Department of Computer Science

Kerstin Andersson

Cellular Automata in Science

Master's Thesis

2005:01

Cellular Automata in Science

Kerstin Andersson

This thesis is submitted in partial fulfillment of the requirements for the Masters degree in Computer Science. All material in this thesis which is not my own work has been identified and no material is included for which a degree has previously been conferred.

Kerstin Andersson

Approved, 2005-04-07

Opponent: Christiane Schweitzer

Advisor: Johan Garcia

Examiner: Donald F. Ross

Abstract

Cellular automata have a widespread use in the description of complex phenomena in disciplines as disparate as, for example, physics and economics. A cellular automaton is a lattice of cells, and the cells can be in a finite number of states. By using simple local rules the states of the cells are updated in parallel at discrete time steps. In short, a cellular automaton can be characterised by the three words—simple, local, and parallel. These three words are one of the reasons for the attractiveness of cellular automata. They are simple to implement and they are well suited for parallel computers (computations). Another reason for using cellular automata are for their spatio-temporal properties. The lattice may represent space and the updating of the cells gives a dimension of time.

In spite of the simplicity of cellular automata they may give rise to a complex macroscopic behaviour. This is illustrated, in this thesis, by an hydrodynamic example, namely the creation of vortices in flow behind a cylinder.

Although cellular automata have the ability to describe complex phenomena it is sometimes hard to find the proper rules for a given macroscopic behaviour. One approach which has been successfully employed is to let cellular automata rules evolve (for example, through genetic algorithms) when finding the desired properties. In this thesis this will be demonstrated for two-dimensional cellular automata with two possible states of the cells. A genetic algorithm is used to find rules that evolve a given initial configuration of the cells to another given configuration.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this Master's thesis cellular automata and their use particular in science and computer science will be discussed. This work is a continuation of the preliminary studies of cellular automata that constituted a Bachelor's project [3] in 2003.

Cellular automata have a widespread use in the description of complex phenomena in disciplines as disparate as physics and economics. The field seems enormous and in this Master's thesis only a small fraction will be covered. The writing of this thesis has been greatly inspired by Stephen Wolfram's book "A New Kind of Science" [32], and in particular by aspects concerning modelling in science. As is indicated by Figure 1.1 the greatest source of inspiration for making models in science is nature, i.e., the physical world and its phenomena. Another source of inspiration is mathematics. With the development of mathematics it has been possible to build more and more appropriate models in science. This can be exemplified by the development of the integral and differential calculus in the 17th and the 18th centuries and the formulation of classical mechanics. Another example concerns the development of differential geometry in the 18th century and the formulation of the general theory of relativity.

The validation of a model in science often means solving a number of equations. This procedure is largely benefited from results obtained in the field of computer science. Actually one can consider the research in quantum chemistry partly as an ongoing validation of the model of quantum mechanics for atomic and molecular systems.

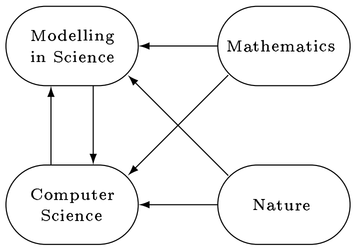


Figure 1.1: *Sources of inspiration in modelling in science.*

The enhancement of computer science, on the other hand, is also largely due to results from other fields. For example, in the 1930s John von Neumann was convinced that numerical work was the most promising way to get a feeling for certain non-linear partial differential equations in hydrodynamics. Therefore he started to study new possibilities for computation on electronic machines. This partly initiated the flourishing development and manufacturing of computers.

The last decades nature has been a steadily growing source of inspiration in computer science with buzz words like neural networks, swarm intelligence, ant colony systems, and genetic algorithms appearing frequently in the literature.

In this Master's thesis cellular automata will be discussed and they can be considered as simple constructions in the field of computer science. Since John von Neumann [28] and Stanislaw Ulam [27] in 1950 first proposed the concept, it has attracted researchers from various disciplines. In this thesis it will be demonstrated how cellular automata can be used in science. In Chapter 2 a background of various aspects of cellular automata is given. In Chapter 3 implementations of some cellular automata are discussed in order to exemplify the workings of them and what kind of problems they can model. In Chapter 4 the evaluation of cellular automata is given. Conclusions and future work is found in Chapters 5 and 6, respectively.

Chapter 2

Background

In this chapter various aspects of cellular automata will be discussed in order to give an understanding of their role and place in science.

2.1 What is a Cellular Automaton?

In order to explain what a cellular automaton is both a simple example and a formal definition will be given in this section.

2.1.1 A Simple Example

A good way to explain what a cellular automaton is is by giving a concrete example. Here a simple set of cellular automata, in the following called elementary cellular automata, will be defined. They were extensively discussed in Wolfram [32] and Andersson [3]. When defining a cellular automaton the lattice of cells, the states of the cells, and the rules have to be defined. For the elementary cellular automata, the lattice of cells, L, is a one-dimensional finite lattice illustrated in Figure 2.1. The finite set of states of the cells is $S = \{0, 1\} = \{white, black\}$. That is, each cell in L can be in either of two states, namely 0 or 1. In the figures below white and black will be used for denoting the states.

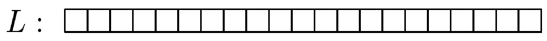


Figure 2.1: *One-dimensional finite lattice.*

Finally, the rules have to be defined. The new state of a cell will be determined by the states of the cell and its two neighbours. The boundary cells are special cases and here the neighbours of the first cell are the last cell and the second cell. Similarly, the neighbours of the last cell is the next last cell and the first cell. What the procedure above does is to connect the first and last cells of the lattice, so that instead of a finite string of cells a circular lattice is obtained. Since the number of states is two there are $2^3 = 8$ possible combinations of the states of a cell and its two immediate neighbours, as illustrated in Figure 2.2. Further, since the new state of a cell can be in either of two states there are

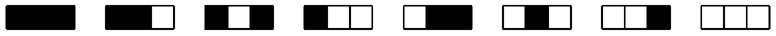


Figure 2.2: *Eight possible combinations of states for a cell and its two immediate neighbours.*

$2^8 = 256$ possible rules for elementary cellular automata. Figure 2.3 gives the definition of the 256 rules, where the top row gives the states of a cell and its two immediate neighbours and the other rows give the new state of the cell. The rule numbers of the cellular automata are given to the right.

The workings of an elementary cellular automaton is best illustrated by an example. Let L be a lattice with an odd number of cells. For starting the evolution of an elementary cellular automaton an initial configuration, C_0, has to be chosen. Let us make the middle cell black and all other cells white. Given the initial configuration, the configurations C_t, $t > 0$, can be easily determined by using Figure 2.3. Let us demonstrate the evolution for the elementary cellular automaton with rule number 90 and with L consisting of 21 cells, see Figure 2.4. For the next configuration, C_1, the rule number 90 has to be considered. Since most cells are white, most cells will remain white. The middle black cell has two

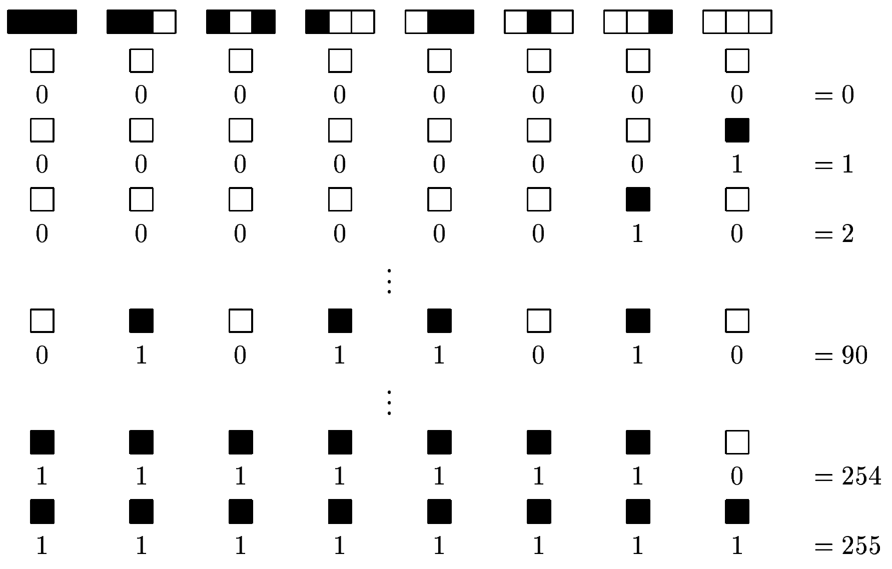


Figure 2.3: *Rules for the elementary cellular automata.*

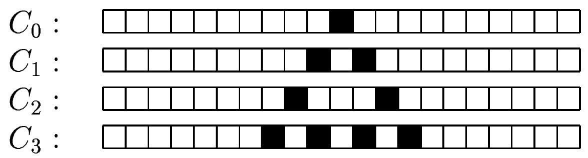


Figure 2.4: *Evolution of an elementary cellular automaton.*

white neighbours and according to the rule the middle cell will turn white. The cell to the left of the middle cell is white with a white neighbour to the left and a black neighbour to the right. This cell will turn black according to rule number 90. Similarly, the cell to the right of the middle cell is white with a black neighbour to the left and a white neighbour to the right. This cell will also turn black. The result is demonstrated in Figure 2.4. This process can be continued forever and in Figure 2.4 a few more configurations are given.

The evolution $(C_0, C_1, \ldots)$ of an elementary cellular automaton can be summarised as in Figure 2.5. Here the top row is the initial configuration. The evolution of the elementary cellular automaton with rule number 90 has a nested fractal structure, see Figure 2.5. Other structures are also possible for elementary cellular automata, for instance uniform

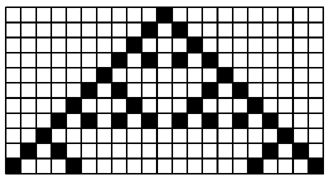


Figure 2.5: *Evolution of the elementary cellular automaton with rule number 90.*

and random structures [32], see Figure 2.6.

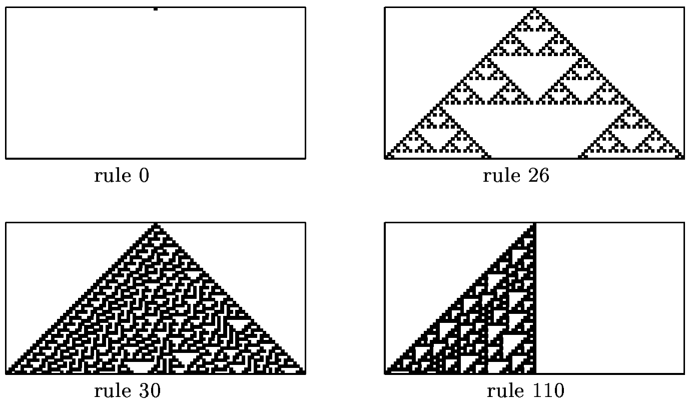


Figure 2.6: *Evolution of elementary cellular automata with simple initial configurations.*

2.1.2 Definition

Several informal definitions of cellular automata can be found and the following [30] gives a fair description of what a cellular automaton is. As the name suggests a cellular automaton is a discrete model with interactions that are uniform in structure. Cellular automata are characterized by the following fundamental properties:

- They consist of a regular discrete lattice of cells.
- The evolution takes place in discrete time steps.

- Each cell is characterized by a state taken from a finite set of states.
- Each cell evolves according to the same rule which depends only on the state of the cell and a finite number of neighboring cells.
- The neighborhood of nearby cells is defined in the same way for each cell, i.e., the neighborhood relation is local and uniform.

Along with this informal definition a formal definition [30] will be given:

Definition 2.1.1 *Let*

- *L be a regular lattice (the elements of L are called cells),*
- *S be a finite set of states,*
- *N be a finite set (of size $n = |N|$) of neighborhood indices such that $\forall c \in N$, $\forall r \in L$: $r + c \in L$,*
- *$f : S^n \rightarrow S$ be a transition function.*

Then the 4-tuple (L,S,N,f) is called a cellular automaton.

2.2 Why use a Cellular Automaton?

Cellular automata are very simple constructions with their lattices of cells which can be in a finite number of states. By using simple local rules the states of the cells are updated in parallel at discrete time steps. In short, a cellular automaton can be characterised by the three words—simple, local, and parallel. These three words are one of the reasons of the attractiveness of cellular automata. They are simple to implement and they are well suited for parallel computers (computations).

Another reason for using cellular automata are their spatio-temporal properties. The lattice may represent a discretisation of the three-dimensional space we are living in and the updating of the cells gives a discretisation of time. In Chapter 3 this will be illustrated by some hydrodynamic examples.

Cellular automata can also be considered as constructions in the field of computer science. One important aspect that has been widely studied is the property of universality [32]. As a matter of fact also among the simpler cellular automata there are universal cellular automata. These are able to calculate everything computable. Another aspect of cellular automata that has been studied is the behaviour of them, i.e., how the cellular automata evolve in time. As is illustrated in Figure 2.6 some cellular automata display a static or repetitive behaviour while others display a more chaotic behaviour. The cellular automata that display a chaotic behaviour can be used in cryptography and in implementations of random number generators. Interestingly, there are cellular automata whose behaviour is somewhere between order and chaos. These are the candidates for universal cellular automata and they are also the ones that may give a life-like impression [29]. In Figure 2.7 the behaviour of a universal elementary cellular automaton is displayed.

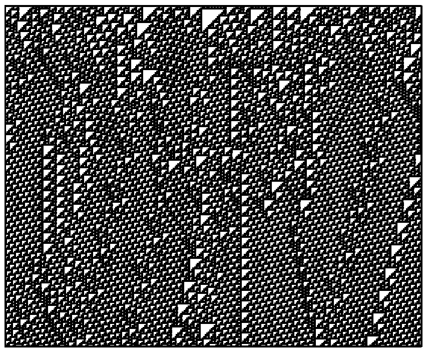


Figure 2.7: *The rule 110 elementary cellular automaton, a universal cellular automaton.*

2.3 Flexibility of Cellular Automata

As is demonstrated in Chapter 2.1.1 a cellular automaton is defined through its lattice, states, and rules. This gives us some flexibility in defining a cellular automaton [11]. Firstly, the dimension, boundary, and shape are factors that can be used in determining the lattice. In Chapter 2.1.1 a one-dimensional lattice is used, but other dimensions are possible also. The boundary can also be chosen to be treated in several ways. In Chapter 2.1.1 a periodic boundary is used, but other possibilities exist as well. The shape of the lattice is another factor that can be used in determining the lattice. In Chapter 3.2 a two-dimensional hexagonal lattice is considered. Secondly, concerning the states, it is the number of them which makes it possible to modify a cellular automaton. Thirdly, the rules give us a wide variety of possibilities of defining cellular automata. The local rules applied to the cells can be either identical or different. This is denoted as uniform and hybrid rules, respectively. In Chapter 2.1.1 uniform rules is used. Further, as in Chapter 2.1.1 the rules can be chosen to be deterministic. But other possibilities exist as well as is illustrated in Chapter 3.2, where probabilistic rules are used. For most cellular automata the next state of a given cell depends on the states of the cells at the previous time step, as is exemplified in Chapter 2.1.1. However, it is also possible to construct cellular automata with time-dependent rules. Most cellular automata considered are synchronous, meaning that all cells are updated simultaneously. However, asynchronous cellular automata may also be constructed.

2.4 Characterisation of Cellular Automata

As is demonstrated in the previous section there are many factors that can be considered when defining a cellular automaton. Despite this fact, Stephen Wolfram has demonstrated that cellular automata fall into four categories [32]: (i) Class 1: uniform behaviour, (ii) Class 2: repetitive behaviour, (iii) Class 3: chaotic behaviour, and (iv) Class 4: complex behaviour. Actually, among the elementary cellular automata all four categories can be found. Figure 2.8 displays four elementary cellular automata, one from each class: rules 0, 26, 30, and 110 belong to classes 1, 2, 3, and 4, respectively.

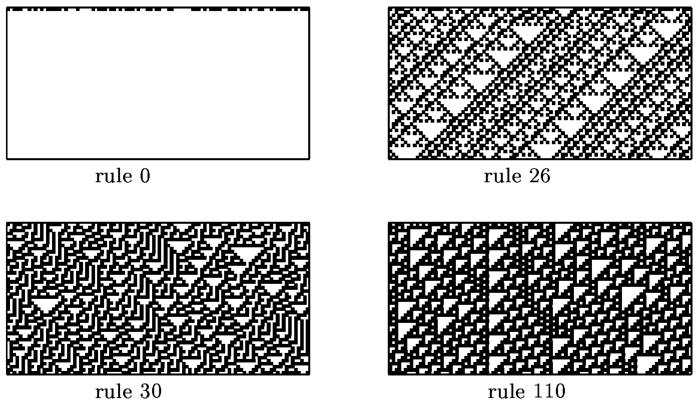


Figure 2.8: *Elementary cellular automata with random initial configurations.*

Besides characterisation of cellular automata through their global behaviours, it is interesting to understand the global dynamics of cellular automata from their local rules. In Figure 2.9 cellular automata are split into several categories and each of them will be discussed separately. The linear/additive cellular automata follow the constraints of

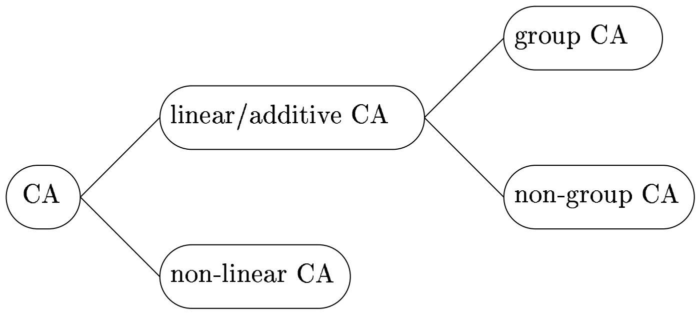


Figure 2.9: *Categories of cellular automata (CA).*

xor/xnor logic for its next state function. For the elementary cellular automata, rule 60 is an example of such a cellular automaton: the new state of a cell is obtained by adding the states of the cells in the neighbourhood modulo 2 with weights either 0 or 1. Linear cellular automata can usually be analysed using algebraic methods [11]. Some of them has gained considerable interest recently because of their application in the field of Very Large Scale Integration (VLSI) of circuits. Linear cellular automata can be split into group and non-group cellular automata, where in the former each of the states has a single predecessor which is not true for the latter [11]. The non-group cellular automata initially received less attention than the group cellular automata. However, recently the trend has been reversed. Non-group cellular automata have been used in a wide range of functions like hashing, classification, and authentication, see Ganguly *et al.* [11] and references therein.

The most important analytical tool for investigating non-linear cellular automata is Langton's λ-parameter. For the elementary cellular automata, consisting of the states 0 and 1, the λ parameter is defined as the probability that the next state of a particular cell will be 1. These probabilities can easily be calculated by investigating the rule table in Chapter 2.1.1. For example, for rule 0 all new states will be zero giving $\lambda = 0$, for rule 90 half of the new states will be 1 and the other half 0 giving $\lambda = 0.5$. In Figure 2.10 the λ-parameters of the rules in Figure 2.8 are indicated.

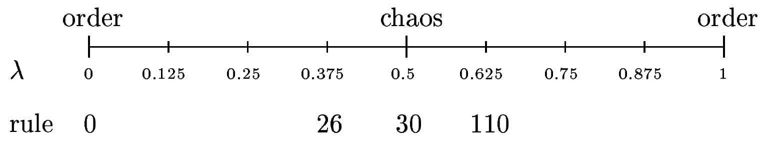


Figure 2.10: *The λ-parameter of various elementary cellular automata.*

Since λ is the probability that the next state of a particular cell will be 1 for an elementary cellular automaton, then $1 - \lambda$ will be the probability for the next state to be 0. The product of these two probabilities will be studied next. Let us denote it $f(\lambda)$,

$$f(\lambda) = \lambda * (1 - \lambda). \tag{2.1}$$

The maximum of f occurs when $\lambda = 0.5$, and the minimum value occurs when $\lambda = 0$ and 1. It has been shown by Langton[15] that with the increase of λ, the cellular automaton changes from "order" to "chaos". Instead of just studying the parameter λ it seems appropriate to study the function f which includes, in the case of two-state cellular automata, all states. The neighbourhood of the maximum of f gives the chaotic regime and the neighbourhoods of the minimum of f give the ordered regimes. Comparing with Figure 2.10 it can be concluded that λ gives a rough measure of the character of a cellular automaton. For the elementary cellular automaton with rule 0 $\lambda = 0$ and indeed rule 0 is a cellular automaton with a ordered behaviour. Rule 30, on the other hand, has $\lambda = 0.5$, and indeed rule 30 is a cellular automaton with a chaotic behaviour. Rule 26 has a large value of λ (0.375) in spite of its orderly behaviour. Rule 110, a class 4 cellular automaton, has $\lambda = 0.625$ which is between the ordered and chaotic regimes. This point Langton calls a transition point, seeing the similarity to phase transitions in matter:

$$Class\ 1\&2 \rightarrow {}''Class\ 4'' \rightarrow Class\ 3. \tag{2.2}$$

Langton even suggested the following transition in dynamical systems:

$$Order \rightarrow {}''Complexity'' \rightarrow Chaos, \tag{2.3}$$

where "complexity" refers to the complicated life-like dynamical behaviour of Class 4 cellular automata [15, 29]. In his work of analysing two-dimensional cellular automata [15], Langton further saw the following parallel to Turing's halting problem [12]. The halting problem is undecidable meaning that there exist computations for which it is not possible to decide whether or not they will halt. This means that computations can be put into three classes: (1) those that will halt, (2) those that will never halt, and (3) those that it is not possible to determine whether they will halt or not. For cellular automata Langton defines a freezing problem, which is a natural analog of Turing's halting problem. Class 1 and 2 cellular automata "freeze up" into a short-period behaviour, and Class 3 cellular automata never freeze into a periodic behaviour. For cellular automata close to the transition (Class

4) both of these outcomes are possible and it will be practically undecidable whether or not a particular cellular automaton will freeze into a periodic behaviour. Langton suspects that the freezing problem is undecidable.

2.5 The Inverse Problem

The inverse problem addresses the problem of finding the rules for a cellular automaton that gives a certain global behaviour. This problem is extremely difficult, see Ganguly *et al.* [11] and references therein. However, there has been work performed in this area and the most popular techniques used are based on evolutionary computation methodologies like genetic algorithms and simulated annealing. The initial work on cellular automata using genetic algorithms was made by Packard and co-workers in the end of the 1980s, see, for example, Meyer *et al.* [17] and Richards *et al.* [22], and it was later on popularised by Mitchell *et al.* [19]. In Chapter 3.3 an implementation of a genetic algorithm for solving the inverse problem for simple global patterns for two-dimensional-lattice cellular automata is described.

2.6 Applications

Cellular automata have been used in a vast number of research areas and here some of them will be mentioned in order to show the diversity of fields where cellular automata are applicable. Most information in this section is taken from Ganguly *et al.* [11].

Cellular automata have been constructed as parallel computing machines (CAMs) and they are ideally suited for simulation of complex systems. Due to the parallelity they can achieve performances that are several orders of magnitude higher than what can be achieved with conventional computers [16]. The advent of nanotechnology also presents new possible applications of cellular automata, and in Benjamin *et al.* [4] an adding cellular automaton, realised as a wireless nanometre-scale computing device, has been proposed. To continue lining up applications of cellular automata an example from the field of computer science will be given. Cellular automata have a potential for acting as language recognisers, and it has been shown that cellular automata can accept context-free languages [26].

We will now continue to the field where cellular automata have been most widely explored, namely the field of dynamical systems. Dynamical systems are often described by non-linear partial differential equations. Because of the non-linearity these systems are hard to analyse and results are very sensitive to round-off errors and the choice of initial conditions. By virtue of their simplicity and stability, cellular automata provide an alternative to partial differential equations [24], which will be demonstrated in Chapter 3.2 for a hydrodynamic example. Other physical systems, like spin systems, reaction-diffusion systems, and systems of galaxies, have been also modelled using cellular automata. Dynamical systems do not only appear in physics but in other sciences, like biology and social sciences, as well. In biology cellular automata have been used to model phenomena at various levels, from the cellular level to the population of organisms. Biological examples concern the immune system, tumour development, detection of genetic disorders in cancerous cells, DNA sequences, fish migration in rivers, and swarms, just to mention a few. In the social sciences cellular automata are used as well and here the following applications can be mentioned: group formation, urban development, and traffic flow.

Chapter 3

Implementation

In this chapter a few well known implementations of cellular automata will be studied in more detail, starting with the perhaps best-known of them all: the Game of Life.

3.1 Game of Life

The Game of Life was developed in the early 1970s by the English mathematician John Conway. The game of Life is actually not a game but a cellular automaton with a two-dimensional lattice. Just as for the elementary cellular automata the cells can be in one of two states, namely black or white. The lattice is finite with periodic boundaries. The neighbourhood of a cell consists of its eight closest neighbours. The rule can be stated as follows: a white cell with 3 black neighbours will become black, and a black cell with 2 or 3 black neighbours remains black, otherwise it will become white. The implementation of the Game of Life is similar to the implementation of the elementary cellular automata (for details, see Andersson [3]).

The Game of Life was very popular in the 1970s, partly because of Martin Gardener's "Mathematical Games" column in Scientific American. What made the Game of Life so popular was that when the simple rules were turned into a program displaying the lattice on the screen, the screen came alive. It boiled with activity. One could see ”gliders”, small clusters of black cells moving across the screen. There were ”glider guns” that fired off new gliders, and other structures that ate the gliders. In Figure 3.1 a few snapshots of the Game of life are presented. The starting configuration is randomly chosen and the three

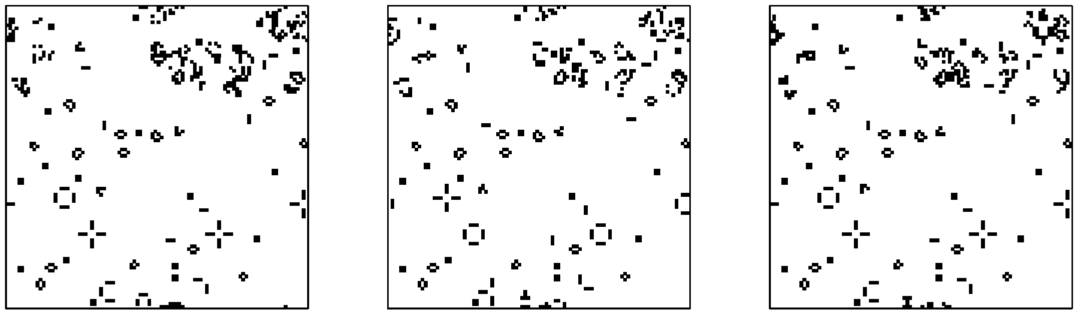


Figure 3.1: *Snapshots of the Game of Life at three consecutive time-steps.*

snapshots are the configurations after 1500, 1501, and 1502 time-steps, respectively. In Figure 3.1 a number of patterns, like the block, boat, blinker, and glider can be identified [1], see Figure 3.2. The block and the boat are stationary structures while the blinker is a

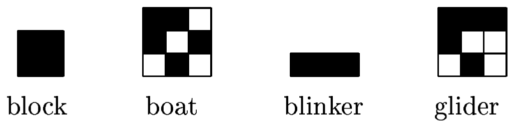


Figure 3.2: *Examples of simple patterns occurring in the Game of Life.*

oscillating stationary structure. The glider is steadily moving across the grid as time goes by, see Figure 3.3. The Game of Life also presents more complicated structures like glider guns, of which one example is shown in Figure 3.4. A glider gun is doing exactly what it says, firing off gliders. As can be seen in Figure 3.4 a new glider is fired off every 30 time-steps or so, in the same direction.

It seems that the game of the Game of Life is to find new patterns and structures. The

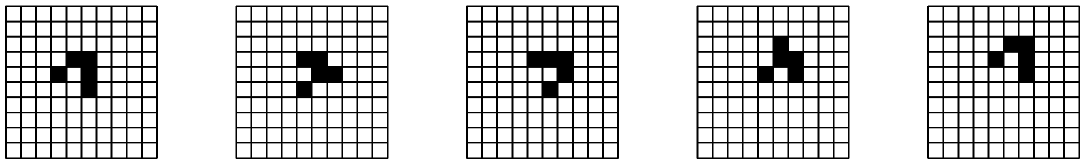


Figure 3.3: *The glider at consecutive time-steps.*

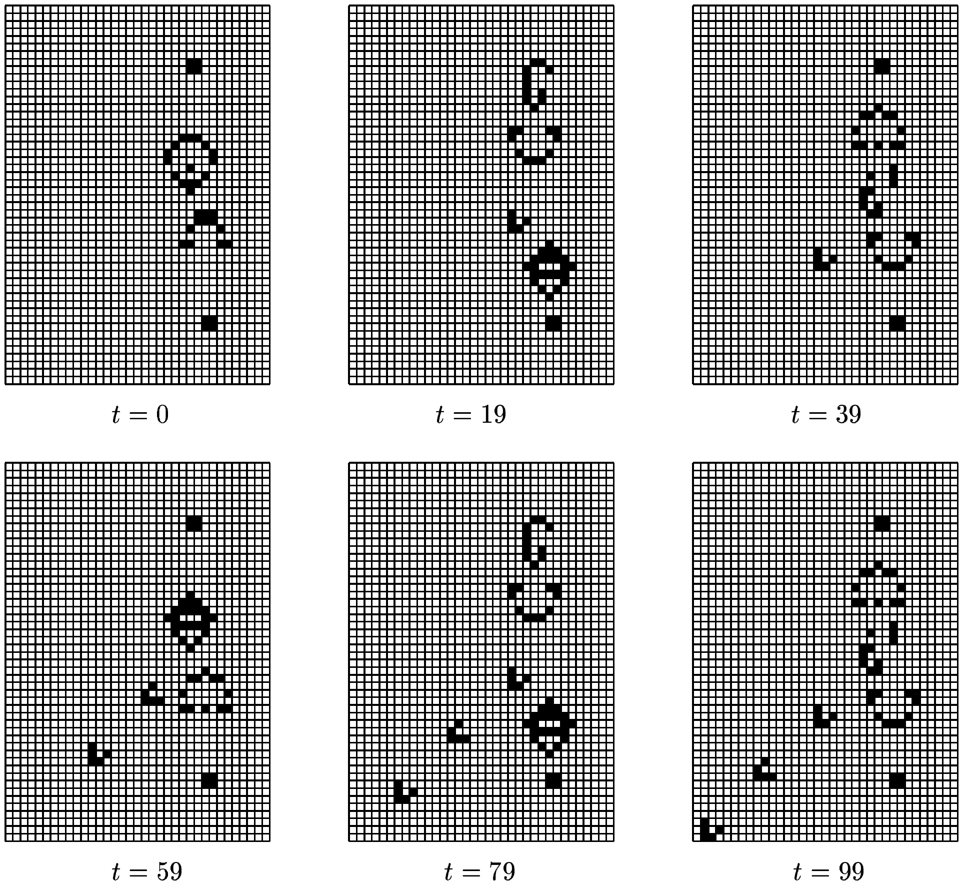


Figure 3.4: *A glider gun at various time-steps.*

glider gun in Figure 3.4 was found by Bill Gosper in November 1970. Since then, various complicated structures like adders and prime number generators have been made. It is also possible to construct the logic gates AND, OR, and NOT using gliders. However, to really become fascinated by the Game of Life, one should see it evolve in real time. In Waldrop [29] Chris Langton, the founding father of artificial life, gives a vivid description of his experience with the Game of Life:

> There's the Game of Life cranking away on the screen. Then I glanced back down at my computer code—and at the same time, the hairs on the back of my neck stood up. I sensed the presence of someone else in the room. (...) I realised that it must have been the Game of Life. There was something *alive* on that screen.

Next the λ-parameter of the Game of Life will be calculated. Since the number of states is two there are $2^9 = 512$ combinations of states for a cell and its eight closest neighbours. Further, there are $\binom{8}{3} = 56$ and $\binom{8}{2} = 28$ combinations for a cell to have exactly three and two black neighbours, respectively. This is enough information for calculating the λ-parameter of the Game of Life: $\lambda = (56 + 56 + 28)/512 = 0.27$, a value in between the ordered ($\lambda = 0$) and chaotic ($\lambda = 0.5$) regimes. The Game of Life is a Class 4 cellular automaton, and it has been shown to be universal. The proof that the Game of Life is universal includes the gliders as propagators of signals and the blinkers as storage elements in the construction of a general purpose computer [15].

3.2 Hydrodynamics

In this section a somewhat more advanced cellular automaton, used in describing, for example, hydrodynamic phenomena, will be discussed.

3.2.1 The Cellular Automaton

Consider the two-dimensional hexagonal lattice in Figure 3.5. This is a typical lattice for modelling fluid flow in two dimensions. In this model the fluid is partitioned into fluid

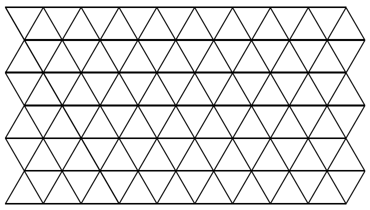


Figure 3.5: *Hexagonal lattice.*

'particles' which are restricted to move along the lines and collide only at the intersections of these lines (nodes). In the collisions Newton's laws of motion have to be obeyed, i.e., mass and momentum have to be conserved. Here the nodes can be considered as the cells and the states of the cells can be chosen in various ways. In the model described in this thesis the number of states is $2^7 = 128$, where 7 is the maximum number of fluid particles heading for a node. A fluid particle can either be at rest or move along the six directions towards a node with a given constant speed. In Figure 3.6 some of the states are shown, where circles denote fluid particles at rest and arrows denote fluid particles in motion in a given direction. Observe that the maximum number of particles at rest is one and that the maximum number of particles in a given direction is one for each node. After discussing

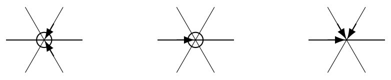


Figure 3.6: *Examples of states.*

the lattice and the states, the rules have to be considered. For this cellular automaton the rules can be split into a collision phase and a propagation phase. In the collision phase the fluid particles at a node collide and in the propagation phase they either remain at the node or propagate to neighbouring nodes. At each time step the collision phase has to be performed first for all nodes before the propagation phase can take place. We will start by discussing the collision phase. All fluid particles have the same mass and are either at rest or move with the same constant speed. In this thesis all possible outcomes of a given collision process, where mass and momentum are conserved, will be taken taken into account. In Figure 3.7 some collision processes are shown, and a complete table is

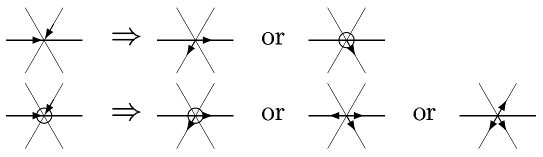


Figure 3.7: *Examples of collisions.*

given in Table A.1 in Appendix A. In Figure 3.7 it is also illustrated that there are several possible outcomes of certain collision processes. For those cases one can make a random choice of the possible collisions, giving them equal probability. In this thesis collisions where no interaction takes place are included (see the outcomes immediately to the right of the arrows in Figure 3.7). Most work in this area has excluded non-interacting collisions whenever it is possible to choose interacting collisions. However, one should realise that the fluid particles do not represent real fluid particles. They are just items for modelling the entire fluid and non-interacting collisions may in those circumstances be realistic. Further, collisions with just one possible outcome are in fact non-interacting but necessarily have to be included for the model to work. The inclusion of non-interacting collisions may give the fluid other properties. This issue is not further discussed here but it might be important when modelling real fluids.

After discussing the collision phase of the rules the propagation phase should be discussed. This phase is actually quite trivial. A particle at rest will remain at the same node, otherwise it will move on to the neighbouring node in the direction it is moving. Figure 3.8 gives an illustration of the collision and propagation phases of a cellular automaton fluid.

For boundaries special rules have to be applied, and in this thesis the rules illustrated in Figure 3.9 will be used, where the solid circles represent solid boundary nodes. That is, a fluid particle will continue in the opposite direction when hitting a boundary node.

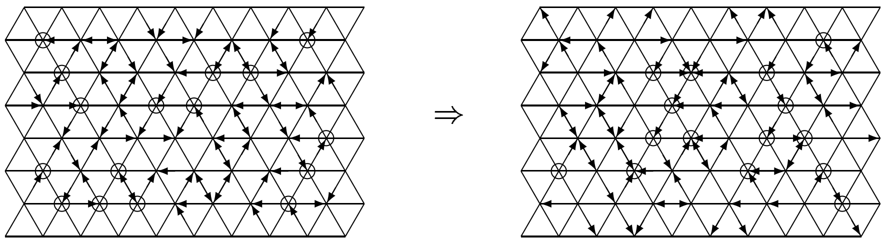


Figure 3.8: *Collision and propagation.*

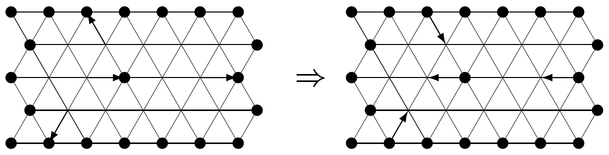


Figure 3.9: *Rules at boundaries.*

3.2.2 Computer Simulations

In order to model flowing fluid pressure has to be introduced. This is achieved here by at each time step introducing new fluid particles to the left and removing the same number of particles to the right. In Figure 3.10 a model of laminar flow is given. In this model the

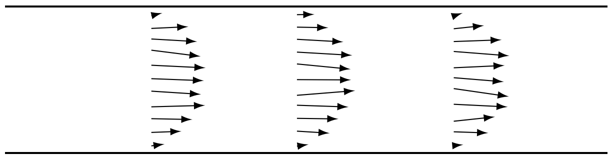


Figure 3.10: *Model of laminar flow.*

hexagonal lattice (which is not printed) consists of 200×57 nodes. The number of particles per node is initially set to 2 and the pressure is modelled by introducing 20 particles to the left and removing 20 particles to the right at each time step. The number of time steps is 1000. The arrows in Figure 3.10 are average velocities of fluid particles at 50×5 nodes. 25×57 nodes close to the inlet to the left and 25×57 nodes close to the outlet to the right are not taken into account in the averaging procedure. The model in Figure 3.10

represents an intersection of flow between two parallel plates. For the case of parallel flow, like the one in Figure 3.10 the fundamental equations in hydrodynamics, the Navier-Stokes equations [20], are extremely simple, leading to a parabolic velocity distribution between the plates. This is also achieved with the cellular automaton model as is illustrated in Figure 3.11, where the parabola resulting from a fitting procedure of the middle velocity

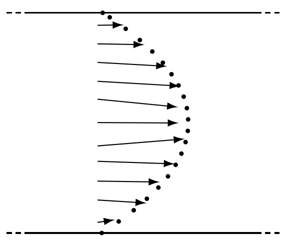


Figure 3.11: *Parabolic velocity distribution of laminar flow.*

profile in Figure 3.10 is included. It is worth while mentioning that the averaging procedure in Figures 3.10 and 3.11 is essential for obtaining the parabolic velocity profiles. Looking at the average velocity at each node will give a quite stochastic impression, see Figure 3.12.

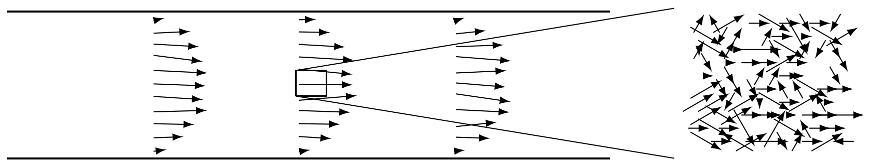


Figure 3.12: *Laminar flow in detail.*

With this cellular automaton it is also possible to investigate when laminar flow turns turbulent. Almost the same model as in Figure 3.10 is used in this investigation. However, to remove disturbances from the inlet and outlet the number of nodes in the horizontal direction is increased from 200 to 500 and the number of nodes close to the inlet and outlet not included in the averaging procedure is increased from 25 $\times$ 57 to 100 $\times$ 57. In addition, all fluid particles in the remaining nodes in the horizontal direction (300 $\times$ 5) are included in the averaging procedure. In Figure 3.13 the velocity distributions for five different pressures ($p = 30$, $p = 50$, $p = 60$, $p = 70$, and $p = 90$) are shown. It seems that

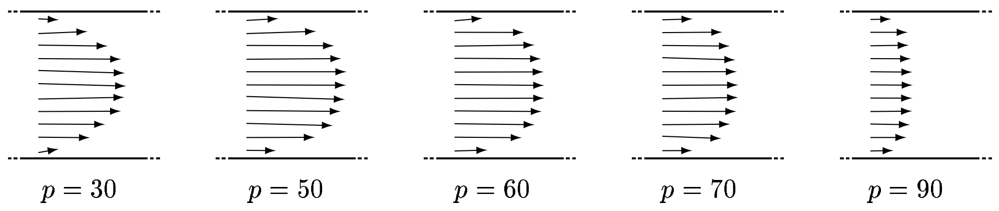


Figure 3.13: *Velocity distributions of fluid flow at different pressures.*

between the pressures $p = 50$ and $p = 70$ the parabolic shape of the velocity distribution is lost. The velocity distribution gets flattened out. This is characteristic of turbulent fluid flow [20]. The collective motion of the fluid is no longer striving for going from left to right but is more irregular leading to smaller average velocities in the centre between the plates. Studying the models more closely one sees that when running the cellular automaton in many time-steps at high pressures, a shortage of fluid particles occur to the right. The particles assemble to the left taking on also other directions than those leading from left to right. This also is an evidence of turbulent motion.

We will now turn to a more interesting phenomenon in hydrodynamics. As is illustrated in Figure 3.9 obstacles can easily be introduced in the cellular automaton model simply by adding some solid boundary nodes. In Figure 3.14 an obstacle in the form of a circle has been included in the lattice. This represents a cylinder in between two parallel plates in three dimensions. Now an interesting behaviour is obtained, namely the creation of vortices behind the cylinder, a regular vortex street is created. In the model the hexagonal lattice consists of 1500×1000 nodes. The number of particles per node is initially set to 2 and the pressure is modelled by introducing 900 particles to the left and removing 900 particles to the right at each time step. The number of time steps is 40000. In the figure lines instead of arrows represent velocities and each line is the average velocity of fluid particles in 30×30 nodes minus the average velocity of the entire fluid. In Chapter 4 a

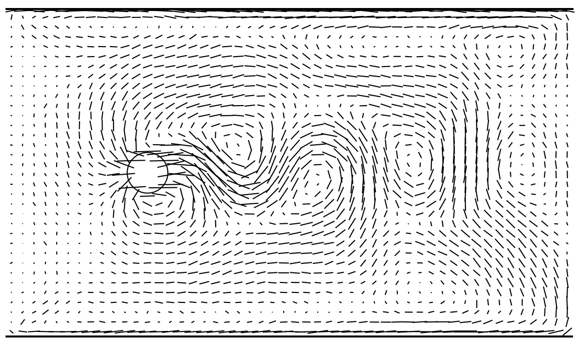


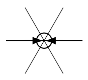
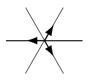
Figure 3.14: *Model of a vortex street.*

comparison with numerical methods and experiment is given.

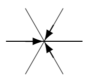
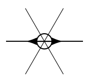
The model described above is referred to as a lattice-gas model in the physics literature. In 1986 Frisch, Hasslacher, and Pomeau [10] introduced the lattice-gas model based on a hexagonal lattice. They created various models depending on which collision rules were used and whether rest particles were used or not. In order to understand the physics behind the lattice-gas model it is worth while mentioning that a fluid can be described on several levels. Firstly, there is the molecular level where the motion is reversible. Secondly, there is the irreversible statistical level with the Boltzmann equation where the fluid is described using a distribution function [6]. Thirdly, we have the macroscopic level where the fluid is considered as a continuum and Navier-Stokes equations can be applied. In the lattice-gas model described here the motion of the fluid particles is not reversible. However, as can be seen in Table A.1 in Appendix A the collision rules satisfy detailed balance, i.e.,

$$A(s \to s') = A(s' \to s), \tag{3.1}$$

where $A(s \to s') \geq 0$ is the probability that an in-state s (arrows pointing towards the node) collides to give an out-state s' (arrows pointing against the node), if for a given collision process all possible outcomes obeying Newton's laws are equally probable. For example,

$A($  $\rightarrow$  $) = 1/5$ (3.2)

and

$A($  $\rightarrow$  $) = 1/5,$ (3.3)

since both these collision processes have five possible outcomes which in this thesis are considered equally probable. Other choices are also possible and it seems that models satisfying only semi-detailed balance, i.e.,

$$\sum_s A(s \rightarrow s') = 1, \forall s', \tag{3.4}$$

should also be appropriate for modelling fluid flow. In Buick [6], Frisch *et al.* [9], and Wolfram [31] all necessary steps and details in describing the derivation of macroscopic equations from lattice-gas models are given, and indeed, within certain approximations, the non-linear Navier-Stokes partial differential equations are obtained. It is also shown that the choice of a hexagonal lattice before a square lattice ensures the property of isotropy.

We will now in more detail study the creation of a vortex street behind a cylinder like the one in Figure 3.14, where 40000 time steps are used. In Figure 3.15 the evolution of the vortex street with time is demonstrated. The pressure is the same as in Figure 3.14 ($p = 900$), but only 1000×600 nodes are displayed and the averaging is done over 50×50 nodes. Just as in Figure 3.14 the lines represent average velocities of fluid particles minus

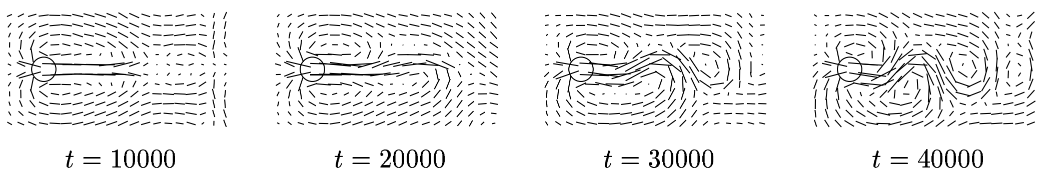


Figure 3.15: *Evolution of a vortex street.*

the average velocity of the entire fluid. As time goes by more and more vortices are created. Since the lattice is finite, vortices will disappear as well (to the right) after about 40000 time steps.

It is also interesting to study how the pressure affects the creation of a vortices. In Figure 3.16 different pressures are used and the number of time steps is 40000, otherwise the computational details are the same as in Figure 3.15. Starting with low pressures the flow is steady. Increasing the pressure gives a regular vortex street which disappears when turbulent behaviour is introduced at high pressures. Whether or not lattice-gas models can be used for giving proper descriptions of turbulence still seems to be an open question [23].

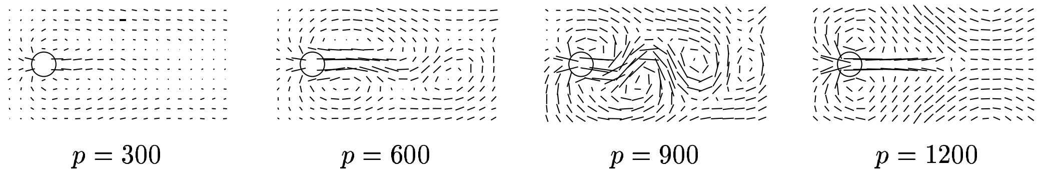


Figure 3.16: *Flow behind a cylinder at various pressures.*

The discussion of vortices will be concluded by a nice example from nature, namely the hair whirls (the twin vortices) on top of the head of baby Viking Andersson. Look in the direction of the arrows in Figure 3.17. One vortex is clockwise and the other one is anti-clockwise.

3.2.3 Program Description

In this section the computer implementation for running the cellular automaton and generating the pictures of the fluid flow will be discussed. In Appendix B the complete C^{++} code is given and the output of the program is LaTeX code of pictures of the flow. Although the hydrodynamic problem is not an ideal problem for C^{++}, classes are used for its implementation. The entire problem consists mainly of one class called Hydro. There is also

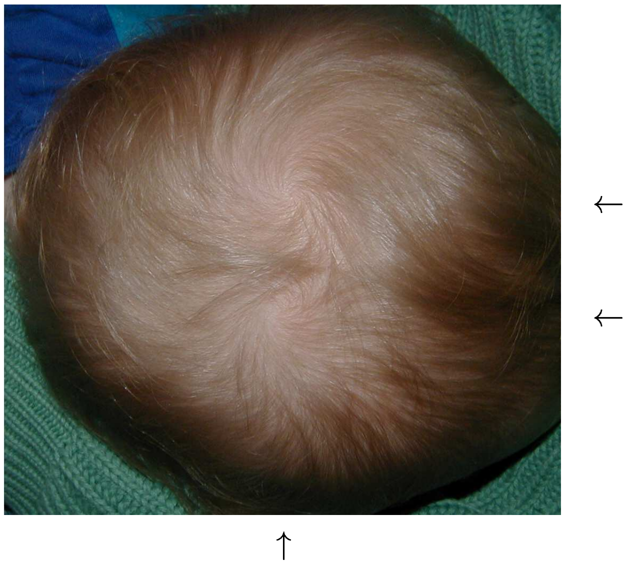


Figure 3.17: *Vortices of baby Viking.*

another class Vector which describes the manipulation of point vectors in two dimensions. Further, in the class Hydro the methods can be split into three groups: (1) methods for dealing with the collisions as described in Appendix A, (2) methods for initialising and evolving the cellular automaton, and (3) methods for printing results in the form of LaTeX code.

For treating the collisions the notation in Figure 3.18 is used. With the use of a

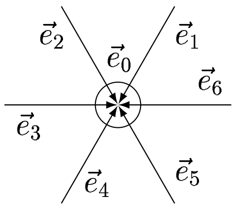


Figure 3.18: *The vectors used in the collisions.*

coordinate system the vectors may be expressed as in Equation 3.5, where all vectors are normalised to length one (except $\vec{e}_0$ which has zero length).

$$\begin{aligned}
\vec{e}_0 &= (0,0), \\
\vec{e}_1 &= (-\frac{1}{2}, -\frac{\sqrt{3}}{2}), \\
\vec{e}_2 &= (\frac{1}{2}, -\frac{\sqrt{3}}{2}), \\
\vec{e}_3 &= (1,0) \\
\vec{e}_4 &= (\frac{1}{2}, \frac{\sqrt{3}}{2}), \\
\vec{e}_5 &= (-\frac{1}{2}, \frac{\sqrt{3}}{2}), \\
\vec{e}_6 &= (-1,0),
\end{aligned} \tag{3.5}$$

For each node the fluid particles are stored as an `unsigned char`, i.e., an 8-bit word, where the second position to the left contains information of the $\vec{e}_0$ fluid particle, the third position to the left contains information of the $\vec{e}_1$ fluid particle, etc. In Figure 3.19 the storage of information of fluid particles is summarised. Here 0 and 1 mean the absence and

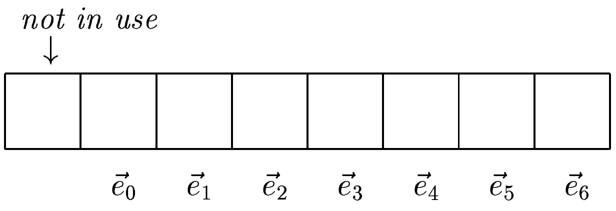


Figure 3.19: *The storage of fluid particle information.*

presence of a fluid particle. Therefore, 01101010, for example, means that the node has 4 fluid particles with the 4 directions $\vec{e}_0$, $\vec{e}_1$, $\vec{e}_3$, and $\vec{e}_5$. The vectors in Figure 3.18 are the direction of fluid particles before a collision. For describing the situation after a collision the reversed directions have to be used. Let us denote a collision

$$(a_0, a_1, a_2, a_3, a_4, a_5, a_6) \Rightarrow (b_0, b_1, b_2, b_3, b_4, b_5, b_6), \tag{3.6}$$

where $a_i, b_i \in \{0, 1\}, \ \forall i \in \{0, 1, 2, 3, 4, 5, 6\}$, denote the number of particles in the various directions $\vec{e}_0, \ldots, \vec{e}_6$. If a collision is allowed according to Newton's laws, then the following two equations must hold for the conservation of mass and momentum:

$$\sum_{i=0}^{6} a_i = \sum_{i=0}^{6} b_i, \text{ and} \tag{3.7}$$

$$\sum_{i=0}^{6} a_i \vec{e}_i = -\sum_{i=0}^{6} b_i \vec{e}_i. \tag{3.8}$$

The method `setNumInnerColl()` calculates the number of possible collisions $(b_0, b_1, b_2, b_3, b_4, b_5, b_6)$ there exist for each $(a_0, a_1, a_2, a_3, a_4, a_5, a_6)$. The method `setInnerColl()` then calculates the $(b_0, b_1, b_2, b_3, b_4, b_5, b_6)$ for all possible $(a_0, a_1, a_2, a_3, a_4, a_5, a_6)$ and puts them in an array for later usage.

The methods for dealing with the initialisation and evolution of the cellular automaton will now be discussed. First, it is determined which nodes are inner nodes and which nodes are boundary nodes. This is done with the method `setInner()`. Next, a number of fluid particles have to be inserted in each node. The number is given by the variable `partPerNode` and the velocities of the fluid particles are chosen randomly among the 7 possible choices for each node. This is performed with the method `setLattice()`. Finally, the evolution of the cellular automaton can be calculated using the method `oneTimeStep()` which, as the name suggests, calculates the next configuration of the cellular automation. If several time-steps have to be calculated then the method `oneTimeStep()` has to be applied several times. At each time-step particles are first inserted to the left and removed to the right if a pressure has to be simulated. The number of particles inserted and removed is given by the variable `p`. The strategy for inserting particles is as follows: start with the leftmost nodes and insert one particle at most in each node in the y-direction. The nodes in the y-direction are chosen randomly. If the pressure is high then probably more layers of nodes in the y-direction have to be considered. The removal of particles is performed similarly. After the pressure has been taken care of the fluid particles may collide. If there are several choices of the outcome of a collision then the choice is made randomly with equal probabilities of the various outcomes. After the collisions have taken place the fluid particles have to propagate to their new locations. This piece of code looks quite nasty because of several possible cases. For instance, the indices of the neighbouring nodes of node (i, j) depends on whether j is even or odd, see Figure 3.20. For boundary nodes special care also has to be taken.

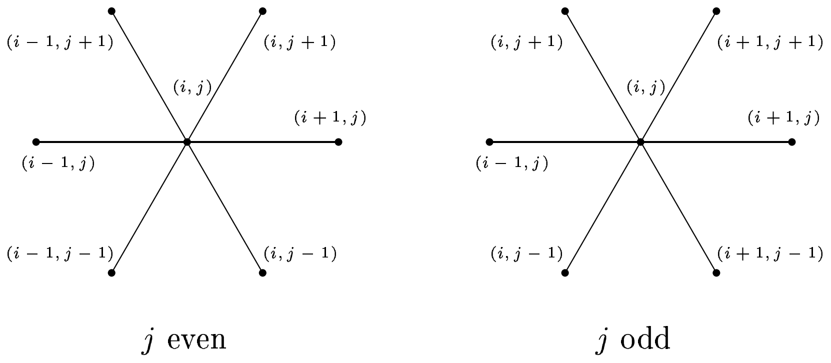


Figure 3.20: *Indices of neighbouring nodes.*

The remaining code in Hydro concerns with the evaluation of the results from the cellular automaton and how they should be presented in LaTeX. A great part concerns making averages of particle velocities over several nodes. The averaging procedure is important for two reasons. First, it makes it possible two reduce the amount of information to be printed. Second, it makes visible phenomena that would be hard to discover otherwise. The method `print(char *, bool)` is the method which is responsible for the evaluation and the creation of LaTeX code. The name of the file where the LaTeX code should be stored has to be provided as well as a boolean value telling whether or not the hexagonal lattice should be printed. The position of the lines that symbolise the average velocities is at the centre of mass of the fluid particles included in the average value. The program also allows that another reference frame is used when printing the average velocities. That reference frame is that given by the average velocity of the entire fluid, which means that what is printed are the deviations from the average velocity.

3.3 Genetic Algorithms

Genetic algorithms are simple computational models of Darwin's evolutionary theory of survival of the fittest. John Holland, one of the inventors of genetic algorithms, was motivated to find algorithms that could be used in the implementation of robust adaptive systems that would be capable to deal with an ever changing environment [13].

3.3.1 Definition

Genetic algorithms can be defined in several ways but the basic ingredient is a fixed-size population of individuals represented by fixed-length strings. These strings may be thought of as strings containing some sort of "genetic" information, like the strings of DNA contain genes for the human being. The population might then evolve by selecting parents among the individuals, according to their "fitness", and these parents then generate "genetically" similar offspring. What fitness means has to be defined from problem to problem. In Figure 3.21 a schematic picture is given of how a population of individuals evolve. To

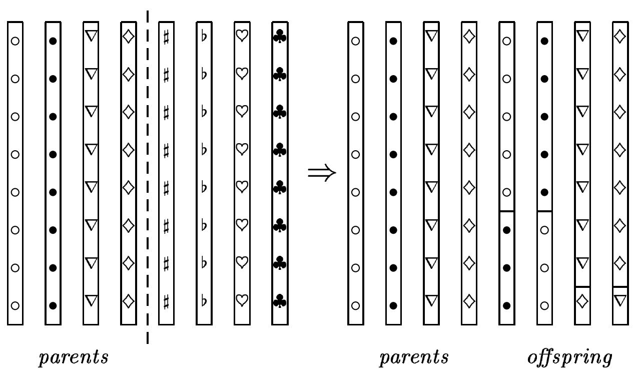


Figure 3.21: *Evolution of a population of individuals.*

make it simple, the size of the population should be dividable by four. The parents are chosen from the half of the population with the fittest individuals. These individuals are grouped together two by two, and two offspring are created from each couple. The parents together with their offspring constitute the new population, which now in turn can be evolved. In this thesis the definition of the genetic algorithm more or less follows the one in Mitchell *et al.* [19]. However, here an elite procedure is applied meaning that only the fittest individuals are allowed to make offspring. Each of the elite individuals only create offspring with one other randomly chosen elite individual.

The next question that has to be addressed is how the offspring are created. The genetic operations crossover and mutation are used for this purpose. The crossover operation takes large pieces of the strings of two individuals and interchanges them creating two new individuals, see Figure 3.22. In this thesis the pieces are taken at the ends of the strings, however, other choices are possible also and it should not make too much of a difference. The size of the pieces that are exchanged is randomly chosen. The mutation

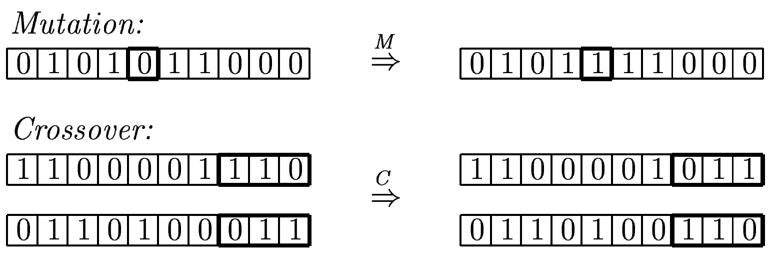


Figure 3.22: *The genetic operations crossover and mutation.*

operator makes only small changes in the individuals by flipping a small number of bits in the strings, see Figure 3.22. New offspring are generated by first applying the crossover operation on the parents and then applying the mutation operator on the two individuals thereby produced. The two genetic operations have different roles when moving around in the fitness landscape, see the schematic picture in Figure 3.23. With the crossover operation it is possible to take large steps in the fitness landscape and therefore the chances of finding the global optimum increase and the risk of getting stuck in local optima is reduced. With the mutation operator local movement is provided.

What has been described above is essentially one iteration in a simple genetic algorithm for finding the fittest individual. The genetic algorithm is summarised in Figure 3.24.

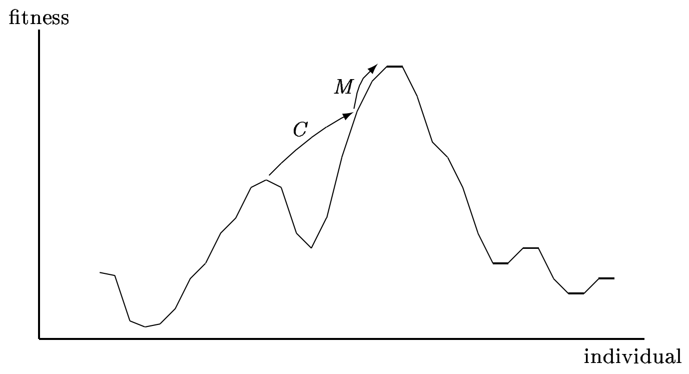


Figure 3.23: *The fitness landscape.*

First, a random initial population of individuals is chosen. Then the iterative procedure

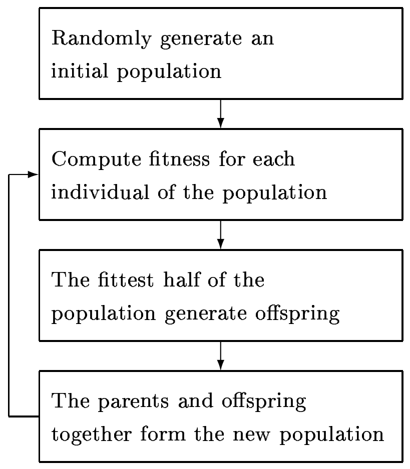

Randomly generate an initial population
Compute fitness for each individual of the population
The fittest half of the population generate offspring
The parents and offspring together form the new population

Figure 3.24: *A simple genetic algorithm.*

starts by calculating the fitness for each individual in the population. Those individuals in the population which belongs to the fittest half of the population are the parents and generate offspring according to the procedure that previously has been described. Finally, the parents and their offspring form the new population and the process repeats itself until an individual has been found whose fitness is in accordance with a given criteria.

What is interesting to note about genetic algorithms is that they are not designed to treat a specific class of problems. Rather, as quoted from Jong [13], "they are best characterised as simulation models of complex, non-linear phenomena with (hopefully) interesting and useful emergent behaviour".

3.3.2 The Cellular Automaton

In this section the genetic algorithm described in Chapter 3.3.1 is applied for solving the inverse problem for a two-dimensional cellular automaton. The lattice is finite and periodic boundary conditions are used. Just as for the elementary cellular automata and the Game of Life, the number of possible states for the cells is two and the states are denoted either 0 and 1 or white and black, i.e.,

$$S = \{0, 1\} = \{white, black\}. \tag{3.9}$$

For the rules, a von Neumann neighbourhood is used, i.e., the new state of a cell depends on the states of the cell itself and its four closest neighbours, see Figure 3.25. Since there

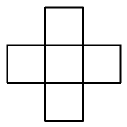


Figure 3.25: *A von Neumann neighbourhood.*

are $2^5 = 32$ combinations of states for such a neighbourhood of a cell, the number of rules that can be generated is $2^{32} \approx 10^{10}$, i.e., a huge number of rules. The rules can be stored in arrays of length 32, see Figure 3.26. If the five cells in the von Neumann neighbourhood of a cell is numbered as in Figure 3.26 then we get a correspondingly ordering of the combination of states of a von Neumann neighbourhood. For example, if the cells numbered 0, 3, and 4 are black and the cells numbered 1 and 2 are white then the outcome is found in position $2^4 + 2^3 + 2^0 = 25$ in the array storing the rule. The reason for this detailed description of

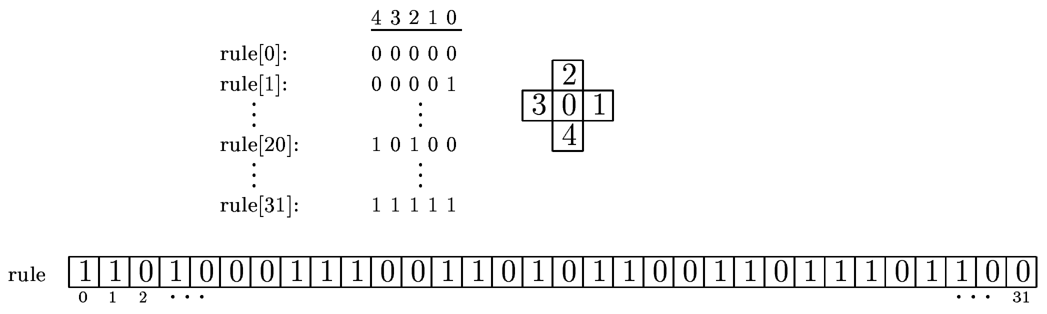


Figure 3.26: *A representation of a cellular automaton rule.*

how to represent the rules is because that the representation can be used directly in the genetic algorithm described in Chapter 3.3.1. Further, in the computer simulations the mutation operator will be applied three times for each offspring.

In order to illustrate the inverse problem, i.e., the problem of finding rules that will generate a particular output, a number a patterns of white and black of the lattice is used. The task of the genetic algorithm is to find a rule that is able to generate a given pattern. The patterns that are investigated in the computer simulations in Chapter 3.3.3 are, for example, the ones in Figure 3.27. In the computer simulations the fitness function, used

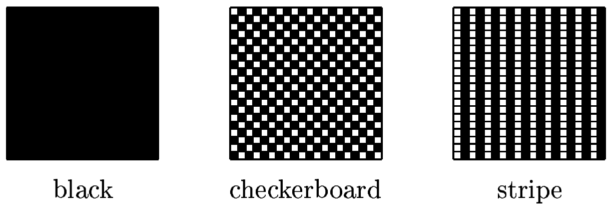


Figure 3.27: *Various patterns for the inverse problem.*

to determine the fitness of an individual (rule) in the genetic algorithm, is simply defined as the fraction of cells in the lattice that coincide with the corresponding cells in a given pattern, like the ones in Figure 3.27. In order to avoid oscillatory behaviour the fitness is calculated for several consecutive time steps and the total fitness is then calculated as the average of these fitnesses.

3.3.3 Computer Simulations

A number of computer experiments is performed for the inverse problem as defined in the previous section. The start configuration for the cellular automaton can be chosen in several ways and in Figure 3.28 the ones used in the computer simulations are presented. One consists of randomly chosen black and white cells and it is called a *random* configuration and the other one, called the *seed* configuration, consists of a lattice with just white cells except for a black cell in the middle.

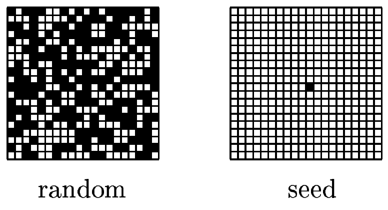


Figure 3.28: *Various starting configurations for the inverse problem.*

The Black Pattern

The first experiment deals with finding a rule, using the genetic algorithm described previously, that after 100 time steps turns a random configuration of a lattice consisting of 20×20 cells to a black configuration (a configuration with only black cells). The iterative procedure stops when the fitness function is equal to 100% or the number of iterations is greater than 400. To avoid rules that oscillates between configurations, the number of consecutive time steps where the fitness is calculated is set to three. In Figure 3.29 the number of iterations versus the population size in the genetic algorithm is given. The number of calculations for each population size is 100, and the minimum, average, and maximum number of iterations for each population size is recorded. As can be seen in Figure 3.29 there is a large spread in the number of iterations required in the genetic algorithm for a given size of the population, see the minimum and maximum number of iterations in Fig-

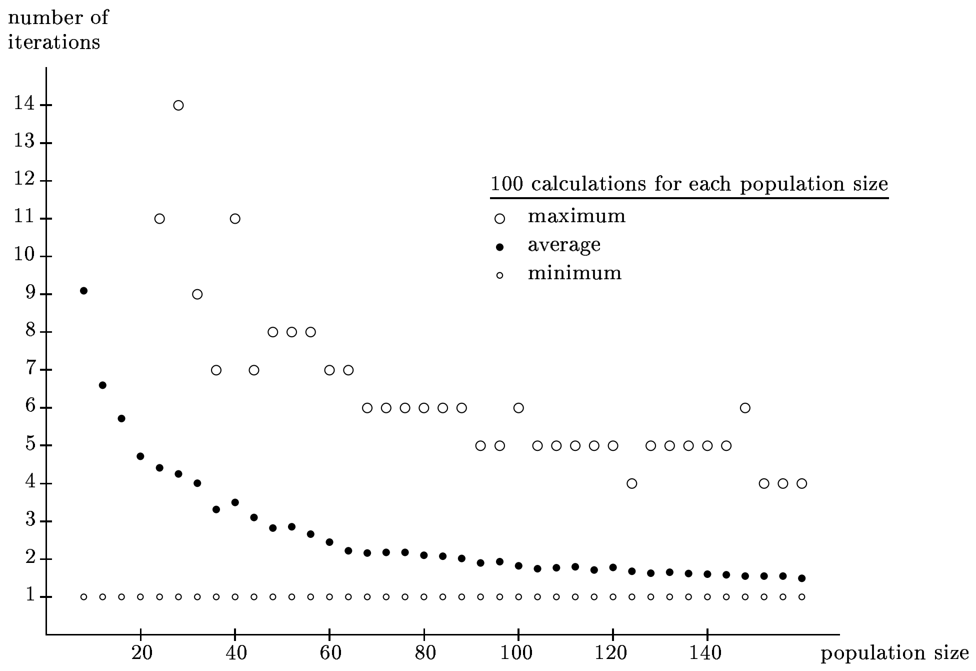


Figure 3.29: *Genetic algorithm for finding a rule that from a random start configuration gives a black configuration.*

ure 3.29. However, the spread decreases when the population size increases. The average number of iterations also decreases when the population size increases, as is expected, and for population sizes larger than 44 the average number of iterations is smaller than three.

The λ value, discussed in Chapter 2.4, for the rules found by the genetic algorithm is also recorded for the experiment described above, and the result is displayed in Figure 3.30. Recall that the λ value is equal to the number of 1's in a rule string divided by 32 (the length of the string). Since the task of the genetic algorithm is to find a rule that turns a random start configuration to a configuration with only black cells, it is natural that the λ value will be somewhere between 0.5 and 1.0. A rule with only 1's ($\lambda = 1.0$) is a solution to the problem but as can be seen in Figure 3.30 this solution is not found by the genetic algorithm. It seems that the λ value for the fittest rule is independent of the population size, at least for population sizes below 160, and that the λ value most likely is between 0.50 and 0.81. The average value is around 0.68.

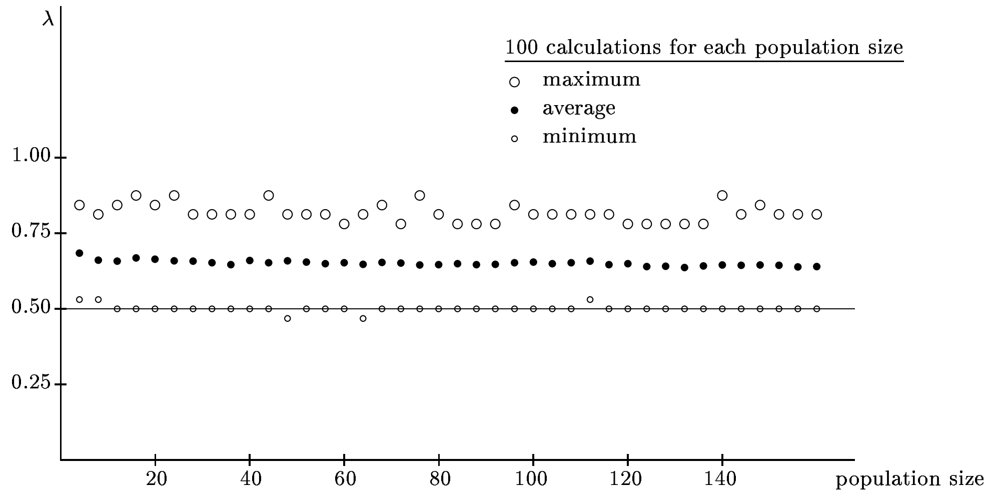


Figure 3.30: *λ values for rules resulting from the genetic algorithm searching for rules that from a random start configuration give a black configuration.*

In Figure 3.29 the fitness is set to the highest possible value (100%). Smaller fitness values can also be used and in Figure 3.31 a comparison of the number of iterations required in the genetic algorithm are displayed for the fitness values 0.90%, 0.95%, and 100% as convergence criteria. For each population size and fitness value, 100 calculations

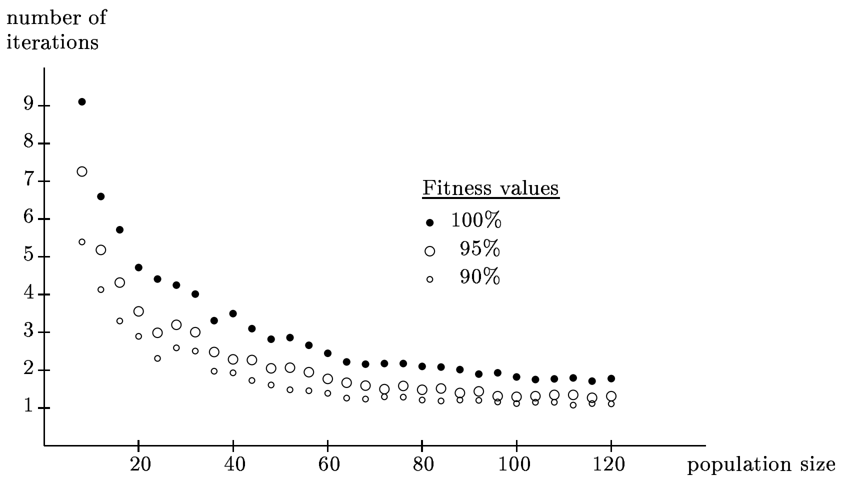


Figure 3.31: *Average number of iterations in the genetic algorithm for finding a rule that from a random start configuration gives a black configuration.*

are performed, and the average number of iterations is obtained. As is illustrated by

Figure 3.31 the number of iterations of the genetic algorithm converges rapidly to 1 iteration with increasing population sizes when the convergence criteria is set to 90% fitness value. However, going from 90% fitness to 100% fitness usually means, in this problem, an increase in the number of iterations by about one or two on average.

Finally, the time evolution of a cellular automaton using the most fittest rule obtained in a genetic algorithm using a population size equal to 40 is calculated and displayed, see Figure 3.32.

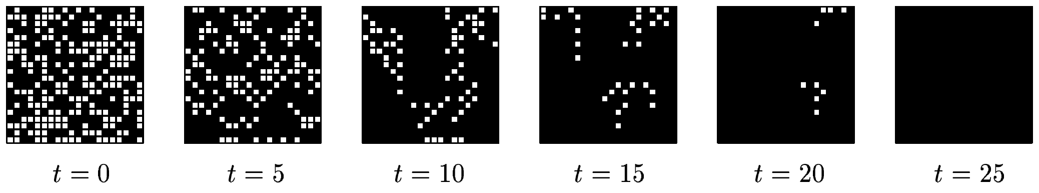


Figure 3.32: *Time evolution of a cellular automaton using the most fittest rule obtained in a genetic algorithm for the black pattern problem.*

The Checkerboard Pattern

In the next experiment a more advanced pattern, namely the checkerboard pattern in Figure 3.27, is used. The start configuration is the *seed* configuration in Figure 3.28, otherwise the parameters are the same as in the previous experiment. In Figure 3.33 the number of iterations required in the genetic algorithm to find a rule versus the size of the population is given. Just as in the previous experiment the spread in data are large for small population sizes. It can also be noted that the number of iterations required is about 20 times larger for this more advanced experiment than the previous one. The λ values are also calculated and they are displayed in Figure 3.34. The λ values ranges between about 0.25 and 0.71, with an average value around 0.5. In this experiment, the symmetry between white and black in the checkerboard pattern is reflected in the λ values. The average value is closer to the value 0.5 than in the previous experiment. For population

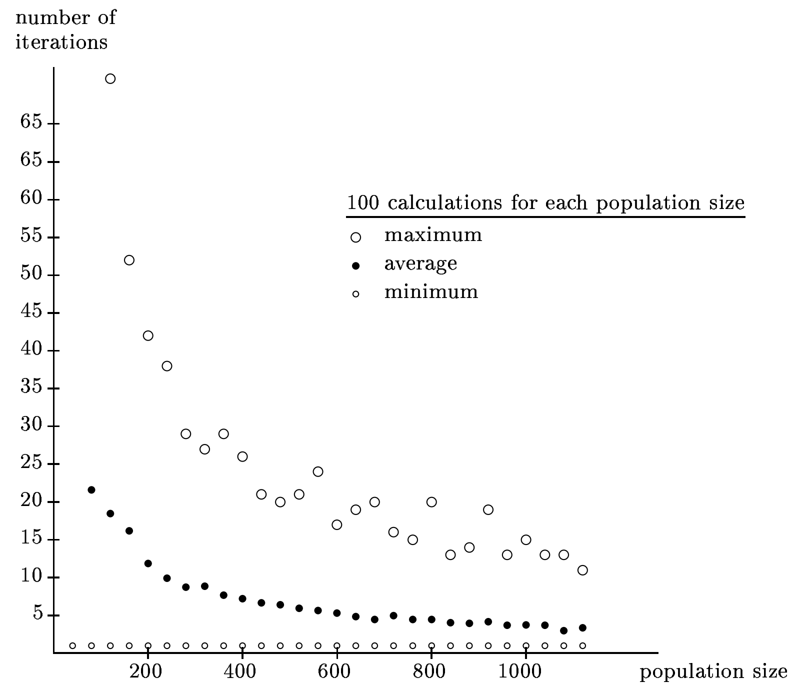


Figure 3.33: *Genetic algorithm for finding a rule that from a seed start configuration gives a checkerboard configuration.*

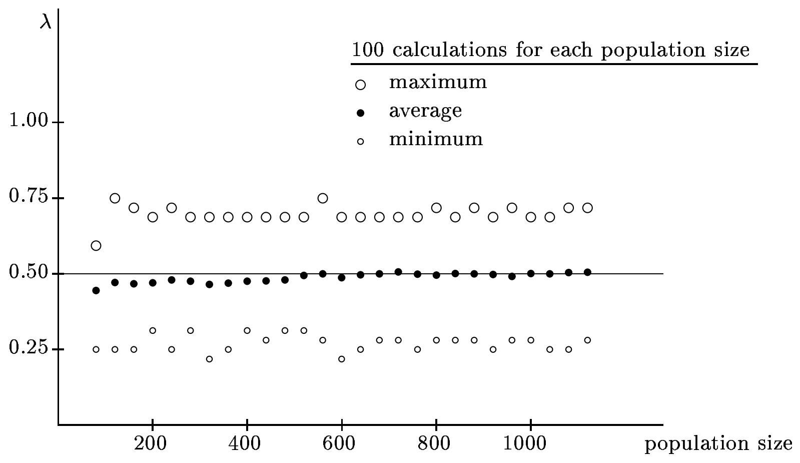


Figure 3.34: *λ values for rules resulting from the genetic algorithm searching for rules that from a seed start configuration give a checkerboard configuration.*

sizes less than about 500 the average value is consistently slightly less than 0.5 and this is probably due to the asymmetry in black and white of the start configuration (only one black cell).

Finally, the time evolution of a cellular automaton using the most fittest rule obtained in a genetic algorithm using a population size equal to 40 is calculated and displayed, see Figure 3.35. Also a random initial configuration will evolve to a checkerboard pattern using the same rule.

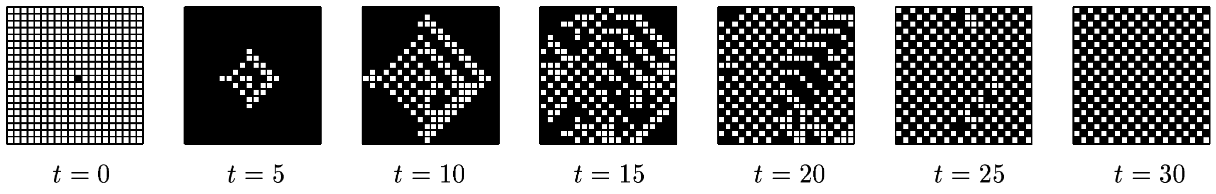


Figure 3.35: *Time evolution of a cellular automaton using the most fittest rule obtained in a genetic algorithm for the checkerboard pattern problem.*

The Stripe Pattern

By exchanging the checkerboard pattern to a stripe pattern (see Figure 3.28) the results in Figures 3.36 and 3.37 are obtained. The number of iterations required in the genetic algorithm for the stripe pattern is about twice as many as for the checkerboard pattern. The λ values ranges between about 0.35 and 0.65, with an average value around 0.5. In this experiment, the minimum and maximum values are more symmetrical located around the value 0.5 then in the checkerboard problem. However, the average value is also for this problem slightly less than 0.5, probably due to the asymmetry in black and white of the start configuration (only one black cell).

Finally, the time evolution of a cellular automaton using the most fittest rule obtained in a genetic algorithm using a population size equal to 40 is calculated and displayed, see Figure 3.38. Also a random initial configuration will evolve to a stripe pattern using the same rule.

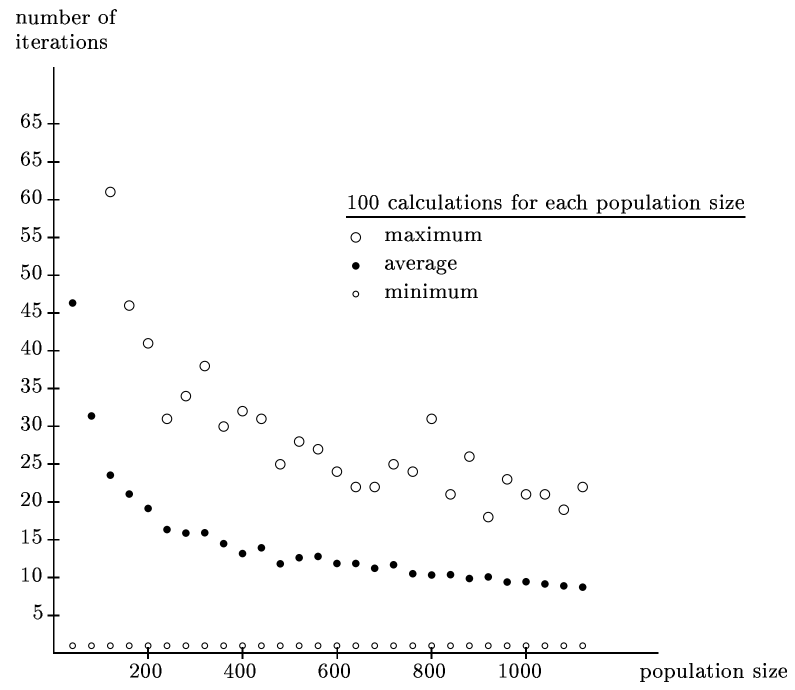


Figure 3.36: *Genetic algorithm for finding a rule that from a seed start configuration gives a stripe configuration.*

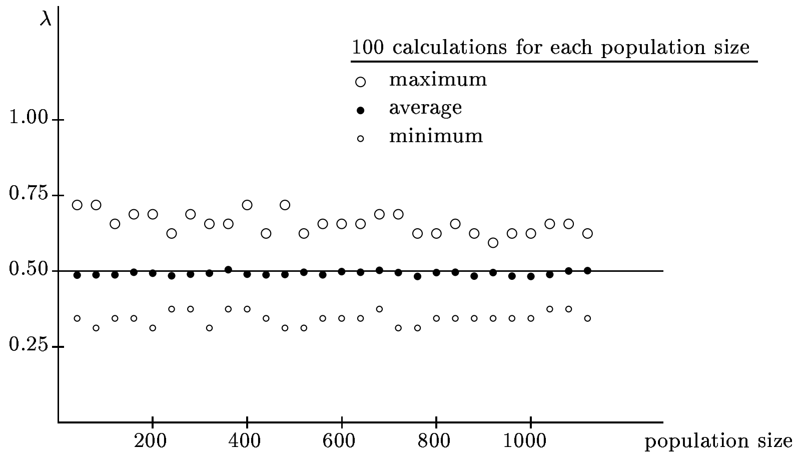


Figure 3.37: *λ values for rules resulting from the genetic algorithm searching for rules that from a seed start configuration give a stripe configuration.*

Distribution of λ values

In all the three experiments described previously, the initial rules in the population are chosen randomly. This means that most rules initially will be centred around $\lambda = 0.5$ since

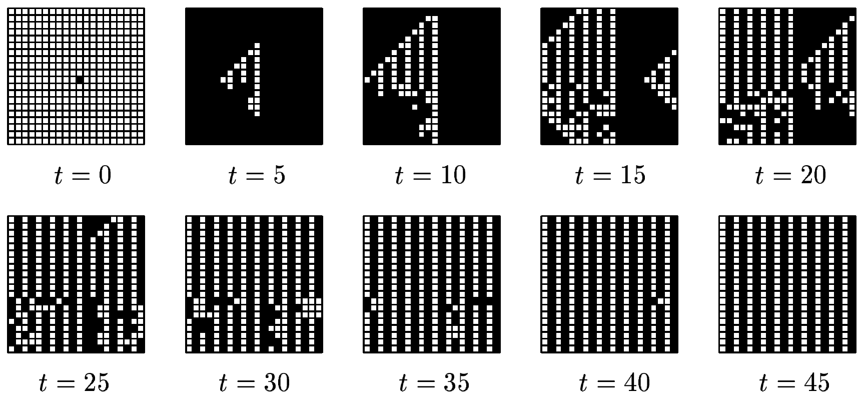


Figure 3.38: *Time evolution of a cellular automaton using the most fittest rule obtained in a genetic algorithm for the stripe pattern problem.*

of the 2^{32} possible rules most will be found here. For $\lambda = 0$ or 1 there is, on the other hand, only one possible rule. In Figure 3.39 the distribution of 100×200 randomly generated rules with respect to the λ value is given. There are only 33 possible values for λ, namely

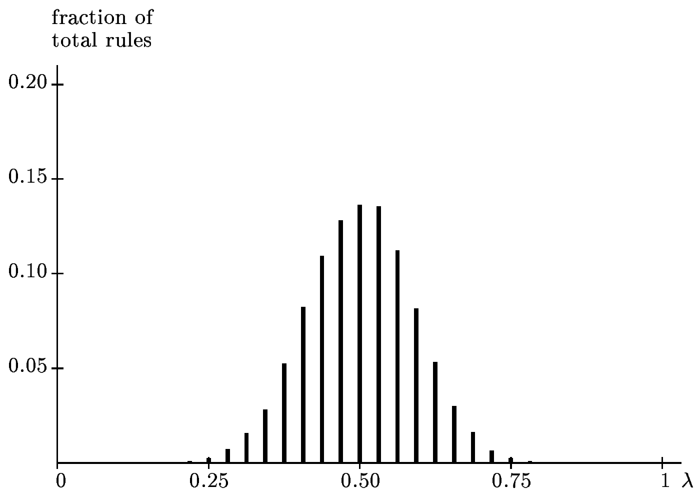


Figure 3.39: *Average distribution of λ for randomly chosen initial populations.*

0, 1/32, 2/32, ..., 31/32, and 1, and each value is represented in Figure 3.39 by a bar. When using a distribution of initial rules like the one in Figure 3.39 the genetic algorithm with the stripe pattern converges after 19 iterations on average with a population size of 200. The distribution of rules in the population after convergence is slightly modified from the initial distribution. In Figure 3.40 the average distribution of the 100 elite rules, obtained when running the stripe pattern problem 100 times using a population size of 200, with respect to the λ value is given. The distribution has been slightly shifted over

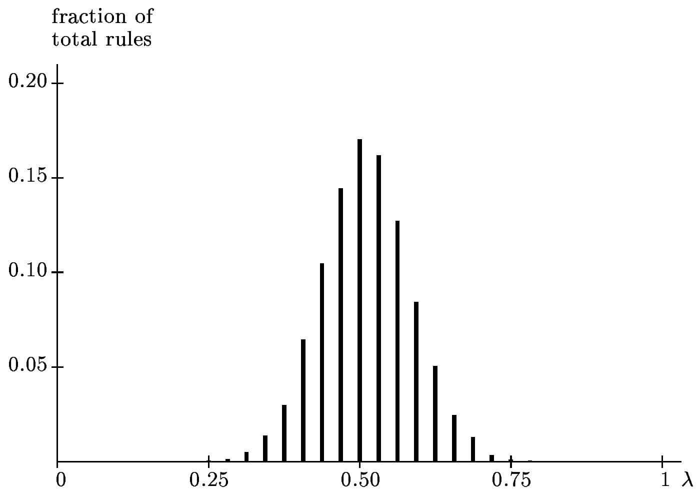


Figure 3.40: *Average distribution of λ for the 100 elite rules in the population after convergence of the stripe pattern problem starting from a random-distributed population.*

to the centre values. An interesting experiment would be to investigate how the shape of the distribution of the initial population affects the outcome of the genetic algorithm. If an evenly distributed initial population, like the one in Figure 3.41, is used instead of a random distribution then the genetic algorithm with the stripe pattern converges after 23 iterations on average with a population size of 200. Thus the genetic algorithm for the

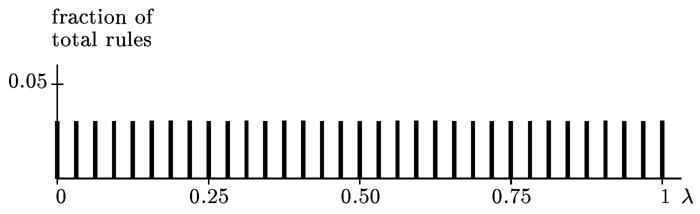


Figure 3.41: *Even distribution of λ for rules in a randomly chosen initial population.*

stripe problem, on average, converges faster using a initial population that is randomly distributed. This makes sence since the fittest rule on average has a λ value of 0.5 and a totally random initial population has a concentration of individuals around this value. In Figure 3.42 the average distribution of the 100 elite rules, obtained when running the stripe pattern problem 100 times using a population size of 200, with respect to the λ value is given. The resulting distribution is broader than the one in Figure 3.40, and it is centered

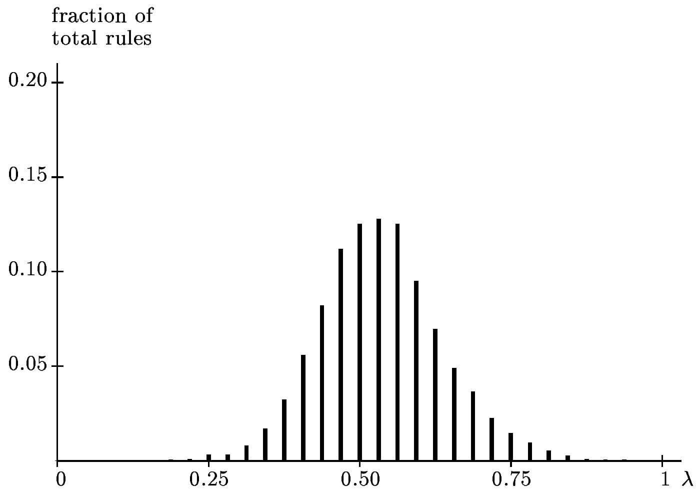


Figure 3.42: *Average distribution of λ for the 100 elite rules in the population after convergence of the stripe pattern problem starting from an even random-distributed population.*

around a value slightly larger than 0.5. The asymmetry of the distribution around $\lambda = 0.5$ might be due to the asymmetry of white and black in the initial configuration.

The result using an evenly distributed initial population for the checkerboard problem is displayed in Figure 3.43. The average number of iterations is 11 for population sizes of 200. In comparison to the stripe pattern problem the distribution is broader for the checkerboard pattern problem. This may be due to the fact that among the 100 calculations there are also calculations requiring only one iteration, and the elite populations from those calculations are probably more evenly distributed among the λ values. However, the asymmetry around $\lambda = 0.5$ is also present in the checkerboard pattern problem. For totally random initial populations the distributions of the elite rules are similar for the stripe and checkerboard pattern problems.

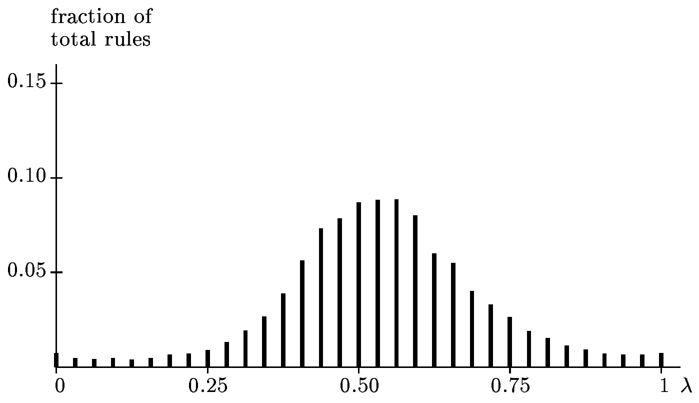


Figure 3.43: *Average distribution of λ for the 100 elite rules in the population after convergence of the checkerboard pattern problem starting from an even random-distributed population.*

A discussion of the black pattern problem in relation to the distribution of λ values in the initial population will be given in the next section.

The $\rho_c = 1/2$ Classification Task

The $\rho_c = 1/2$ classification task was originally discussed by Packard [21] and later on reexamined by Mitchell and *et al.* [19]. The $\rho_c = 1/2$ classification task is defined as follows: if the number of black cells is larger then the number of white cells initially then the cellular automaton will evolve to an entirely black configuration, and conversely, if the number of black cells is less than the number of white cells initially then the cellular automaton will evolve to an entirely white configuration.

A genetic algorithm is used in determining a rule that solves the $\rho_c = 1/2$ classification task. Each rule in the population is tested against ten random initial configurations with 5%, 15%, . . ., and 95% white cells, respectively. The total fitness for a rule is calculated as the average of the fitnesses obtained in the ten tests. To get a high fitness score, a rule has to do well in all tests. The size of the population is set to 100. In order for comparison the black pattern problem is calculated as well. In Figure 3.44 the results from the calculations are summarised, using both distributions of λ values discussed previously. To get enough

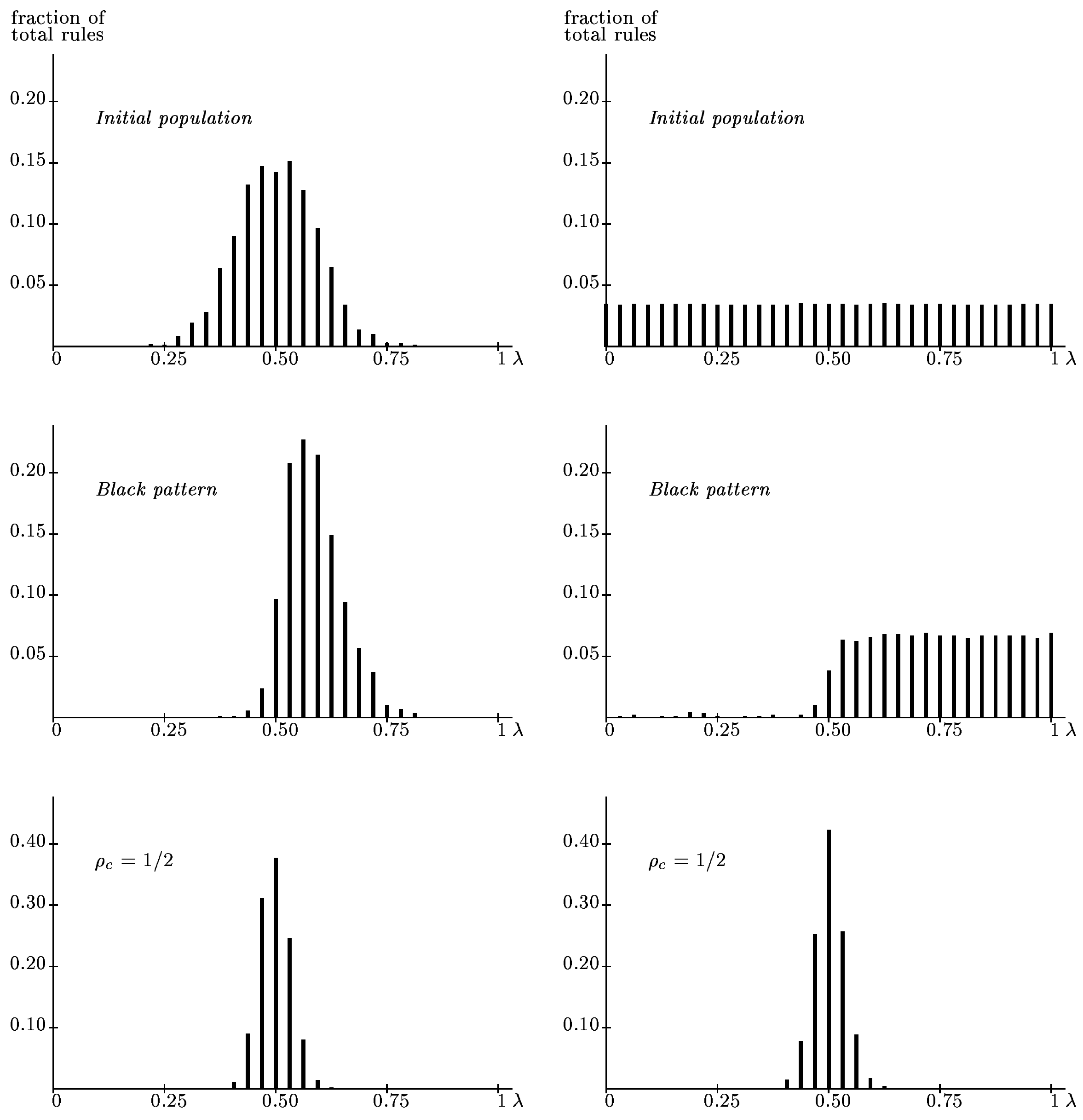


Figure 3.44: *Distribution of λ for the elite rules in a population of 100 rules after convergence of the black pattern problem and the $\rho_c = 1/2$ classification task, respectively.*

statistics the calculation is performed 20 times with different random-number seeds for each calculation. This means that $20 \times 100 = 2000$ rules are included in the initial populations in Figure 3.44. Since only the elite rules of the population are taken into account after convergence, only $20 \times 50 = 1000$ rules are included in the converged populations in Figure 3.44. The black pattern problem needs 1–5 iterations to converge using an initial population with random-distributed λ values in the 20 calculations performed. With evenly random-distributed λ values the black pattern problem converges immediately after just one iteration. The number of iterations required in the $\rho_c = 1/2$ classification task is 8–36 and 11–41, respectively, for the random-distributed and the evenly random-distributed λ values for the rules in the initial populations.

As can be seen from Figure 3.44 the λ distribution of the fittest rules of the $\rho_c = 1/2$ classification task is more or less symmetric around $\lambda = 0.5$. For this value the distribution also has a maximum. These two facts apply independently of which distribution of λ values that is used in the initial population. This makes sense since the problem is symmetric with respect to black and white. However, this result is in contrary to the results obtained by Mitchell and *et al.* [19] and Packard and [21]. Instead of using a 2-dimensional lattice these two groups used a 1-dimensional lattice and the neighbourhood of a cell consisted of 7 cells, which in total gives rules of length $2^7 = 128$ and the total number of possible rules 2^{128}. Both groups used an initial population of evenly distributed λ values. The calculational details differ somewhat in all calculations but the main parts are similar. The λ distribution of the fittest rules of the $\rho_c = 1/2$ classification task from these two groups has a minimum at $\lambda = 0.5$ and obviously two maxima. Mitchell and *et al.* obtained the maxima around $\lambda = 0.43$ and $\lambda = 0.57$, i.e., fairly close to $\lambda = 0.5$. Packard, on the other hand, obtained the maxima around $\lambda = 0.24$ and $\lambda = 0.76$, i.e., much further away from $\lambda = 0.5$. Packard interpreted these results as giving evidence of the hypotheses that

- cellular automata rules that are able to perform complex computations are most likely to be found in the critical area between the ordered and chaotic regimes (see Chapter 2.4),

- when cellular automata rules are evolved to perform complex computations, evolution will tend to select rules in the critical area between the ordered and chaotic regimes.

Since Mitchell and *et al.* did not arrive at the same results as Packard, even though their intention was to redo Packards experiment, it is hard to believe that the hypotheses above are valid. The results in Figure 3.44 also give no evidence of the hypotheses. On the contrary, the λ value for the most fittest rule for the $\rho_c = 1/2$ classification task is most likely around 0.5, i.e., in the middle of the chaotic regime.

In Figure 3.44 the results from the black pattern problem are presented as well. This problem is not symmetric with respect to white and black and this is clearly seen in the λ values for the fittest rules, rules with $\lambda > 0.5$ are usually selected.

3.3.4 Program Description

The C^{++}-code for the genetic algorithm for 2-dimensional cellular automata with a von Neumann neighbourhood and with two possible states for the cells can be found in Appendix C. The code is based on the description given in Chapters 3.3.1 and 3.3.2 and it is fairly straight forward consisting of a large number of small functions.

Chapter 4

Evaluation

After the demonstration in Chapter 3.2 of the workings of a cellular automaton on a real physical problem it is appropriate to wonder how well the cellular automaton model do in comparison with other numerical methods and experiment. Since no numerical results are presented in Chapter 3.2 only a qualitative comparison is made.

A fluid flow is often characterised by a dimensionless quantity called the Reynolds number, Re. This quantity was introduced by Osborne Reynolds when making fluid flow experiments using glass tubes. Reynolds number is defined as

$$Re = \rho v d/\mu, \tag{4.1}$$

where ρ is the density of the fluid, v the average velocity of the fluid, d the glass tube diameter, and μ the viscosity of the fluid. Reynolds number is used also for other flows than flows in tubes and then the diameter d is replaced by another representative size of the problem.

In the cellular automaton model of fluid flow the pressure, modelled as the insertion and deletion of fluid particles to the left and right, respectively, is used to vary the velocity of the fluid. Large pressures give large velocities and therefore large Reynolds numbers.

By modifying the pressure fluid flows of different Reynolds numbers can be simulated.

For exact descriptions of fluid flows, the Navier-Stokes equations of motion have to be solved simultaneously with the continuity equation. However, the Navier-Stokes equations are non-linear and it is difficult to solve them analytically. For small and large Re certain terms can be neglected and analytical solutions can be obtained. For intermediate values of Re the equations cannot be simplified and they have to be solved numerically. There are a number of approaches for doing that, for example, the finite difference method and the finite element method. In Allievi *et al.* [2] a finite element method calculation of flow around a circular cylinder at $Re = 100$ is described. The finite element grid consists of quadrangular elements of varying sizes with the smallest elements close to the cylinder and above and below the cylinder perpendicular to the flow. The resulting time evolution of the streamlines of the flow is presented. In comparison, it seems that the flow in Figure 3.15 has a larger Reynolds number. In order to facilitate a comparison the calculations in Figure 3.15 are redone with a smaller value of the pressure, $p = 600$. The resulting time evolution of the flow is presented in Figure 4.1, but the lines printed now represent average velocities (laboratory reference frame) instead of average velocities minus the average velocity of the entire fluid (moving reference frame). Figure 4.1 should be compared to Figures 6 and 7

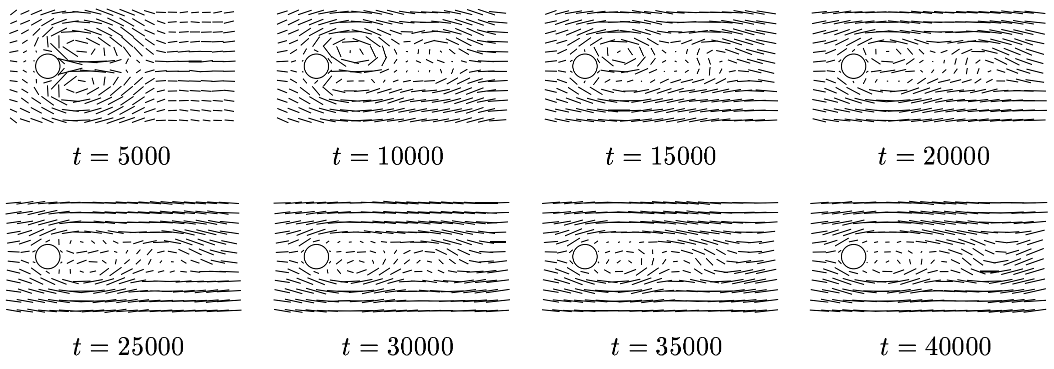


Figure 4.1: *Evolution of flow around a circular cylinder (laboratory reference frame).*

in Allievi *et al.* [2]. In both models, a pair of symmetric vortices initially grow behind the cylinder. At a certain time the vortices become asymmetrical and then unstable. From then onwards they begin shedding vortices alternatively, giving rise to the wavy pattern. It should be pointed out that no artificial triggering was necessary, neither to introduce the asymmetry in the wake nor to provoke vortex shedding in either of the models. Thus it can be concluded that the two models qualitatively give similar descriptions of flow around a circular cylinder.

Flow around cylinders has also been investigated experimentally. In Mills *et al.* [18] flow around rectangular cylinders has been investigated. I Figure 3 in Mills *et al.* [18] both velocity vector and streamline plots using different reference frames of flow at $Re = 490$ around a rectangular cylinder are given. The similarities with Figure 3.14 is striking for the flow behind the cylinder. In the experiment vortices are also created above and below the rectangular cylinder which, of course, are not present in Figure 3.14 since a circular cylinder is used. However, in the cellular automaton model it is straightforward to modify the shape of the cylinder. In Figure 4.2 the calculations in Figure 3.14 are repeated but with a rectangular cylinder instead. Because of the elongated obstacle a larger lattice is

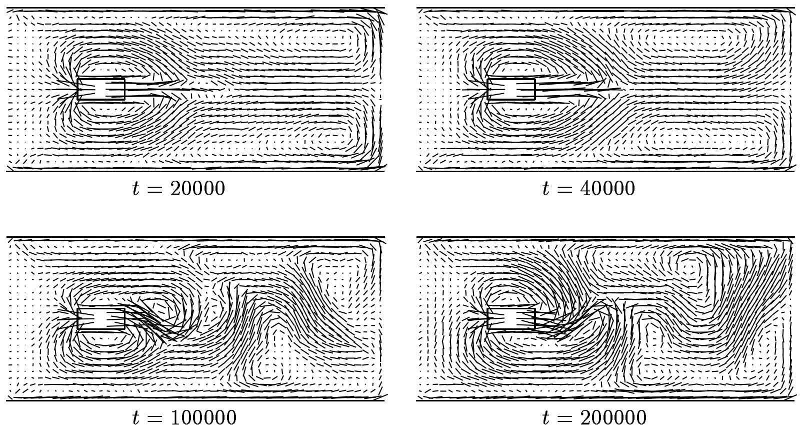


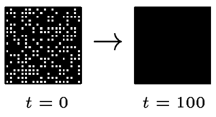
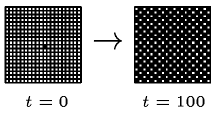
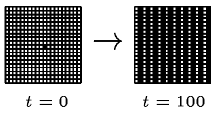
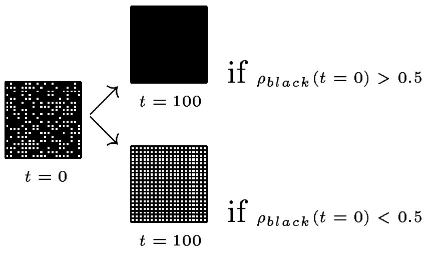
Figure 4.2: *Flow around a rectangular cylinder (moving reference frame).*

used, 2000×1000 nodes instead of 1500×1000. The lines in Figure 4.2 represent average velocities of fluid particles from 40×40 nodes minus the average velocity of the entire fluid. In Mills *et al.* [18] a large value of the Reynolds number was used ($Re = 490$) in comparison to the value ($Re = 100$) used in Allievi *et al.* [2] using a cicular cylinder. In Figure 4.2 the same pressure ($p = 900$) is used as in Figure 3.14. Increasing the pressure too much above this value leads to what seems turbulent behaviour. It turned out that the creation of vortices is sensitive to the dimensions of the rectangle in the cellular automaton model. The rectangle should not be too thin in the direction perpendicular to the flow and not too wide in the other direction. The dimensions in Figure 4.2 turned out to be all right. However, the creation of vortices is still sensitive to other parameters in the model, for example, the number of nodes in the direction of the flow. The critical point in the generation of vortices is the origin of some asymmetry which is clearly seen after 40000 time steps in Figure 4.2. From then on vortices can be generated and after 100000 time steps a vortex street is created. Vortices are also present below and above the rectangular cylinder. Thus, the cellular automaton model for flow around rectangular cylinders gives results that are in qualitative agreement with experiment.

For the hydrodynamic example in Chapter 3.2 it seems that the cellular automaton model is able to reproduce the characteristics of fluid flow which are in agreement with results from numerical methods, like the finite element method, and experiment. Since no Reynolds number is defined in the cellular automaton model it is hard to make any quantitative comparisons. The advantages of the cellular automaton model is that it is extremely simple and straightforward both to use and to implement, while methods like the finite element method with its equations and iterative procedures are more difficult to grasp. For future comparisons of various methods for describing fluid flow, it will be mentioned that the calculation described in Figure 3.14 took about 2 hours on an Intel(R) Pentium(R) 4 Microprocessor (1.70 GHz).

A cellular automaton is by its very definition spatio-temporal. The lattice represents the space and the evolution gives the dimension of time. Problems that are truly spatio-temporal should be ideally suited for a cellular automaton. For other problems, maybe a

Table 4.1: *Application of a genetic algorithm.*

Problem	Number of iterations[a]			CPU time[b] (s)
	Min	Max	Average	
	1	6	2	0.8
	1	220	24	5.6
	1	73	26	6.0
 $t = 100$ if $\rho_{black}(t = 0) > 0.5$ $t = 0$ $t = 100$ if $\rho_{black}(t = 0) < 0.5$	5	43	18	44.6

[a]Each problem is run 100 times and Min, Max, and Average refer to the minimum, maximum, and average number of iterations required in the genetic algorithm with a population size of 100.
[b]On an Intel(R) Pentium(R) 4 Microprocessor (1.70 GHz) for one single run on average.

first transformation to some kind of spatio-temporal problem will make also them suitable to cellular automata. What might be the bottleneck is to find the proper rules for a given problem. Here genetic algorithms may be useful. In Chapter 3.3 a demonstration of the workings of a genetic algorithm for finding cellular automaton rules for some simple problems is given. The results are summarised in Table 4.1. Each problem can be described as finding a rule that evolves a given start configuration to another given final configuration after 100 time steps. The two-dimensional lattice consists of 20×20 cells and a von Neumann neighbourhood is used when updating the cells. Each problem is run 100 times and the minimum, maximum, and average number of iterations required in the genetic algorithm to solve the problem is recorded. The population size of the genetic algorithm is set to 100. Although the problems are simple it is still impressive to see how stable the genetic algorithm is and actually how fast rules can be found. The black pattern problem (see the first row in Table 4.1) is, not surprisingly, the fastest, while the checkerboard and stripe pattern problems (see the second and third rows in Table 4.1) requires about ten times more iterations and computing time. The $\rho_c = 1/2$ classification task (see the last row in Table 4.1) requires most computing time and this is not surprising since many start configurations (with different densities of black cells) are used in each iteration of the genetic algorithm.

In the literature there are examples where the procedure of using genetic algorithms has been used on real problems. For example, in Meyer *et al.* [17] and Richards *et al.* [22] a procedure is described of finding cellular automaton rules directly from experimental data. The experiment concerns the solidification of NH_4Br from a supersaturated aqueous solution. The solution was put between two glass plates (thus made two-dimensional) and put in a water bath where the temperature could be modified. Below a certain temperature and by tapping on the apparatus the growth of the crystal is initiated. The evolution of the growth is recorded by a video camera, and thus a two-dimensional pattern evolving in time is obtained. The pattern is dendritic, where the dendrites form sidebranches, which in turn may form sidebranches, and so on. The recorded evolution of the pattern in time can be used to find a cellular automaton rule using some kind of genetic algorithm. In Richards *et al.* [22] a two-dimensional lattice was used and the new state of a cell does not only depend on the states of the cells in the von Neumann neighbourhood, but also on states of cells further away both in space and time. This, of course, gives rise to a huge number of possible rules. Nevertheless, the genetic algorithm is able to find a rule that gives an evolution that is in line with the one experimentally recorded.

Chapter 5

Conclusion

Cellular automata are powerful computational constructs and even the simpler ones, like the rule 110 elementary cellular automaton and the Game of Life, do have the property of being universal. That means that providing the right input (starting configuration) it is possible for them to calculate whatever that is computable. This is indeed a nice property, but maybe in practice it is of less use since the task of providing the cellular automaton with the right input might be tremendously difficult.

Another property of cellular automata that impresses is the locality of the rules. The state of a cell is affected only by its immediate neighbourhood when going from one time step to another. This does not mean that a cell is not influenced by cells far away, on the contrary, after enough time steps the effect of a particular cell might be far reaching, see for example the gliders in the Game of Life that steadily move on without changing the direction. The locality of rules also seems sensible in the description of many natural phenomena, from the growth of snowflakes and crystals to the movement of particles like in the hydrodynamic example in Chapter 3.2. To use a discrete model like a cellular automaton for describing natural phenomena might seem inappropriate trained as we are as seeing natural laws as continuous equations. However, these continuous equations are often solved using computers, which means that in one way or the other a discrete representation is used. As a matter of fact there are several scientists, starting with Konrad Zuse in the 1960s, that have proposed that the physical laws of the universe are discrete by nature. Konrad Zuse is well known for his pioneering work in several areas [25]. During the second world war he built programmable computers from electromechanical relays and in 1945 he developed the first high-level programming language called Plankalkül. Zuse was in many respects ahead of his time and in 1969 he proposed in "Rechnender Raum" (Calculating Space) that the entire universe is just the output of a deterministic calculation on a cellular automaton [33]. Zuse's work was followed by others, for example Edward Fredkin and his "Digital Mechanics" [7] and Stephen Wolfram and his "A New Kind of Science" [32]. Stephen Wolfram has formulated a principle which he calls the Principle of Computational Equivalence, which states that

> ... all processes, whether they are produced by human effort or occur spontaneously in nature, can be viewed as computations.

So far there is no physical evidence against the possibility that everything is just a computation. The following citation by Richard Feynmann is illustrative [8].

> It always bothers me that, according to the laws as we understand them today, it takes a computing machine an infinite number of logical operations to figure out what goes on in no matter how tiny a region of space and no matter how tiny a region of time. How can all that be going on in that tiny space? Why should it take an infinite amount of logic to figure out what one tiny piece of space-time is going to do?

These are well-founded questions and indeed one may wonder whether there exists alternative descriptions of the universe that are simpler than those presently in use.

The final property of cellular automata that will be discussed here is the simplicity of the rules. As has been demonstrated in this thesis, despite the simplicity of the rules, cellular automata may display enormous complexity. The programming effort is also considerably small and therefore cellular automata might provide cost effective and safe programs for describing various phenomena, not only in science and computer science, but in other areas, like social sciences and biology, as well.

Cellular automata are indeed powerful, but what could possibly be a drawback in their usage is the problem of finding the proper rules for a given computational task. As is demonstrated in Chapter 3.3, genetic algorithms may provide solutions to that problem.

Chapter 6

Future Work

The field of cellular automata is huge and in this thesis only a tiny fraction of this immense exciting field has been covered. There are a number of items that have been left out and that remain to be studied:

- For the hydrodynamic example it would be interesting if something that corresponds to Reynolds number could be worked out. Further, the effect of using detailed balance when deriving macroscopic equations starting from the cellular automaton model should be investigated.

- In order to gain experience in how well cellular automata are suited for simulating dynamical systems, some more systems, physical and others, should be studied.

- Quantum mechanical systems have not been mentioned at all in this thesis, but in fact there are applications of cellular automata also in this field, see for example Boghosian *et al.* [5] and references therein. Calculations for solving the many-particle non-relativistic Schrödinger equation are expensive and alternative approaches using, for example, cellular automata, might be a step in the right direction. But before attempting to solve the Schrödinger equation using cellular automata, basic principles like the Pauli exclusion principle has to be understood in the framework of cellular automata.

- Genetic algorithms seem powerful and a more in-depth analysis of them would be desirable. An interesting application of genetic algorithms is in finding computer programs in an automated way that enables a computer to solve a given problem. This application area is called genetic programming [14] and the goal here is to make computers do what needs to be done without telling them exactly how to do it. Here the population, which the genetic algorithm operates on, consists of computer programs of varying sizes.

References

[1] Conway's Game of Life. http://en.wikipedia.org/wiki/Conway's_Game_of_life, 2004.

[2] Alejandro Allievi and Rodolfo Bermejo. Finite Element Modified Method of Characteristics for the Navier-Stokes Equations. *Int. J. Numer. Meth. Fluids*, 32:439–464, 2000.

[3] Kerstin Andersson. Cellular Automata. Bachelor's Project 2003:26, Karlstad university, Sweden, 2003.

[4] S. C. Benjamin and N. F. Johnson. A Possible Nanometer-scale Computing Device based on an Adding Cellular Automaton. *Applied Physics Letters*, 70(17):2321–2323, 1997.

[5] Bruce M. Boghosian and Washington Taylor IV. A Quantum Lattice-Gas Model for the Many-Particle Schrödinger Equation in d Dimensions. Preprint quant-ph/9604035, Jan 1997.

[6] James Maxwell Buick. *Lattice Boltzmann Methods in Interfacial Wave Modelling.* PhD thesis, University of Edinburgh, Edinburgh, 1997.

[7] Edward Fredkin. Digital Mechanics. http://digitalphilosophy.org/download_documents/digital_mechanics_book.pdf, 2000.

[8] Edward Fredkin. Digital Mechanics. http://www.digitalphilosophy.org/dm_paper.htm, 2004.

[9] U. Frisch, D. d'Humières, B. Hasslacher, P. Lallemand, Y. Pomeau, and J.-P. Rivet. Lattice Gas Hydrodynamics in Two and Three Dimensions. *Complex Systems*, 1(4):649–707, 1987.

[10] U. Frisch, B. Hasslacher, and Y. Pomeau. Lattice-Gas Automata for the Navier-Stokes Equation. *Phys. Rev. Letters*, 56(14):1505–1508, 1986.

[11] Niloy Ganguly, Biplab K. Sikdar, Andreas Deutsch, Geoffrey Canright, and P. Pal Chaudhuri. A Survey on Cellular Automata. http://www.cs.unibo.it/bison/publications/CAsurvey.pdf, 2001.

[12] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, New York, 2001.

[13] Kenneth De Jong. Genetic Algorithms: a 30 Year Perspective. http://www.pscs.umich.edu/jhhfest/Papers/dejong.ps.gz, 1999.

[14] John R. Koza. Human-Competitive Machine Intelligence by Means of Genetic Algorithms. http://www.genetic-programming.com/jkpdf/hollandfest1999.pdf, 1999.

[15] Chris G. Langton. Computation at the Edge of Chaos: Phase Transitions and Emergent Computation. *Physica D*, 42:12–37, 1990.

[16] Norman Margolus, Tommaso Toffoli, and Gérard Vichniac. Cellular-Automata Supercomputers for Fluid-Dynamics Modeling. *Phys. Rev. Letters*, 56(16):1694–1696, 1986.

[17] Thomas P. Meyer, Fred C. Richards, and Norman H. Packard. Learning Algorithm for Modeling Complex Spatial Dynamics. *Phys. Rev. Letters*, 63(16):1735–1738, 1989.

[18] Richard Mills, John Sheridan, and Kerry Hourigan. Particle Image Velocimetry and Visualization of Natural and Forced Flow Around Rectangular Cylinders. *J. Fluid Mech.*, 478:299–323, 2003.

[19] M. Mitchell, P. T. Harber, and J. P. Crutchfield. Revisiting the Edge of Chaos: Evolving Cellular Automata to Perform Computations. *Complex Systems*, 7:89–130, 1993.

[20] Y. Nakayama and R. F. Boucher. *Introduction to Fluid Mechanics.* Arnold, London, 1999.

[21] N. H. Packard. *Adaptation Toward the Edge of Chaos.* World Scientific, Singapore, 1988. In J. A. S. Kelso, A. J. Mandell, and M. F. Schlesinger, editors, Dynamic Patterns in Complex Systems.

[22] Fred C. Richards, Thomas P. Meyer, and Norman H. Packard. Extracting Cellular Automaton Rules Directly from Experimental Data. *Physica D*, 45:189–202, 1990.

[23] Daniel H. Rothman and Stéphane Zaleski. *Lattice-Gas Cellular Automata: Simple Models of Complex Hydrodynamics.* Cambridge University Press, 1997.

[24] Frank Schweitzer and Jörg Zimmermann. *Communication and Self-Organization in Complex Systems: A Basic Approach.* Springer, Berlin, 2001. In M. M. Fischer and J. Fröhlich, editors, Knowledge, Complexity and Innovation Systems.

[25] Robert W. Sebesta. *Concepts of Programming Languages.* Addison-Wesley, 2002.

[26] A. R. Smith(III). Real-Time Language Recognition by One-Dimensional Cellular Automata. *J. Comput. Syst. Science*, 6:233–253, 1972.

[27] S. Ulam. *Random Processes and Transformations.* Cambridge, MA, 1950. In L. M. Graves, E. Hille, P. A. Smith and O. Zarski, editors, Proceedings of the International Congress of Mathematicians (Providence, RI: AMS 1952)II.

[28] J. von Neumann. *Theory of Self-reproducing Automata.* University of Illinois Press, Urbana, IL, 1966. A. W. Burks, editor.

[29] M. Mitchell Waldrop. *Complexity: the Emerging Science at the Edge of Order and Chaos.* Simon & Schuster, New York, 1992.

[30] J. Weimar. Simulations with Cellular Automata. http://www.tu-bs.de/institute/WiR/weimar/ZAscriptnew/intro.html, 1998.

[31] Stephen Wolfram. Cellular Automaton Fluids 1: Basic Theory. *J. of Stat. Phys.*, 45(3/4):471–525, 1986.

[32] Stephen Wolfram. *A New Kind of Science.* Wolfram media, 2002.

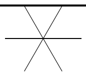
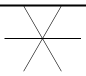
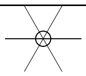
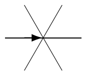
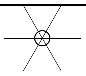
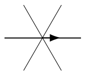
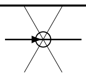
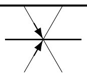
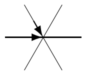
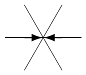
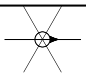
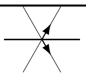
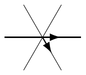
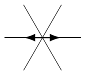
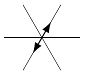
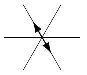
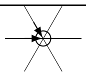
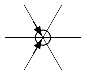
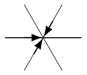
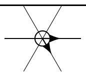
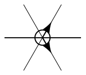
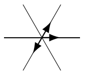
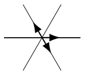
[33] Konrad Zuse. *Rechnender Raum.* Vieweg & Sohn, Braunschweig, 1969. Translated as Calculating Space, AZT-70-164-GEMIT MIT Project MAC (1971).

Appendix A

Collision Rules

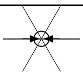
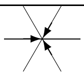
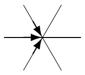
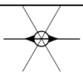
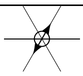
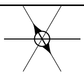
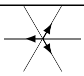
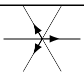
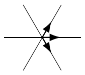
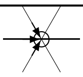
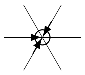
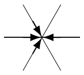
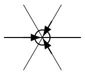
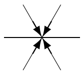
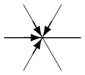
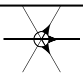
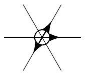
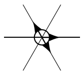
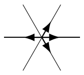
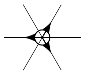
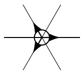
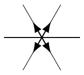
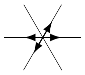
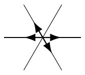
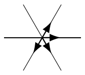
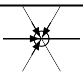
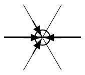
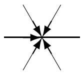
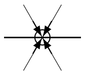
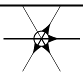
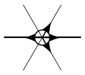
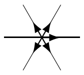
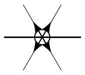
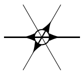
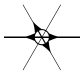
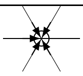
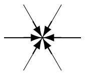
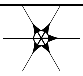
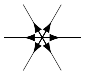
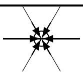
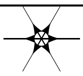
In Table A.1 the collision rules for the fluid 'particles' in Chapter 3.2 are given.

Table A.1: *Collision rules for fluid 'particles' in a hexagonal lattice (N is the number of particles).*

N	Before collision	After collision
0		
1	 	 
2	 ,   	 or    or  or 
3	  , 	  or  or 

continued on next page

Table A.1: *continued*

N	Before collision	After collision
	 ,  	 or  or  or  or  
4	  ,   ,  	  or  or   or  or  or  or  
5	  ,  	  or   or  or 
6	 	 
7		

Appendix B

Code for Hydrodynamics

```
//************************************
//Header file Hydro.h for class Hydro.
//************************************
#ifndef HYDRO_H
#define HYDRO_H
//Class for simulating a cellular automaton fluid

#include <string>
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

#include "Vector.h"

class Hydro
{
   public:
      typedef unsigned char neighbours; //type for fluid particle information in the nodes
      enum { MAX_X_SIZE = 2500 };       //the maximum number of nodes in the x-direction
      enum { MAX_Y_SIZE = 1000 };       //the maximum number of nodes in the y-direction
      enum { NEIGHBOURS = 7 };          //the maximum number of particles at each node
      enum { TWO_POW_NEIGHBOURS = 128 };//2**NEIGHBOURS

      //Description: Constructor
      //Pre: 0 < a <= MAX_X_SIZE, 0 < b <= MAX_Y_SIZE, 0 <= c <= NEIGHBOURS, d >= 0,
      //     0 < e <= a, e <= f <= a, 0 < g <= f - e + 1,
//     0 < h <= b, h <= i <= b, 0 < j <= i - h + 1, 0 <= k <= 100
//Post: The cellular automation is initialised
Hydro(int a, int b, int c, int d, int e, int f, int g, int h, int i, int j, int k,
      bool, bool, bool);

//Description: Destructor
//Pre: true
//Post: memory is released
~Hydro();

//Description: sets array numInnerColl
//Pre: true
//Post: numInnerColl is initialised
void setNumInnerColl();

//Description: sets array innerColl
//Pre: numInnerColl is initialised
//Post: innerColl is initialised
void setInnerColl();

//Description: sets the array inner
//Pre: true
//Post: inner is initialised (true for inner nodes and false for border nodes)
void setInner();

//Description: initialises the array lattice with partPerNode particles/node
//Pre: inner is initialised
//Post: lattice is initialised with partPerNode particles/node
void setLattice();

//Description: calculates the next configuration of the cellular automaton
//Pre: lattice is initialised
//Post: the next configuration of the cellular automaton has been calculated
void oneTimeStep();

//Description: generates a LaTeX-file with a picture of average velocities,
//             file name has to be provided and the boolean value indicates
//             whether or not the hexagonal lattice should be printed
//Pre: lattice is initialised
//Post: result = "" (a LaTeX-file is generated), otherwise an error message
std::string print(char *, bool);
```

```
private:
   int sx;                          //the number of nodes in the x-direction
   int sy;                          //the number of nodes in the y-direction
   int partPerNode;                 //the number of particles per node initially
   int p;                           //the pressure
   int startx;                      //starting node for averaging in x-direction
   int stopx;                       //final node for averaging in x-direction
   int averx;                       //number of nodes in x-direction used in averaging
   int starty;                      //starting node for averaging in y-direction
   int stopy;                       //final node for averaging in y-direction
   int avery;                       //number of nodes in y-direction used in averaging
   int size;                        //size used in the created LATEX-output
   bool rect;                       //a rectangular cylinder (yes or no)
   bool circ;                       //a circular cylinder (yes or no)
   bool newref;                     //new reference frame for velocities
   int numInnerColl[TWO_POW_NEIGHBOURS]; //number of possible outcomes of a given collision
   neighbours * innerColl[TWO_POW_NEIGHBOURS]; //states resulting from collisions
   neighbours lattice[MAX_X_SIZE][MAX_Y_SIZE]; //the states of the cells of the lattice
   neighbours lattCol[MAX_X_SIZE][MAX_Y_SIZE]; //the states after collision
   bool inner[MAX_X_SIZE][MAX_Y_SIZE];         //is a node a inner node or a border node?

   //Description: initialises innerColl
   void initInnerColl();

   //Description: initialises a node (starting configuration)
   //Pre: 0 <= x < sx, 0 <= y < sy
   //Post: the node is set to a random state with partPerNode fluid particles
   void initNode(int x, int y);

   //Description: inserts particles to the left
   void pressureIn();

   //Description: removes particles to the right
   void pressureOut();

   //Description: the particles at a given node are colliding
   void collide(int, int);

   //Description: the particles at a given node are propagating after collision
   void propagate(int, int);
```

```
      //Description: functions for calculating mass end velocity of particles
      int mass(int, int);
      int mass(int, int, int, int);
      Vector centerOfMass(int, int, int, int);
      Vector cartCoord(int, int);
      Vector velocity(int, int);
      Vector velocity(int, int, int, int);

      //Description: functions for creating a LATEX file
      void printFrame(FILE *);
      void printMesh(FILE *);
      void printObstacle(FILE *);
      void printAverage(FILE *);
      void printAll(FILE *);
      double closeAngle(Vector, int[]);

      //Description: returns the minimum of two integers
      //Pre: true
      //Post: result = minimum of a and b
      int min(int a, int b);

      //Description: generates a random number between 1 and m.
      //Pre: 1 <= m
      //Post: result = number between 1 and m
      int random(int m);

      //Description: swaps to elements in an array
      void swap(int [], int, int);

      //Description: puts the numbers 0, ..., integer - 1 in random order in the array.
      //Pre: integer > 0, size of array >= integer
      //Post: the array contains the numbers 0,..., integer - 1 in random order in its
      //      first positions
      void randomVector(int [], int);
};

#endif

//**********************************************
//Implementation file Hydro.cpp for class Hydro.
//**********************************************
```

```
#include "Hydro.h"

Hydro::Hydro(int sx, int sy, int partPerNode, int p, int startx, int stopx, int averx,
             int starty, int stopy, int avery, int size, bool rect, bool circ, bool newref)
{
   this->sx = sx;
   this->sy = sy;
   this->partPerNode = partPerNode;
   this->p = p;
   this->startx = startx;
   this->stopx = stopx;
   this->averx = averx;
   this->starty = starty;
   this->stopy = stopy;
   this->avery = avery;
   this->size = size;
   this->rect = rect;
   this->circ = circ;
   this->newref = newref;
   setNumInnerColl();
   initInnerColl();
   setInnerColl();
   setInner();
   setLattice();
}

Hydro::~Hydro()
{
   for (int i = 0; i < TWO_POW_NEIGHBOURS; i++)
      delete [] innerColl[i];
}

void Hydro::setNumInnerColl()
{
   int a0, a1, a2, a3, a4, a5, a6;
   int asum, ax, ay;
   int count;
   int b0, b1, b2, b3, b4, b5, b6;
   int bsum, bx, by;
   for (int a = 0; a < TWO_POW_NEIGHBOURS; a++) {
      a0 = (a & 0100) >> 6;
      a1 = (a & 0040) >> 5;
      a2 = (a & 0020) >> 4;
      a3 = (a & 0010) >> 3;
      a4 = (a & 0004) >> 2;
      a5 = (a & 0002) >> 1;
      a6 = a & 0001;
      asum = a0 + a1 + a2 + a3 + a4 + a5 + a6;
      ax = - a1 + a2 + 2 * a3 + a4 - a5 - 2 * a6;
      ay = - a1 - a2 + a4 + a5;
      count = 0;
      for (int b = 0; b < TWO_POW_NEIGHBOURS; b++) {
         b0 = (b & 0100) >> 6;
         b1 = (b & 0040) >> 5;
         b2 = (b & 0020) >> 4;
         b3 = (b & 0010) >> 3;
         b4 = (b & 0004) >> 2;
         b5 = (b & 0002) >> 1;
         b6 = b & 0001;
         bsum = b0 + b1 + b2 + b3 + b4 + b5 + b6;
         bx = b1 - b2 - 2 * b3 - b4 + b5 + 2 * b6;
         by = b1 + b2 - b4 - b5;
         if (bsum == asum && bx == ax && by == ay)
            count++;
      }
      numInnerColl[a] = count;
   }
}

void Hydro::setInnerColl()
{
   int a0, a1, a2, a3, a4, a5, a6;
   int asum, ax, ay;
   int count;
   int b0, b1, b2, b3, b4, b5, b6;
   int bsum, bx, by;

   for (int a = 0; a < TWO_POW_NEIGHBOURS; a++) {
      a0 = (a & 0100) >> 6;
      a1 = (a & 0040) >> 5;
      a2 = (a & 0020) >> 4;
      a3 = (a & 0010) >> 3;
      a4 = (a & 0004) >> 2;
      a5 = (a & 0002) >> 1;
      a6 = a & 0001;
      asum = a0 + a1 + a2 + a3 + a4 + a5 + a6;
      ax = - a1 + a2 + 2 * a3 + a4 - a5 - 2 * a6;
      ay = - a1 - a2 + a4 + a5;
      count = 0;
      for (int b = 0; b < TWO_POW_NEIGHBOURS; b++) {
         b0 = (b & 0100) >> 6;
         b1 = (b & 0040) >> 5;
         b2 = (b & 0020) >> 4;
         b3 = (b & 0010) >> 3;
         b4 = (b & 0004) >> 2;
         b5 = (b & 0002) >> 1;
         b6 = b & 0001;
         bsum = b0 + b1 + b2 + b3 + b4 + b5 + b6;
         bx = b1 - b2 - 2 * b3 - b4 + b5 + 2 * b6;
         by = b1 + b2 - b4 - b5;
         if (bsum == asum && bx == ax && by == ay)
            innerColl[a][count++] = b;
      }
   }
}

void Hydro::setInner()
{
   // Inner nodes.
   for (int x = 1; x < sx - 1; x++)
      for (int y = 1; y < sy - 1; y++)
         inner[x][y] = true;

   // Border nodes.
   for (int x = 0; x < sx; x++) {
      inner[x][0] = false;
      inner[x][sy - 1] = false;
   }
   for (int y = 0; y < sy; y++) {
      inner[0][y] = false;
      inner[sx - 1][y] = false;
   }

   // Obstacle.
   if (rect) {
      int xc = sx/5;
      int yc = sy/2;
      //int xL = sx/8;
      int xL = sx/16;
      //int yL = sy/16;
      int yL = sy/16;
      for (int x = xc - xL; x <= xc + xL; x++)
         for (int y = yc - yL; y <= yc + yL; y++)
            inner[x][y] = false;
   }
   if (circ) {
      int xc = sx/4;
      int yc = sy/2;
      int r = sy/16;
      for (int x = xc - r; x <= xc + r; x++) {
         for (int y = yc - r; y <= yc + r; y++) {
            Vector v = cartCoord(x - xc, y - yc);
            if (v.length() < (double)r * sqrt(3.0) / 2.0)
               inner[x][y] = false;
         }
      }
   }
}

void Hydro::setLattice()
{
   for (int x = 0; x < sx; x++)
      for (int y = 0; y < sy; y++)
         initNode(x, y);
}

void Hydro::oneTimeStep()
{
   pressureIn();
   pressureOut();

   for (int x = 0; x < sx; x++)
```

```
      for (int y = 0; y < sy; y++)
         collide(x, y);

   for (int x = 0; x < sx; x++)
      for (int y = 0; y < sy; y++)
         propagate(x, y);
}

std::string Hydro::print(char * fn, bool mesh)
{
   FILE * fd = fopen(fn, "w+");
   std::string s = "";

   if (fd == NULL)
      s = "error in opening file\n";
   else {
      Vector v = cartCoord(sx, sy - 1);
      fprintf(fd, "\\setlength{\\unitlength}{%i.%imm}\n", size / 10, size % 10);
      fprintf(fd, "\\begin{picture}(%f,%f)(0,0)\n", v.getX(), v.getY());

      // print frame
      printFrame(fd);

      // print mesh
      if (mesh) printMesh(fd);

      // print obstacle
      if (rect || circ) printObstacle(fd);

      // print result
      printAverage(fd);
      //printAll(fd);

      fprintf(fd, "\\end{picture}\n");
   }

   fclose(fd);

   return s;
}
```

```
void Hydro::initInnerColl()
{
   for (int i = 0; i < TWO_POW_NEIGHBOURS; i++)
      innerColl[i] = new neighbours[numInnerColl[i]];
}

void Hydro::initNode(int x, int y)
{
   lattice[x][y] = 0;

   if (inner[x][y]) {
      int vec[NEIGHBOURS];
      randomVector(vec, NEIGHBOURS);

      for (int i = 0; i < partPerNode; i++)
         lattice[x][y] += (1 << (NEIGHBOURS - 1 - vec[i]));
   }
}

void Hydro::pressureIn()
{
   int vec[MAX_Y_SIZE];
   int rvec[NEIGHBOURS];
   int count = 0;
   int x = 0;
   int y;

   while (count < p) {
      x++;
      randomVector(vec, sy - 2);

      for (int j = 0; count < p && j < sy - 2; j++) {
         y = vec[j] + 1;
         if (inner[x][y]) {
            bool notfound = true;
            randomVector(rvec, NEIGHBOURS);
            for (int i = 0; notfound && i < NEIGHBOURS; i++) {
               if (((lattice[x][y] >> (NEIGHBOURS - 1 - rvec[i])) & 0001) == 0) {
                  lattice[x][y] += (0001 << (NEIGHBOURS - 1 - rvec[i]));
                  count++;
                  notfound = false;
               }
            }
         }
      }
   }
}

void Hydro::pressureOut()
{
   int vec[MAX_Y_SIZE];
   int rvec[NEIGHBOURS];
   int count = 0;
   int x = sx - 1;
   int y;

   while (count < p) {
      x--;
      randomVector(vec, sy - 2);

      for (int j = 0; count < p && j < sy - 2; j++) {
         y = vec[j] + 1;
         bool notfound = true;
         randomVector(rvec, NEIGHBOURS);
         for (int i = 0; notfound && i < NEIGHBOURS; i++) {
            if (((lattice[x][y] >> (NEIGHBOURS - 1 - rvec[i])) & 0001) == 1) {
               lattice[x][y] -= (0001 << (NEIGHBOURS - 1 - rvec[i]));
               count++;
               notfound = false;
            }
         }
      }
   }
}

void Hydro::collide(int x, int y)
{
   if (inner[x][y]) {
      int ind = lattice[x][y];
      int num = numInnerColl[ind];
      int r = random(num);
      lattCol[x][y] = innerColl[ind][r - 1];
   }
   else
      lattCol[x][y] = lattice[x][y];
}

void Hydro::propagate(int x, int y)
{
   if (inner[x][y]) {
      if (y % 2 == 0)
         lattice[x][y] = (lattCol[x][y] & 0100) +
                         ((lattCol[x][y + 1] & 0004) << 3) +
                         ((lattCol[x - 1][y + 1] & 0002) << 3) +
                         ((lattCol[x - 1][y] & 0001) << 3) +
                         ((lattCol[x - 1][y - 1] & 0040) >> 3) +
                         ((lattCol[x][y - 1] & 0020) >> 3) +
                         ((lattCol[x + 1][y] & 0010) >> 3);
      else
         lattice[x][y] = (lattCol[x][y] & 0100) +
                         ((lattCol[x + 1][y + 1] & 0004) << 3) +
                         ((lattCol[x][y + 1] & 0002) << 3) +
                         ((lattCol[x - 1][y] & 0001) << 3) +
                         ((lattCol[x][y - 1] & 0040) >> 3) +
                         ((lattCol[x + 1][y - 1] & 0020) >> 3) +
                         ((lattCol[x + 1][y] & 0010) >> 3);

   }
   else {
      lattice[x][y] = 0;
      if (y % 2 == 0) {
         if (y + 1 < sy && inner[x][y + 1])
            lattice[x][y] += ((lattCol[x][y + 1] & 0004) << 3);
         if (x - 1 >= 0 && y + 1 < sy && inner[x - 1][y + 1])
            lattice[x][y] += ((lattCol[x - 1][y + 1] & 0002) << 3);
         if (x - 1 >= 0 && inner[x - 1][y])
            lattice[x][y] += ((lattCol[x - 1][y] & 0001) << 3);
         if (x - 1 >= 0 && y - 1 >= 0 && inner[x - 1][y - 1])
            lattice[x][y] += ((lattCol[x - 1][y - 1] & 0040) >> 3);
         if (y - 1 >= 0 && inner[x][y - 1])
            lattice[x][y] += ((lattCol[x][y - 1] & 0020) >> 3);
         if (x + 1 < sx && inner[x + 1][y])
            lattice[x][y] += ((lattCol[x + 1][y] & 0010) >> 3);
      }
      else {
         if (x + 1 < sx && y + 1 < sy && inner[x + 1][y + 1])
            lattice[x][y] += ((lattCol[x + 1][y + 1] & 0004) << 3);
         if (y + 1 < sy && inner[x][y + 1])
            lattice[x][y] += ((lattCol[x][y + 1] & 0002) << 3);
         if (x - 1 >= 0 && inner[x - 1][y])
            lattice[x][y] += ((lattCol[x - 1][y] & 0001) << 3);
         if (y - 1 >= 0 && inner[x][y - 1])
            lattice[x][y] += ((lattCol[x][y - 1] & 0040) >> 3);
         if (x + 1 < sx && y - 1 >= 0 && inner[x + 1][y - 1])
            lattice[x][y] += ((lattCol[x + 1][y - 1] & 0020) >> 3);
         if (x + 1 < sx && inner[x + 1][y])
            lattice[x][y] += ((lattCol[x + 1][y] & 0010) >> 3);
      }
   }
}

int Hydro::mass(int x, int y)
{
   int count = 0;

   if (((lattice[x][y] & 0100) >> 6) == 1) count++;
   if (((lattice[x][y] & 0040) >> 5) == 1) count++;
   if (((lattice[x][y] & 0020) >> 4) == 1) count++;
   if (((lattice[x][y] & 0010) >> 3) == 1) count++;
   if (((lattice[x][y] & 0004) >> 2) == 1) count++;
   if (((lattice[x][y] & 0002) >> 1) == 1) count++;
   if ((lattice[x][y] & 0001) == 1) count++;

   return count;
}

int Hydro::mass(int xmin, int xmax, int ymin, int ymax)
{
   int totmass = 0;

   for (int x = xmin; x <= xmax; x++)
      for (int y = ymin; y <= ymax; y++)
         totmass += mass(x, y);
   return totmass;
}

Vector Hydro::centerOfMass(int xmin, int xmax, int ymin, int ymax)
{
   double mx = 0.0;
   double my = 0.0;

   for (int x = xmin; x <= xmax; x++) {
      for (int y = ymin; y <= ymax; y++) {
         int m = mass(x, y);
         Vector v = cartCoord(x, y);
         mx += m * v.getX();
         my += m * v.getY();
      }
   }

   int totmass = mass(xmin, xmax, ymin, ymax);

   return Vector(mx / (double)totmass, my / (double)totmass);
}

Vector Hydro::cartCoord(int x, int y)
{
   double cartX = x;
   double cartY = y * sqrt(3.0) * 0.5;

   if (y % 2 == 1)
      cartX += 0.5;

   return Vector(cartX, cartY);
}

Vector Hydro::velocity(int x, int y)
{
   double vx, vy;

   vx = 0.0;
   vy = 0.0;

   if (((lattice[x][y] & 0040) >> 5) == 1) {
      vx -= 0.5;
      vy -= 0.5;
   }
   if (((lattice[x][y] & 0020) >> 4) == 1) {
      vx += 0.5;
      vy -= 0.5;
   }
   if (((lattice[x][y] & 0010) >> 3) == 1) vx += 1.0;
   if (((lattice[x][y] & 0004) >> 2) == 1) {
      vx += 0.5;
      vy += 0.5;
   }
   if (((lattice[x][y] & 0002) >> 1) == 1) {
      vx -= 0.5;
      vy += 0.5;
   }
   if ((lattice[x][y] & 0001) == 1) vx -= 1.0;

   vy *= sqrt(3.0);

   return Vector(vx / (double)NEIGHBOURS, vy / (double)NEIGHBOURS);
}

Vector Hydro::velocity(int xmin, int xmax, int ymin, int ymax)
{
   double vx = 0.0;
   double vy = 0.0;

   for (int x = xmin; x <= xmax; x++) {
      for (int y = ymin; y <= ymax; y++) {
         Vector v = velocity(x, y);
         vx += v.getX();
         vy += v.getY();
      }
   }

   int number = (xmax - xmin + 1) * (ymax - ymin + 1);

   return Vector(vx / (double)number, vy / (double)number);
}
```

```
void Hydro::printFrame(FILE * fd)
{
    Vector v = cartCoord(0, sy - 1);

    fprintf(fd, "\\put(0,0){\\line(1,0){%i}}\n", sx - 1);
    fprintf(fd, "\\put(%f,%f){\\line(1,0){%i}}\n", v.getX(), v.getY(), sx - 1);
}

void Hydro::printMesh(FILE * fd)
{
   //Horizontal lines
   for (int y = 1; y < sy - 1; y++) {
      Vector v = cartCoord(0, y);
      fprintf(fd, "\\put(%f,%f){\\line(1,0){%i}}\n", v.getX(), v.getY(), sx - 1);
   }

   //Negative slope lines
   Vector v = Vector(-(sy - 1) * 0.5, (sy - 1) * sqrt(3.0) * 0.5);
   int cl[2];
   double length = closeAngle(v, cl);
   for (int x = sy/2; x < sx; x++)
      fprintf(fd, "\\put(%i,0){\\line(%i,%i){%f}}\n", x, cl[0], cl[1], length);

   //Positive slope lines
   v = Vector((sy - 1) * 0.5, (sy - 1) * sqrt(3.0) * 0.5);
   length = closeAngle(v, cl);
   for (int x = 0; x <= sx - (sy + 1)/2; x++)
      fprintf(fd, "\\put(%i,0){\\line(%i,%i){%f}}\n", x, cl[0], cl[1], length);

   //The rest
   for (int y = 2; y < sy; y += 2) {
      Vector w = cartCoord(0, y);
      v = Vector(y * 0.5, - y * sqrt(3.0) * 0.5);
      length = closeAngle(v, cl);
      fprintf(fd, "\\put(0,%f){\\line(%i,%i){%f}}\n", w.getY(), cl[0], cl[1], length);
      v = Vector((sy - 1 - y) * 0.5, (sy - 1 - y) * sqrt(3.0) * 0.5);
      length = closeAngle(v, cl);
      fprintf(fd, "\\put(0,%f){\\line(%i,%i){%f}}\n", w.getY(), cl[0], cl[1], length);
   }
   for (int y = 1; y < sy - 1; y += 2) {
      Vector w = cartCoord(sx - 1, y);
      v = Vector(- y * 0.5, - y * sqrt(3.0) * 0.5);
      length = closeAngle(v, cl);
      fprintf(fd, "\\put(%f,%f){\\line(%i,%i){%f}}\n", w.getX(), w.getY(), cl[0], cl[1], length);
      v = Vector(- (sy - 1 - y) * 0.5, (sy - 1 - y) * sqrt(3.0) * 0.5);
      length = closeAngle(v, cl);
      fprintf(fd, "\\put(%f,%f){\\line(%i,%i){%f}}\n", w.getX(), w.getY(), cl[0], cl[1], length);
   }
}

void Hydro::printObstacle(FILE * fd)
{
   if (rect) {
      int xc = sx/5;
      int yc = sy/2;
      //int xL = sx/8;
      int xL = sx/16;
      //int yL = sy/16;
      int yL = sy/16;
      Vector v = cartCoord(xc - xL, yc - yL);
      fprintf(fd, "\\put(%f,%f){\\line(1,0){%i}}\n", v.getX(), v.getY(), 2 * xL);
      fprintf(fd, "\\put(%f,%f){\\line(0,1){%f}}\n", v.getX(), v.getY(), sqrt(3.0) * yL);
      v = cartCoord(xc + xL, yc + yL);
      fprintf(fd, "\\put(%f,%f){\\line(-1,0){%i}}\n", v.getX(), v.getY(), 2 * xL);
      fprintf(fd, "\\put(%f,%f){\\line(0,-1){%f}}\n", v.getX(), v.getY(), sqrt(3.0) * yL);
   }
   if (circ) {
      int xc = sx/4;
      int yc = sy/2;
      int r = sy/16;
      //int r = sy/32;
      Vector v = cartCoord(xc, yc);
      fprintf(fd, "\\put(%f,%f){\\circle{%f}}\n", v.getX(), v.getY(), sqrt(3.0) * r);
   }
}

void Hydro::printAverage(FILE * fd)
{
   Vector av = velocity(0, sx - 1, 0, sy - 1);

   double factor = 30.0 * min(averx, avery);
   double x, y, length;
   int cl[2];

   for (int xmin = startx - 1; xmin < stopx; xmin += averx) {
      for (int ymin = starty - 1; ymin < stopy; ymin += avery) {
         int xmax = min(xmin + averx, stopx) - 1;
         int ymax = min(ymin + avery, stopy) - 1;
         int totmass = mass(xmin, xmax, ymin, ymax);
         if (totmass > 0) {
            Vector cm = centerOfMass(xmin, xmax, ymin, ymax);
            Vector v  = velocity(xmin, xmax, ymin, ymax);
            if (newref) {
               v.setX(v.getX() - av.getX());
               v.setY(v.getY() - av.getY());
            }
            length = closeAngle(v, cl);
            x = cm.getX() - 0.5 * factor * v.getX();
            y = cm.getY() - 0.5 * factor * v.getY();
            fprintf(fd, "\\put(%f,%f){\\line(%i,%i){%f}}\n", x, y, cl[0], cl[1], factor * length);
         }
      }
   }
}

void Hydro::printAll(FILE * fd)
{
   int cl[2];
   double length;
   double factor = 0.5;
   double xc, yc;

   for (int x = 0; x < sx; x++) {
      for (int y = 0; y < sy; y++) {
         Vector r = cartCoord(x, y);
         if (((lattice[x][y] & 0100) >> 6) == 1) {
            fprintf(fd, "\\put(%f,%f){\\circle{%f}}\n", r.getX(), r.getY(), 0.4);
         }
         if (((lattice[x][y] & 0040) >> 5) == 1) {
            Vector v(-0.5, -0.5 * sqrt(3.0));
            length = closeAngle(v, cl);
            xc = r.getX() - factor * v.getX();
      yc = r.getY() - factor * v.getY();
      fprintf(fd,"{\\linethickness{0.15mm}");
      fprintf(fd, "\\put(%f,%f){\\vector(%i,%i){%f}}}\n", xc, yc, cl[0], cl[1], factor * length);
   }
   if (((lattice[x][y] & 0020) >> 4) == 1) {
      Vector v(0.5, -0.5 * sqrt(3.0));
      length = closeAngle(v, cl);
      xc = r.getX() - factor * v.getX();
      yc = r.getY() - factor * v.getY();
      fprintf(fd,"{\\linethickness{0.15mm}");
      fprintf(fd, "\\put(%f,%f){\\vector(%i,%i){%f}}}\n", xc, yc, cl[0], cl[1], factor * length);
   }
   if (((lattice[x][y] & 0010) >> 3) == 1) {
      Vector v(1.0, 0.0);
      length = closeAngle(v, cl);
      xc = r.getX() - factor * v.getX();
      yc = r.getY() - factor * v.getY();
      fprintf(fd,"{\\linethickness{0.15mm}");
      fprintf(fd, "\\put(%f,%f){\\vector(%i,%i){%f}}}\n", xc, yc, cl[0], cl[1], factor * length);
   }
   if (((lattice[x][y] & 0004) >> 2) == 1) {
      Vector v(0.5, 0.5 * sqrt(3.0));
      length = closeAngle(v, cl);
      xc = r.getX() - factor * v.getX();
      yc = r.getY() - factor * v.getY();
      fprintf(fd,"{\\linethickness{0.15mm}");
      fprintf(fd, "\\put(%f,%f){\\vector(%i,%i){%f}}}\n", xc, yc, cl[0], cl[1], factor * length);
   }
   if (((lattice[x][y] & 0002) >> 1) == 1) {
      Vector v(-0.5, 0.5 * sqrt(3.0));
      length = closeAngle(v, cl);
      xc = r.getX() - factor * v.getX();
      yc = r.getY() - factor * v.getY();
      fprintf(fd,"{\\linethickness{0.15mm}");
      fprintf(fd, "\\put(%f,%f){\\vector(%i,%i){%f}}}\n", xc, yc, cl[0], cl[1], factor * length);
   }
   if ((lattice[x][y] & 0001) == 1) {
      Vector v(-1.0, 0.0);
      length = closeAngle(v, cl);
      xc = r.getX() - factor * v.getX();
      yc = r.getY() - factor * v.getY();
```

```
            fprintf(fd,"{\\linethickness{0.15mm}");
            fprintf(fd, "\\put(%f,%f){\\vector(%i,%i){%f}}}\n", xc, yc, cl[0], cl[1], factor * length);
         }
      }
   }
}

double Hydro::closeAngle(Vector v, int cl[2])
{
   double angle = v.angle();
   double speed = v.length();

   cl[0] = (int)(1000.0 * cos(angle));
   cl[1] = (int)(1000.0 * sin(angle));

   return speed * fabsf(cos(angle));
}

int Hydro::min(int a, int b)
{
   if (a < b)
      return a;
   else
      return b;
}

int Hydro::random(int m)
{
//   srand(time(0));
   return 1 + (int)((double)m * rand() / (RAND_MAX + 1.0));
}

void Hydro::swap(int vec[], int i, int j)
{
   int temp = vec[i];
   vec[i] = vec[j];
   vec[j] = temp;
}

void Hydro::randomVector(int vec[], int size)
{
   for (int i = 0; i < size; i++)
      vec[i] = i;

   for (int i = 0; i < size; i++) {
      int r = random(size - i);
      swap(vec, r - 1, size - i - 1);
   }
}

// **************************************
// Header file Vector.h for class Vector.
// **************************************
#ifndef VECTOR_H
#define VECTOR_H
// Class for describing position vectors in two dimensions.

#include <math.h>

class Vector
{
   public:
      //Creates a position vector with given coordinates.
      Vector(double, double);

      //Destroys position vector.
      ~Vector();

      //Returns x-coordinate of position vector.
      double getX();

      //Returns y-coordinate of position vector.
      double getY();

      //Sets x-coordinate of position vector.
      void setX(double);

      //Sets y-coordinate of position vector.
      void setY(double);

      //Returns length of position vector.
      double length();
      //Returns angle of position vector.
      double angle();
   private:
      double x; //x-coordinate
      double y; //y-coordinate
};

#endif

// ************************************************
// Implementation file Vector.cpp for class Vector.
// ************************************************

#include "Vector.h"

Vector::Vector(double x, double y)
{
   this->x = x;
   this->y = y;
}

Vector::~Vector()
{
}

double Vector::getX()
{
   return x;
}

double Vector::getY()
{
   return y;
}

void Vector::setX(double x)
{
   this->x = x;
}

void Vector::setY(double y)
{
   this->y = y;
}

double Vector::length()
{
   return sqrt(x * x + y * y);
}

double Vector::angle()
{
   return atan2(y, x);
}

//*********************************************************************
//   The main program
//*********************************************************************
#include "Hydro.h"

#define MAX_FILE_NAME    20

int main()
{
   char fn[MAX_FILE_NAME];
   int sx, sy, partPerNode, p;
   int startx, stopx, averx;
   int starty, stopy, avery;
   int  size, step;
   char crect, ccirc;
   bool rect = false;
   bool circ = false;
   char cmesh;
   bool mesh = false;
   bool newref = false;
   char cnewref;

   printf("write a file name: ");
   scanf("%s", fn);
   printf("write the number of nodes (x-direction) (<= %d, > 0): ", Hydro::MAX_X_SIZE);
   scanf("%d", &sx);
   printf("write the number of nodes (y-direction) (<= %d, > 0): ", Hydro::MAX_Y_SIZE);
   scanf("%d", &sy);
   printf("write the number of particles per node (on average) (0, ..., %d): ", Hydro::NEIGHBOURS);
```

```
scanf("%d", &partPerNode);
printf("write the pressure (0, ..., ~%d): ", sy);
scanf("%d", &p);
printf("write a step number (>= 0): ");
scanf("%d", &step);
printf("write the start node for averaging in the x-direction (1, ..., %d): ", sx);
scanf("%d", &startx);
printf("write the stop node for averaging in the x-direction (%d, ..., %d): ", startx, sx);
scanf("%d", &stopx);
printf("write the number of nodes for averaging (x-direction) (1, ..., %d): ", stopx - startx + 1);
scanf("%d", &averx);
printf("write the start node for averaging in the y-direction (1, ..., %d): ", sy);
scanf("%d", &starty);
printf("write the stop node for averaging in the y-direction (%d, ..., %d): ", starty, sy);
scanf("%d", &stopy);
printf("write the number of nodes for averaging (y-direction) (1, ..., %d): ", stopy - starty + 1);
scanf("%d", &avery);
printf("write a size number (1, 2, ..., 100): ");
scanf("%d", &size);

printf("rectangular cylinder? (y/n): ");
scanf("\n%c", &crect);
if (crect == 'y')
    rect = true;
else {
    printf("circular cylinder? (y/n): ");
    scanf("\n%c", &ccirc);
    if (ccirc == 'y') circ = true;
}

printf("mesh? (y/n): ");
scanf("\n%c", &cmesh);
if (cmesh == 'y') mesh = true;
printf("new reference frame? (y/n): ");
scanf("\n%c", &cnewref);
if (cnewref == 'y') newref = true;

if (sx > 0 && sx <= Hydro::MAX_X_SIZE && sy > 0 && sy <= Hydro::MAX_Y_SIZE &&
    partPerNode >= 0 && partPerNode <= Hydro::NEIGHBOURS && p >=0 && step >= 0 &&
    startx > 0 && startx <= sx && stopx >= startx && stopx <= sx && averx > 0 &&
    averx <= stopx - startx + 1 &&
     starty > 0 && starty <= sy && stopy >= starty && stopy <= sy && avery > 0 &&
     avery <= stopy - starty + 1 &&
     size >= 1 && size <= 100)
  {
    Hydro test = Hydro(sx, sy, partPerNode, p, startx, stopx, averx,
                       starty, stopy, avery, size, rect, circ, newref);

    for (int i = 0; i < step; i++)
      test.oneTimeStep();

    test.print(fn, mesh);
  }
  else
    printf("something wrong with input data\n");

  return 0;
}
```

Appendix C

Code for Genetic Algorithms

```
//*****************************************************************
//* Genetic Algorithm for a 2D cellular automaton with a von      *
//* Neumann neighbourhood                                         *
//*****************************************************************
#ifndef GA_H
#define GA_H

#define MAX_ARRAY       300
#define SIZE             32  //2^5
#define MUTATIONS         3  //MUTATIONS << SIZE

typedef bool rule[SIZE];

//Generates a random number between 1 and m
//Pre: m > 0
int random(int m);

//Swaps the elements i and j in array F
//Pre: 0 <= i < size of F, 0 <= j < size of F
void swap(int F[], int i, int j);

//Puts the numbers 0, ..., s - 1 in random order in the array vec.
//Pre: s > 0
//Post: vec contains the numbers 0,..., s - 1 in random order in its
//      first positions
void randomVector(int vec[], int s);
```

```
//Initializes a vector of rules with random values
void init_rules_rand(rule R[], int popsize);

//Initializes a vector of rules with random values evenly distributed
void init_rules_even(rule R[], int popsize);

//Initializes a matrix A of size s*s with random values
//Pre: 0 < s <= MAX_ARRAY
void init_rand(bool A[][MAX_ARRAY], int s);

//Initializes a matrix A of size s*s with random values (fracw % is white)
//Pre: 0 < s <= MAX_ARRAY, 0 <= fracw <= 100
void init_rand(bool A[][MAX_ARRAY], int s, int fracw);

//Initializes a matrix A of size s*s with only white
//Pre: 0 < s <= MAX_ARRAY
void init_white(bool A[][MAX_ARRAY], int s);

//Initializes a matrix A of size s*s with only black
//Pre: 0 < s <= MAX_ARRAY
void init_black(bool A[][MAX_ARRAY], int s);

//Initializes a matrix A of size s*s with a black cell in the center
//Pre: 0 < s <= MAX_ARRAY
void init_seed(bool A[][MAX_ARRAY], int s);

//Initializes a matrix A of size s*s with a checkerboard pattern
//Pre: 0 < s <= MAX_ARRAY
void init_check(bool A[][MAX_ARRAY], int s);

//Initializes a matrix A of size s*s with stripes
//Pre: 0 < s <= MAX_ARRAY
void init_stripe(bool A[][MAX_ARRAY], int s);

//Returns update of matrix element at position (i,j) of matrix A of size s*s
//Pre: 0 <= i < s, 0 <= j < s, 0 < s <= MAX_ARRAY
bool update_el(rule ru, int i, int j, bool A[][MAX_ARRAY], int s);

//Updates a matrix A of size s*s
//Pre: 0 < s <= MAX_ARRAY
```

```
void update_mat(rule r, bool A[][MAX_ARRAY], int s);

//Evolves the CA in time time-steps
//Pre: 0 < s <= MAX_ARRAY, time > 0
void evolve(rule r, bool A[][MAX_ARRAY], int s, int time);

//Calculates the fitness of A to T.
//Pre: 0 < s <= MAX_ARRAY
//Post: result = value between 0 and s*s
int fitness(bool A[][MAX_ARRAY], bool T[][MAX_ARRAY], int s);

//Calculates the fitness of a rule
//Pre: 0 < s <= MAX_ARRAY, time > 0, t > 0, sc = {seed, rand}, 0 <= w <= 100
int fitness(rule r, bool T[][MAX_ARRAY], int s, int time, int t, char * sc, int w);

//Calculates the fitness of several rules and stores in an array
//Pre: 0 < s <= MAX_ARRAY, time > 0, t > 0, 0 <= start < popsize, sc = {seed, rand},
//     0 <= w <= 100
void fitness(int f[], rule r[], bool T[][MAX_ARRAY], int s, int time, int t, int start,
             int popsize, char * sc, int w);

//Swaps the elements i in rules r1 and r2
//Pre: 0 <= i < SIZE
void swap(rule r1, rule r2, int i);

//Swaps the rules i and j in R
//Pre: 0 <= i < size of R, 0 <= j < size of R
void swap(rule R[], int i, int j);

//Sorts F and R in descending order in F
void sort(int F[], rule R[], int popsize);

//Two offsprings are generated from two parent rules
void crossover(rule r1, rule r2);

//A rule is mutated
void mutation(rule r);

//Copies rule r2 to rule r1
void copy(rule r1, rule r2);
```

```
//First half of the rules generate offsprings which will be the second half
//Pre: popsize > 0, popsize = size of R
void offsprings(rule R[], int popsize);

//F and R are sorted and the new rules are generated
//Pre: 0 < popsize, popsize = size of F and R
void new_rules(int F[], rule R[], int popsize);

//Calculates the number of 1's in rules R and prints them
//Pre: popsize = size of R
void popDistr(rule R[], int popsize, int D[]);

//Genetic algoritm for calculating fittest rule
//Pre: 4 <= popsize, popsize%4 = 0, 0 < s <= MAX_ARRAY, time > 0, t > 0,
//     sc = {seed, rand}, fc = {black, check, stripe}, 0 <= w <= 100
//Post: result = the number of iterations
int genAlg(rule r, int popsize, int s, int time, int t, char * sc, int w, char * fc,
           int DI[], int DF[]);

#endif

//*****************************************************************
//* Genetic Algorithm for a 2D cellular automaton with a von      *
//* Neumann neighbourhood                                         *
//*****************************************************************
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "ga.h"

int random(int m)
{
   return (1 + (int)((double)m * rand() / (RAND_MAX + 1.0)));
}

void swap(int F[], int i, int j)
{
    int temp = F[i];
    F[i] = F[j];
    F[j] = temp;
}
```

```
void randomVector(int vec[], int s)
{
   for (int i = 0; i < s; i++)
      vec[i] = i;

   for (int i = 0; i < s; i++) {
      int r = random(s - i);
      swap(vec, r - 1, s - i - 1);
   }
}

void init_rules_rand(rule R[], int popsize)
{
    for (int i = 0; i < popsize; i++) {
        for (int j = 0; j < SIZE; j++) {
            int r = (int)((2.0 * rand()) / (RAND_MAX + 1.0));
            R[i][j] = (bool)(r);
        }
    }
}

void init_rules_even(rule R[], int popsize)
{
    int n = popsize/(SIZE + 1);
    int m = popsize%(SIZE + 1);
    int Lamb[popsize + SIZE + 1 - m];
    for (int i = 0; i <= n; i++)
        randomVector(Lamb + i * (SIZE + 1), SIZE + 1);

    int Rule[SIZE];
    for (int i = 0; i < popsize; i++) {
        int black = Lamb[i];
        randomVector(Rule, SIZE);
        for (int j = 0; j < SIZE; j++)
            R[i][j] = false;
        for (int k = 0; k < black; k++)
            R[i][Rule[k]] = true;
    }
}

void init_rand(bool A[][MAX_ARRAY], int s)
```

```
{
    for (int i = 0; i < s; i++) {
        for (int j = 0; j < s; j++) {
            int r = (int)((2.0 * rand()) / (RAND_MAX + 1.0));
            A[i][j] = (bool)(r);
        }
    }
}

void init_rand(bool A[][MAX_ARRAY], int s, int fracw)
{
    int nw = (fracw * s * s) / 100;
    init_black(A, s);
    int vec[s*s];
    randomVector(vec, s*s);
    for (int i = 0; i < nw; i++)
        A[vec[i]/s][vec[i]%s] = false;
}

void init_white(bool A[][MAX_ARRAY], int s)
{
    for (int i = 0; i < s; i++)
        for (int j = 0; j < s; j++)
            A[i][j] = false;
}

void init_black(bool A[][MAX_ARRAY], int s)
{
    for (int i = 0; i < s; i++)
        for (int j = 0; j < s; j++)
            A[i][j] = true;
}

void init_seed(bool A[][MAX_ARRAY], int s)
{
    init_white(A, s);
    A[s/2][s/2] = true;
}

void init_check(bool A[][MAX_ARRAY], int s)
{
    init_black(A, s);
    for (int i = 0; i < s; i++) {
        if (i%2 == 0) {
            for (int j = 0; j < s; j += 2)
                A[i][j] = false;
        }
        else {
            for (int j = 1; j < s; j += 2)
                A[i][j] = false;
        }
    }
}

void init_stripe(bool A[][MAX_ARRAY], int s)
{
    init_black(A, s);
    for (int i = 0; i < s; i++)
        for (int j = 0; j < s; j += 2)
            A[i][j] = false;
}

bool update_el(rule ru, int i, int j, bool A[][MAX_ARRAY], int s)
{
    int l = j - 1;
    int r = j + 1;

    if (l < 0) l = s - 1;
    if (r > s - 1) r = 0;

    int u = i - 1;
    int d = i + 1;

    if (u < 0) u = s - 1;
    if (d > s - 1) d = 0;

    int index = 0;
    if (A[i][j]) index += 1;
    if (A[i][r]) index += 2;
    if (A[u][j]) index += 4;
    if (A[i][l]) index += 8;
    if (A[d][j]) index += 16;
    return ru[index];
}

void update_mat(rule r, bool A[][MAX_ARRAY], int s)
{
    bool Ac[s][MAX_ARRAY];

    for (int i = 0; i < s; i++)
        for (int j = 0; j < s; j++)
            Ac[i][j] = A[i][j];

    for (int i = 0; i < s; i++)
        for (int j = 0; j < s; j++)
            A[i][j] = update_el(r, i, j, Ac, s);
}

void evolve(rule r, bool A[][MAX_ARRAY], int s, int time)
{
    for (int t = 1; t <= time; t++)
        update_mat(r, A, s);
}

int fitness(bool A[][MAX_ARRAY], bool T[][MAX_ARRAY], int s)
{
    int f = 0;

    for (int i = 0; i < s; i++)
        for (int j = 0; j < s; j++)
            if (A[i][j] == T[i][j]) f++;

    return f;
}

int fitness(rule r, bool T[][MAX_ARRAY], int s, int time, int t, char * sc, int w)
{
    int f;
    bool A[s][MAX_ARRAY];

    //initialize the CA
    if (strcmp(sc, "seed") == 0) init_seed(A, s);
    //else if (strcmp(sc, "rand") == 0) init_rand(A, s);
    else if (strcmp(sc, "rand") == 0) init_rand(A, s, w);

    //evolve the CA
    evolve(r, A, s, time);

    //calculate fitness
    f = fitness(A, T, s);
    for (int i = 1; i < t; i++) {
        //one more time-step
        evolve(r, A, s, 1);
        //calculate fitness
        f += fitness(A, T, s);
    }

    return (f/t);
}

void fitness(int f[], rule r[], bool T[][MAX_ARRAY], int s, int time, int t, int start,
             int popsize, char * sc, int w)
{
    for (int i = start; i < popsize; i++)
        f[i] = fitness(r[i], T, s, time, t, sc, w);
}

void swap(rule r1, rule r2, int i)
{
    bool temp = r1[i];
    r1[i] = r2[i];
    r2[i] = temp;
}

void swap(rule R[], int i, int j)
{
    for (int k = 0; k < SIZE; k++)
        swap(R[i], R[j], k);
}

void sort(int F[], rule R[], int popsize)
{
    for (int i = 0; i < popsize; i++) {
        for (int j = i + 1; j < popsize; j++) {
            if (F[i] < F[j]) {
                swap(F, i, j);
                swap(R, i, j);
            }
        }
    }
}

void crossover(rule r1, rule r2)
{
    int p = random(SIZE);

    for (int i = 0; i < p; i++)
        swap(r1, r2, i);
}

void mutation(rule r)
{
    int p;

    for (int i = 0; i < MUTATIONS; i++) {
        p = random(SIZE) - 1;
        if (r[p])
            r[p] = false;
        else
            r[p] = true;
    }
}

void copy(rule r1, rule r2)
{
    for (int i = 0; i < SIZE; i++)
        r1[i] = r2[i];
}



void offsprings(rule R[], int popsize)
{
    int vec[popsize/2];
    randomVector(vec, popsize/2);
    for (int i = 0; i < popsize/4; i++) {
        copy(R[popsize/2 + 2*i], R[vec[2*i]]);
        copy(R[popsize/2 + 2*i + 1], R[vec[2*i + 1]]);
        crossover(R[popsize/2 + 2*i], R[popsize/2 + 2*i + 1]);
        mutation(R[popsize/2 + 2*i]);
        mutation(R[popsize/2 + 2*i + 1]);
    }
}

void new_rules(int F[], rule R[], int popsize)
{
    //sort F and R in descending order in F
    sort(F, R, popsize);
    //create offsprings
    offsprings(R, popsize);
}

void popDistr(rule R[], int popsize, int D[])
{
    int C[SIZE + 1];
    for (int n = 0; n < SIZE + 1; n++)
        C[n] = 0;

    int num;
    for (int j = 0; j < popsize; j++) {
        num = 0;
        for (int i = 0; i < SIZE; i++) {
            if (R[j][i]) num++;
        }
        C[num]++;
    }

    for (int n = 0; n < SIZE + 1; n++)
        D[n] += C[n];
}

int genAlg(rule r, int popsize, int s, int time, int t, char * sc, int w, char * fc,
           int DI[], int DF[])
{
    rule R[popsize];  //rules
    int F[popsize];   //fitness of rules
    bool T[s][MAX_ARRAY];
    int start = 0;
    int count = 0;

    //randomly generate an initial population of rules
    init_rules_rand(R, popsize);
    popDistr(R, popsize, DI);

    //initialize the template (that the CA should evolve to)
    if (strcmp(fc, "black") == 0) init_black(T, s);
    else if (strcmp(fc, "check") == 0) init_check(T, s);
    else if (strcmp(fc, "stripe") == 0) init_stripe(T, s);

    do {
        //calculate the fitness for the rules
        fitness(F, R, T, s, time, t, start, popsize, sc, w);
        start = popsize/2;
        //produce a new set of rules
        new_rules(F, R, popsize);
        count++;
    } while(!(F[0] == s*s || count > 400));

    popDistr(R, popsize/2, DF);

    for (int i = 0; i < SIZE; i++)
        r[i] = R[0][i];

    return count;
}

//******************************************************************
//* LATEX files from evolution of 2D cellular automata with a von  *
//* Neumann neighbourhood                                          *
//******************************************************************
#include <stdio.h>
#include "ga.h"

#ifndef PRINT_H
#define PRINT_H

//Prints instructions in LATEX for a frame of height h and width w to a file
```

```
//Pre: h > 0, w > 0
void print_frame(FILE * fd, int h, int w);

//Prints instructions in LATEX for a mesh to a file
//The mesh should fit in a frame with height h and width w and the
//size of a square is 1 x 1
//Pre: h > 0, w > 0
void print_mesh(FILE * fd, int h, int w);

//Prints instructions in LATEX for filling a row of l squares starting with
//square with coordinates x and y of the lower left corner to a file
//Pre: l > 0, 0 < size < (about) 100
void fill_squares(FILE * fd, int x, int y, int l, int size);

//Prints instructions in LATEX for a 2D cellular automaton to a file
//The step number is given by step,
//the result of a step is stored in matrix A with size s*s
//Pre: Matrix A initialized, s > 0, step >= 0, 0 < size < (about) 100
void print_cell_aut(FILE * fd, rule r, bool A[][MAX_ARRAY], int s, int step, int size);

//Prints instructions in LATEX for a 2D cellular automaton to a file.
//The name of the file is given by fn, the step number is given by step,
//the number of cells is given by width,
//the size of a square is given by size (in tenths of mm), mesh indicates
//whether a mesh should be printed or not.
//Pre: step >= 0, 0 < width <= MAX_ARRAY, 0 < size < (about) 100,
//     sc = {seed, rand}, 0 <= w <= 100
void cell_aut(char * fn, rule r, int step, int width, int size, bool mesh, char * sc, int w);

#endif

//*****************************************************************
//* LATEX files from evolution of 2D cellular automata with a von  *
//* Neumann neighbourhood                                          *
//*****************************************************************
#include <string.h>
#include "print.h"

void print_frame(FILE * fd, int h, int w)
{
    fprintf(fd, "%s%i%s%i%s\n", "\\put(0,", 0, "){\\line(1,0){", w, "}}");
    fprintf(fd, "%s%i%s%i%s\n", "\\put(0,", h, "){\\line(1,0){", w, "}}");
    fprintf(fd, "%s%i%s%i%s\n", "\\put(", 0, ",0){\\line(0,1){", h, "}}");
    fprintf(fd, "%s%i%s%i%s\n", "\\put(", w, ",0){\\line(0,1){", h, "}}");
}

void print_mesh(FILE * fd, int h, int w)
{
    for (int i = 1; i < h; i++)
        fprintf(fd, "%s%i%s%i%s\n", "\\put(0,", i, "){\\line(1,0){", w, "}}");
    for (int i = 1; i < w; i++)
        fprintf(fd, "%s%i%s%i%s\n", "\\put(", i, ",0){\\line(0,1){", h, "}}");
}

void fill_squares(FILE * fd, int x, int y, int l, int size)
{
    int size_i = size / 10;
    int size_d = size % 10;

    fprintf(fd, "%s%i.%i%s\n", "{\\linethickness{", size_i, size_d, "mm}");
    fprintf(fd, "%s%i,%i%s%i%s\n", "\\put(", x, y, ".5){\\line(1,0){",l,"}}}");
}

void print_cell_aut(FILE * fd, rule r, bool A[][MAX_ARRAY], int s, int step, int size)
{
    int l = 0;
    int j0;
    bool pblack = false;

    evolve(r, A, s, step);

    for (int i = 0; i < s; i++) {
        for (int j = 0; j < s; j++) {
            if (A[i][j]) {
                if (l == 0) {
                    j0 = j;
                    pblack = true;
                }
                l++;
            }
            else {
                if (pblack) {
                    fill_squares(fd, j0, s - i - 1, l, size);
                    l = 0;
                    pblack = false;
                }
            }
        }
        if (pblack) {
            fill_squares(fd, j0, s - i - 1, l, size);
            l = 0;
            pblack = false;
        }
    }
}

void cell_aut(char * fn, rule r, int step, int width, int size, bool mesh, char * sc, int w)
{
    FILE * fd = fopen(fn, "w+");
    bool A[width][MAX_ARRAY];

    int size_i = size / 10;
    int size_d = size % 10;

    // initialize the cellular automaton
    if (strcmp(sc, "seed") == 0) init_seed(A, width);
    else if (strcmp(sc, "rand") == 0) init_rand(A, width, w);

    if (fd == NULL)
        printf("error in opening file %s\n", fn);
    else
    {
        fprintf(fd, "%s%i.%i%s\n", "\\setlength{\\unitlength}{", size_i, size_d, "mm}");
        fprintf(fd, "%s%i,%i%s\n", "\\begin{picture}(", width, width, ")(0,0)");
        // print frame
        print_frame(fd, width, width);
        // print mesh
        if (mesh) print_mesh(fd, width, width);
        // print cellular automaton
        print_cell_aut(fd, r, A, width, step, size);
        fprintf(fd, "%s\n", "\\end{picture}");
    }

    fclose(fd);
```

```
}

//***********************************************************************
//*    The rho_c = 1/2 classification task                               *
//***********************************************************************
#ifndef CLASSTASK_H
#define CLASSTASK_H
#include "ga.h"

//Calculates the fitness of several rules and stores in an array
//Pre: 0 < s <= MAX_ARRAY, time > 0, t > 0, 0 <= start < popsize
void fitnessCT(int f[], rule r[], bool T[][MAX_ARRAY], int s, int time, int t, int start, int popsize);

//Genetic algoritm for calculating fittest rule for the rho_c = 1/2 classification task
//Pre: 4 <= popsize, popsize%4 = 0, 0 < s <= MAX_ARRAY, time > 0, t > 0
//Post: result = the number of iterations
int classTask(rule r, int popsize, int s, int time, int t, int DI[], int DF[]);

#endif

//***********************************************************************
//*    The rho_c = 1/2 classification task                               *
//***********************************************************************
#include "ga.h"

void fitnessCT(int f[], rule r[], bool T[][MAX_ARRAY], int s, int time, int t, int start, int popsize)
{
    init_white(T, s);
    for (int i = start; i < popsize; i++) {
        f[i] = 0;
        for (int w = 55; w < 100; w += 10) {
            f[i] += fitness(r[i], T, s, time, t, "rand", w);
        }
    }

    init_black(T, s);
    for (int i = start; i < popsize; i++)
        for (int w = 5; w < 50; w += 10)
            f[i] += fitness(r[i], T, s, time, t, "rand", w);

    for (int i = start; i < popsize; i++)
        f[i] /= 10;
```

```
}

int classTask(rule r, int popsize, int s, int time, int t, int DI[], int DF[])
{
    rule R[popsize];  //rules
    int F[popsize];   //fitness of rules
    bool T[s][MAX_ARRAY];
    int start = 0;
    int count = 0;

    //randomly generate an initial population of rules
    init_rules_rand(R, popsize);
    popDistr(R, popsize, DI);

    do {
        //calculate the fitness for the rules
        fitnessCT(F, R, T, s, time, t, start, popsize);
        start = popsize/2;
        //produce a new set of rules
        new_rules(F, R, popsize);
        count++;
    } while(!(F[0] == s*s || count > 400));

    popDistr(R, popsize/2, DF);

    for (int i = 0; i < SIZE; i++)
        r[i] = R[0][i];

    return count;
}
```