Comenius University
Faculty of mathematics, physics and informatics

Evolution of growing and irregular cellular automata

Master thesis

2014

Miroslav Beka, Bc.

Comenius University
Faculty of mathematics, physics and informatics

Evolution of growing and irregular cellular automata

Master thesis

Study program:	applied informatics
Study field:	2511, applied informatics (artificial intelligence)
Department:	department of applied informatics
Supervisor:	Mgr. Pavel Petrovič, PhD.

Bratislava 2014

Miroslav Beka, Bc.

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Bc. Miroslav Beka
Študijný program: aplikovaná informatika (Jednoodborové štúdium, magisterský II. st., denná forma)
Študijný odbor: 9.2.9. aplikovaná informatika
Typ záverečnej práce: diplomová
Jazyk záverečnej práce: anglický
Sekundárny jazyk: slovenský

Názov: Evolution of Growing and Irregular Cellular Automata

Cieľ: The thesis will contain an overview of studies on irregular cellular automata (CA). It will further focus on using CA as a representation of shapes for evolutionary design and to generate CA using Evolutionary Algorithms (EA). Student will design original methods (or modify existing ones) for generating CA using EA with changing structure, CA that can develop from en embryo to a final phenotype. The goal is to explore such methods that use CA with irregular grid. The proposed/analysed methods will be experimentally tested and the experiment will be evaluated.

Literatúra: Beatens J.M., De Baets, B.: Phenomenological study of irregular cellular automata based on Lyapunov exponents and Jacobians, Chaos, 20, 2010.
Mitchell M.: Computation in Cellular Automata: A Selected Review. In T. Gramss, S. Bornholdt, M. Gross, M. Mitchell, and T. Pellizzari, Nonstandard Computation, pp. 95–140. Weinheim: VCH Verlagsgesellschaft, 1998.
Mitchell M., Crutchfield J.P., Rajarshi D.: Evolving Cellular Automata with Genetic Algorithms: A Review of Recent Work, EuCA'96.

Kľúčové slová: cellular automata, evolutionary design, development

Vedúci: Mgr. Pavel Petrovič, PhD.
Katedra: FMFI.KAI - Katedra aplikovanej informatiky
Vedúci katedry: doc. PhDr. Ján Rybár, PhD.

Dátum zadania: 10.10.2012

Dátum schválenia: 23.10.2012

doc. RNDr. Roman Ďurikovič, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

Acknowledgements

I would like to thank to my supervisor Pavel Petrovič for valuable advices and directions. Also I thank to my friends and family for their support during developing and writing this thesis, especially Lenka Nemečková for her help with translation.

Abstract

Evolutionary design (ED) is still young promising field already with multiple achievements. This work contains overview of this evolutionary strategy, focused mainly on ED with cellular representations. Framework for experimenting with cellular representations was implemented and successfully used to support several hypotheses about evolvability of different configurations.

Keywords: cellular automata, evolutionary design, ontogeny

Abstrakt

Evolučný dizajn (ED) je stále mladý nádejný odbor, s mnohými úspechmi v riešení problémov. Táto práca obsahuje prehľad tejto evolučnej stratégie, zameranej hlavne na ED s celulárnou reprezentáciou. Bol implementovaný framework na experimentovanie s celulárnymi reprezentáciami, ktorý bol úspešne použitý na analýzu niekoľkých hypotéz ohľadom CA reprezentácií

Kľúčové slová: cellularne automaty, evolučný dizajn, ontogenéza

Foreword

In human effort to look for a new and better ways to solve problems, nature has been always source of inspiration. Many useful algorithms, structures and processes were inspired by nature. By taking the idea, abstracting it and using in the right way, we were able to solve many problems and move on to more difficult problems that lie ahead of us. Nevertheless, complexity that nature is operating at is still beyond what we can achieve right now. Our ultimate goal is to create system at biological complexity, such as human brain, which consists of hundreds billion of neurons and from 100 to 500 trillion connections between them. But how do we get to this point? How nature managed to create such complexity? The answer is evolution. This blind optimisation through search space took approximately 3.6 billion years to get from simple prokaryotes to human beings. We can consider ourselves as state of the art of evolution. Beings capable of understanding processes that created them and consciously use them for their own benefits.

Contents

1. Introduction

1.1 What is Design?

People usually imagine design as process of creating something beautiful and pleasant for an eye. But we can't limit the notion of design just to an art. In this context we will understand design from wider point of view. We will define it as process of creating solution to given problem. That basically includes everything that was ever made by human beings. Starting with primitive tools used by Neanderthals, through business strategies to astronauts and rocket science. We were able to design all of it with our mind.
In 2009 Paul Ralph and Yair Wand proposed definition of a design. [1]

"Design is a specification of an object, manifested by an agent, intended to accomplish goals, in a particular environment, using a set of primitive components, satisfying a set of requirements, subject to constraints"

The design process consists usually from setting or defining the problem that we want to solve. Then we produce ideas and solutions that are suitable for given problem. We also use knowledge base about the environment. We take different ideas, slightly modify them or combine them into solution that serves our purpose.
For example we can take a look at bridges constructions. There are several basic types that are regularly used [2]. One of the most basic bridges is a Beam bridge. This bridge is just strong beam put over river or other obstacle to connect small distance. Truss bridge is combination of beam and truss construction which is more stable and can sustain more stress. This improvement also opens up possibility of connecting greater lengths. Later, more advanced bridges, were

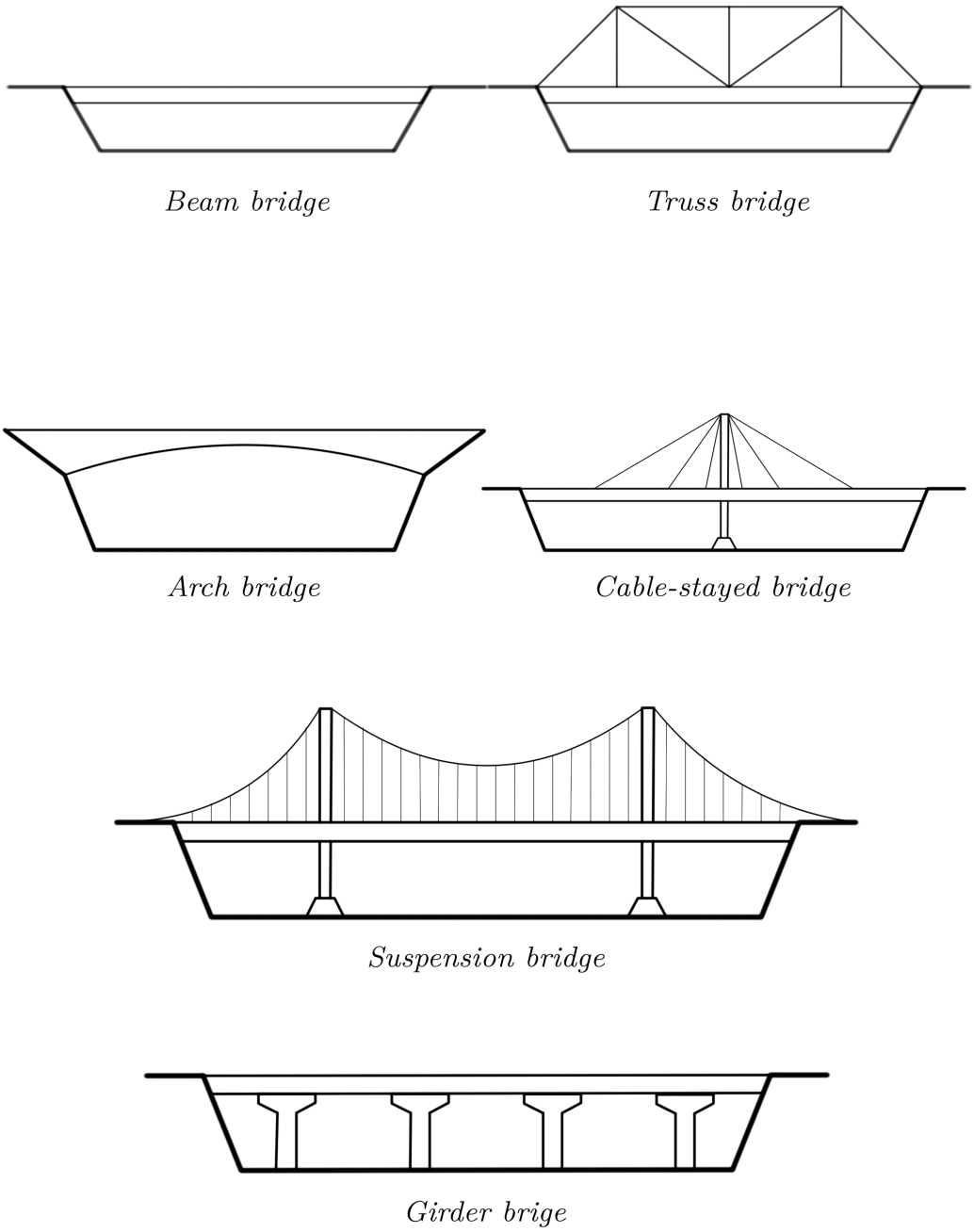
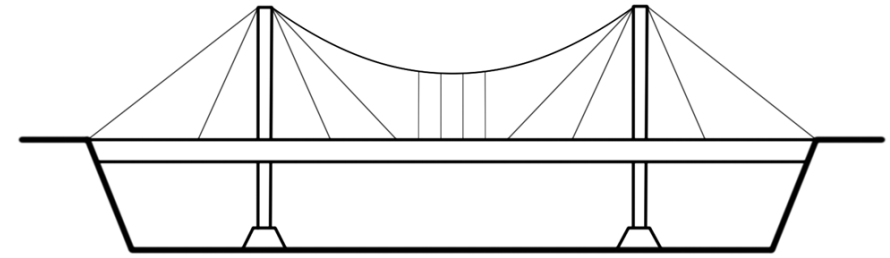
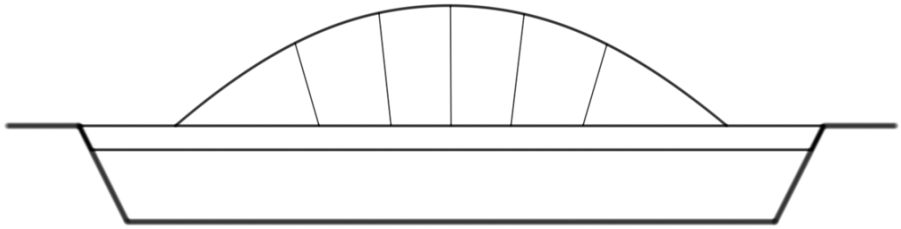


Figure 1.1 basic designs of bridge

created such as Arch bridge, Cable-stayed bridge, Girder bridge, Suspension bridge. Those are common types of designs ([3]) that can be found all around us. In the capital of Slovakia, Bratislava we can identify several designs such as Truss, Arch-suspension, Cable-stayed and Arch. Design of the following bridges is result of selection, recombination and variation of previous ideas.



Cable-stayed + Suspension bridge



Arch + Suspension bridge

Figure 1.2 combined design of bridges

By combining arch and suspension design we get new, slightly different construction that uses both of them. Same applies for cable-stayed and suspension construction.

Designs were combined together as solution to a particular problem, where properties of both designs were used to overcome other difficulties where previous design failed. For example to connect greater distances or support higher load. On figure 1.2 are examples of such bridges.

One might already notice similarity between design process and evolution. Translating definition of design proposed by Paul Ralph and Yair Wand, into terms of evolution is not that difficult. "*Design is a specification of an object, manifested by an agent*". Specification of an object is something that describes how to construct such an object. In case of evolution, we have DNA as an specification.

Although, in this case, it is not manifested by any agent. But we might say, that evolution is the agent *"intended to accomplish goals, in particular environment."* Translation of this sentence into terms of nature could be for example this: "Every creature is in some particular environment where it tries to survive and reproduce it's genes using a set of primitive components, satisfying a set of requirements, subject to constraints." Here we can define primitive set of components on a several levels. Depending of what nature tries to solve, components could be on cellular level, on basic body structures such as bones, specialised tissue and organs or even on behavioural level such as techniques how to survive. It is required that these creatures have met particular requirements in order to function correctly and stay under constraints of our physical world. Indeed, Goldberg actually attempts to formally define human design in terms of evolution by the genetic algorithm (Goldberg, 1991). He compares the recombination of genetic material from parents when forming a new child, with a designer combining ideas from two different solutions to form a new one. Evolution is the change in the inherited characteristics of biological populations over successive generations. Evolutionary processes give rise to diversity at every level of biological organisation, including species, individual organisms and molecules such as DNA and proteins. [4]

1.2 Evolution, mother of all designers

Since Evolution created truly remarkable solutions, why not to inspire ourselves? Abstracting principles of the Evolution in terms of computer science, we are getting to evolutionary computation. This particular field is a collection of optimisation algorithms that strongly uses meta-heuristics based on evolution. There are many variations of evolutionary algorithms that differ from one another by search space, evaluation methods, way how solutions are combined, etc. This evolutionary algorithms are optimisation, adaptation and search strategy in any

type of space. The last property is one of the most important. For algorithm to be able to search in **any type of search space**.

The whole group of Evolutionary Algorithms is categorised by those properties. There are 4 main categories among EA, which are divided by particular type of problem. Basic mechanisms are implemented in each one of this methods. Such as reproduction, mutation, recombination, natural selection and survival of the fittest. The main categories are:

- Genetic algorithm
- Evolutionary strategies
- Genetic programming
- Evolutionary programming

As the basic example of evolutionary algorithm, we could take Genetic Algorithm (GA). GA presents all the necessary mechanisms in straightforward way. This algorithm works in iterative process. It generates population of solutions that is adapted and improved in time to find the best solution for given problem. Like in nature, each of this individuals/solutions are somehow represented. We will review options of representation later in this chapter.

How does evolution work? Evolution consist of producing initial population of solutions to given problem, selection operator that selects the best solutions in actual generation. Crossover operator is applied after selection and the best solutions are combined together to produce better solutions. Candidate solutions are slightly mutated in direction towards better solution.

The next question is interesting to think about. Does all designs made by human, existed before? And all the designs we have made is just copying, combining and modifying of things that we have already saw, heard, touched, tasted or smelled?

Well this is good question, but impossible to answer. Instead of speculating, let me propose small experiment. Try to think of new animal that never existed before and it really scares you. Give yourself few minutes and then compare it to other animals. Does it resemble any existing animals? Or combination? I'm sure that animal that you thought of, never existed before, but do you consider it original? You can see that process of designing has some limitations. How can one imagine something that he has never seen before? We can create solutions or in other words "designs" in some specific way and at some point we can get into the dead end. Then we have to start again and take different decisions to create different and better design. We also use previous knowledge and incorporate it into our designs. This is called knowledge base and is applied in form of constraints and requirements.

It's not really hard to combine evolution with design after all. It's almost exactly the same. Both have similarities and differences.

Evolution	**Design**
completely automated	supervised
highly parallel	sequential
slow	much faster
better search space coverage	search space limitations (i.e. animal experiment)
breading only within species	can combine several contexts

Interesting fact about design is that we can combine very different ideas from very different contexts and even multiple ideas. Evolution on the other hand, can combine individuals from same context. In reproduction step are involved two parents that produce at least one offspring.

In order to design solution, we need a designers that carries out the process.

Evolution in this case has advantage, because it is happening naturally and all the time. So it is completely automated and it requires no conscious attention of participants.

On the other hand, Evolution unlike the designing, requires much more time. Creating solution by evolution usually takes years and results are in smaller incremental steps. But even if evolution takes years, we can witness some remarkable designs by natural evolution.

1.3 Evolutionary design

The fundamental difference against other strategies is that representation is rendered and simulated in Computer Aided Design (CAD).The problem is still optimisation task like in other methods, but in ED the evaluation of the design lies in physical simulation.

Physical simulation is system of objects, that are free to move (usually in 3 dimensions) according Newton's laws of dynamics. Creative innovation is the main property of evolutionary design that gives it advantage over other methods. ED proved that it's capable of creatively solve problems, too hard for us to solve. Some of them are mentioned in section 1.5

Here is one example of evolutionary design. Let's imagine we have to evolve design of a bridge. We set our goal to make a bridge for heavy loads. That means our objective function would be measuring, how much weight is needed for bridge to collapse. Then we set up our representation of genotype. It could be either L-system or graph representation. In simulated environment with physics, we would evaluate every design and see its performance. Based on score, designs are selected, combined and varied into next generation. After sufficient number of generations,

we just review best solutions and then we can construct better, more stable, durable and innovative bridges.

1.3.1 Representation

At first we need to define some expressions we will use. In EA we talk about genotype as a representations of given solution. **Genotype** can consists from vector of binary values, real vector, tree or graph structures. Could be any kind of structured information. **Phenotype** is resulting representation that is evaluated by fitness function in environment. In some cases, phenotype is same as genotype which is called direct representation. Genotype is directly evaluated by fitness function. Indirect representation adds process of transformation from genotype to phenotype.

Genotype is compact representations, or instructions (in case of indirect representation) and is better as a search space. Phenotype is solution space. There is a mapping function transforming genotype into phenotype.

Evolutionary algorithms has been successfully used for search in structured space many times. But the true power of EA is in its ability to optimise solutions in unstructured search space. Since we are searching in complex search space, solutions itself are more complex which brings us to question of feasibility. Can we use today's computers without the computation taking years?

One of the solutions how to lower complexity is to use sparse variable-length genotype. This type of representations requires operators that can operate on such structures. Mutation operator can make small change in genotype, but resulting phenotype could be radically changed. Also crossover operator or initialisation procedure needs to meet some constraints to produce meaningful feasible solutions.

With the right representation one could get better scalability. For complex representations is this property most of the time problematic, to achieve because

complexity grows with the size of the problem. For more scalable solutions, one could use list of elementary items and put them into groups. Using for example Genetic Programming is also an alternative representation for compact genotype and with well-formed variation operators, this is an another step towards useful features such as modularity, recursion and ultimately to the more complex solutions. As an additional value to these features is less resource wasting. With more symmetry in a solution, the evolution does not have to "design" every instance of this symmetry. Anyway, more recent research direction is heading toward so called morphogenetic approach. To evolve more complex solutions, instead evolving parts of solution, one evolves programs that yields resulting phenotype as we run them. One of the early research in cellular representation was done by F. Gruau, where topology of neural network was evolved. By comparing direct encoding of ANN with fixed topology and cellular representation, Gruau showed significant improvement on pole balancing problem.

Over the period of several years, many successes have been reported using morphogenic approaches. This approaches are appealing to build complex solutions to difficult problems. However increase in scalability, modularity and recursion goes together with a loss in causality.Influence of small changes in the genotype are impossible to track in resulting phenotype.

1.4 Cellular Automata

CAs are dynamical systems in which space and time are discrete. A cellular automaton consists of a regular grid of cells, each of which can be in one of a finite number of k possible states, updated synchronously in discrete time steps according to a local interaction rule. The state of a cell is determined by the previous states of a surrounding neighbourhood of cells.

The infinite or finite cellular array (grid) is n-dimensional, where n = 1, 2, 3 is used in practice. The rule contained in each cell is essentially a finite state machine, usually specified in the form of a rule table (also known as the transition function), with an entry for every possible neighbourhood configuration of states. The neighbourhood of a cell consists of the surrounding cells. The term configuration refers to an assignment of states to cells in the grid. When considering a finite-sized grid, spatially periodic boundary conditions are frequently applied, resulting in a circular grid for the one-dimensional case, and a toroidal one for the two-dimensional case. Combination of properties of CA (neighbourhood, transition rule, state, dimensions of the grid) creates specific versions of CA. One of the most popular is Conway cellular automaton called Game of Life [5]. This particular CA has binary state, each cell has 8 neighbour cells and transition rules according table 1.3

1.4.1 History

Cellular automata have been really extensively researched and studied. Notion of regular grid has been known for long time. It was Ulam ([6]), who tried to model crystal growth on regular grid. Ulam was coworker of von Neumann, who was researching self-replication. But his research required big amount of robotic parts that could be used to build up a robot, so Neumann dropped this approach and started to use Ulam's grid with some states and transition rules and the very first cellular automaton was born. This particular self-replicating automaton was constructed from 200000 cells and had 29 different states.

Von Neumann was so pleased with concept of cellular automaton that he actually did most of the initial research about cellular automata and proposed many theorems and many of them were proved.

Some people were so impressed by modelling capabilities of cellular automata that

# of living neighbours	0	1	2	3	4+
living cell	dies	dies	stays alive	stays alive	dies
dead cell	stays dead	stays dead	stays dead	come to life	stays dead

Figure 1.3 game of life rule table

they took the concept of CA even further. Konrad Zuse in his book Calculating Universe pioneered notion, that whole universe is just computer operating in 3 dimensional space where the smallest particles are basic “cells” flipping through different states according rules .[7]

1.4.2 Cell states

In the most general way, cells can carry any type of information. Usually, cell is in one state of finite set in any point in time. As for the well-known example, Game of Life, state of cell can be either 1 or 0. In this case, we are talking about binary set of states. This type of states are more likely to simulate discrete state space. In case of continuous state space, cell states are represented as real values on some interval. Even if the nature of CA is discrete, we are able to simulate continuous phenomena such as reaction-diffusion system. RD system are using notion of "concentration of chemicals" as cell state, in which case it is vector of real values. Even the definition of state of CA cell dictates use of fixed finite set of, we can carry arbitrary piece of information in cell state.

1.4.3 Geometry

Space in which CA operates can be a d-dimensional (possibly infinite) grid. In case of finite grids, it is needed to specify a boundary conditions. Periodic boundary on a grid, in some dimension, is folded in that dimension.

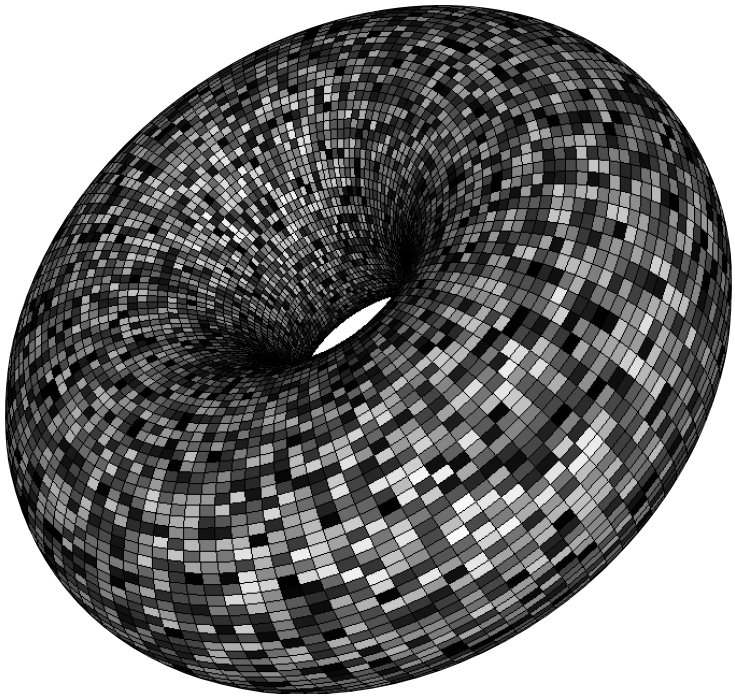
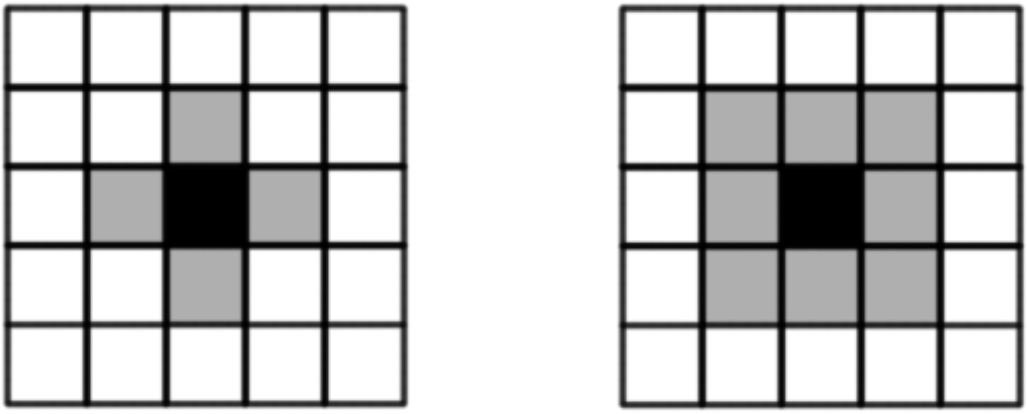


Figure 1.4 wrapped 2D CA into torus

The dimension has a fixed boundary condition if the border cells are adjacent to cells in some specified state whose value does not change over the time. There is also third possibility, CA with no boundary restrictions. This type of geometry is less practical and studied. Even though there are implementations of infinite grid, such as hashlife. The most notable property about cellular automaton is dimension and tessellation of that particular dimensions. Wolfram did most of his research with 1-dimensional CA. Other popular dimension and tessellation is the one in Game of Life. This is two dimensional space divided to regular square grid.

With 2 and higher dimensional space, much more possibilities start to show up. It is because tessellation can have many different forms. We could start with regular tessellation such as square, hexagonal or triangular and many more. Also irregular tessellation of space like voronoi diagram is very interesting and gets a lot of attention in research of CA.



(a) von Neumann neighbourhood

(b) Moore neighbourhood

Figure 1.5 types of neighbourhood in 2D

1.4.4 Neighbourhood

In some cases the cells neighbourhood itself is determined by the geometry of CA. However, let's stay at d-dimensional grid. It is possible to define many different kinds of neighbourhood. More basic definition of neighbourhood is von Neumann or Moore neighbourhood. Von Neumann neighbourhood is orthogonal and cell has to have adjacent edges with neighbouring cells. Cell with Moore neighbour has more neighbours and in this case neighbouring cells has to have at least one common point. We can easily push this definition in higher third dimension.

It is possible to more complex neighbourhood like, input and output neighbourhoods. A cell takes its input from its input neighbourhood and its state is available to the cells of its output neighbourhood. If the sizes of the input and output neighbourhoods are equal, then the CA is balanced.

A variant of CA where the transition rule operates on the sum of the states of the neighbouring cells is called totalistic CA. This type of CA was introduced by Stephen Wolfram in his book New Kind Of Science [5].

1.4.5 Transition rule

The most important part of a CA is the transition function or simply "rule". This is what determines the behaviour most. Even though the rule depends on the lattice geometry, neighbourhood and cell state set, rule significantly determines the evolution over time. In the most cases prediction how CA will look like is not really feasible. The only way is to explicitly simulating it.

Any function, that takes as input states of neighbour cells and return new state for cell can be used. Wolfram did extensive research about "basic" rules for 1-dimensional CA. By size of neighbourhood of cell and number of states, we can calculate total number of rule outcomes. For example 1-dimensional CA with 3 neighbour cells and binary state has total number of rules = 223 which is 256.

Wolfram examined behaviour of each one of this rules and created 4 classes [5].

- **Class 1**: Nearly all initial patterns evolve quickly into a stable, homogeneous state. Any randomness in the initial pattern disappears.
- **Class 2**: Nearly all initial patterns evolve quickly into stable or oscillating structures. Some of the randomness in the initial pattern may filter out, but some remains. Local changes to the initial pattern tend to remain local.
- **Class 3**: Nearly all initial patterns evolve in a pseudo-random or chaotic manner. Any stable structures that appear are quickly destroyed by the surrounding noise. Local changes to the initial pattern tend to spread indefinitely
- **Class 4**: Nearly all initial patterns evolve into structures that interact in complex and interesting ways, with the formation of local structures that are able to survive for long periods of time

With each rule that is used, comes into question reversibility of rule. Reversible rule or reversible CA is such property, that one is able to reverse CA configuration

backwards. One can get previous configuration just based on rule and current configuration. In other words, each configuration has one one unique predecessor. Decidability if rule is reversible or not is algorithmically possible for one dimensional CA in polynomial time. However, in tow dimensional CA is this proved to be an undecidable problem. J. Kari proves in [4, p 379] that this problem is undecidable.

1.5 Recent contributions

In this section we introduce some selection of successful Evolutionary Design research. As we mentioned earlier, the true power of Evolutionary Algorithms is ability to search in any kind of representation space. As such, EA have been used to solve design problems where traditional methods were least helpful.

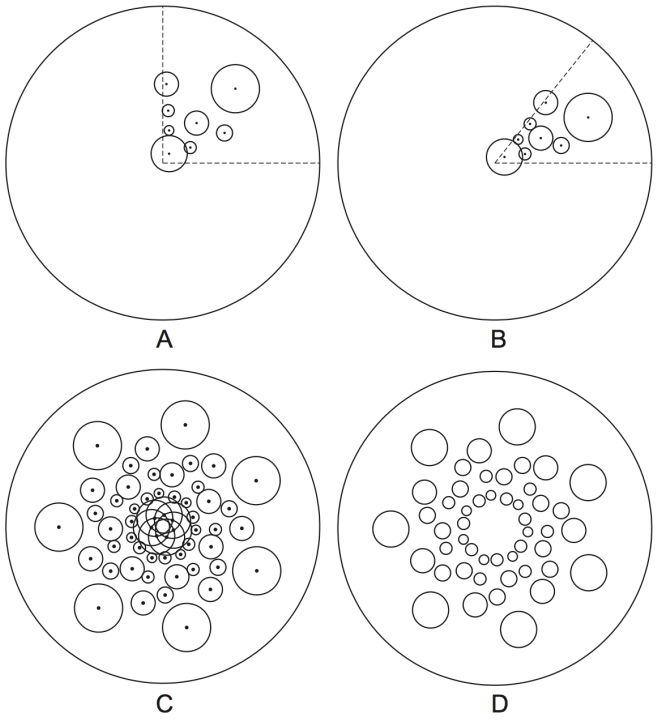
Each of selected design problem was solved by using different representation, to demonstrate available variability using Evolutionary Design. Almost any design problem can be matched with EA flexibility.

1.5.1 Optical fibres design

In this work is used direct representation. The genotype explicitly represents location of air hole which guide light trough fibre and the desired radius of the hole.This holes are positioned on a fibre optics section on the first quadrant of circle. Last property in coded in genotype is symmetry, which means how much is the quadrant compressed and repeated over the hole fibre section.

The process of making final design is depicted on figure 1.6

Even though this is explicit representation, genotype thanks to symmetry stays relatively small to be feasible and EA can explore most of search space. This



A) positions of holes is placed according genotype inside one quadrant of circle

B) circle quadrant is compressed based on amount of symmetry encoded in genotype

C) compressed quadrant is distributed along whole circle

D) shows the result in order to satisfy hole overlapping constraints

Figure 1.6 optical fibre design

representation was used with success also in other engineering problems related to optical fibres.

1.5.2 Karl Sim's

This work is more of a exploratory nature. Rather than address some precise engineering problem, this work tries to validate ideas on the possible ways to design implicit representations.

Karl Sims' is one of the most interesting research when it comes to question of creativity in evolutionary design. The goals in this work was to create various creatures that can move as fast as possible, jump into the air, swim in water environment or competing for a object of interest. All of this goals were satisfied with great success. Evolved behaviour remarkably resembles nature running style, swimming or competition.

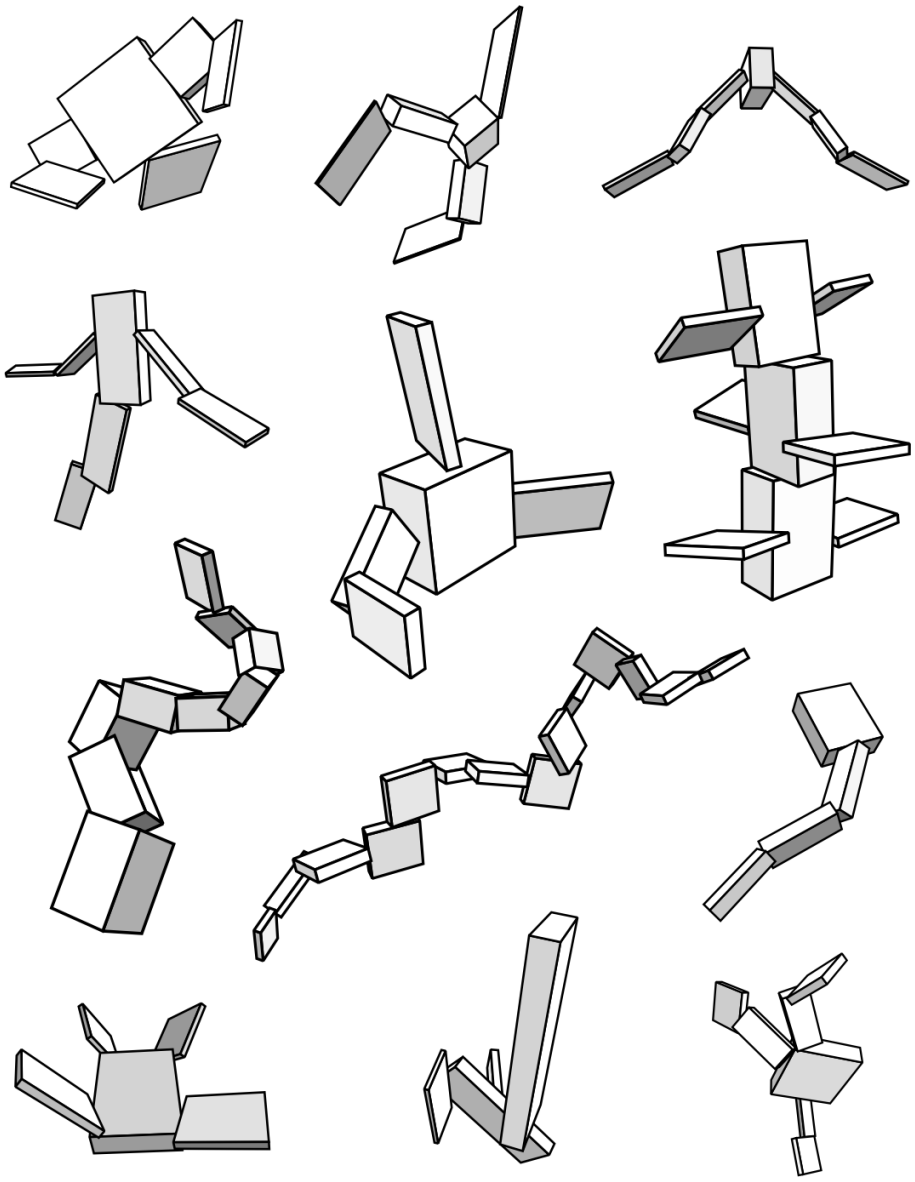


Figure 1.7: Creatures evolved for swimming

In this implicit representation, genotype was directed graph where each node represented cube and connections were mechanical joints. Graph could be cyclic, but just to the same node. This kind of connection in graph represented recursion and that particular part of body was replicated multiple times.

The resulting phenotype was constructed based on graph blueprint. Each cube also contained neural network connected to other neural networks in other parts,

simulating neural system. The neural network have different activation functions like oscillators or step functions.

1.5.3 Antennae

Maxwell's equations are designed to well describe electromagnetic phenomena, nevertheless building antennae with great reception is one of the difficult problems to solve. We have little clues about how to design a good antennae using Maxwell's equations. This creates nice playground for evolutionary design.

Jason Lohn used ED for creating superior quality antennae comparing previous fully human designed antennae.

Interesting part about this setup is usage of co-evolution. The fitness function was evolved parallel to antennae. The idea is to smoothly guide evolution from more simpler problems to harder, original problem.

2. Hypothesis formulation

In this chapter we formulate our assumptions and raise questions about ability of CA to solve complex problems. The results of experiments will be analysed in Chapter 4.

The main question is about global communication and organisation of CA just with local interactions. This means, that each cell does not have any information about its position on grid, size of grid or any other data about its environment. The only information available to cell is transition rule and states of neighbouring cells. However, it has to set its state correctly to match desired colour in that point of corresponding pattern. Organisation on global scale requires complex solution.

We have decided to examine 2 different settings on which we will analyse emergent phenomena. One setting of CA is using real valued state cells and feed forward neural network as a transition rule. The other setting has implemented Reaction Diffusion system (RD system) that uses chemical values of cells. The state of cell consists of real vector of chemical concentrations and internal state real valued vector. As transition rule is used more complicated neural network topology. All details about experiment settings are in sections of Chapter 4.

2.1 Problem definition

Dijkstra [8] has defined problem, which he calls the "Problem of the Dutch National Flag". The problem is stated as following

"There is a row of N buckets numbered from 1 to N. The buckets are arranged in numerical order with bucket 1 on left and bucket N on the right. Each bucket contains exactly one pebble and each pebble is coloured either red, white or blue. The problem is to arrange the pebbles in the buckets so that

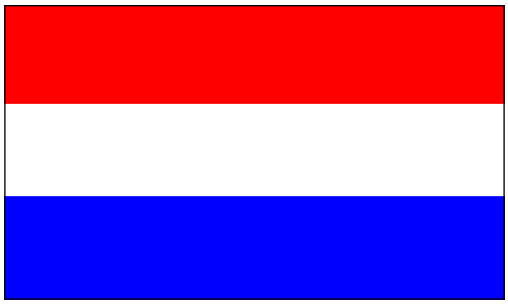


Figure 2.1 Dutch National Flag

(i) all of the red pebbles are to the left of all of the white pebbles and all of the blue pebbles, and

(ii) all of the white pebbles are to the left of all of the blue pebbles." [9]

We are using slightly modified original problem in our experiments. Colours are strictly in grayscale, represented by numbers in interval <0, 255>. The desired pattern of flag is in our experiments variable. We are using 4 basic shapes such as depicted in figure 2.2. Formulation of original problem also changes. Goal is to find such cell controller (transition rule) that assigns correct colours to cells according to given pattern.

2.2 Local and global communication

To colour cells of CA into some simple shape is very difficult task. Especially, if CA is forced to use only local interactions. Even though, for human beings it is quite easy task, CA would require complicated behaviour and many different states of cells. A complex mechanism would be needed to substitute for global communication and organisation on higher level.

Reaction-diffusion system is mathematical model describing behaviour of chemical concentration under influence of 2 processes, **reaction** and **diffusion**. The chemical reaction causes substances to be transformed into each other. The system

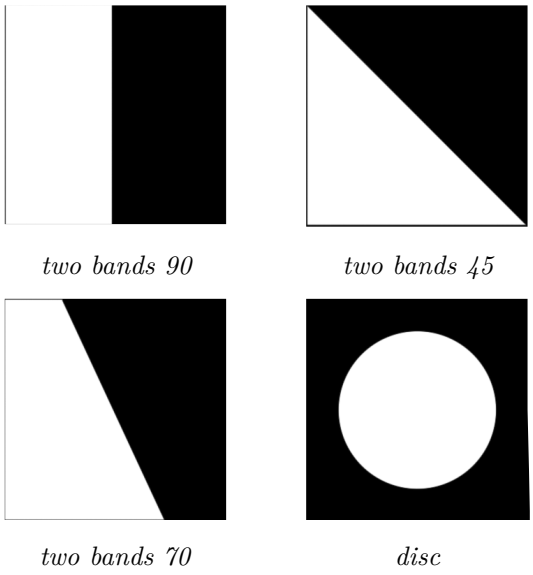


Figure 2.2 desired patterns

only with reactions and no diffusion, has two stable states. Either whole system has only chemical A or chemical B. But the diffusion, which spreads out chemicals, can cause interesting behaviours at the borders between system components. Most often the RD systems are configuration of 2 components, which means that there are 2 different chemicals. Models with one component are much simpler and create environment for basic emergent phenomenon. One of such phenomenon is emergence of homogeneous parts that travel through the environment at constant speed. In context of CA the one component RD system could model the main mechanism for global communication.

2.3 Desired pattern difficulty

On figure 2.2 are 4 simple patterns which are modelled by the evolving CA. The question is, how difficult is to model desired pattern and which properties are critical.

Since we are evolving weights for neural network (NN) that controls state of cell (colour) this network should be able to detect edges of patterns. NN are good for pattern recognition and hence the right tool [10] for detecting edges.

The assumption is that straight edge on *twobands90* and 45 degrees edge on *twobands45* should be the easiest to detect. If we take closer look on patterns as rasterised pixels (cells), we can recognise several patterns.

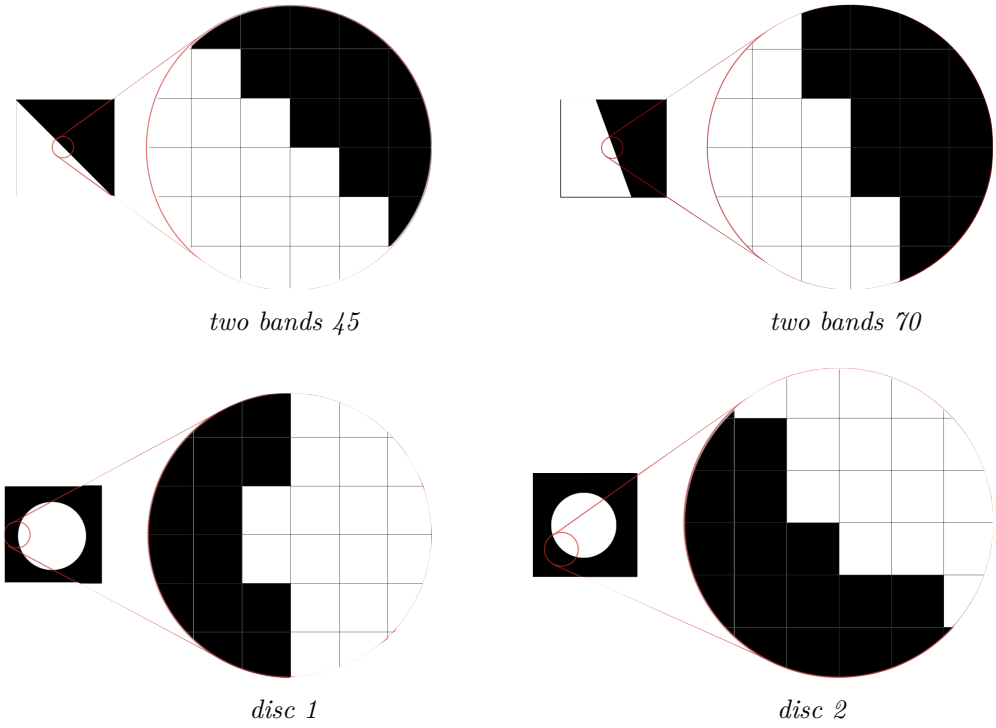


Figure 2.3 rasterised target patterns

Disc appears to be the most difficult pattern. It has 2 different patterns which are rotated 4 times by 90 degrees. It makes 8 patterns that neural network needs to be able to recognise. This applies to CA configuration without RD. Again, RD might provide some underlying mechanism that would made this assumption false.

2.4 Static borders as triggers

There are several possibilities of how to take care of border cases. Cells at edges of grid could be neighbours with cells on the other side of grid. Another way is to set static values on edge cases. When cell asks for state outside of grid, it gets static state, predefined for that position. This method could also work as triggering method for CA to start with initial information. From this initial configuration, CA can build up target pattern with evolved rules. The assumption here is that some of the evolved rules, could activate particularly on this initial configurations and start to propagate information further into grid.

To determine sensitivity of CA to values on edges, first configuration consists of all values equal to 0. Since values on borders are neutral, no triggering point would be available, hence the resulting CA would have higher overall error or would not evolve to feasible solution at all. Second configuration would deliberately create multiple possible triggering point what would result in shorter evolving time and more precise solutions. For each static border value we would choose same value as

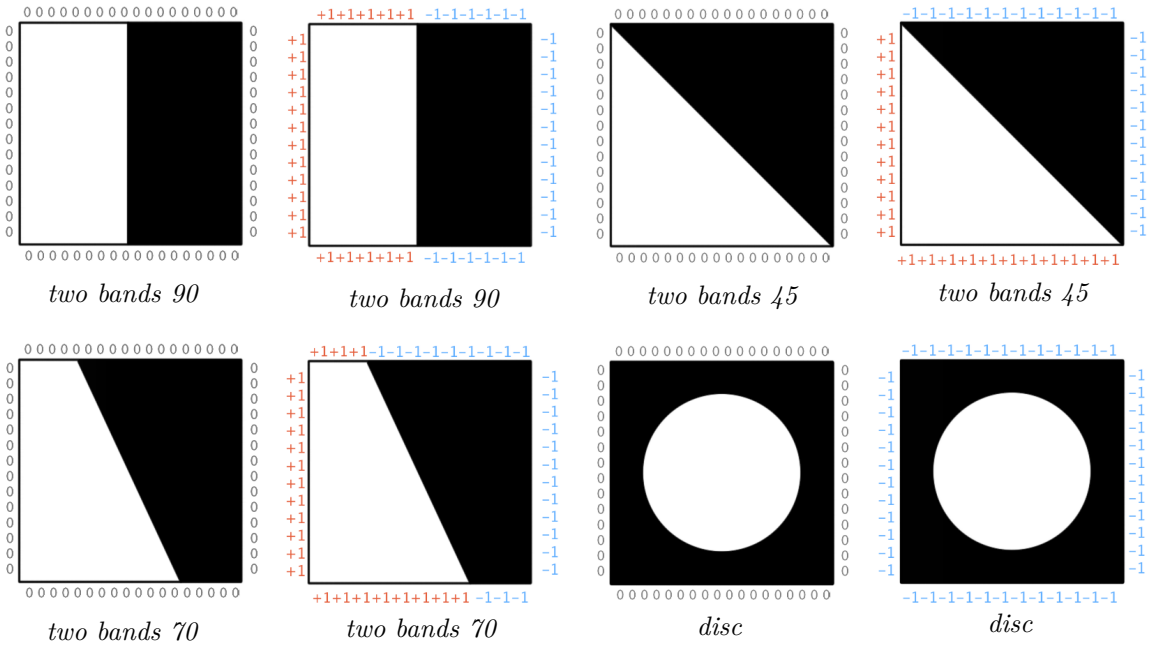


Figure 2.4 static border configurations

at nearest point in target pattern. Possible configurations of static border states are depicted on figure 2.4

3. Solution design and implementation

This chapter focuses on our development of CA Framework (CAF), its requirements and features. It describes concept of framework with easily reusable modules that is capable simulating wide range of experiments with evolving CA, evaluating statistical data and visualising the simulation.

One of the main goals of CAF is to replicate previous experiments and their settings. In order to recreate this experiments and provide tool for further exploration, flexibility is a key feature. To support this variety of settings, our solution was designed in the most general way possible. This, on the other hand, increases complexity. Nevertheless, changing properties for a experiment should be as easy as possible.

Selection of programming language for such task is important decision. We've chosen Python as our language because of its flexibility, meta programming capabilities and other high level functionality. Further, using other libraries and packages is great deal of help in rapid prototyping. This also helps to deliver quality solutions that are stable and tested. One of the biggest disadvantages over compiled languages is slow execution speed. Simulation of CA with neural networks on grid with significant size is complex computation and can be very CPU intensive. However, the grid like nature of CA enables us to execute computations on specialised hardware for parallel computing such as graphics cards. This is the way to speed up computation and scale CA experiments with more complex structures.

3.1 CA Framework

CAF is organised into 7 python packages (*cellular_automata, evo, methods, objectives, projects, wica, utils*). Each package implements different part of CAF. *cellular_automata* implements different sizes of grids, transition rules, individual cells and simulation of CA. *methods* contains different evolution methods like GA, CMAES, NES. *objectives* package implements objectives for different image goals (two band, triangle, structure). Each objective has its own error method and objective function. *evo* package uses packages *methods, cellular_automata* and *objectives* to evolve weights for given objective. *wica* and *projects* are packages for building web user interface and organising experiments into projects.

Core package is *cellular_automata* which implements CA functionality for simulation. Basic parts of this package are:

- Cell
  - base.py, regullar.py, irregular.py
- State
  - base.py, grayscale.py, color.py
- Lattice
  - base.py, equiangular.py, voronoi.py
- Rule
  - base.py, gol_rule.py, neural_rule.py, number_rule.py, stochastic.py
- Neighbourhood
  - base.py, vonneumann.py, moore.py

Since package cellular_automata is complex enough by itself, Python enables us to use nested packages. Whole framework is based on object oriented design pattern. Each one of the parts have their own base abstract classes (base.py), where public API is defined. Cells, rules, states, lattices and neighbourhoods have different

abstract classes where are required methods, but not yet implemented. These are implemented on different inheritance levels, when needed.

OOP allows us to use very interesting design pattern called duck-typing or duck punching. Despite our intentions to create compatible parts that can be assembled together, there are certain restrictions of how can be parts of CA framework combined. CAF doesn't prevent user from creating CA with incompatible components that can misbehave or raise exceptions during run.

Cell is basic building element of CA framework. Underlying abstract class defines position, state, pointer to rule. This class itself is not used. **SquareCell** inherits from Cell base class and implements neighbouring and adds more coordinates functionality. **VariableSquareCell** inherits from SquareCell. This class implements merging mechanism and brings irregularity into CA. Such cell introduce property that says if cell wants merge with others or divide into 4 equally sized cells. If 4 cells are aligned and one of the cells checks if its neighbours wants to merge, these are replaced with one double sized cell. State of new cell is median of parent cells.

State of cell is implemented with same design pattern as cell class. This class is intended for carrying information about cell state. State can be represented as binary, integer, real value or vector composed of these types. It has colour value needed for visualisation which is either RGB or grayscale. Underlying abstract class defines only one method *euclidian_distance* which is implemented only for states that have vector values. Higher classes are **BinaryState**, **GrayscaleState**, **ColourState**, **ColourTopologyState**, **ChemicalState**. All of these classes define only different set of stored information in state.

Lattice class is main building block in CAF. It covers the most important functionality for simulation of CA. All other classes are instantiated and assembled

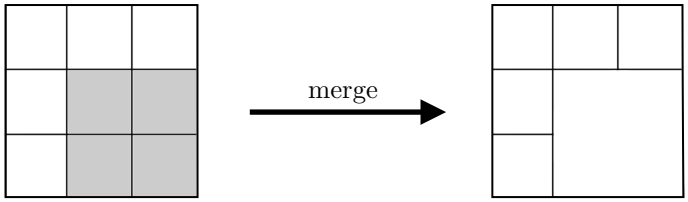


Figure 3.1 Merging of 4 cells

here. Base class implements initialisation, where based on specified dimensions and resolution the required number of cells (Cell class) is created, also with their initial states (State class). Pointers to neighbours are generated for each cell (Neighbourhood class). Reference to singleton of transition rule (Rule class) is assigned to cells as well. After initialisation is lattice ready for applying transition rule synchronously to each cell. More complex classes are build upon Lattice class. **VariableSquareLattice** implements CA with changing structure. Size of cells is variable and is determined with each step in time. **DiffusionSquareLattice** is based on concept of reaction-diffusion system (RD). In this class, with each step, function for chemical reaction is applied. After reaction, chemicals are diffused into cell's neighbourhood.

Rule is responsible for computing next state of cell. On the input are states of neighbours and state of cell itself. Output is used to set colour and state of cell. Depending on cell state type, other properties can be set as well. Such as merge/ division values, rgb or chemical vectors. Abstract class defines interface for rule, which consists of only from one function, *get_next_state.* All other classes implement different strategies for rules. **GameOfLifeRule** is using binary states and selects if cell lives or dies according to rule table (Figure 1.3). Totalistic cellular automaton rule is in class **NumberRule**. The most interesting rules are in module *neural_rule.py.* Here classes implements different neural network topologies.

Neighbourhood is simple class that is used in lattice initialisation process. **VonNeumann** and **EdieMoore** gathers neighbours for each cell and creates dictionary structure where key is direction (north, east, south, west) and value is list of neighbours in that direction. Normally this class would be not necessary, because neighbours could be hardcoded into cells what would simplify overall complexity. Nevertheless, ability to change neighbourhood gives us more freedom with experiments. For example, we could easily simulate second-order cellular automaton.

Visualisation is important for studying of evolved behaviour. This functionality is included in CAF as part of web interface. Simulation of CA is recorded into "replays" (steps of CA stored into file). With web interface this file can be loaded and rendered in HTML canvas with javascript.

3.2 Evolutionary Algorithms

We have decided to use reliable libraries for evolving weights of neural network instead of implementing algorithms by ourselves. The reason is that matured libraries are more optimised, tested and significantly shortens development time. **PyBrain** provides a toolbox for black-box, multi-objective optimisation and evolutionary methods besides other capabilities such as supervised learning, reinforcement learning, architectures of neural networks, compositionally, benchmarks and speed optimisation. Thanks to maturity of development, PyBrain has already been used at least in 14 scientific peer-reviewed publications [11].

For optimisation, we used Covariance Matrix Adaptation Evolutionary Strategy (CMA-ES) which is also part of the PyBrain toolbox. Following properties of our objective function makes CMA-ES the right solution:

- problem specific knowledge can't be used in black-box optimisation
- evaluation function time complexity makes exhaustive search infeasible

- search surface is not smooth
- dimensionality of search space
- non-separability of the problem where are dependencies between the objective variables

CMA-ES address besides these problems with objective function also

Detailed explanation of CMA-ES is in [12].

citation needed!!!

3.3 Experiments and simulations

For executing experiments we were using our framework, which covers all necessary functionality to perform such tasks.

Each experiment is defined as **project**. Each project has its own configuration file. Example configuration file in on figure 3.2

Here we define CA properties such as size of lattice, cell type, which state to use, transition rule and neighbourhood. If neural network is used for computing next state (transition rule), we can optionally define weights for it. Several options for evolving are in sections *stop_criterion* and *evolution*. Not to confuse, stop criterion is used to determine if running simulation of single CA should be stopped. There are other parameters for evolutionary algorithms to check if solution diverges and stops. Initial values are mandatory option only for strategies that require them.

One can either simulate CA and create *replay* file or evolve new weights with this configuration.

Evolved weights with corresponding value of objective function are stored with other information into *weights* file. Besides weights, we are storing objective function value, number of CA steps to get to static configuration and progress of evolving (generation number, objective function value and weights). The whole

```
[lattice]
width = 400
height = 400
resolution = 20
type = SquareLattice

[cells]
state = GrayscaleState
rule = FeedForwardNetwork
neighbourhood = VonNeumann

[network]
weights = [0.2398,...,0.3986]
```

```
[stopcriterion]
criterion = EnergyCriterion
max_time =1024

[evolution]
objective = PatternObjective
pattern = patterns/triangle.txt
strategy = cmaes
initial_values = [1,...,1]
```

Figure 3.2 configuration file for triangle pattern

progress is stored, because later one can examine development of solution from random values into final one.

As we can see, multiple configuration option gives user ability to quickly test and compare different hypothesis.

Replay files are just steps of simulation of CA, so user do not have to simulate it each time. This saves significant amount of time, since CA simulation on bigger grids is lengthy process. Behaviour of evolved CA can be visually analysed.

3.4 Statistics

Most of the statistical data is gathered into log file. Each run of evolutionary algorithm or simulation of CA, generates logs, with different levels of verbosity. Logging is redirected into file or standard output, from which (with different set of scripts) data is filtered and processed.

The most difficult part was to get information from PyBrain libraries, where we had to add some functionality to get information about generations and population size. Since objective function was running our code, all required logging could be

easily accessible for every evaluation. Most of logged information is ignored, however, in later development, when new hypothesis is raised, we could easily evaluate new data.

After all logs were processed, we have used python matplotlib library for plotting statistical data.

3.5 Complexity

Number of cells depends on size of CA grid. In our case, in 2 dimensions the number of cells is quadratic function of side of square what gives us

$$\#ofcells(n) = n^2$$

Since activation of neural network on any input happens in constant time, this does not depends on number of cells. To get next state for each cell in one iteration we get following upper time bound

$$T(n) = n^2 \times C$$

Worst case scenario, makes 1024 iterations, what is constant time.Overall complexity of 1 CA run depends on cell count and has quadratic upper bound

$$T(n) = n^2 \times C \times 1024$$
$$T(n) = O(n^2)$$

During our experiments, the average number of steps was 575 iterations.

3.6 SIMD architecture

Single Instruction, Multiple Data (SIMD) is a class of parallel computers, that exploits data parallelism. Single instruction is applied on multiple data points simultaneously. Bottleneck of our application is in simulation of CA time steps. Single time step requires quadratic amount of time. In one time step, for each cell needs to be computed next state. All other time consuming tasks are outsourced by external libraries that provide optimised code and even ability to use faster implementations for even bigger speed boost.

Simulation of CA is ideal candidate for SIMD architecture and could be solution to CAF speed problem. In terms of SIMD, the instruction is neural network that requires input values and outputs next state for cell. This instruction is applied to every one of cells in grid in given time step. The data are obviously the cells of CA. States of cells are updated synchronously, what means that next states are applied after all of them are computed. This property enables us to compute states in parallel. In case of asynchronous CA, the parallelism would not be possible, because data would change dynamically.

The most popular options for integrating such architecture are CUDA toolkit or OpenCL. Other options are mostly vendor specific solutions. While CUDA is arguably the incumbent standard, OpenCL is an industry wide effort, supported by rage of companies. OpenCL has another advantage - it can work with multicore CPU as well as with GPU. OpenCL creates abstract architecture layer over CPU and GPU, so usage on different devices is transparent. The last argument why to use OpenCL is that there is Python wrapper, pyopencl that lets us access the OpenCL parallel computation API from Python.

The only possible issue with integration of General Purpose computing on GPU is the mapping of complex CA structure into GPU memory architecture. In following section is explained how OpenCL memory model works. But the problem is, that

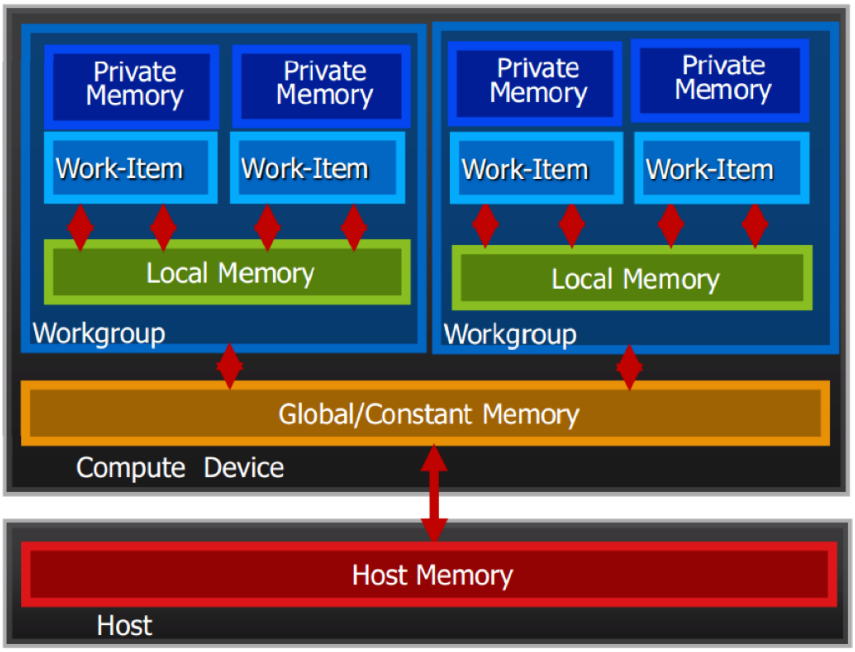


Figure 3.3 OpenCL memory model

each cell needs also states of surrounding cells to be able to apply transition rule. Since neighbourhood is defined dynamically, this would require significant amount of development time and expertise to implement mapping of such structure to GPU memory. We have made some initial implementation, but the more complex solution was, the amount of copying and moving data in GPU memory made it slow.

3.6.1 OpenCL

OpenCL is a programming framework and standard set from Khronos, for heterogeneous parallel computing on cross-vendor and cross-platform hardware. It provides a top level abstraction for low level hardware routines as well as consistent memory and execution models for dealing with massively-parallel code execution.The advantage of this abstraction layer is the ability to scale code from simple embedded microcontrolers to general purpose CPUs from Intel and AMD, up to massively-parallel GPGPU hardware pipelines, all without reworking code.

The execution model [13] of OpenCL consists of:

- Kernel - basic executable unit (~ C function)
- Program - collection of kernels and functions
- Command queue - application queue kernels & data transfers
- Work-item - an execution of a kernel by a processing element (~ thread)
- Work-group - a collection of related work-items that execute on a single compute unit (~ core)

On figure 3.2 (from [14]) is depicted memory model, where each work-item operates with its private memory (fastest) and has access to local memory (shared within work items) and global memory. Work item has write access to its private memory, workgroup memory and allocated output memory, where results of computation are stored. Work items can be synchronised, but only within work group.

This model could be exploited to boost speed in our implementation of CA simulation. We propose draft for matching of CA to OpenCL memory model. Single work item would represent cell with its state in work items private memory. The whole grid would be divided into chunks of size of work group. Additionally each work group needs values of border cells from neighbouring work groups, what would make acceptable memory overheat. Also, underlying libraries for neural networks would need to be ported into OpenCL C language. Fortunately, OpenCL provides basic math functions used in neural networks.

Minimal example of program that uses pyopencl is on figure 3.4 (from [15]). Program is adding 2 vectors and writing result into output vector.

```
import pyopencl as cl
import numpy
import numpy.linalg as la

a = numpy.random.rand(50000).astype(numpy.float32)
b = numpy.random.rand(50000).astype(numpy.float32)

ctx = cl.create_some_context()
queue = cl.CommandQueue(ctx)

mf = cl.mem_flags
a_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=a)
b_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=b)
dest_buf = cl.Buffer(ctx, mf.WRITE_ONLY, b.nbytes)

prg = cl.Program(ctx, """
    __kernel void sum(__global const float *a,
    __global const float *b, __global float *c)
    {
      int gid = get_global_id(0);
      c[gid] = a[gid] + b[gid];
    }
    """).build()

prg.sum(queue, a.shape, None, a_buf, b_buf, dest_buf)

a_plus_b = numpy.empty_like(a)
cl.enqueue_copy(queue, a_plus_b, dest_buf)

print(la.norm(a_plus_b - (a+b)), la.norm(a_plus_b))
```

Figure 3.4 minimal pyopencl program
adding 2 vectors

4. Experimental setup and results

In second chapter, we have raised assumptions that some particular configuration of settings could have positive or negative effect on ability to evolve transition rule that could solve our task. In this chapter we discuss each one of those settings and experimentally determine if they are true or false.

Many other configurations of settings for embryogenesis can be formed and experimentally analysed. At the end of chapter are several other properties, that could be tested with our framework.

4.1 Settings

Detailed settings of experiments are in following sections. For each combination, we have ran the experiments at least 10 times. Average evolving time without the RD system was 7 hours while lattices with the RD system took more than twice as long, in average 16 hours.

For evolving were used multiple computers of shared robotics laboratory of FMFI UK and FEI STU and some of our personal machines.

4.1.1 Lattice

For each experiment was size of the lattice 20 by 20, with square cells if fixed size. The type of lattice was *SquareLattice* for configurations without RD system and *DiffusionSquareLattice* with the RD system.

In CA are used 2 types of cell states. First one has only one real valued state variable

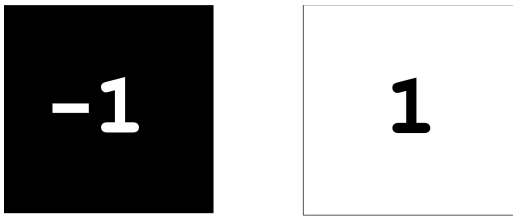


Figure 4.1 state mapping

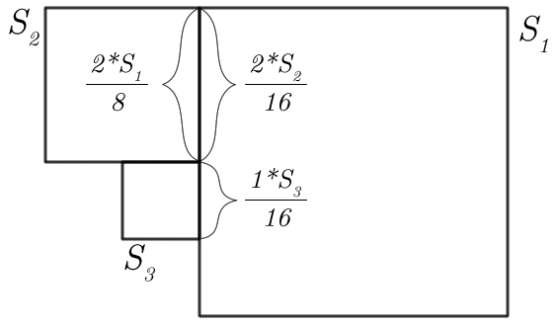


Figure 4.2 model of information flow

and colour representation on grayscale. State value is in interval <-1, 1> which is then linearly interpolated to grayscale value <0,255>. Value -1 is then displayed as black colour and 1 is white (figure 4.1).

4.1.2 Neighbourhood

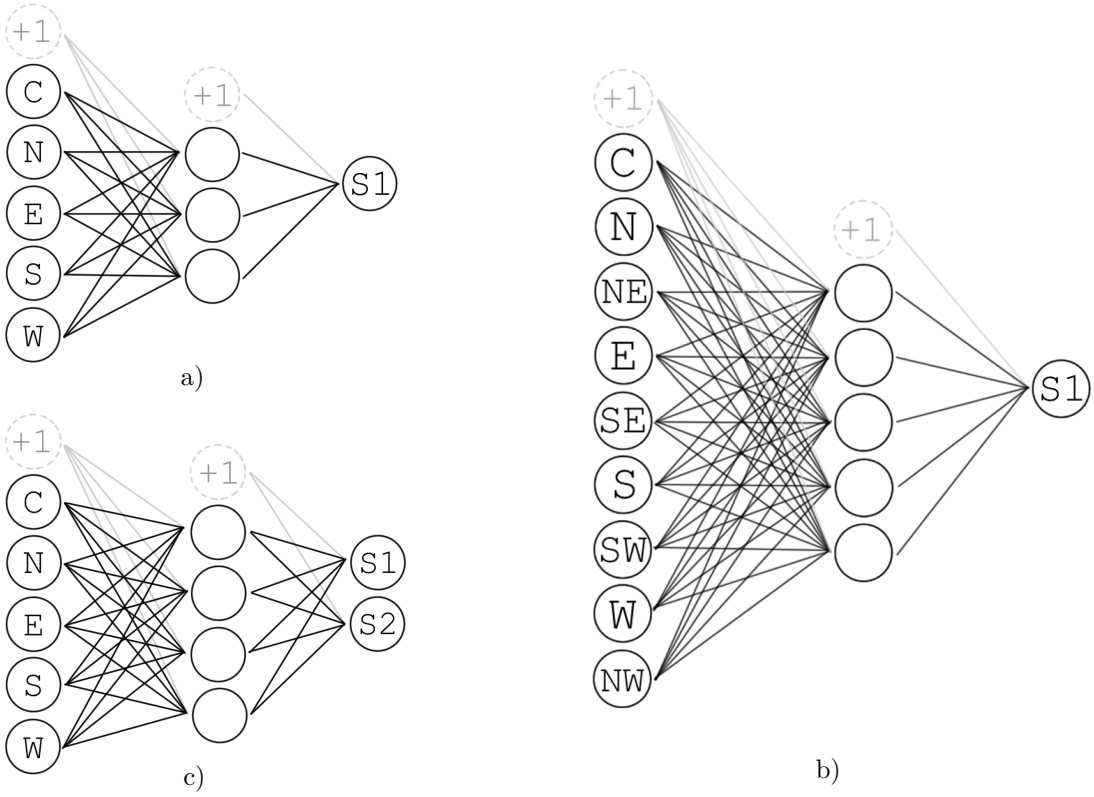
The default neighbourhood used in most of the experiments was von Neumann (figure 1.5) unless is otherwise stated. The reason for selecting the von Neumann neighbourhood was to resemble real cellular interactions in developing embryo. Cells share common membrane, through which are transferred chemicals. Depending on length of common membrane, the amount of information flows trough it. Mathematical model describing this information flow is on figure 4.2

Moore neighbourhood was used only in several experiments.

4.1.3 Static borders

Experiments was executed with both configurations of static border cells. One configuration was with all values set to 0. This was neutral configuration that would not give CA any triggering points.

Second configuration was supposed to give CA more triggering point to start colouring of grid. Setting of values of border cells was in this case for each



NW - north west	*N - north*	*NE - north east*
W - west	*C - cell own state*	*E - east*
SW - south west	*S - south*	*SE - south east*
S1 - out state 1	*S2 - out state 2*	

Figure 4.3 topology of used neural networks

experiment different. For each state on border cell, was assigned value often nearest neighbourhood on pattern. Configurations are depicted on figure 2.4

4.1.4 Transition rule

Three different topologies were used as a controller of cell state. On figure 4.3 are:

a) Feed forward neural network that is used as default setting for every experiment unless otherwise stated.

b) Feed forward neural network for Moore neighbourhood. This option is used in same manner as first one, but is larger because of longer input vector.

c) Feed forward neural network for controlling state and chemicals of cell. On the input is state of cell and chemical values of neighbours. This network is used in experiments with RD system.

Input vector consists of states of neighbouring cells in clock-wise direction. First value is cells own state (bias is not counted as first). Length of hidden layer depends on length of input, multiplied by factor of 0,6. Additionally, if output layer was longer than one, additional neuron was added.

4.1.5 Evolution algorithms

For evolving weights of neural network was used CMA-ES black-box optimiser. This optimiser is considered state of the art [16] for non-linear non-convex problems in continuous domain. Parameters of CMA-ES were adjusted in previous works ([17] [18]) and we use them as default settings for all experiments. Dimension of search space is equal to number of weights for neural network. Based on dimensionality of search space is population size and number of parents for selection in following table.

	POPULATION SIZE	# OF PARENTS FOR SELECTION
NETWORK TYPE A)	13	6
NETWORK TYPE B)	16	8
NETWORK TYPE C)	14	7

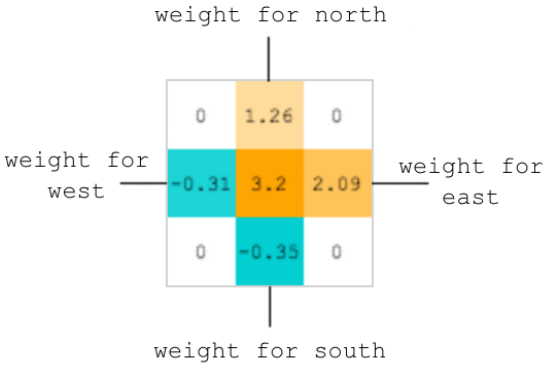


Figure 4.4 visualisation of first layer neuron

4.1.6 Objective function

When creating instance of CMA-ES one of the arguments is objective function. This function gets called for each evaluation of weights. Afterwards, instance of CA is created and weights of neural network are set. CA is simulated and stopped after it gets to stable configuration. Evaluating of stable configuration is done by stop criterion, described in following section. Simulation can be halted by exceeding maximum number of steps. Such weights are then evaluated as unstable and maximum error is returned. For stable configurations, error function is called that compares final colour of cells with target pattern. Error is calculated by following equation:

$$\sum_{i=1}^{C}\sum_{j=1}^{P}(C_i - P_j)^2$$

C is list of cells of simulated CA and P are cells of target pattern. Error value is normalised to interval <0, 1> by linear interpolation.

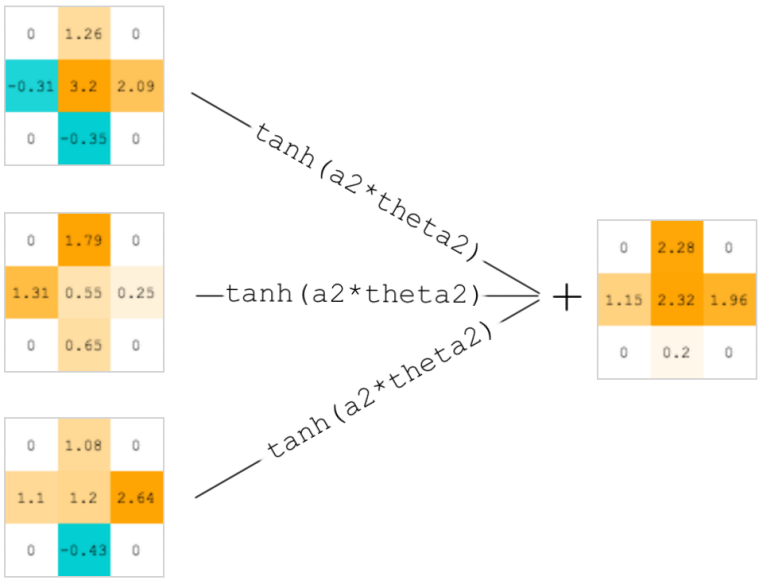


Figure 4.4 visualisation of second layer

4.1.7 Stoping criterion

Halting simulation of CA is based on 2 criteria. Maximum time of CA simulation and energy of CA. Energy of CA is calculated from euclidean distance of state vector for each cell. Sum of distances represents energy of CA in single time step. After variance of energy in time window of 15 steps exceeds threshold, simulation of CA is halted.

4.2 Results

Select several experiments that explores given hypothesis. Each experiment consists of at least 16 runs of experiment. We select just 200 generations to execute it in feasible time.

Visualisation of neurons is as follows. First layer is displayed as map of weights for each neighbour. Positive values are in shades of orange and negative values are in shades of blue. Each neuron has it's *min* and *max* value. Weight for given neighbour is then linearly interpolated to this colour domain (figure 4.2)

Second layer is multiplied by its corresponding weight in *theta2* and summed with all other neurons (figure 4.3).

Statistical results as number of generations, error values, evolving time, etc. is from log files. Behavioural analysis was done by our tool that takes evolved weights, visualise them and gives used ability to interactively change input values (states of neighbours) and immediately see change on colour of cell.

4.2.1 Two bands 45

Following table describes configuration of experiment.

Lattice	square grid without RD
Rule	feed forward network type a)
Neighbourhood	von Neumann
Static borders	with triggers

Regularity of 45 degrees line was the most successful shape to model. Almost perfect solution was evolved in first 20 generations and improved over the time.

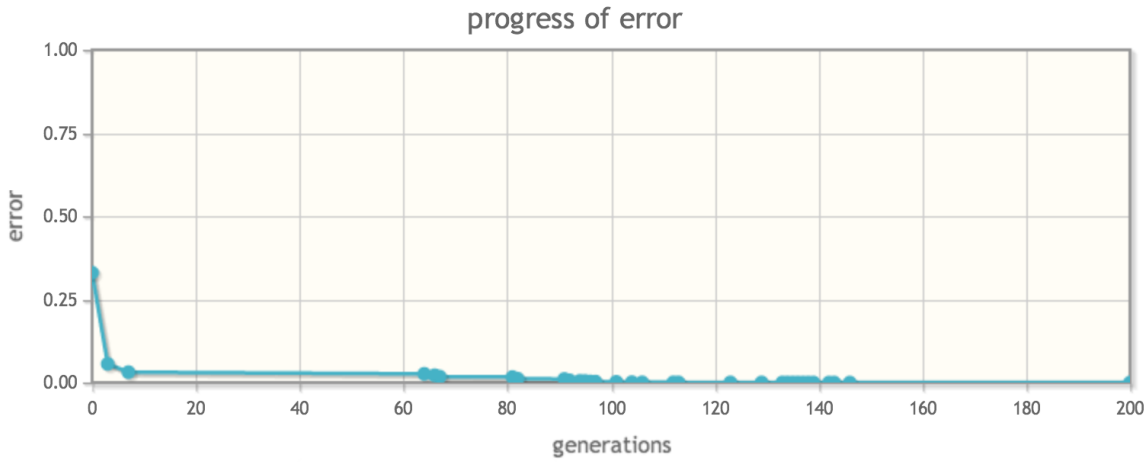


Figure 4.5a

On figure 4.5a is plot of error over generations. Each point represents improvement of solution. CA can solve target pattern in 90 steps. Figure 4.5c

depicts progress of CA in 9 snapshots. By analysing which neighbour has the most influence on cell (figure 4.5b) we discovered 3 main behavioural patterns.

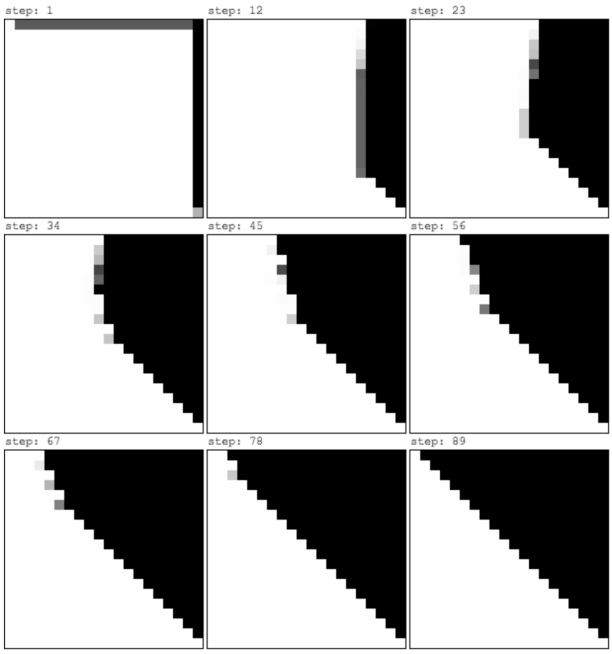
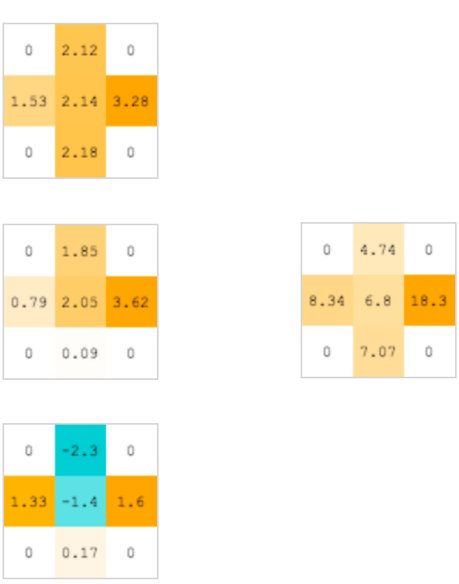
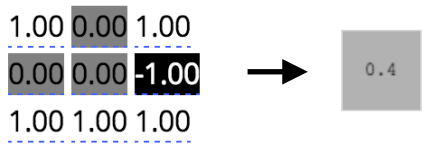


Figure 4.5c

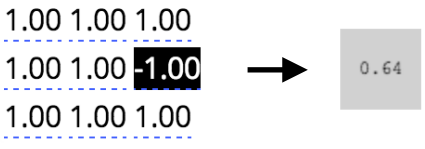
Evolved solution is using simple strategy based on propagation and stabilisation. of the edges. At first, it engages triggering rule at start of simulation (figure 4.5d). This rule assign white colour to right bottom cell. Second rule (figure 4.5e) is propagating states from right to left. Right neighbour has the most influence on cell state. Figure 4.5b supports this assumption. Highest weights are assigned to right neighbour. The last rule (figure 4.5f) is ensuring stable diagonal edge. The first assumption about detecting edges by neural network was correct. However, solution was based on triggering points. Setting static border cells to support this behaviour was critical. With neutral border states, most of evolved solutions were in average with 24% error rate.



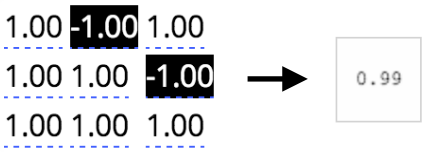
4.5b evolved weights



4.5d initial trigger



4.5e the propagator



4.5f stabiliser

Figure 4.6a

Nevertheless, we experimented also with RD configuration. These solutions were merely as good as those with previous configuration. Figure 4.6a depicts error development over generations. Here we can see rather flat development even though this was the best solution.

Also the progress depicted on figure 4.6b shows very poor results matching the target pattern. The effects of RD are clearly visible and there are no sharp edges.

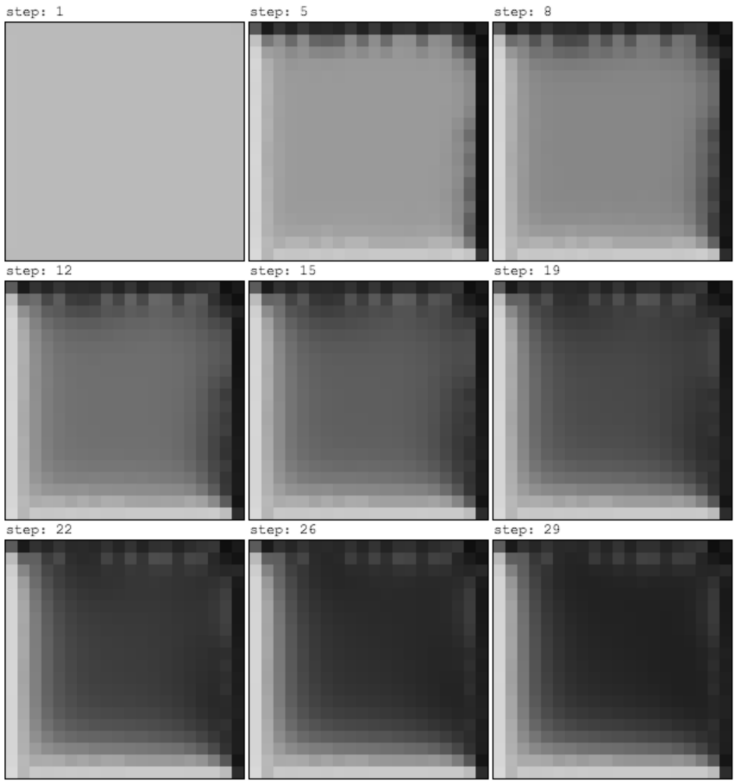


Figure 4.6b progress of simulation

The diffusion smoothes transition of white and black. The black part in desired pattern is here represented on by darker region and opposite part is lighter colour. This configuration uses lattice with RD system and FF type c).

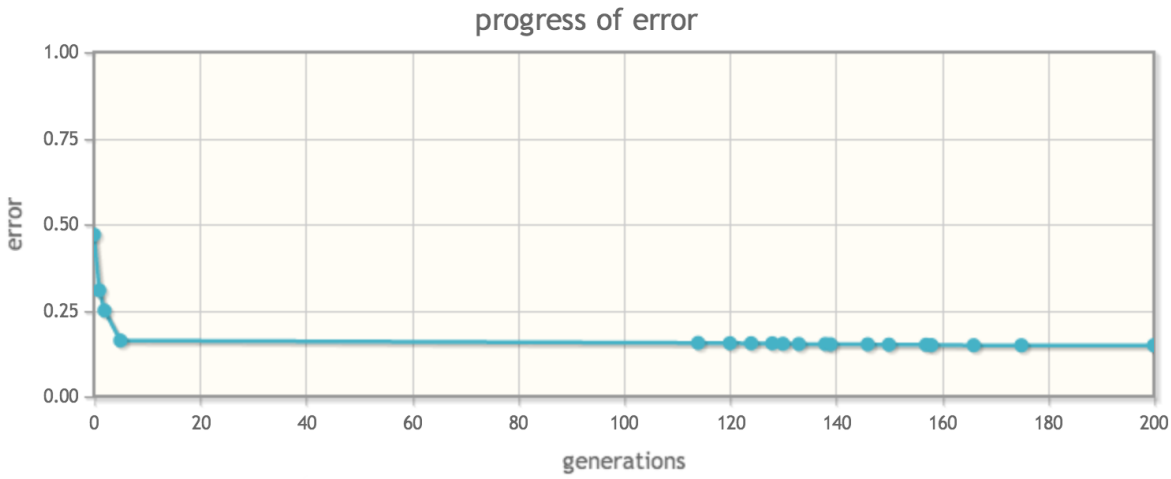


Figure 4.7a

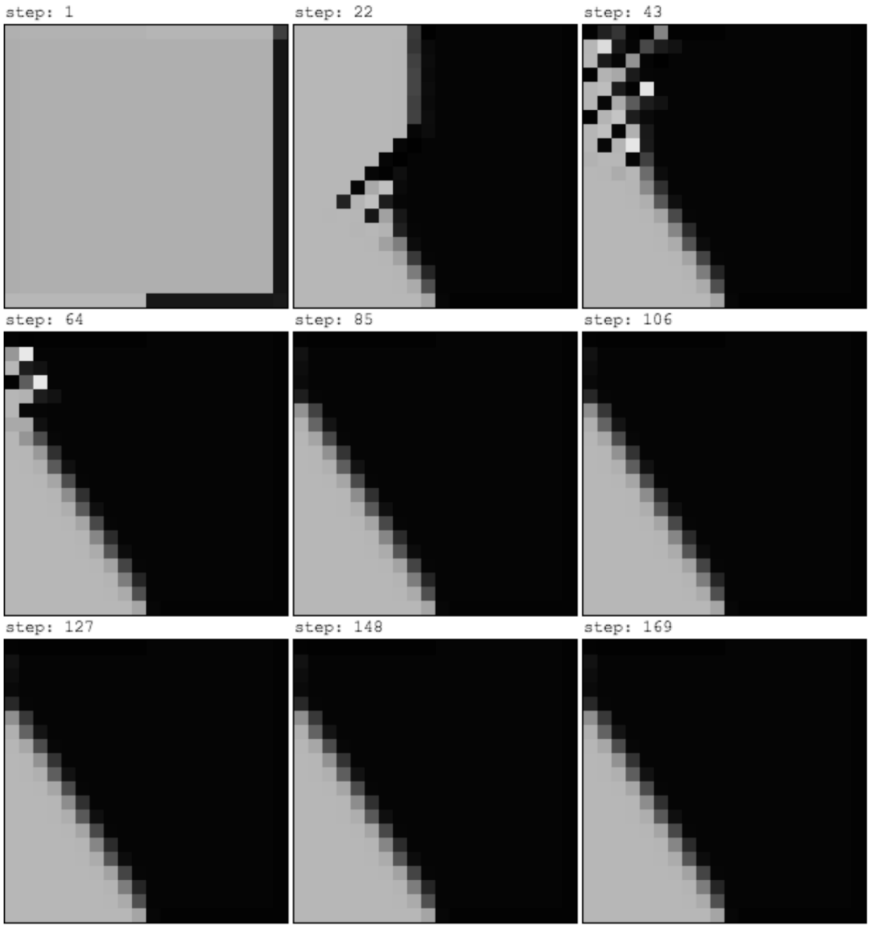


Figure 4.7b progress of simulation

4.2.2 Two bands 90

Following table describes configuration of experiment.

Lattice	square without RD
Rule	feed forward network type a)
Neighbourhood	von Neumann
Static borders	without triggers

Sensitivity of evolved weights on static borders are most obvious in this experiment. Most of the solutions were around 20% of error. The best solution had error value 0.1482134 what makes approximately 15% error with target pattern. Resulting solutions were using propagating strategy with angled edge along transition of white and black. As we can see on figure 4.7b, the simulation looks chaotic at the beginning, but the result is clear after CA gets into stable configuration. Very similar rules were evolved to those in two band 45 experiment. This particular solution more resembles two bands 70 pattern. The next configuration is using triggering borders and same neural network as before. Error immediately dropped to values close to 0 and we are getting 100% match with target pattern. Figure 4.8a shows that only in 2 generations we get the best solution. In all runs of experiment we get variations of same behaviour with

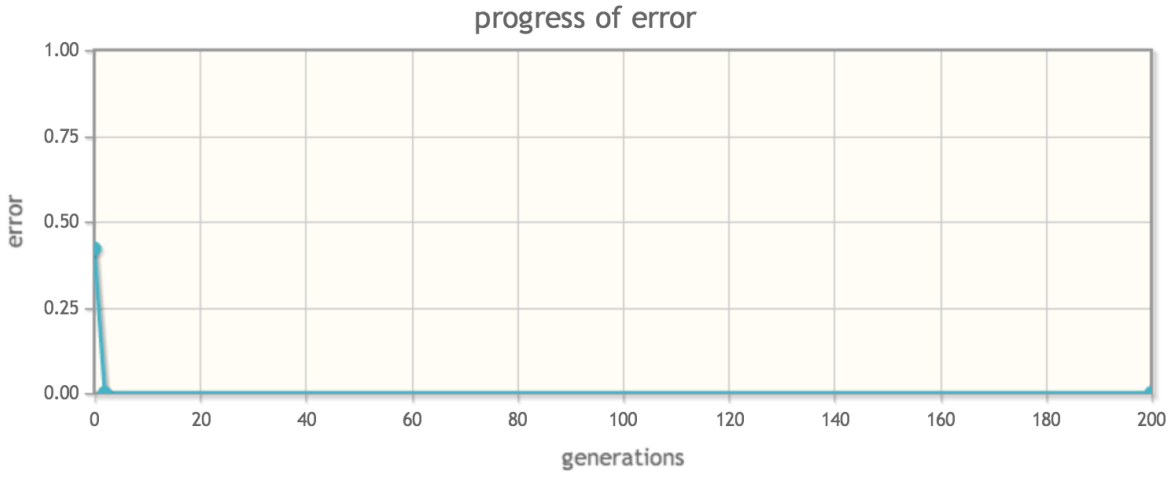


Figure 4.8a

different speeds of colouring. With triggering borders set, we were expecting the most simple solution and that would be just propagating information from top to bottom and vice versa. This assumption might be true, if objective function would take into account also speed of colouring. But the evolved solution were based on propagating information from right side and bottom neighbourhood. The biggest influence has right neighbour with weights of 1.62 on first neuron, 1.14 on second neuron and only 0.5 on the third. Figure 4.8c clearly show how information in automaton is propagated. Only in 49 steps

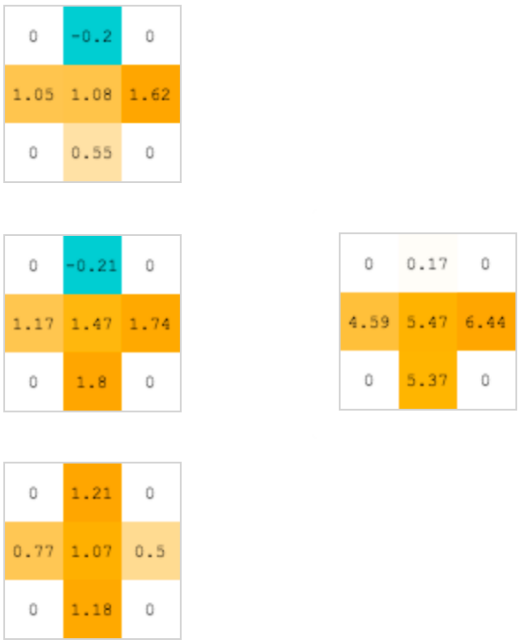


Figure 4.8b evolved weights

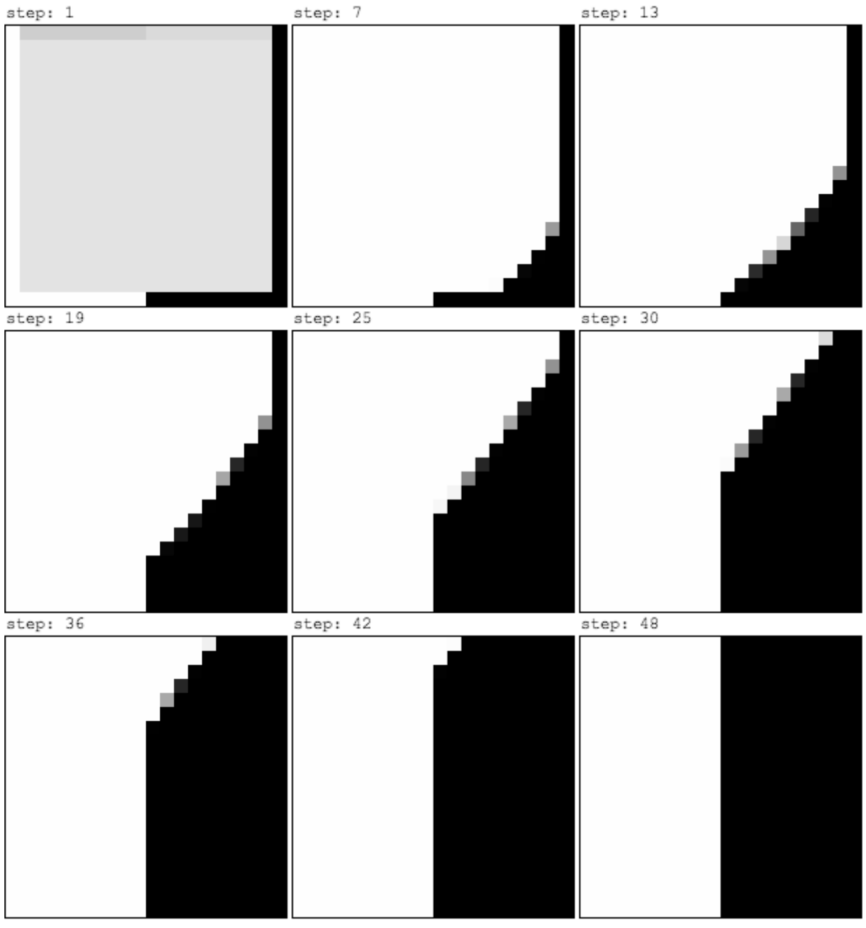


Figure 4.8c progress of ca with triggers.

was grid correctly coloured as target pattern. Again, static border cells shows strong influence on success of evolved solutions. Following settings are experimenting with lattice with RD system.

Again, the RD system seems not to react to static border triggers or to any other configuration. Following graph shows poor error drop over generations.

Figure 4.9a poor error development over generations

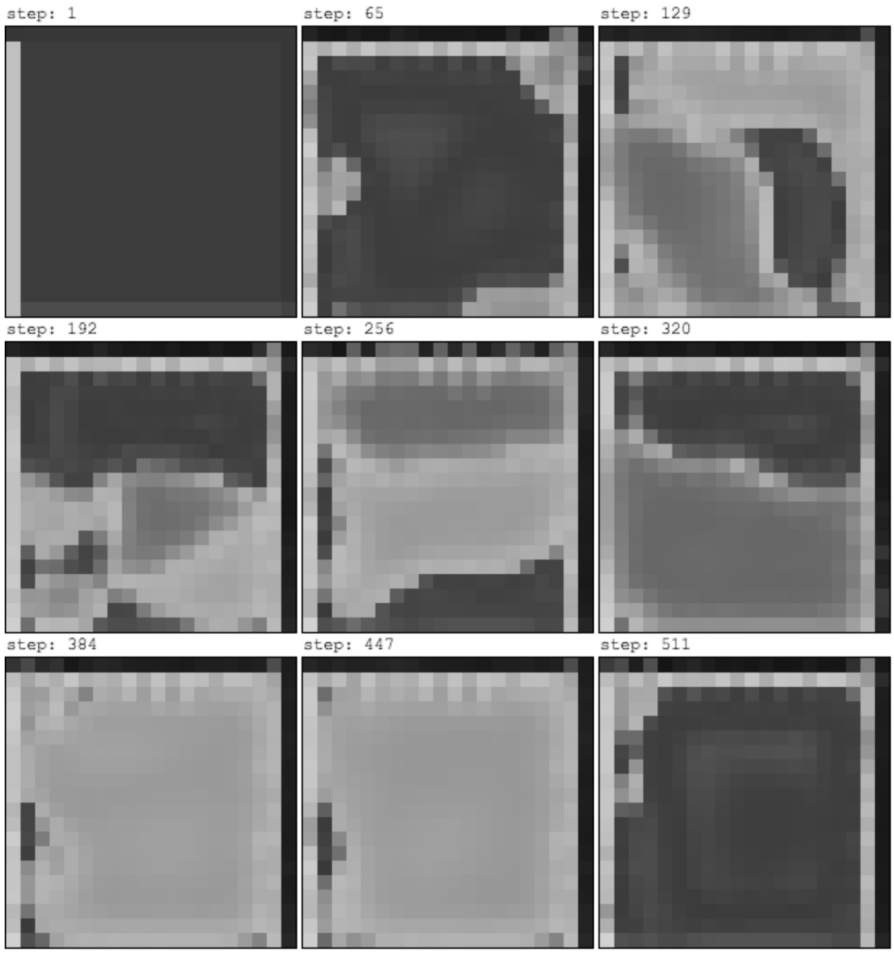


Figure 4.9b progress of CA with reaction diffusion system

The behaviour was also very chaotic and showed blinking configurations (figure 4.8b). This was caused by our stop criterion, because variance over selected time window would seems very low and would exceed stopping threshold.

Following analysis of evolved weights (figure 4.9c) shows connection to such blinking behaviour. On second layer we can see that one neuron is activating upon negative values, converting surrounding colours of neighbour to opposite. Second neuron does same thing for positive values.

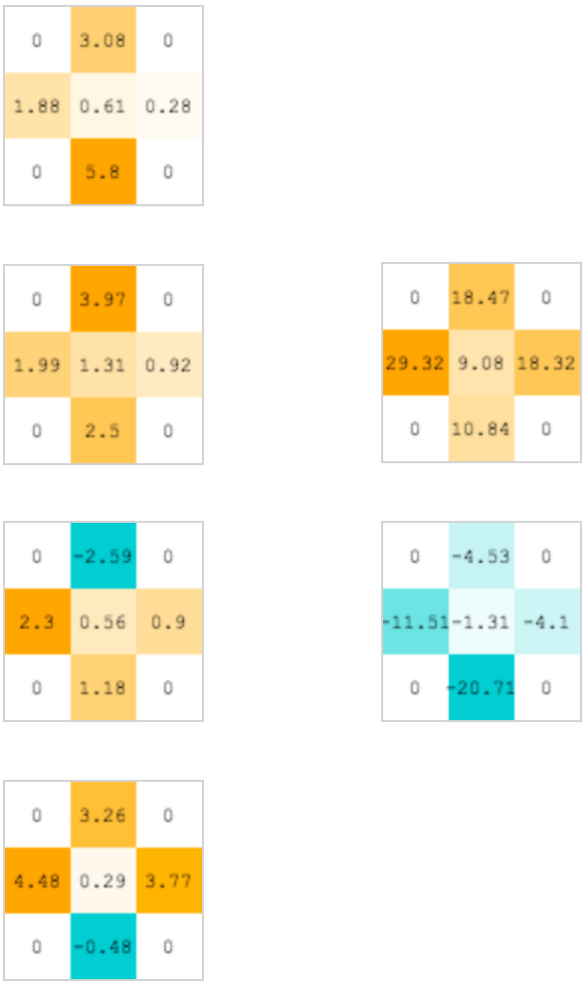


Figure 4.9c evolved weights

4.2.3 Two bands 70

Setting of this experiments are in following table.

Lattice	square grid without RD
Rule	feed forward network type a)
Neighbourhood	von Neumann
Static borders	with triggers

Experiment to replicate edge with 70 degree angle were one of the less successful. Neural network has to recognise rasterised pattern to be able to determine where to stop propagating information and create edge. The task is same as with 45 degree angle, but rasterised pattern is more complicated. Figure 4.10a shows that error rate was moving around quite low numbers, but the final pattern was not so

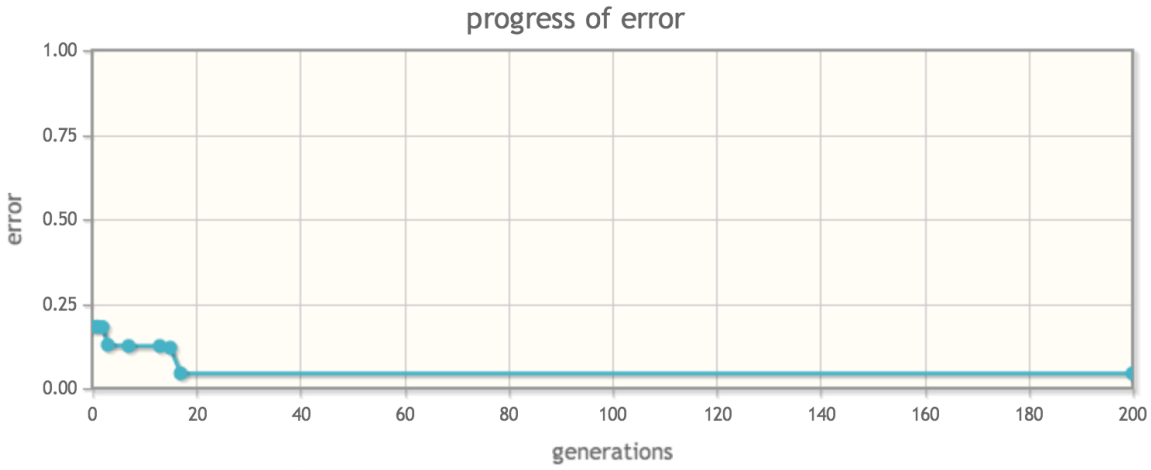


Figure 4.10a error rate over generations

accurate. Results were more similar to those for 45 degree angle. Final pattern is shown on figure 4.10b with other snapshots of simulation.

In this particular experiment we have tried changing neighbourhood to Moore's,. The reason for this change is that with Moore's neighbourhood there are more possible patterns to recognise by neural network. However the results were practically unchanged comparing to fist setting.

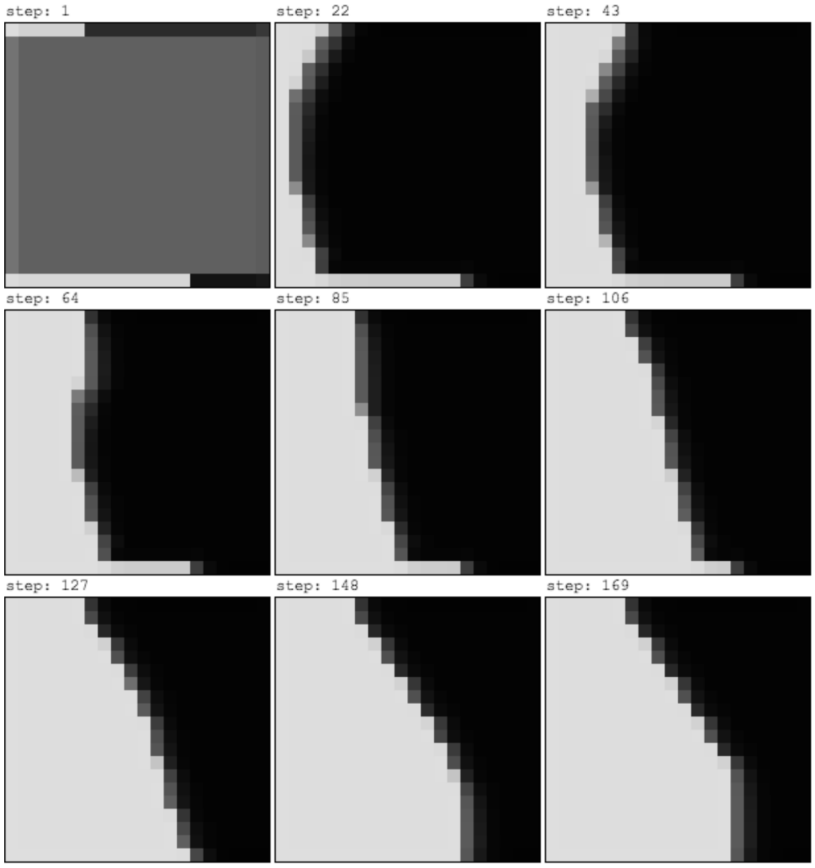


Figure 4.10b progress of 70 degree angle pattern

The RD version of lattice seems to bring similar unsatisfactory results as in previous trials. The average error in all trials was moving around 20% from which the best was with 13% error. However in this case, we could observe similar propagating behaviour as in previous experiments.

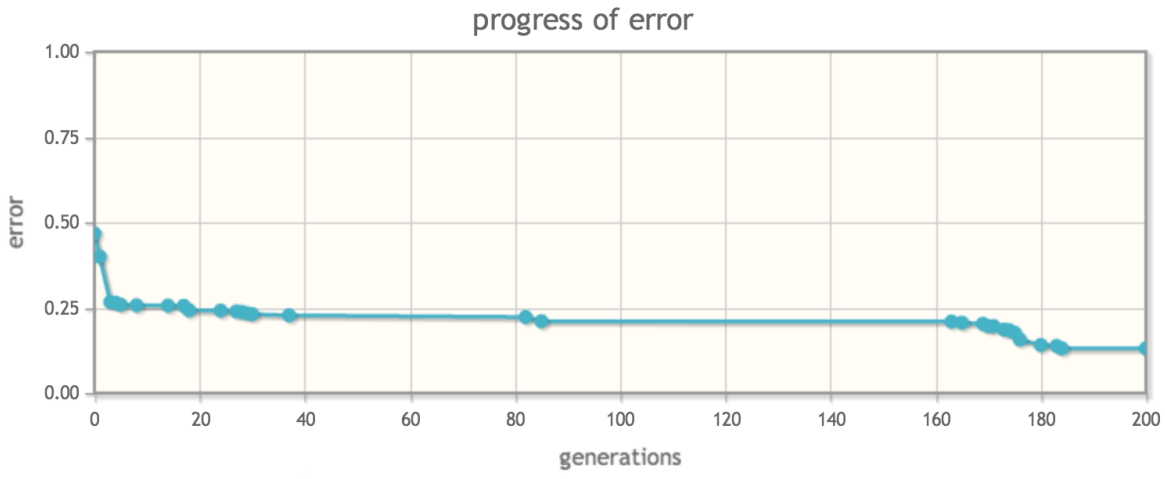


Figure 4.11a error rate over generations

From the right side, with static borders coloured to black, is this state propagated to the right side. Weight values are greater for right sided neighbours, but also it amplifies state values of cell own state. This means that states from right are decreasing values in cell (colouring them to black) and afterwards cell increases this state even more, serving (black) state to another neighbour.

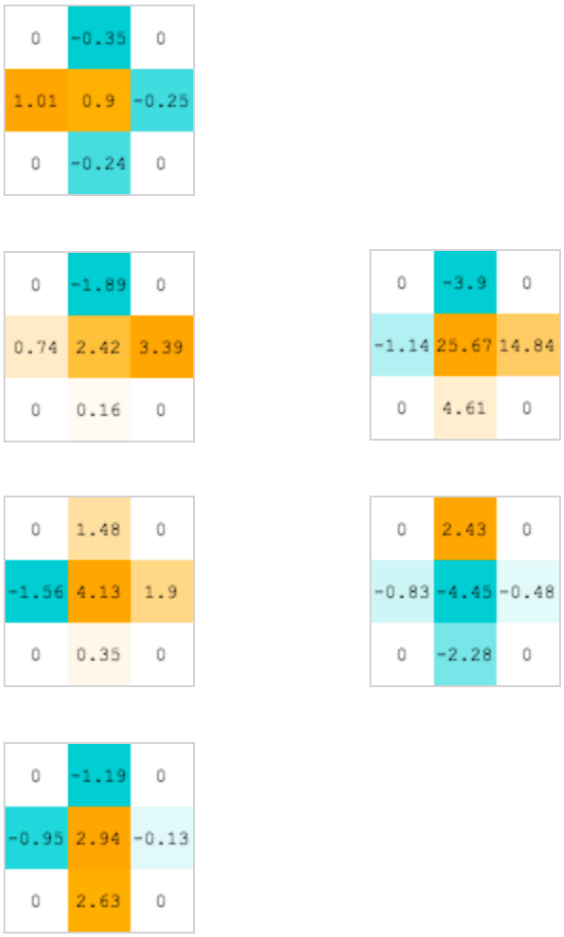


Figure 4.10c weights of first and second layer.

This feed forward network is showing complex behaviour and not all “rules” can be exactly described. Nevertheless we can infer global behaviour from the second layer, which gives highest values to the right and center cell.

4.2.4 Disc

Following table describes configuration of experiment.

Lattice	square lattice with RD
Rule	feed forward network type b)
Neighbourhood	vonNeumann
Static borders	with triggers

This was the best candidate for pattern that would be difficult for configuration without RD and FFN type a) and more easily solvable by configuration with the RD. In the most of previous solutions we can see, that system without RD has better error rate for straight lines. Disc pattern would require to detect 2 different edges (described in section 2.3) 4 times rotated by 90 degree angle. But the symmetric rotations would be easier to model for lattice with the RD. In figure 4.11a is depicted decrease of error over generations. The error dropped into 8% overall difference against target pattern. However examination of progress and actually evolved phenotype reveals that solutions was so close to the target pattern as one might say. Figure 4.11b shows progress of a CA over only 8 time steps.

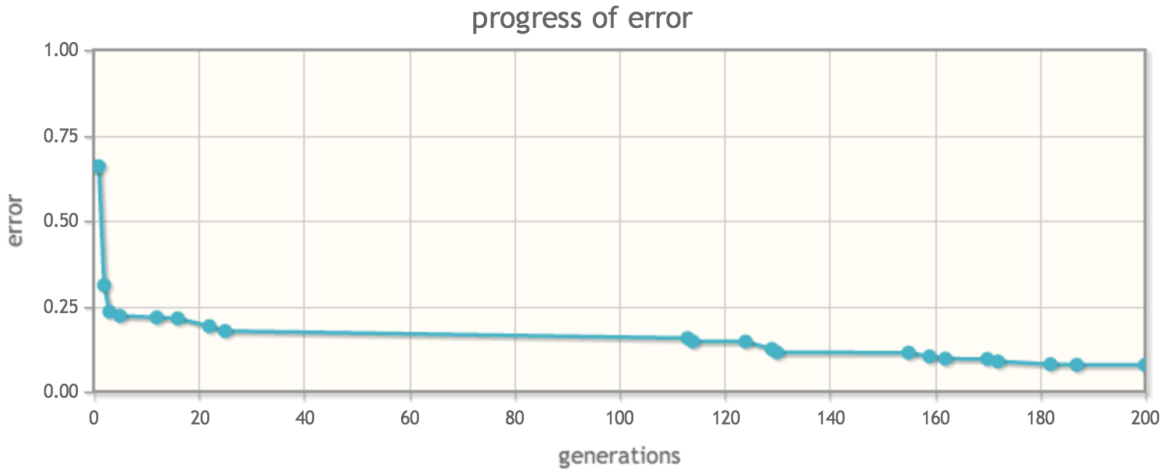


Figure 4.11a error over generations

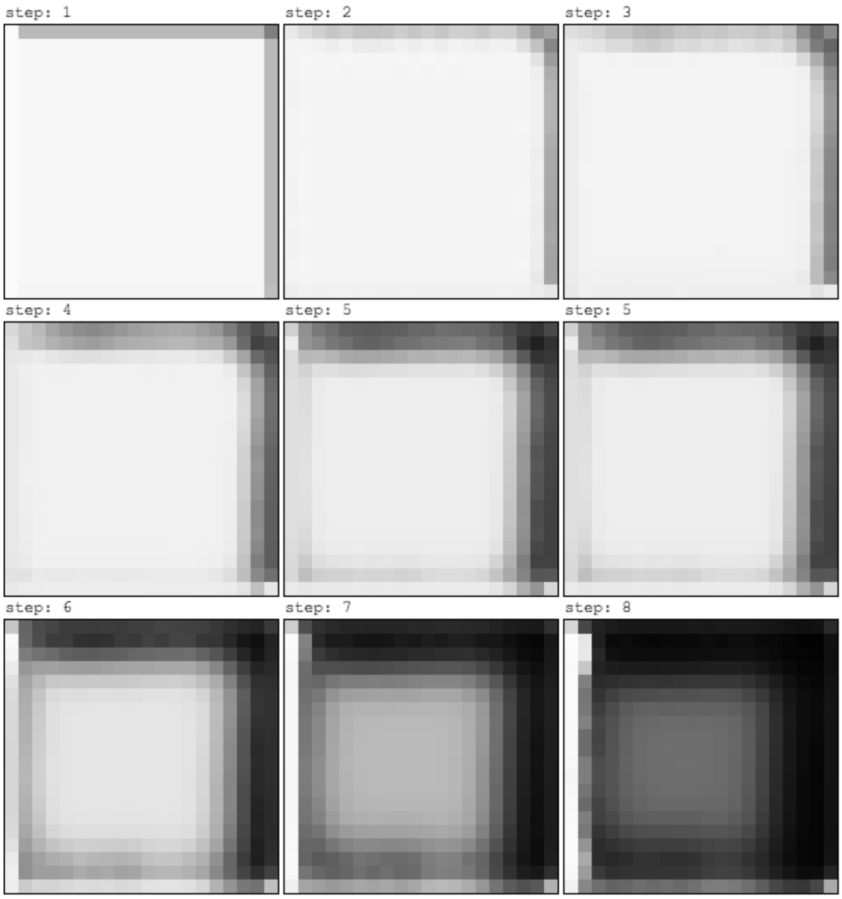


Figure 4.11b progress of CA over 8 time steps

Behaviour of this particular automaton shows tendency to propagate in formation from the edges towards the centre, however the right side and top side is propagated faster, resulting in slightly dislocated centre. Since states are propagated at constant speed from borders of a square, resulting shape resembles more of a square than disc. Figure 4.11c shows evolved weights. Here we can see, that the most important was state of a centre cells.

Figure 4.11c evolved weights

Because lattice with RD system has tendency to has more smooth edges, we tried configuration without to RD. Following error rate (figure 4.12a) shows progress of an error over generations for lattice without the RD system and FFN of type a)

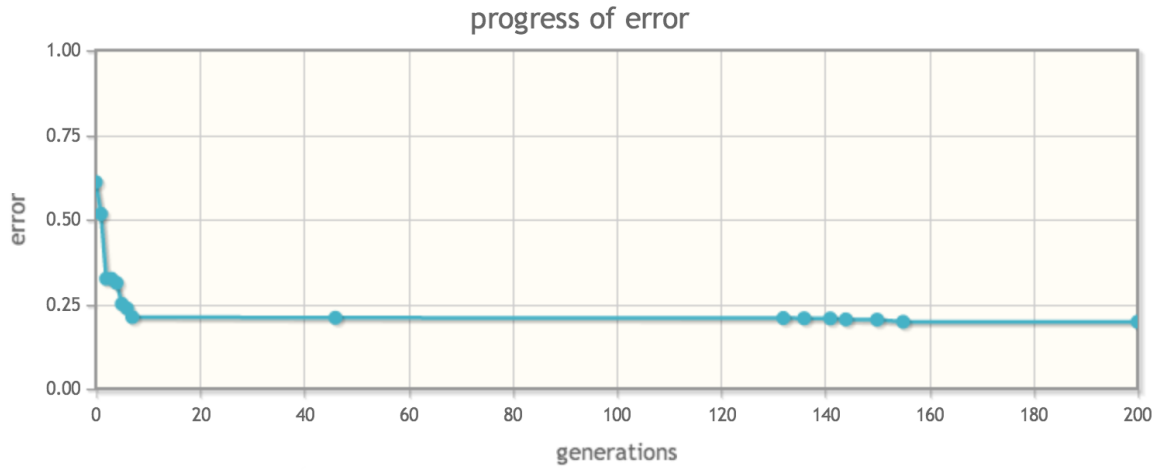


Figure 4.12a error rate over generations

This shape is particularly difficult for this set of parameters to solve correctly. Results in this case were poor, however if we take a look at figure 4.12b, complex behaviour seems to emerge. In one point it almost seems to get the disc shape in the right position, but this was not stable CA configuration and simulation resulted in lattice, mostly coloured by single grey shade.

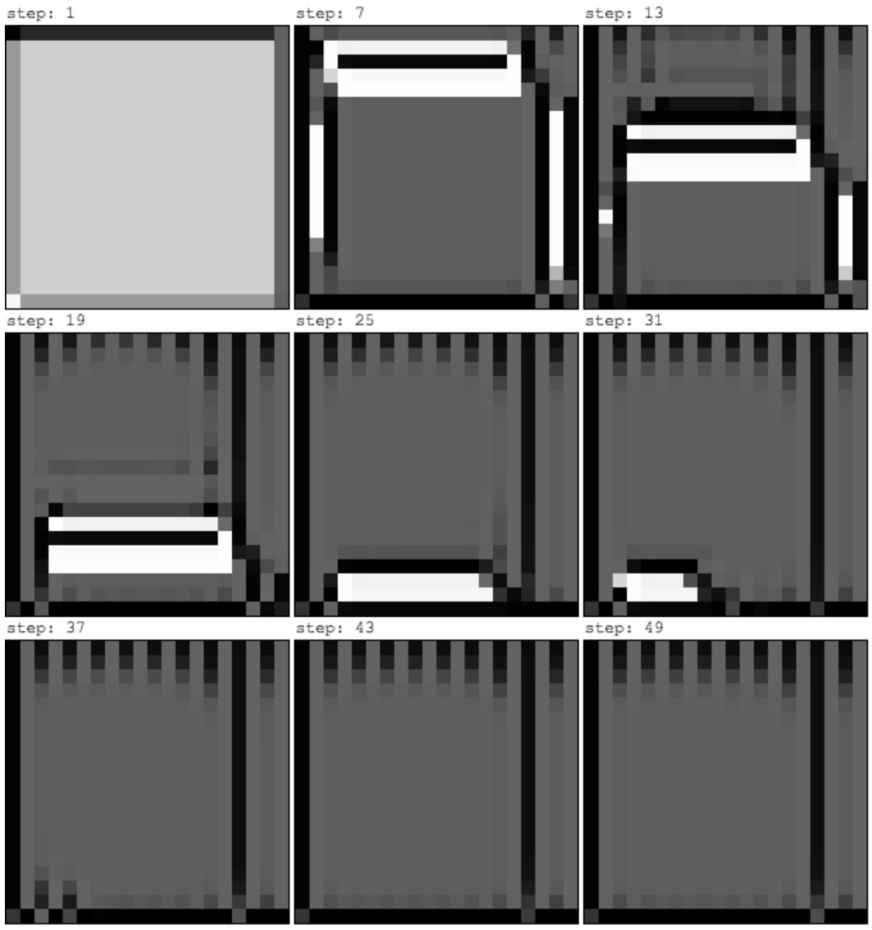


Figure 4.12b progress of a simulation

4.3 Discussion

Behaviour analysis of evolved solutions reveals, that incorporating RD system into lattice did not significantly improve solution. In some particular configurations was error rate smaller, but in overall RD did not provide mechanism to help emerge better solutions. Also sensitivity of RD configuration to static border states which introduced triggers was moderate or even weak. Introducing RD into lattice had also another consequence on quality of the solutions. This system could not model sharp and straight edges with clearly dividing white from black. Diffusion transferred the chemical concentrations into its sounding.

Introducing trigger points into grid, as in our assumption, helped to emerge more robust and faster solutions, that without triggers. This proved to be true. However, only to a solutions with higher sensitivity to triggers. Multiple solution exploited this and behaviour was based on trigger points.

5. Conclusion

We have implemented CA framework for evolving weights of neural controller where user can easily try out many different options. Evolutionary Design is still young and unexplored field and this makes CAF a good tool for such exploratory tasks where one can easily support or refute hypothesis. This framework was successfully used for examination of our hypotheses and gave us deep insight of what is happening on the background of simulated CA. We have raised several assumptions about ability to evolve some of the target patterns and each one of them was supported or refute by the behavioural analysis.

Because of implementation difficulties, CAF does not contains options for exploring CA on irregular tesselations. The architecture was intended to handle both types of CA, but irregular CA would not be able to run in feasible time.

Bibliography

1. Ralph, P. and Y. Wand, A proposal for a formal definition of the design concept, in Design requirements engineering: A ten-year perspective. 2009, Springer. p. 103-136.

2. Chen, W.-F. and L. Duan, Bridge Engineering Handbook: Construction and Maintenance. 2014: CRC press.

3. Podolny, W. and J.B. Scalzi, Construction and design of cable-stayed bridges. 1900.

4. Gutowitz, H., Cellular Automata. 1991: The MIT Press.

5. Wolfram, S., A new kind of science. 2002: Wolfram Media. 1197-1197.

6. Pickover, C.A., The math book: from Pythagoras to the 57th dimension, 250 milestones in the history of mathematics. 2009: Sterling Publishing Company, Inc.

7. Zuse, K., Calculating space. 1970: Massachusetts Institute of Technology, Project MAC.

8. Dijkstra, E., A discipline of programming. 1976.

9. McMaster, C.L., An analysis of algorithms for the Dutch National Flag Problem. Commun. ACM, 1978. 21(10): p. 842-846.

10. Bishop, C.M., Neural networks for pattern recognition. 1995: Oxford university press.

11. Schaul, T., et al., PyBrain. J. Mach. Learn. Res., 2010. 11: p. 743-746.

12. Auger, A. and N. Hansen, Tutorial: CMA-ES — Evolution Strategies and Covariance Matrix Adaptation. 2011.

13. Tompson, J. and K. Schlachter, An Introduction to the OpenCL Programming Model. 2012.

14. Trevett, N., OpenCL Introduction. 2013.

15. Kloeckner, A., PyOpenCL documentation. 2014.

16. Hansen, N. and A. Ostermeier, Completely derandomized self-adaptation in evolution strategies. Evolutionary computation, 2001. 9: p. 159--195.

17. Jana, H., Cellular Embryogenic Representations in Evolutionary Design. 2010.

18. Devert, A., Building processes optimization: Toward an artificial ontogeny based approach. 2009.