r^2 + \frac{1}{6}r\Big] \\
&\quad + \frac{92}{15}\Big[\frac{1}{2}r^2 + \frac{1}{2}r\Big] - \frac{4}{15}r^5 - \frac{2}{3}r^4 - \frac{8}{3}r^3 - \frac{10}{3}r^2 - \frac{16}{15}r + 1 \\
&= \Big[\frac{4}{45}r^6 + \frac{4}{15}r^5 + \frac{2}{9}r^4 - \frac{2}{45}r^2\Big] + \Big[\frac{4}{15}r^5 + \frac{2}{3}r^4 + \frac{4}{9}r^3 - \frac{2}{45}r\Big] \\
&\quad + \Big[\frac{4}{3}r^4 + \frac{8}{3}r^3 + \frac{4}{3}r^2\Big] + \Big[\frac{20}{9}r^3 + \frac{10}{3}r^2 + \frac{10}{9}r\Big] \\
&\quad + \Big[\frac{46}{15}r^2 + \frac{46}{15}r\Big] - \frac{4}{15}r^5 - \frac{2}{3}r^4 - \frac{8}{3}r^3 - \frac{10}{3}r^2 - \frac{16}{15}r + 1 \\
&= \frac{4}{45}r^6 + \Big[\frac{4}{15} + \frac{4}{15} - \frac{4}{15}\Big]r^5 + \Big[\frac{2}{9} + \frac{2}{3} + \frac{4}{3} - \frac{2}{3}\Big]r^4 \\
&\quad + \Big[\frac{4}{9} + \frac{8}{3} + \frac{20}{9} - \frac{8}{3}\Big]r^3 + \Big[-\frac{2}{45} + \frac{4}{3} + \frac{10}{3} + \frac{46}{15} - \frac{10}{3}\Big]r^2 \\
&\quad + \Big[-\frac{2}{45} + \frac{10}{9} + \frac{46}{15} - \frac{16}{15}\Big]r + 1 \\
&= \frac{4}{45}r^6 + \frac{4}{15}r^5 + \frac{14}{9}r^4 + \frac{8}{3}r^3 + \frac{196}{45}r^2 + \frac{138}{45}r + 1
\end{aligned}
$$

Bibliography

[1] D. Andre, F. H. Bennett, and J. R. Koza. Discovery by genetic programming of a cellular automata rule that is better than any known rule for the majority classification problem. Technical report, Stanford University, 1996.

[2] P. J. Angeline. Evolving fractal movies. In J. R. Koza, D. E. Goldberg, Fogel D. B., and Riolo R. L., editors, *1st Annual Conference on Generic Programming*, pages 503–511, Stanford University, July 1996.

[3] T. Bäck. Self-adaptation in genetic algorithms. In *Toward a Practice of Autonomous Systems: Proceedings of the first European Conference on Artificial Life*, pages 263–271, Cambridge, MA, USA, 1992. MIT Press.

[4] T. Bäck and M. Schutz. Intelligent mutation rate control in canonical genetic algorithms. In *Proceedings of the 9th International Symposium, ISMIS 96*, pages 158–167, Berlin, 1996. Springer-Verlag.

[5] Th. Bäck. *Evolutionary Algorithm in Theory and Practice*. Oxford University Press, 1996.

[6] Th. Bäck, D. B. Fogel, and editors Michalewicz, Z., editors. *Handbook of Evolutionary Computation*. Oxford University Press and Institute of Physics Publishing, Bristol/New York, 1997.

[7] Th. Bäck, Breukelaar R., and Willmes L. Inverse design of cellular automata by genetic algorithms: an unconventional programming paradigm. *UPP proceedings in the 'Hot Topics' subline of LNCS*, 2005.

[8] W. Banzhaf. *Interactive evolution. in: Handbook of Evolutionary Computation (T. Bäck,, D. Fogel, and Z. Michalewicz)*, chapter C2.10, pp. 1-5. Oxford University Press, 1997.

[9] D. Barca, G. M. Crisci, S. Gregorio, and C. Nicoletta. Cellular automata for simulating lava flows: A method and examples of the etnean eruptions. *Transport Theory and Statistical Physics*, 23(1):195 – 232, 1994.

[10] H.-G. Beyer. *The Theory of Evolution Strategies*. Springer, Berlin, 2001.

[11] R. Breukelaar and Th. Bäck. Evolving transition rules for multi dimensional cellular automata. In *6th International Conference on Cellular Automata for Research and Industry, ACRI*, Amsterdam, The Netherlands, 2004. Springer.

[12] A.E. Eiben, R. Hinterding, and Z. Michalewicz. Parameter control in evolutionary algorithms. *Evolutionary Computation, IEEE Transactions on Evolutionary Computation*, 3(2):124–141, July 1999.

[13] B. Filipic and D. Juricic. An interactive genetic algorithm for controller parameter optimization. In *International Conference on Artificial Neural Nets and Genetic Algorithms*, pages 458–462, Innsbruck, Austria, 2003.

[14] D. B. Fogel. *Evolutionary Computation: Toward a New Philosophy of Machine Intelligence*. IEEE Press, Piscataway, New York, 1995.

[15] L. Fogel, A. Owens, and M. Walsh. *Artificial Intelligence through Simulated Evolution*. John Wiley and Sons, 1966.

[16] P. Gacs, G. L. Kurdyumov, and L. A. Levin. One dimensional uniform arrays that wash out finite islands. *Problemy Peredachi Informatsii*, 14:92–96, 1978.

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

[18] D. E. Goldberg. *The Design of Invocation:Lessons from and for Competent Genetic Algorithms*. Kluwer Academic Publishers, 2002.

[19] N. Hansen and A. Ostermeier. Completley derandomized selfadaptation in evolution strategies. *Evolutionary Computation*, 9(2):159–195, 2001.

[20] M. Herdy. Evolution strategies with subjective selection. In *Parallel Problem Solving from Nature - PPSN IV*, pages 22–31, Germany, 1996. Springer.

[21] M. Herdy. Evolutionary optimization based on subjective selection - evolving blends of coffee. In *5th European Congresson Intelligent Techniques and Soft Computing EUFIT'97*, pages 640–644, 1997.

[22] J. H. Holland. *Adaptation in Natural and Artificial Systems*. The University of Michigan Press, Ann Arbor, 1975.

[23] D. Horowitz. Generating rhythms with genetic algorithms. In *International Computer Music Conference*, pages 142–143, Aarhus Denmark, 1994.

[24] S. Inverso, D. Kunkle, and C. Merrigan. Evolutionary methods for 2-d cellular automata computation. www.cs.rit.edu/~drk4633/mypapers/gacaProj.pdf, 2002.

[25] Anderson J.R. *Cognitive Psychology and its implications*. Worth Publishers, UK, 2004.

[26] J. R. Koza. *Genetic Programming: On the Programming of Computers by Natural Selection*. MIT Press, Cambridge, MA, 1992.

[27] J. R. Koza, M. A. Keane, M. J. Streeter, W. Mydlowec, J. Yu, and G. Lanza. *Genetic Programming IV: Routine Human-Competitive Machine Intelligence*. Kluwer Academic Publishers, 2003.

[28] W. Li, N. H. Packard, and C. G. Langton. Transition phenomena in cellular automata rule space. *Physica D*, 45:77–94, 1990.

[29] M. Mitchell and J.P. Crutchfield. The evolution of emergent computation. Technical report, Proceedings of the National Academy of Sciences, SFI Technical Report 94-03-012, 1994.

[30] M. Mitchell, J.P. Crutchfield, and P.T. Hraber. Evolving cellular automata to perform computations: Mechanisms and impediments. *Physica D*, 75:361–391, 1994.

[31] I. C. Parmee and C. R. Bonham. Cluster oriented genetic algorithms to support designer/evolutionary computation. In *Proceedings of CEC'99*, pages 546–555, Washington D.C., USA, 1999.

[32] J. Reason. *Human Error*. Cambridge University Press, Cambridge UK, 1990.

[33] I. Rechenberg. *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. Fromman-Holzboog Verlag, Stuttgart, 1973.

[34] I. Rechenberg. *Evolutionsstrategie '94*. Fromman-Holzboog Verlag, Stuttgart, 1994.

[35] G. Rudolph. On interactive evolutionary algorithms and stochastic mealy automata. In *Parallel Problem Solving from Nature - PPSN IV*, pages 218–226, 1996.

[36] G. Rudolph. Self-adaptive mutations may lead to premature convergence. *Evolutionary Computation, IEEE Transactions*, 5(4):410 – 414, August 2001.

[37] R. Salomon. The curse of high-dimensional search spaces: observing premature convergence in unimodal functions. In *Evolutionary Computation, CEC2004*, pages 918 – 923, 2004.

[38] H. P. Schwefel. Numerische optimierung von computer-modellen mittels der evolutionsstrategie. *Interdisciplinary Systems Research*, 26, 1977.

[39] H. P. Schwefel. *Evolution and Optimum Seeking*. Wiley, New York, 1995.

[40] H. Takagi. Interactive evolutionary computation: Fusion of the capabilities of ec optimization and human evaluation. In *Proceedings of the IEEE*, pages 1275–1296, 2001. vol.89, no.9.

[41] S. Wolfram. Statistical mechanics of cellular automata. *Reviews of Modern Physics*, 55, 1983.

Acknowledgments

As is true for so many pleasant surprises in life, this dissertation would not have been possible without a string of now seemingly unlikely events, all of which I will be forever thankful for. I am for instance thankful for my art teacher convincing me that painting was not my calling, for my first internship convincing me I was certainly not going to work right away, thankful for studying to be boring enough to start taking long walks in nature, and for nature finally showing me what I had been studying for.

This dissertation would not have been possible without the network of PhD students and staff at LIACS. I would like to thank in particular Ofer Shir, Michael Emmerich and Rui Li for their invaluable insights and companionship.

I thank the FOM foundation for partially funding this research, I thank BLUERIDGE Analytics Inc. for giving me the opportunity to pursue my PhD while being under full employment in America and I thank the LIACS institute for having the patience needed for this arrangement.

I do not believe my interest in evolution was a genetic trait, yet I know that without the continuous trust and support from my family (my mom in particular) I would not be where I am right now.

My distinction between friends and family has blurred over the years as I believe only good friendships have the power to, which makes me thank a few people in particular. Anne, thank you for so much, but especially for helping me think. Lennert, thanks for helping me enjoy thinking. Tess, thanks for giving me a reason to think.

Samenvatting

Evolutie en Interactie zijn twee processen die veelvuldig gebruikt worden om oplossingen te creeren, vorm te geven, te vinden en te optimalizeren voor problemen in de wetenschap en industrie. Zo is evolutie bijvoorbeeld succesvol toegepast als een zoekalgoritme in onderzoeksgebieden als natuurkunde, scheikunde en biologie, maar ook in commerciele applicaties zoals viegtuig aerodynamica en grondwerken in de bouw. Het definieren van interactie is een belangrijk onderdeel van het ontwerpen van dergelijke algoritmen. Niet alleen zijn de definities van de startwaarden en de antwoorden belangrijk, maar ook de interacties tussen de verschillende onderdelen binnen in een algoritme.

Dit proefschrift houdt zich bezig met de het onderzoeksgebied waar evolutie en interactie elkaar overlappen. Het onderzoekt hoe evolutie in staat is om een interactie te creeren, het onderzoekt hoe de interactie binnenin een evolutionair algoritme invloed heeft op de efficientie van dat algoritme en het laat zien hoe de mens met een evolutionair algoritme kan communiseren. Door deze drie vormen van overlap te behandelen, probeert dit proefschrift inzicht te geven in de wereld van evolutie en interactie.

Evolutionaire Algoritmen

Evolutionaire algoritmen (EA) gebruiken het natuurlijke evolutie principe om betere antwoorden te vinden voor moeilijke problemen. In de natuur werkt evolutie met behulp van een rigoreuze selectie. Als een dier ziek is of verminkt, zal het meer moeite hebben om in leven te blijven en heeft daardoor een kleinere kans om zich voort te planten. Dit betekent dat in

het algemeen fittere individuen meer nakomelingen hebben. Door het gebruik van DNA worde de karakteristieken van de ouders doorgegeven aan de kinderen en dit geeft de nakomelingen een vergelijkbare kans op overleven. Dit process zorgt er in theorie voor dat elke nieuwe generatie beter in staat is om te overleven en zich voor te planten.

Evolutionaire algoritmen werken precies hetzelfde, maar in plaats van dieren en planten gebruikt een EA antwoorden en oplossingen en in plaats van honger of verminking gebruikt een EA probleemdefinities om individuen te evalueren. De antwoorden heten nog steeds 'individuen', de selectie procedure heet een 'fitness functie' en het principe is precies hetzelfde. Een EA heeft een 'pool' (groep) met antwoorden (individuen) voor een bepaald probleem welke allemaal een 'fitness waarde' krijgen toegekend door een 'fitness functie'. Een selectie stap in het algoritme selecteert de beste individuen en gebruikt die om nieuwe antwoorden te maken voor een volgende stap in het algoritme. Deze nieuwe antwoorden lijken erg op de oude (beste) antwoorden, maar zijn een klein beetje veranderd ('gemuteerd'). Doordat de slechtste antwoorden weggegooid zijn en de nieuwe antwoorden veel lijken op de beste antwoorden, gaat de gemiddelde fitness waarde van de antwoorden omhoog en is het EA in staat steeds betere antwoorden te vinden voor een moeilijk probleem.

Evolutie van Interactie

Een Cellulaire Automaat (CA) is een abstract model dat de interactie tussen enkelcellige organismen beschrijft. In de simpelste vorm beschrijft het een ring van cellen in de vorm van een *array* (rij) met binaire waarden. Elke cel in die array heeft een waarde (0 of 1) en heeft een connectie met de twee buur-cellen (een links en een rechts). Tijd wordt gesimuleerd door stapsgewijze interacties tussen de cellen die allemaal tegelijktijd (synchroon) plaats vinden. Elke stap krijgt elke cel een nieuwe waarde gebaseerd op de waarden van de cellen in de buurt door het toepassen van een transitieregel. Het is gebruikelijk dat elke cel dezelfde transitieregel heeft en daardoor hetzelfde 'gedrag'. Dit simpele maar krachtige model is in staat om complexe interacties te simuleren en is daardoor succesvol toegepast in verschillende natuurkundige, scheikundige en biologische applicaties.

De interactie (en daardoor het 'gedrag') van een dergelijk CA is vastgelegd in de transitieregel en deze wordt vaak door experts in het betreffende vakgebied ontworpen om het juiste gedrag te kunnen simuleren. Dit proefschrift beschrijft een manier om evolutionaire algoritmen dit gedrag te laten ontwerpen, zodat enkel een beschrijving van het probleem in combinatie

met het toepassen van evolutie genoeg is om het probleem op te lossen. De generieke methode beschreven in Hoofdstuk 4 laat niet alleen zien dat een dergelijke aanpak werkt voor verschillende communicatieve problemen, maar geeft ook een intrigerend inzicht in hoe interactie kan ontstaan vanuit niets meer dan evolutie en een probleemstelling.

Interactie in Evolutie

Zelf-regulatie ('self-adaptation') is een goed voorbeeld van interactie binnenin een evolutionair algoritme. Met zelf-regulatie kunnen sommige waarden in een EA die normaal niet veranderen of veranderen aan de hand van een wiskundige formule, nu veranderen door het gebruik van evolutie. Zo kan bijvoorbeeld de mate waarin een individu gemuteerd wordt ('mutatie-snelheid') deel uit maken van het individu zelf, zodat de individu die de beste mutatie-snelheid heeft ook de grootste kans heeft om het beste gemuteerd en daardoor geselecteerd te worden, wat dus uiteindelijk een zelf-regulerende mutatie-snelheid tot gevolg heeft. Dit werkt voor waarden als de mutatie-snelheid, maar ook voor meer complexe waarden in andere stappen van een EA. In alle gevallen zorgt het voor flexibelere EA die een grotere verscheidenheid van problemen kunnen oplossen, maar er zijn een aantal mogelijke complicaties die moeilijk te begrijpen of oplossen zijn.

Hoofdstuk 5 beschrijft hoe zelf-regulatie is toegepast op de experimenten uit Hoofdstuk 4. Ook al waren deze experimenten succesvol, ze hadden een aantal onverwachte bijwerkingen als gevolg. Door ditzelfde effect in een veel simpeler experiment te dupliceren, onderzoekt dit proefschrift de limieten van zelf-regulatie en vindt een algemeen probleem dat alleen duidelijk naar voren komt als de interactie binnenin een EA tussen verschillende individuen bestudeerd wordt. Naast het naarvoren brengen van dit probleem, laat dit onderzoek ook het belang zien van het beschrijven van algoritmen in interacties.

Interactie met Evolutie

Veel complexe problemen hebben specifieke complexe algoritmen om het probleem op te lossen. Het ontwerpen van een dergelijk specifiek algoritme betekent vaak dat veel specieke informatie in het algoritme gestopt moeten worden en dat het algoritme dan alleen werkt voor het specifieke probleem. Met steeds sneller wordende computers en een exponentiële groei in

data opslag is het geen verrassing dat de problemen die we met computers proberen op te lossen ook steeds complexer worden. Dit betekent dat het ook steeds moeilijker wordt het juiste algoritme te ontwerpen om die groeiende problemen op te lossen, om maar te zwijgen over te problemen waar een specifiek algoritme onvindbaar of onmogelijk is. Hoofdstuk 6 geeft een korte introductie in een onderzoeksgebied dat dit probeert aan te pakken door de expert deel uit te laten maken van het algoritme. *Human Algorithm Interaction* (HAI - 'Mens Algoritme Interactie') richt zich op de mogelijkheden om algoritmen te 'sturen' door een expert gebruiker. Door de mens deel uit te laten maken van een algoritme, is het niet langer nodig om alle specifieke informatie in het algoritme zelf te vertalen. In plaats daarvan dan de mens de rol van 'controleur' of soms 'data bank' op zich nemen.

Evolutionaire algoritmen zijn uitermate geschikt voor de gevallen waar het moeilijk (of onmogelijk) is specifieke informatie te gebruiken om een specifiek algoritme te ontwerpen en zijn om diezelfde reden ook geschikt voor het gebruik in HAI. Hoofdstuk 6 onderzoekt hoe EA gebruikt kunnen worden voor HAI en identificeert een aantal problemen en potentiele oplossing met betrekking tot de waarneming en analyse limieten van de mens en wat voor een effect die kunnen hebben op een EA.

Curriculum Vitae

Ron Breukelaar was born on the 1^{st} of December 1978 in Winterswijk, The Netherlands. He completed his HAVO (high school) degree in 1996 on the Schaersvoorde College in Aalten, The Netherlands. He immediately started his Bachelor in Computer Science on the HIO (college degree) in Enschede, The Netherlands, which he completed in 2000.

He then moved to Leiden, The Netherlands, to pursue a Master degree in Computer Science at the LIACS institute of Leiden University, which he completed in 2004 with a major in Natural Computing. He started his PhD that same year joining the project "An evolutionary approach to many-parameter physics" funded by the FOM foundation (project nr.: 03TF78-2, werkgroep FOM-L-24) and set out to apply Evolutionary Algorithms to high-dimensional optimization problems in physics.

In 2007 he moved to Charlotte, NC, USA, to develop full fledged optimization solutions for conceptual civil engineering problems at BLUERIDGE Analytics Inc., while in parallel continuing work on his PhD as summarized in this dissertation.