NTNU – Trondheim
Norwegian University of
Science and Technology

Evolution of Cellular Automata using Lindenmayer Systems and Fourier Transforms

Sivert Berg

Master of Science in Computer Science
Submission date: June 2013
Supervisor: Gunnar Tufte, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Sammendrag

Cellulære automater (CAer) er en gruppe veldig parallelle datasystemer. De består av mange små regneelementer som kalles celler. Cellene kan bare kommunisere med naboceller, noe som betyr at det ikke finnes noe global kommunikasjon i systemet. Å programmere et slikt system slik at det kan løse komplekse problemer kan være en vanskelig oppgave, derfor brukes ofte indirekte metoder. I denne masteroppgaven bruker vi evolusjonære algoritmer til å utvikle cellulære automater. Vi vil også se på bruk av L-systemer som en måte å utvikle komplekse CAer med et relativt lite genom. Inn- og utdata håndteres ved å streame dem gjennom kantceller, og vi ser på bruk av en diskret Fourier-transformasjon (DFT) som en måte å tolke utdata på. Eksperimentene viser at det er mulig å utvikle uniforme og semi-uniforme CAer som løser ulike problemer. Semi-uniforme CAer gjør det bedre enn uniform CAer på vanskeligere problemer, og bruk av L-systemer øker ytelsen ytterligere. På lettere problemer derimot virker det som den ekstra kompleksiteten til semi-uniform CAer bare hemmer utviklingen. Eksperimentene viser også at bruk av en DFT til å tolke utdata fungerer bra, og utkonkurrerer en mer direkte tolkning.

Abstract

Cellular automata (CAs) are a class of highly parallel computing systems consisting of many simple computing elements called cells. The cells can only communicate with neighboring cells, meaning there is no global communication in the system. Programming such a system to solve complex problems can be a daunting task, and indirect methods are often applied to make it easier. In this thesis we use evolutionary algorithms (EAs) to evolve CAs. We also look at the possibility of employing L-systems to develop complex CAs while maintaining a relatively small genome. Input and output are handled by streaming them through the edge cells, and we look at the use of a discrete Fourier transform (DFT) as a way to interpret the output. Experiments show that it is possible to evolve uniform and semi-uniform CAs that solve various problems. On harder problems semi-uniform CAs outperform uniform CAs, and using an L-system further improves the performance. However, on simpler problems the extra complexity of semi-uniform CAs seem to only hinder evolution. The experiments also show that interpreting the output with a DFT works well, and outperforms a more direct approach.

Contents

Chapter 1

Introduction

Cellular computing is a computational philosophy consisting of three principles: simplicity, vast parallelism and locality. It promises to provide new means of doing computation more efficiently, while simultaneously offering the potential of addressing much larger problem instances than previously possible [1].

The first example of cellular computing came in the form of cellular automata (CAs) conceived in the late 1940s by Stanislew Ulam and John von Neumann [2]. Since then massive amounts of research have focused on these seemingly simple machines, proving among other things that certain CAs are Touring complete [3].

Although CAs are able to do general computation, it is not always obvious how to program them [1]. This is partly because CAs rely on *emergent computing* [1]. Even though a single cell is deceivingly simple, the system as a whole can display complex global processing capabilities that is not explicit in the system definition [4, 5].

Because of this emergent property, CAs can be very hard to program directly. This is where so called *adaptive programming* comes into play. Instead of specifying what every cell should do, the system is only partly specified. It is then subjected to an adaptive process such as learning, evolution or self-organization [1]. Evolutionary algorithms (EAs) are a class of heuristic search algorithms used with success by other researchers [6] to program CAs, and the experiments in this thesis use EAs to find CAs.

As non-uniform CAs grow, the information needed to describe them also grows. If a simple 1-1 mapping between the genotype evolved by the EA and the resulting CA is used, the size of the genotype will grow linearly with the

number of cells in the CA. However, looking to nature we see organisms made up of trillion of cells developing from a single initial cell. This developmental stage is a crucial part in the creation of multi-cellular organism, and allows a relatively small genome to control the development of a vast organized structure. The genome contains the instruction for how to build the organism, while development carries them out [7].

Some of the experiments in this thesis looks at the possibility of using Lindenmayer systems, or L-systems, as a simple developmental-like mapping between genotypes and phenotypes. L-systems are parallel rewriting systems used to, among other things, model plant development. There are examples of simple L-systems producing remarkably sophisticated plant-like structures [8]. Along with their ease of mapping to CAs this will hopefully make them produce large complex CAs faster and better than a large genome with a 1-1 mapping between genotype and phenotype.

Various ways of passing input and extracting output from CAs have been devised. A common method is to encode the input into the initial state of the CA and decode the output from the final state after running the CA for a while [1, 6]. However, one could argue that this breaks with the distributed nature of the cellular computing paradigm. Having to touch all cells when setting the input and gathering the output requires a global view of the system. In addition there is no way to stream input to or extract output from the CA while it runs. Another approach, and the one taken in this thesis, is to use cells on the edges of a CA as the input and output. This keeps input and output local to the edge cells, and allows continuous streaming of data while the CA runs. The details are explained in Section 3.1.

Once an output stream is produced the question is how to interpret it. One possibility is to throw away all but the last value and use that as the output value. This will however throw away information about the behavior prior to the last output. What we really want is a way to somehow condense the output stream in a way that shows the CA's behavior over time. The discrete Fourier transform (DFT) can be used to convert a list of finite samples into a frequency spectrum. This would allow more than one output value to be taken into account. By using a DFT along with a loose output specification (for example the output should have 4 frequency peaks, but their location does not matter) we hope to give the EA enough freedom to find good solutions. A DFT could also be used to create multiple output values, for example by using the value of the two highest peaks as two output values. In this thesis we look at the possibility of using a DFT as a way to interpret the output stream. There have been some research done in this area earlier.

Aleksander Lunøe Waage looked at how different parameters for one dimensional CAs affected the DFT spectrum, and used EAs to evolve CAs with specific output spectrums [9]. He found that he was able to evolve the desired CAs in a binary rule-space, but not a ternary one.

Ole Henrik Jahren looked at boolean networks (BNs) instead of CAs, and used EAs to evolve BNs capable of producing different numbers of peaks in the DFT spectrum [10]. He was successful in evolving BNs that could produce different number of peaks dependent on the initial state.

Prior to this thesis, as part of TDT4501, I worked on a project concerned with evolution of cellular automata, and in particular the use of DFTs to interpret the output. Two-dimensional CAs were used, and the input was encoded in the initial state. The DFT was run on all the states the CA visited. This resulted in a large three-dimensional spectrum, and visualizing this spectrum to see what was going on turned out to be a challenge. Calculating the spectrum was also time consuming, and no clear advantages compared to a simpler mapping was found. The project also utilized a developmental model based on CAs, but it was outperformed by a genome with a direct 1-1 mapping to the CA. This thesis hopes to improve on both these areas. By only using the edge cells for output, and converting the output vectors to scalar values, the DFT spectrum will be one-dimensional. This should both make the spectrum easier to visualize and cut down considerably on the computation time. Hopefully it will also result in improved performance. Secondly, using L-systems instead of CAs in the developmental model might improve upon the simpler 1-1 mapping.

The rest of this thesis is organized as follows: background about CAs, EAs, discrete Fourier transforms and L-systems are presented in Chapter 2, Chapter 3 gives a detailed description of the experimental setup, Chapter 4 introduces the experiments along with their results and finally Chapter 5 concludes the thesis and gives some suggestions for future work.

Chapter 2

Background

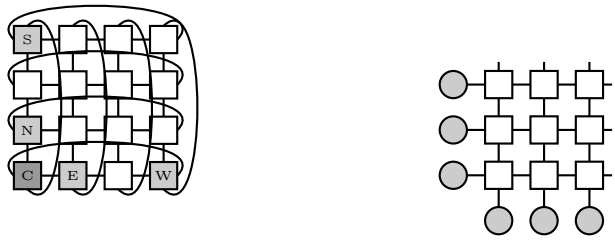
2.1 Cellular Automata

A cellular automaton is a D-dimensional grid of cells. Each cell can be in one out of a finite set Σ of possible cell states. For every cell a neighborhood is defined. We let N denote the number of cells in the neighborhood. A transition function $\Delta : \Sigma^N \to \Sigma$ maps the neighborhood of a cell to the next state of the cell. Figure 2.1 show two common neighborhoods for two-dimensional CAs. In a Von Neumann neighborhood the cells are only connected to the cells to the left and right, as well as above and below them, making $N = 5$. A Moore neighborhood also connects the cells diagonally, resulting in $N = 9$. The CA is updated synchronously, meaning every cell is set to its next state simultaneously.

Since CAs have to be simulated on computers with finite resources, we can not be using infinite grids. This gives us edge-cases where we have to decide what happens with cells that lie on the edge of the grid. This is called the boundary

(a) Von Neumann (b) Moore

Figure 2.1: Common neighborhoods in two-dimensional CAs



(a) Periodic (b) Constant

Figure 2.2: Boundary conditions

conditions of the CA. A common way of handling this is with periodic boundary conditions, where cells on the edge are connected to cells on the opposite edge. This is illustrated on a two-dimensional CA with a von Neumann neighborhood in Figure 2.2a. Another way is with constant boundary conditions, where the edge cells are connected to a constant value. An illustration of this can be seen in Figure 2.2b, where the lower left corner of the CA is pictured. The CA cells (white boxes) on the edges are connected to cells with constant values (gray circles). The CAs studied in this thesis uses a combination of the two, and will be described in more detail in Section 3.1.

The transition function Δ can be the same for all cells, resulting in what is called a uniform CA, or it could be specific to each cell, giving us a non-uniform CA [3]. The number of possible cell states is another important parameter. Von Neumanns's self reproducing automaton used cells with 29 possible states [2]. We will only concern ourselves with two cell states, that is $\Sigma = \{0, 1\}$. The Δ function can be represented by a lookup table (LUT). When we use two cell states it becomes a 2^N bit string, where N is the neighborhood size.

An example of computation in CAs is Melanie Mitchell's use of CAs to solve the majority problem [6]. The majority problem is the problem of determining if the initial state has a majority of 0s or 1s. If the majority of initial cells are set to 1 the end state should have all 1s, and if the majority of initial cells are 0 the end state should be all 0s. Solving this problem with a conventional computer that has a global view of the state is trivial. Just count the number of 1s and 0s and compare. In a system without global communication it is non-trivial, and with the CA type Mitchell used it is actually impossible to solve the problem for all inputs [11].

2.2 Evolutionary Algorithms

Evolutionary algorithms (EAs) are a class of search algorithms drawing inspiration from biological evolution. We will look at two subgroups of EAs, evolution strategy (ES) and genetic algorithms (GA).

An evolution strategy (ES) is an optimization technique inspired by evolution. One of the simplest ES models is called $(1 + \lambda)$ [12] and follows these steps:

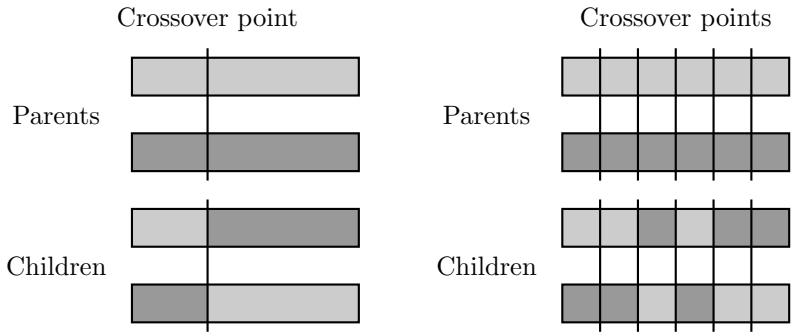
1. Generate random parent.
2. Produce λ offsprings from parent with a mutation operator.
3. Calculate fitness of parent and offsprings, and select the fittest individual. If there is an offspring with the same fitness as the parent, the offspring is selected.
4. Go to 2

In this thesis we use the model with $\lambda = 4$, and will refer to it as $(1 + 4)$.

The best individual in the $(1 + \lambda)$ algorithm discussed above will only be replaced if a better individual is found. This means an individual could survive many generations. This is not biologically plausible, and could lead to the search getting stuck at a local optimum. The standard genetic algorithm (GA) tries to solve this by using a generational model where parents live for only a single generation. The parent population is completely replaced by its offspring [13, 14]. A typical GA follow these steps:

1. Generate random population of size n
2. Map genotype to phenotype
3. Calculate fitness for every individual in the population
4. Produce n offsprings by selecting parents from the population
5. Replace the old population by the new offsprings
6. Go to 2

Producing the new population from the old population consists of two substeps; selecting which parents to reproduce and creating new individuals from these parents.



(a) One-point crossover (b) Uniform crossover

Figure 2.3: Crossover operators

Many selection mechanisms have been devised. The one used in this thesis is the *tournament selection* [15]. To select an individual, m individuals are pulled randomly from the population. The fittest of those m individuals are then the winner, and is selected. The parameter m is called the tournament size. When $m = 2$ it is called binary tournament selection.

After selecting which parents to reproduce, one or more genetic operators are used to produce those new offsprings. Common genetic operators are crossover and mutation.

The crossover operator is a recombination operator, combining two parents A and B into a new child. There are several variations on the crossover scheme. The simplest one is the one-point crossover shown in Figure 2.3a, where a single crossover-point is selected. Every gene before that point is taken from parent A and everything after is taken from parent B. There is also the uniform crossover illustrated in Figure 2.3b, where you can imagine a coin is flipped for every gene in the genome. If it comes up heads the gene from A is selected, and if it is tails the one from B.

The other genetic operator used in this thesis is mutation. This is the classic one-parent reproductive mechanism [16]. For every gene in the genotype there is a μ probability that the gene is mutated. The μ is called the mutation rate. What constitutes a mutation depends on the genotype. E.g. if it is a binary coded genotype it usually involves flipping a bit.

2.3 Discrete Fourier Transforms

We will be using discrete Fourier transforms (DFTs) to transform the output values into frequency spectrums. The output values are one dimensional bit vectors. We want to map these to a single scalar value. One way to do this would be to interpret them as binary coded integers. We will instead count the number of bits set in the vector, and use this number as the output value. If the output has 8 bits, this maps 256 possible output vectors to 9 different output values. By making it a many-to-one mapping we hope to give the system extra freedom, allowing it to explore many different solutions.

The standard DFT is defined as

$$F(u) = \sum_{x=0}^{N-1} f(x) e^{\frac{2\pi i u x}{N}}$$

where i is the imaginary unit, N is the number of output values, and $f(x)$ is the number of bits in the xth output vector [17]. Because we are mainly interested in the strength of the spectrum we will be using the absolute value of $F(u)$. On real values $|F(u)| = |F(N-u)|$. This allows us to ignore the spectrum values $F(u)$ for $u > \frac{N}{2}$.

To calculate the fitness some of the experiments use the weighted mean frequency of the spectrum, defined as:

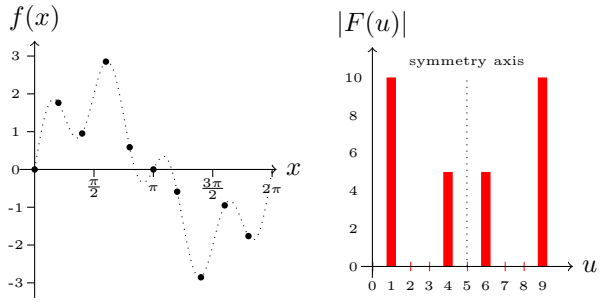
$$\overline{F} = \frac{1}{\sum_{u=1}^{\frac{N}{2}} |F(u)|} \sum_{u=1}^{\frac{N}{2}} u \cdot |F(u)|$$

This condenses the one-dimensional spectrum into a single scalar value, making it easy to use as an output value. Note that $F(0)$ is not included. This is because $F(0) = \sum_{x=0}^{N-1} f(x)$, and it does not tell us anything about the dynamic behavior of the output. To avoid it overshadowing the rest of the spectrum it is not included in $\overline{F}$. In the case where $\sum_{u=1}^{\frac{N}{2}} |F(u)| = 0$, $\overline{F}$ is defined as $\frac{N+2}{4}$.

Figure 2.4 shows an example of a DFT applied to the function

$$f(x) = 2\sin x + \sin 4x$$

The function is sampled at 10 points with equal distance between the samples. We see that the absolute values of the spectrum are symmetrical along an axis going through $x = 5$. As expected there are two peaks (if we ignore the symmetrical values), one for each of the sines.



(a) Function (b) DFT spectrum

Figure 2.4: DFT of $f(x) = 2\sin x + \sin 4x$ with 10 samples

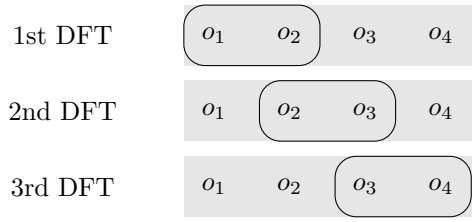


Figure 2.5: Sliding DFT with window size 2

Some of the experiments use a sliding DFT. A sliding DFT looks at the data through a sliding window. At the start this window is placed over the first N values, then the DFT is calculated on these values. N is called the window size. Then the window slides one value over, and the DFT is calculated again. If there are a total of M values and $M \geq N$ it will calculate $M - N + 1$ DFTs. Figure 2.5 shows an example of a sliding DFT with a window size of 2 and $M = 4$.

2.4 L-Systems

L-systems, or Lindenmayer systems, are parallel rewriting systems originally developed as a mathematical tool to model plant development [8]. The L-system rewrites strings by replacing all symbols in parallel according to some rules. The rewriting from the start string, called the axiom, and into the final

string can be viewed as a developmental stage. Mapping the produced string to a CA can be achieved in different ways. One method is to use a constant mapping from string position to cell position, and have one symbol for every cell type. Another more advanced method is using turtle graphics. This will be described in more detail below and in Section 3.2. A formal definition of L-systems is given below [8].

Let V denote an alphabet, V^* is the set of all words over V and V^+ is the set of all non-empty words over V. An L-system can be defined by a tuple $G = (V, \omega, P)$ where V denotes the alphabet of the system, $\omega \in V^+$ is a non-empty word called the axiom of the system and P is a set of productions. Productions are functions from V^+ to V^* and are written as $a \to \chi$. a is called the predecessor while χ is called the successor of the production. If no production is defined for an $a \in V$, the identity production $a \to a$ is assumed. A system where all productions have predecessors with only a single symbol ($a \in V$) is called a context free system. This is also termed a 0L-system. A 0L-system is called deterministic if for all $a \in V$ there exist only one production in P with a as its predecessor. This is referred to as a D0L-system.

An example of a D0L-system

$$\begin{aligned} V &= \{F, [,], +, -\} \\ \omega &= F \\ P &= \{F \to F[+F]F[-F]\} \end{aligned}$$

We can write out the 2 first iterations of this systems:

$$\begin{aligned} &0 : F \\ &1 : F[+F]F[-F] \\ &2 : F[+F]F[-F][+F[+F]F[-F]]F[+F]F[-F][-F[+F]F[-F]] \end{aligned}$$

It might not be immediately obvious how one could use this system to model plant growth. A common way is to interpret the string as instructions for a turtle. F tells the turtle to move one step forward while drawing a straight line. $+$ and $-$ tells it to turn left and right δ degrees, [makes it push its current heading and position onto a stack and] pops the most recently pushed position and heading. Figure 2.4 shows the resulting figure when we run the system for 4 iterations, set $\delta = 25°$ and run the turtle on the resulting string. As we can see it does indeed resemble a plant.

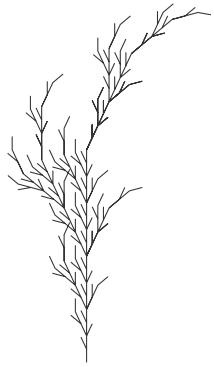


Figure 2.6: Turtle interpretation of an L-system

Chapter 3

Description

This chapter gives a detailed description of the experimental setup. The experiments try to answer different questions, but use the same framework. Figure 3.1 shows an overview of the system. An evolutionary algorithm searches the genome-space. To evaluate a genome it must first be mapped to a CA, then a fitness function feeds this CA input and inspects the output. Finally the fitness is returned to the EA. It is easy to replace parts of the system. This allows easy experimentation with different EAs, genome to CA mappings and fitness functions. The only constant block throughout all the experiments is the CA, only one kind of CA is used. The rest of this chapter will discuss the individual parts in detail.

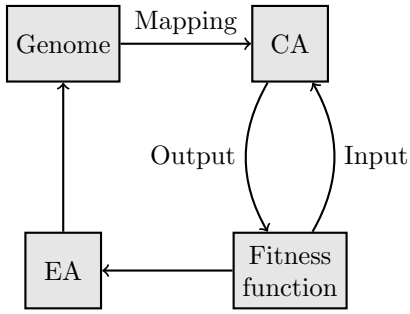


Figure 3.1: Overview of system

3.1 CA

The CA used in the experiments is a two-dimensional CA with periodic boundary conditions on two of the edges, and constant boundary conditions on the other two. It uses Von Neumann neighborhoods. Figure 3.2 shows an example of a 4x4 CA of this kind. The darker gray boxes are CA-cells, while the lighter circles are constant values.

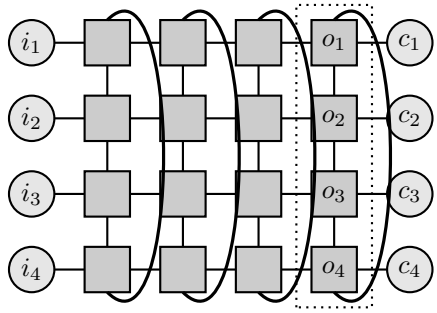


Figure 3.2: CA

The constant values on the left, labeled i_1 to i_4 are used for inputs. It is important to note that these values are constant only in the notion that they do not belong to another cell. They could change value from iteration to iteration, allowing streams of data to be fed to the CA. The constant values on the right (c_1 to c_4) are constant throughout the entire run of the CA, and we will refer to these bits as the configuration values. The initial value of the CA cells are 0.

The right-most row of cells, labeled o_1 to o_4 in the figure, are the output values. As we can see even though the CA is 2D, the input and output are one dimensional bit vectors. The hope is that by treating most of the CA as a blackbox, evolution is allowed greater freedom when searching for possible solutions. In addition having input and output on the edges allows streaming input and output continuously without having to look at the entire CA.

An $m \times n$ CA can be parameterized by a tuple (M, C), where $M : (\mathbb{N} \times \mathbb{N}) \to L$ is a function mapping cell position to elements in the set of lookup tables L and $C \in \{0, 1\}^m$ is a bit string with m bits. C is used to set the configuration values.

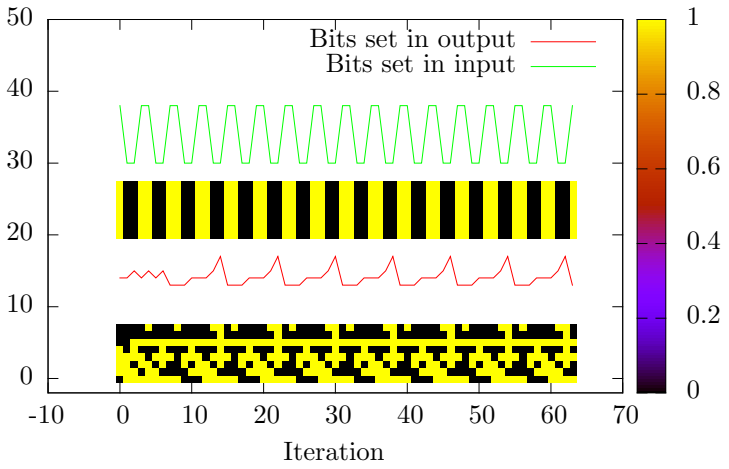


Figure 3.3: Example of CA input and output

Figure 3.3 shows an example of a CA run. The green line shows the number of bits set in the input. The yellow and black lines just below the green line is the actual input values. In Figure 3.2 this is the boxes labeled i_1 to i_4, except the CA in Figure 3.3 is 8×8, so there are 8 input bits. Yellow means the bit is set to 1, while black means it is 0. The red line shows the number of bits set in the output, and the yellow and black figure below is the actual output. Looking at Figure 3.2 the output cells are marked o_1 to o_4. Unlike the input it is easy to see that the output is made up of 8 bits, as the output does not have all the bits set to the same value in an iteration. If we look at the number of bits set in the output, which is what the DFT is run on, we see that the output has half the frequency of the input.

3.2 Genome and Mapping

To be able to evolve a CA, a representation of the genotype along with mutation and crossover operators have to be defined. In addition there have to be a mapping from the genotype to the phenotype, an 8×8 CA. Three different genotypes are defined.

The first genotype $gen_{1,k}$ is the simplest. It consists of a k-tuple $l \in L^k$ with

k LUTs. The second genotype $gen_{2,k}$ builds on the first one. It can be written as a tuple (l, i) where l is defined as above and $i \in \{0,1\}^8$ is a bit string used to specify the configuration values for the CA. The third genotype $gen_{3,k}$ is a further extension, and can be written as (l, i, P) where P is a set of productions for a D0L-system. There are exactly k productions in P, with predecessors 1 through k.

Remember that a CA is parameterized by a tuple (M, C) where M is a function mapping cell position to LUT and C is a bit string used to set the configuration values of the CA. Mapping the genotypes to such a parameterized CA is achieved with the following functions

$$\begin{aligned} f_1(l) &= (\mu(l), 0) \\ f_2(l, i) &= (\mu(l), i) \\ f_3(l, i, P) &= (\rho(l, P), i) \end{aligned}$$

where μ is a function taking a set of LUTs and returns a function taking cell position and returning a LUT and ρ is a function taking both a set of LUTs and a set of productions, returning a function mapping cell position to LUT.

We define three different μ functions for use with uniform, non-uniform and what we term semi-uniform CAs. A semi-uniform CA is a CA where some cells have the same LUT, but not all. We will use the notation l_i to mean the ith LUT in l, where l has been given a deterministic ordering. The first function

$$\mu_{uni}(l) = (x, y) \mapsto l_0$$

maps all positions to the same LUT, essentially creating a uniform CA. This function should be used together with a genome with $k = 1$. The second function

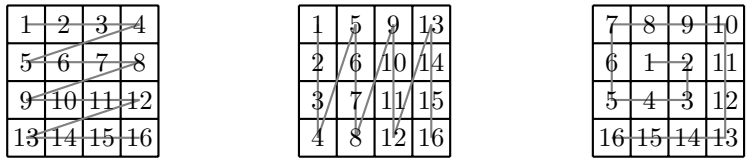
$$\mu_{nu}(l) = (x, y) \mapsto l_{x+y \cdot 8}$$

maps each position to a different LUT. This creates a non-uniform CA. When using this function the genome must have $k = 8 \cdot 8$. The third function

$$\mu_{su}(l) = (x, y) \mapsto l_{x \bmod k}$$

creates a semi-uniform CA, where all cells with the same x value have the same lookup table. If $k = 1$ the CA is uniform.

We also define four different $\rho(l, P)$ functions for use with $gen_{3,k}$. The first three functions are very similar. They first evaluate the L-system (V, ω, P) where $V = \{1, \ldots, k\}$, $\omega = 1$ and P is given by the genome. The L-system is



(a) Horizontal mapping (b) Vertical mapping (c) Spiral mapping

Figure 3.4: Mappings from cell position to string position

evaluated for 64 iterations, or until the produced string is at least 64 symbols long. If the produced string is shorter than 64 symbols it is padded with 1s. The string is then mapped to the CA by using a mapping from cell position to string position. The three different mappings used for this are shown in Figure 3.4. Once the string position is found, the symbol at that position is mapped to a LUT. This is done by assigning one of the k LUTs in l to each of the k symbols in V. This gives us the three functions ρ_v, ρ_h and ρ_s which use the vertical mapping, horizontal mapping and spiral mapping respectively.

The fourth ρ function uses turtle graphics to lay out the CA. We will call this function ρ_t. It uses a D0L-system (V, ω, P) with $V = \{1, \ldots, k, +, -, [,]\}$ and $\omega = 1$. The set of productions P is taken from the genome. The system is run for 4 iterations, or until the string is at least 16384 characters long. This string is used as instructions for a turtle. The $+$, $-$, [and] symbols have the same meaning as in Section 2.4, and the turning angle δ is set to 90°. The symbols 1 to k make the turtle set the LUT of the cell it is currently on to the LUT corresponding to the symbol, and move one step forward.

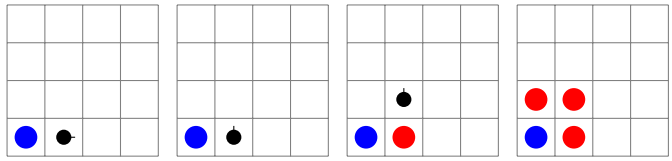
Let us look at an example of the ρ_t mapping. We use a 4x4 CA with the productions

$$P = \{a \to a + b, b \to b\}$$

where we use a and b to represent cell type 1 and 2 respectively. If we run this system for three iterations we get

$$\begin{aligned} 0 &: a \\ 1 &: a + b \\ 2 &: a + b + b \\ 3 &: a + b + b + b \end{aligned}$$

The final string is $a+b+b+b$. Figure 3.5 shows the three first steps in the turtle interpretation of this string. The black circle with the line marks the turtle's



(a) $\underline{a} + b + b + b$ (b) $a\underline{+}b + b + b$ (c) $a + \underline{b} + b + b$ (d) Final CA

Figure 3.5: Turtle interpretation of $a + b + b + b$

$$abc[+b] - a \rightarrow aba[b - c][+b] - a$$

(a) Mutating a single symbol

$$abc[+b] - a \rightarrow abca[b - c] - a$$

(b) Mutating a branch

Figure 3.6: Mutation of productions

position and heading. It starts with a, resulting in the start cell being set to cell type 1 (blue), and the turtle taking one step forward. Next it encounter $+$, which makes it turn 90° left. After that it finds b, so it sets the current cell to cell type 2 (red) and moves on step forward. Figure 3.5d shows the result after interpreting the entire string. Cells that have not received a cell type when the run is over are automatically given cell type 1.

Finally we have to define the mutation and crossover operators. The mutation operator randomly selects one of the elements in the genotype. For gen_1 this will always be l, for gen_2 its either l or i and for gen_3 it is l, i or P. Once one of those is selected, they are mutated. After carrying out the mutation there is a 50% chance it will run the mutation operator again. This gives us a 50% chance of a single mutation happening, 25% chance of two mutations, 12.5% chance of three mutations and so on. l is mutated by picking a random bit in a random LUT and flipping it. i is mutated by picking a random bit and flipping it. P is mutated by picking a random production in P, then picking a random symbol in the successor of this production. If the symbol is $+$, $-$ or one of the symbol 1 through k, it is replaced by a random string that is one to five symbols long. However, if it is [or] the whole branch is replaced by a random string that is one to five symbols long. Figure 3.6 shows an example of these two

operations. This mutation operator is similar to the mutation operator used by Gabriela Ochoa [18].

The crossover operator is similar to a one-point crossover operator, except it uses two one-point crossovers. One for the i part and one for the l and P parts of the genotype. The i part of the genotype is crossed by selecting a crossover point $p \in \{1, \ldots, 8\}$ and taking the first p bits from the first parent and the last $8 - p$ bits from the second parent. The l and P parts are crossed by selecting a crossover point $q \in \{1, \ldots, k\}$ and taking the first q LUTs and productions from the first parent and then taking the last $k - q$ LUTs and productions from the second parent.

3.3 EA

Two different evolutionary algorithms are used. One is the 1+4 ES described in Section 2.2. The other is a standard genetic algorithm with binary tournament selection. Population size, mutation rate and crossover rate are specified in the experiments chapter.

Chapter 4

Experiments

This chapter presents a series of experiments that tries to answer questions related to the evolvability and general behavior of the system described in Chapter 3. To enhance readability the chapter is divided into four sections. The first section contains experiments concerned with the evolvability of uniform and non-uniform CAs. The second section looks at semi-uniform CAs and their evolvability compared to uniform and non-uniform CAs. Of special interest is the performance when we use L-systems as a developmental step. While the first two sections use the 1+4 ES to evolve CAs, the third section experiments with a genetic algorithm. Finally the fourth section applies the system to the majority problem, and compares how various fitness functions and output interpretations affect the system's performance.

4.1 Uniform vs. Non-uniform

One of the most obvious ways to partition the set of CAs are into uniform and non-uniform CAs. In uniform CAs all cell have the same function. In our case these functions are encoded by lookup-tables. Non-uniform CAs on the other hand do not have the same function in all cells. Both classes have their benefits and drawbacks. Uniform CAs are easy to represent, to define the entire CA only a single LUT is needed. To define a non-uniform CA you need one LUT for every cell. This leads to a representation that grows linearly with the number of CA cells. The simplified representation of uniform CAs does however come at a price. Only a minuscule fraction of all CAs are uniform CAs, while

the non-uniform representation can actually be used to represent any CA, even uniform ones. In this experiment we look at the evolvability of these two classes of CAs and compare them.

4.1.1 Experiment 1: Input replication and inversion

Input replication is a very simple problem, requiring only the cells to copy the value of the cell to the left of them. This could easily be solved by both uniform and non-uniform CAs. Inversion of the input is more complicated, and it is not immediately obvious how a uniform CA could do this, or even if it is at all possible. A non-uniform CA on the other hand requires only that the last column of cells invert the value of the cell to the left of them, while the other columns simple pass them through like before. We will try to evolve both uniform and non-uniform CAs that solve the two problems and compare their performance.

The CAs are represented by the gen_1 genome representation. $gen_{1,k}$ consists of a tuple with k LUTs. To represent uniform CAs we use $gen_{1,1}$ and the mapping μ_{uni} which maps every cell position to the only LUT in the genome. Non-uniform CAs are represented with $gen_{1,64}$ and μ_{nu}, where μ_{nu} maps a cell position to a distinct LUT,

Fitness functions

This experiment uses two different fitness functions, f_{thru} and f_{inv}.

f_{thru} is calculated by running the CA for 128 iterations. The first 16 output values are discarded to allow the CA time to settle. This leaves 112 output values, making it possible to see if the CA has a steady output once it has settled. The fitness is calculated by counting the number of bits equal to the input in those 112 output values. This is repeated for 16 different input values, making the maximum fitness $16 \cdot 112 \cdot 8 = 14336$.

f_{inv} is defined similarly to f_{thru}, except we count the number of bits in the output not equal to the input. It could also be defined as $f_{inv} = 14336 - f_{thru}$.

Results

Figure 4.1 shows how well the uniform and non-uniform CAs evolve using 1+4 when f_{thru} is used as the fitness function. We see that the search finds an optimal rule for the uniform CA in less than 50 generations. The non-uniform CA is however unable to make much progress.

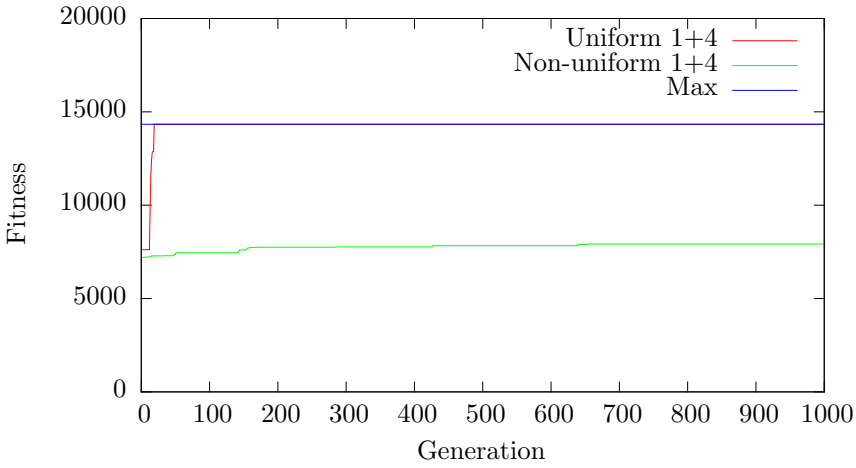


Figure 4.1: Evolution with fitness function f_{thru}

Figure 4.2 shows the results with f_{inv}. We see that the uniform CA has a much harder time compared to earlier. The non-uniform seems to have a slightly slower start too, but ends up at around the same value as for f_{thru}.

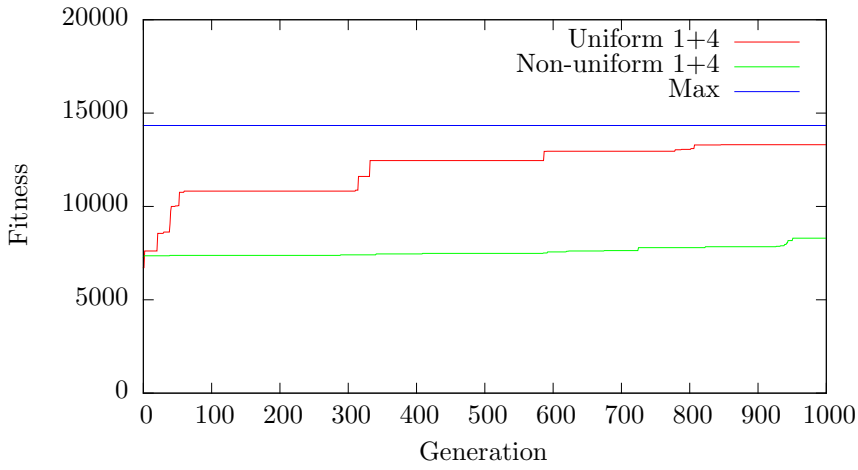


Figure 4.2: Evolution with fitness function f_{inv}

4.1.2 Experiment 2: DFT replication and inversion

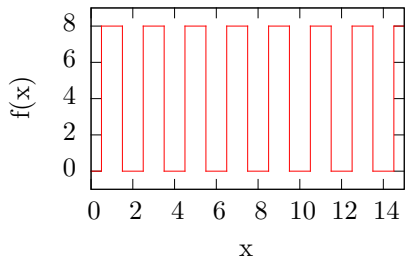
This experiment tries to determine evolvability of CAs when using a DFT to interpret the output instead of a literal interpretation as in the previous experiment. Square waves with four different frequencies are used as input. The frequencies used are 1, 2, 4 and 8. The fitness will be calculated by comparing the main frequency in the DFT of the output to the main frequency of the input square wave. The spectrum value $F(0)$ is simply a sum of $f(x)$ and will be ignored when finding the main frequency. Figure 4.3 shows the square waves with the highest and lowest frequency as well as their DFT spectrums. We see that the square wave with frequency of 1 has main frequency of 1, and with $f = 8$ the main frequency is also 8. We will define two different fitness functions, one where the goal is to have the same main frequency as the input, and one where the frequency relation is inverted, i.e. input with frequency 1 should have output with main frequency 8, input with frequency 2 should have output with main frequency 4 and so on.

Producing the same main frequency on the output as on the input can be achieved by simply replicating the input. As we saw in the previous experiment input replication is a trivial problem, and a uniform CA solving it was found in less than 50 generations. Frequency inversion on the other hand is likely harder, and it is not obvious how a CA would solve it.

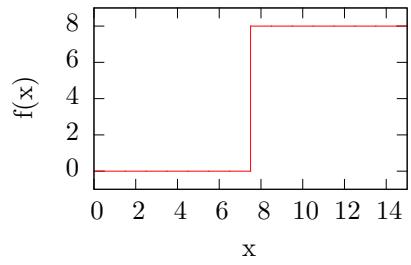
Fitness functions

Two fitness functions are defined. The first one is $f_{fft-thru}$ which is calculated by running the CA for 127 iterations, throwing away the first 16 output values and then running a sliding DFT with window size 16 on the rest of the output. This produces 96 DFT spectrums, which should tell us if the CA behaves the same over time. All those spectrums are then inspected to see if the main frequency is the same as in the input. The number of spectrums with the correct main frequency is the fitness value. With 4 different input waves this gives us a maximum fitness of $4 \cdot 96 = 384$.

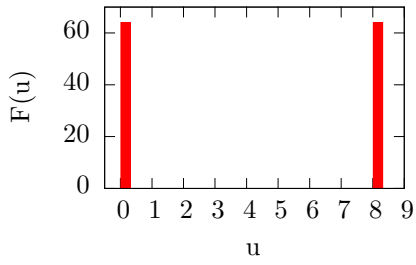
The second fitness function $f_{fft-inv}$ is largely the same, except it counts the number of spectrums with opposite main frequencies.



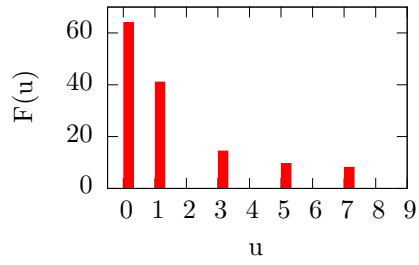
(a) $f = 8$



(b) $f = 1$



(c) DFT spectrum for $f = 8$



(d) DFT spectrum for $f = 1$

Figure 4.3: Input square waves and DFT spectrums

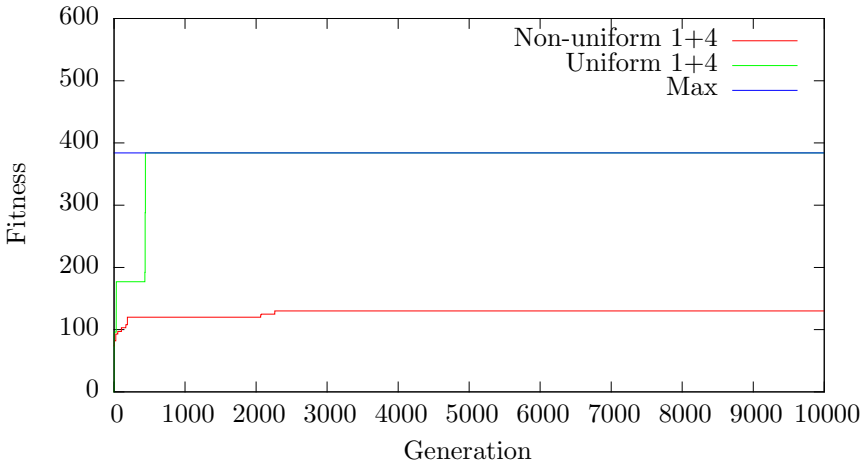


Figure 4.4: Evolution with $f_{fft-thru}$

Results

Figure 4.4 shows the performance of a uniform and non-uniform CA using $f_{fft-thru}$. We see that the uniform CA is able to find a perfect solution quickly. The non-uniform CA however is unable to find a good solution.

Figure 4.5 shows the performance when $f_{fft-inv}$ is used. We see that the non-uniform CA evolves pretty much the same, but the uniform CA gets a much lower fitness.

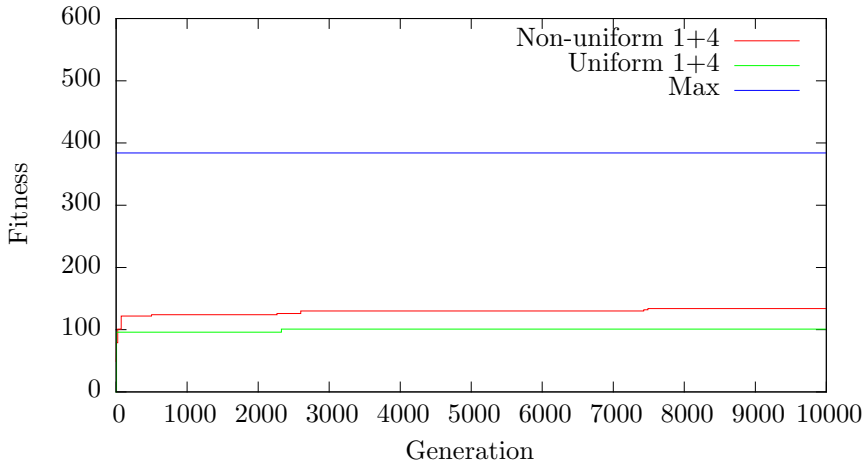


Figure 4.5: Evolution with $f_{fft-inv}$

4.1.3 Experiment 3: Configuration values

In experiment 1 and 2 the configuration values of the CA are all set to 0s. If we look at Figure 3.2 this means all the cells on the right side of the CA have the same inputs when the CA is started, because the cells all have their initial state set to 0. The cells in the middle of the CA will also have the same inputs because they are only connected to other cells which are set to 0. If in addition the input values are the same, for example oscillating between all 1s and 0s, the cells on the left also have the same inputs. This makes it impossible for the cells in a uniform CA to have different behavior in a column, because all the cells in a column have the same inputs. This experiment looks at what happens when the top most configuration value is set to 1, while leaving the rest set to 0. This will allow the top-most cell on the right to behave differently, and this could ripple throughout the CA, creating more complex behavior.

Fitness function

$f_{fft-inv}$ from experiment 2 will be used as the fitness function.

Results

Figure 4.6 shows the difference between the two different configuration values when using $f_{fft-inv}$ and a uniform CA. We see that using configuration values with not all 0s results in a much improved fitness.

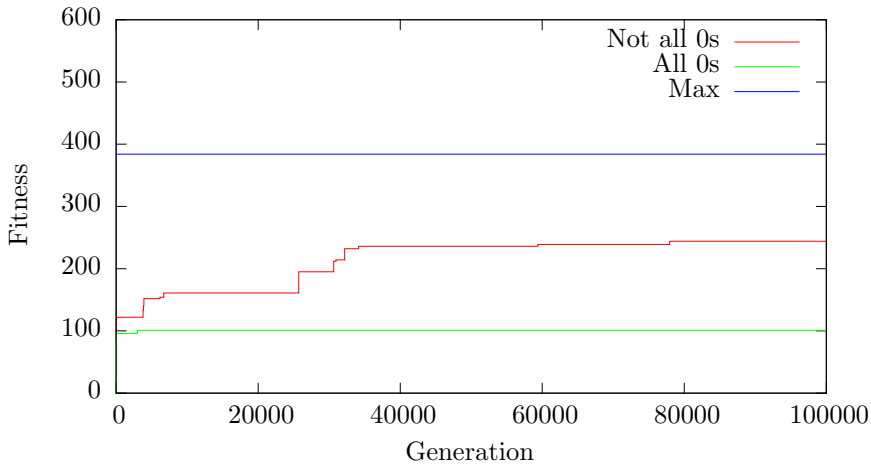


Figure 4.6: Behavior with different configuration values

4.1.4 Experiment 4: Dunn Index

This experiment introduces three new fitness functions. They are all based on the Dunn Index (DI), which is used to evaluate how good a data-set is clustered. Tighter clusters with larger distance between them have a higher Dunn Index. It is defined as

$$DI = \min_{1 \leq i \leq m} \left\{ \min_{1 \leq j \leq m, j \neq i} \left\{ \frac{\delta(C_i, C_j)}{\max_{1 \leq k \leq m} \Delta_k} \right\} \right\}$$

where m is the number or clusters, $\delta(C_i, C_j)$ is the distance between the centers of the clusters i and j and Δ_k is the maximum distance between two points in cluster k. The cluster center is defined as the average of its values. The clusters consists of the weighted average frequency of the output with one cluster for each of the four input frequencies. If the output has a consistent frequency the cluster will have a small size.

Encouraged by the results in experiment 3 this experiment tries the gen_2 genome representation in addition to the gen_1 representation used in the earlier experiments. Remember that the $gen_{2,k}$ representation consists of a k-tuple of LUTs just like $gen_{1,k}$ in addition to an 8-bit string containing the configuration values for the CA.

Fitness functions

The Dunn Index can be used as a fitness function directly. fit_{DI} will be used as the name for this fitness function. However, if the Dunn Index were to be used alone it could result in a non-linear ordering of the output frequencies when ordered by their input frequencies. Therefore two other fitness functions are defined that use linear regression along with the Dunn Index. Linear regression calculates the α and β that makes $y = \alpha + \beta x$ best fit a set of points. In our case the y is the cluster center and the x is the input number. The highest frequency square wave is defined as input 1, the second highest as input 2 and so on. We are mainly interested in the β. The two fitness functions can be defined as $fit_+ = \beta DI$ and $fit_- = -\beta DI$.

A CA that pass through the input unmodified will have a $\beta < 0$. Since the outputs will also be well clustered (the input is already clustered) this will result in a good fitness if we used fit_-. In this regard fit_- is similar to fit_{thru} in that a good solution is trivial. It is harder to find a CA that has a good fitness with fit_+ since a frequency inversion is necessary. As we saw in experiment 2 this is a significantly harder problem.

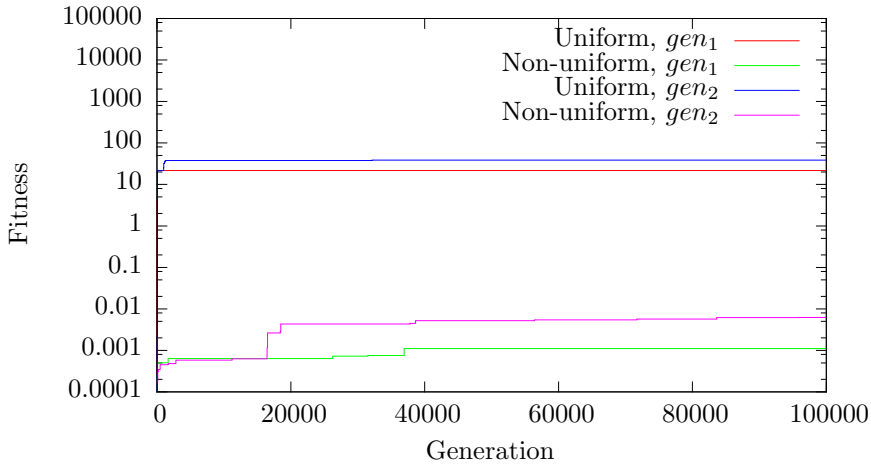


Figure 4.7: Fitness with fit_{-}

Results

Figure 4.7, Figure 4.8 and Figure 4.9 shows the fitness of the different fitness functions, as well as the difference between evolving the configuration values and setting them to 0. We see that evolving the configuration values results in a better fitness in all cases, for uniform CAs using fit_{+} the improvement is staggering. We also see that uniform CAs performs better than non-uniform CAs.

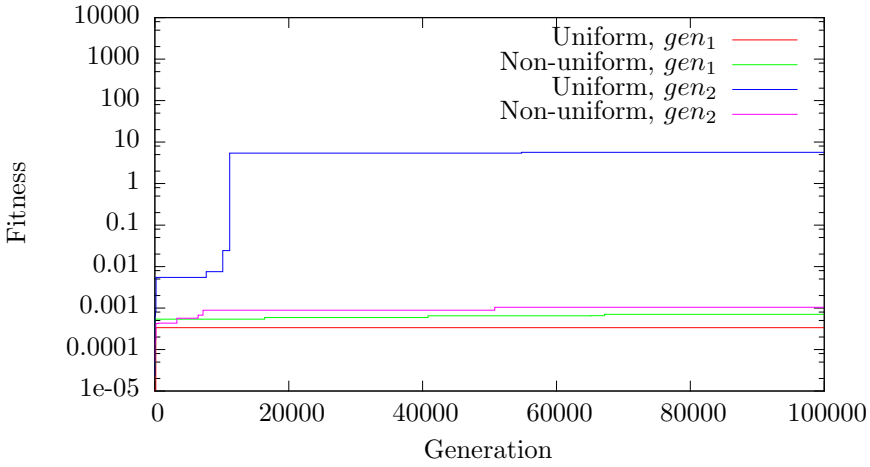


Figure 4.8: Fitness with fit_+

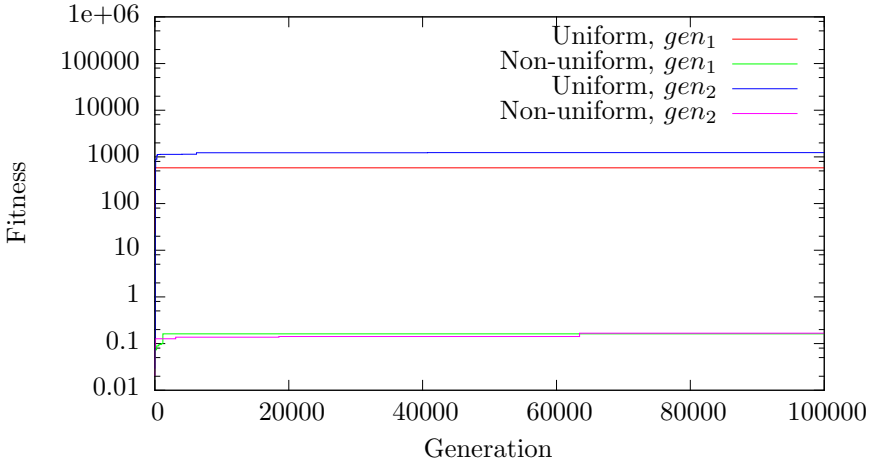


Figure 4.9: Fitness with fit_{DI}

4.1.5 Discussion

The experiments show that uniform CAs are much easier to evolve than non-uniform CAs. An 8×8 non-uniform CA has a genome containing 64 32bit LUTs. This results in a genome space that contains $(2^{32})^{64} = 2^{2048}$ genomes. This is enormous, and it makes sense that such a vast space is harder to search than the smaller genome space for uniform CAs that contains only 2^{32} genomes.

With our representation the set of uniform CAs is a proper subset of the set of non-uniform CAs. That means that when we use non-uniform CAs we can find at least as good a solution as when using uniform CAs, but since the set of non-uniform CAs is so much larger than the set of uniform CAs there is a good possibility an even better solution exists. We saw that a uniform CA just passing through the input was easy to evolve, however the EA was unable to find a uniform CA inverting the input. If we look at the fitness for the non-uniform CAs evolved they are almost the same for the two problems, suggesting that with non-uniform CAs the two problems have similar difficulty.

The experiments also showed that evolving the configuration values along with the LUTs resulted in a better fitness in all cases. This seems to be especially beneficial for uniform CAs. This could be because the non-uniform cell inputs allows it to somehow differentiate the cell behavior based on position, allowing a uniform CA to be somewhat non-uniform.

4.2 Semi-uniform CAs

Earlier experiments failed to find a good uniform CA with fitness function fit_+. One reason for this could be that no uniform CA able to solve it exists. Non-uniform CAs come with a much larger number of possible CAs, making it more likely there exists a CA that solves a certain problem, but as we saw the genome space is harder to search. Instead of using a different LUT in every cell like a non-uniform CA, the LUT could be shared by more than one cell, but not all. E.g. we could use two different LUTs where half the cells use the first and the other half the second. This would change the size of the genome space from 2^{32} to 2^{64}, significantly increasing the number of possible CAs, while hopefully keeping the genome space small enough to easily search. Being a blend of uniform and non-uniform CAs we will call them semi-uniform CAs. This section contains experiments that look at the evolvability of such semi-uniform CAs. Cells with the same LUT will be said to have the same cell type, and the k in the genome representation will be the number of different cell types in the CA.

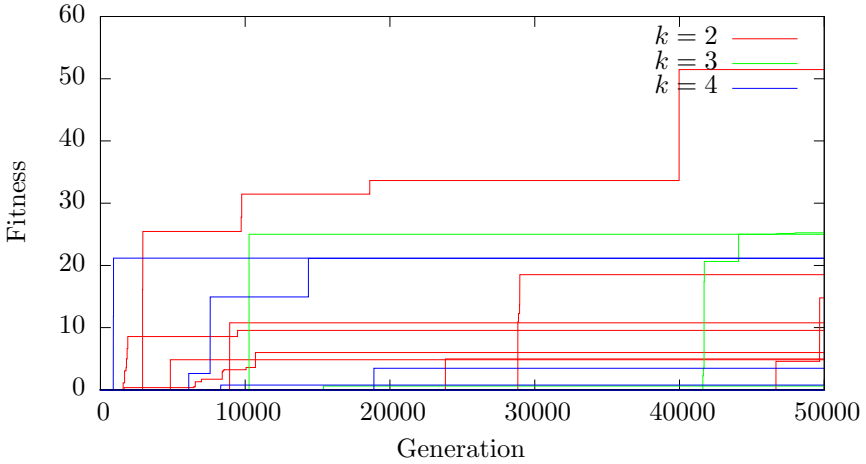


Figure 4.10: Fitness with fit_+ and $gen_{2,k}$, 10 runs for every k

4.2.1 Experiment 5: Constant mapping

The first experiment uses $gen_{2,k}$ with $k \in \{2, 3, 4\}$ and the μ_{su} mapping function. The μ_{su} is a constant mapping where all cells with the same x value have the same cell type.

Fitness function

The fitness function used is fit_+ from experiment 4.

Results

Figure 4.10 shows the results from 10 different runs of a 1+4 ES with 2 to 4 cell types. We see that with 3 and 4 cell types we get a maximum fitness around 20. One of the solutions for 2 cell types manage to a fitness over 50, more than twice that of any other solution. We also observe that there is a large spread in how far a 1+4 ES has come in 50000 generations.

4.2.2 Experiment 6: L-systems

Unlike the previous experiment which used a constant, implicit mapping from cell position to cell type, this experiment will look at a way to make this mapping a part of the genome without scaling with the CA size.

We use the $gen_{3,k}$ genome representation with $k \in \{2, 3, 4\}$. $gen_{3,k}$, like $gen_{2,k}$, contains a k-tuple of LUTs and configuration values, but also contains a set of productions P for an L-system. This allows the cell position to cell type mapping to be evolved along with the LUTs and configuration values. We will try four mapping functions, ρ_v, ρ_h, ρ_s and ρ_t. These are explained in detail in Section 3.2 and Figure 3.4.

Fitness function

We use the same fitness function as the previous experiment, fit_+.

Results

Figure 4.11 shows the results when using a vertical mapping ρ_v, Figure 4.12 shows the same for the horizontal mapping ρ_h, Figure 4.13 shows the results when using a spiral mapping ρ_s and Figure 4.14 shows the results when using the turtle mapping ρ_t. We see that the horizontal mapping has more results in the 10-20 bracket, but the vertical mapping has a better maximum fitness. The spiral mapping does poorly, with only a few solutions getting a fitness above 0. The turtle mapping gets the best results, finding 5 solutions with fitness 25 or better.

4.2.3 Discussion

The experiments show that semi-uniform CAs perform very well. They are easily evolvable and in some cases the 1+4 ES is able to find much better semi-uniform CAs than uniform CAs. Especially interesting are the good results when using an L-system and turtle graphics to lay out the CA. Compared to non-uniform CAs they are much more evolvable, and a great way to get some of their power while maintaining the evolvability of uniform CAs.

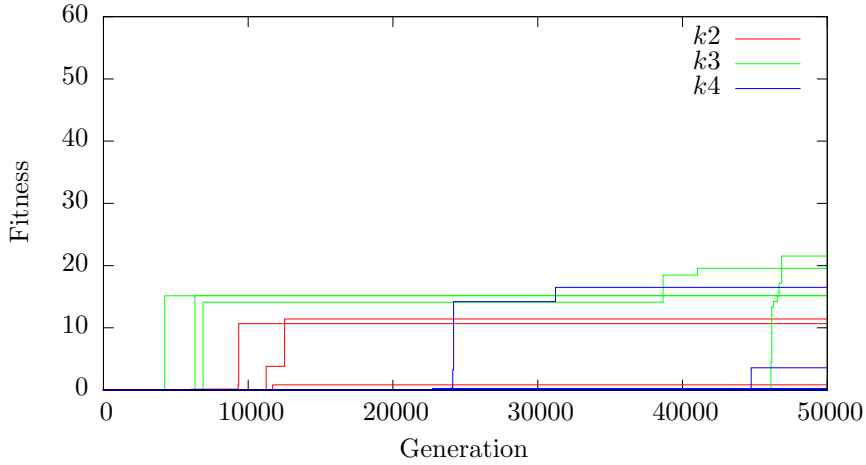


Figure 4.11: Fitness with fit_+, $gen_{3,k}$ and ρ_v, 10 runs for every k

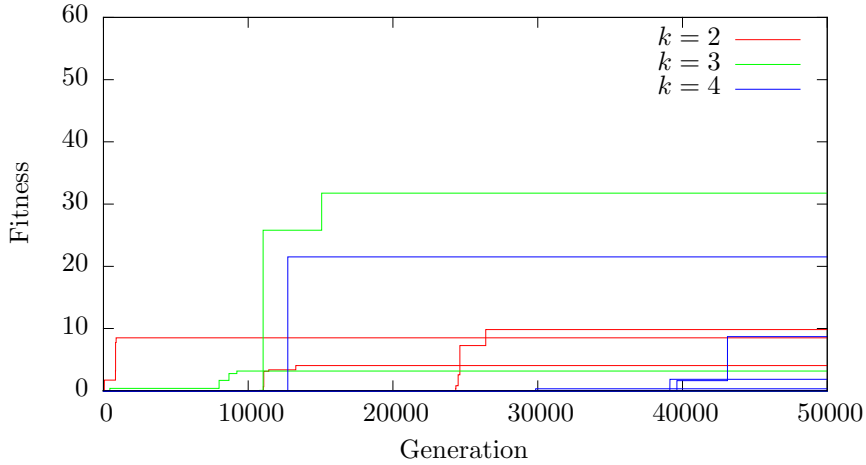


Figure 4.12: Fitness with fit_+, $gen_{3,k}$ and ρ_h, 10 runs for every k

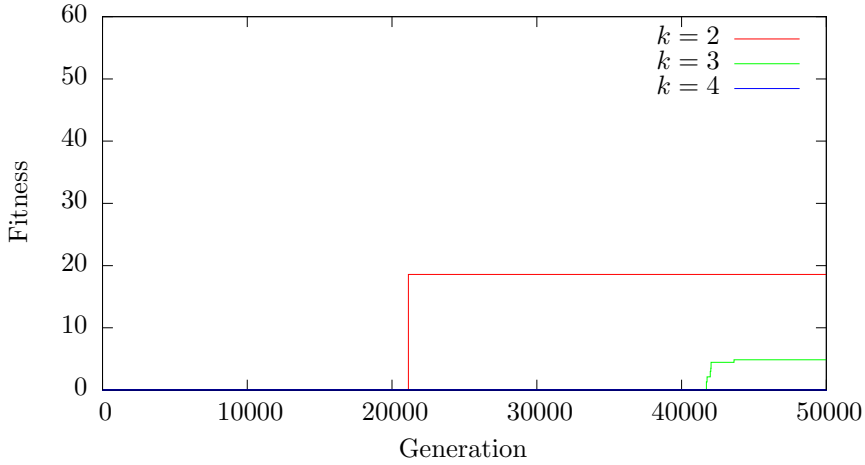


Figure 4.13: Fitness with fit_+, $gen_{3,k}$ and ρ_s, 10 runs for every k

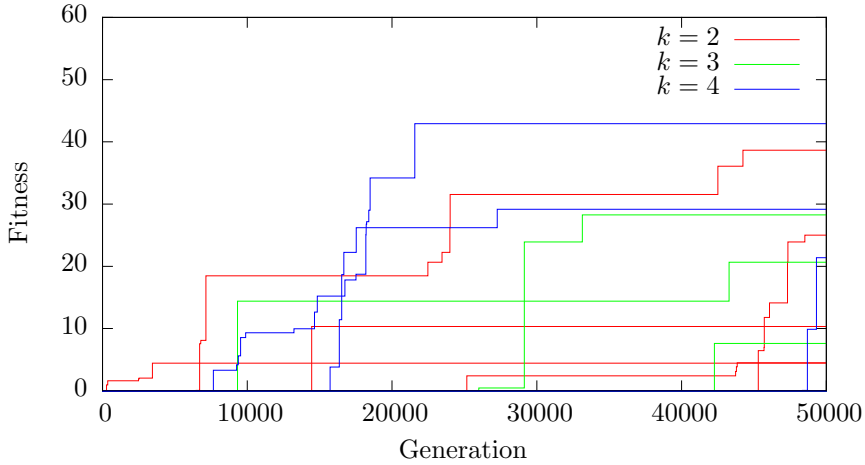


Figure 4.14: Fitness with fit_+, $gen_{3,k}$ and ρ_t, 10 runs for every k

4.3 Genetic Algorithm

The 1+4 ES used in earlier experiments has a performance that is very dependent on the randomly selected start genome. As experiment 5 and 6 showed, 10 runs with different start genomes result in big differences in the fitness values. Using a genetic algorithm instead of 1+4 could help decrease the variance between different runs. When random populations are used there will always be some differences, but the larger population of the GA makes it more likely the start populations have similar characteristics compared to 1+4. A GA also adds the possibility of two-parent genetic operators such as crossover in addition to the single-parent mutation operator used in 1+4.

4.3.1 Experiment 7: GA with fit_+

This experiment looks at the performance of a GA compared to 1+4. The GA uses binary tournament selection, a population size of 64, a mutation rate of 0.2 and a crossover rate of 0.5. It uses the $gen_{3,k}$ genome representation along with the ρ_t mapping function as this had the best results in experiment 6.

Fitness function

This experiment uses the fit_+ fitness function from experiment 4.

Results

Figure 4.15 shows the maximum fitness from the GA run. If we compare it to Figure 4.14 we see that the GA has a much worse fitness.

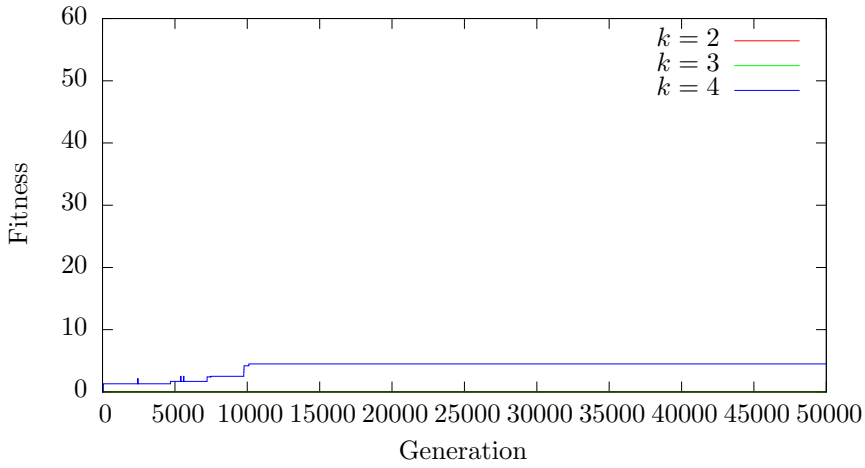


Figure 4.15: Fitness with GA using fit_{+}, $gen_{3,k}$ and ρ_t

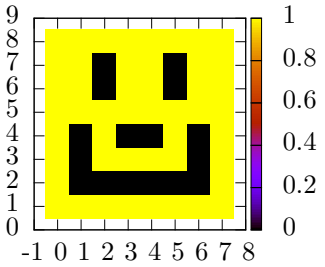


Figure 4.16: The desired smiley

4.3.2 Experiment 8: GA Testing

Experiment 7 showed that just plugging in a GA along with the fit_+ did not work. The reason for this could be one of many. It could be a bug in the GA implementation, a bad fitness function that does not play well with the GA or a search space that is hard to traverse, just to mention a few. This experiment tries several simpler fitness functions to determine if the fitness function is to blame. We use the $gen_{3,k}$ genome representation with $k \in \{2, 3, 4\}$ along with the ρ_t mapping.

Fitness functions

The first fitness function is very simple: $fit_{gl} = gl$, where gl is the genome length. The genome will have four cell types, and therefore also four productions. Each production can grow up to 128 symbols. This means the maximum genome size is 512 characters.

The second fitness function, which we will call fit_{ls-img}, counts the number of correct cells in the CA the L-system produces compared to a goal image. Two different goal images are used, one is a checkerboard pattern and another is the smiley in Figure 4.16.

The third fitness function, dubbed fit_{ca-img}, is similar to the second one. The difference is this time we run the CA for 8 iterations and compare the 8×8 output bit array to a goal image and count the number of correct pixels. The same goal images are used.

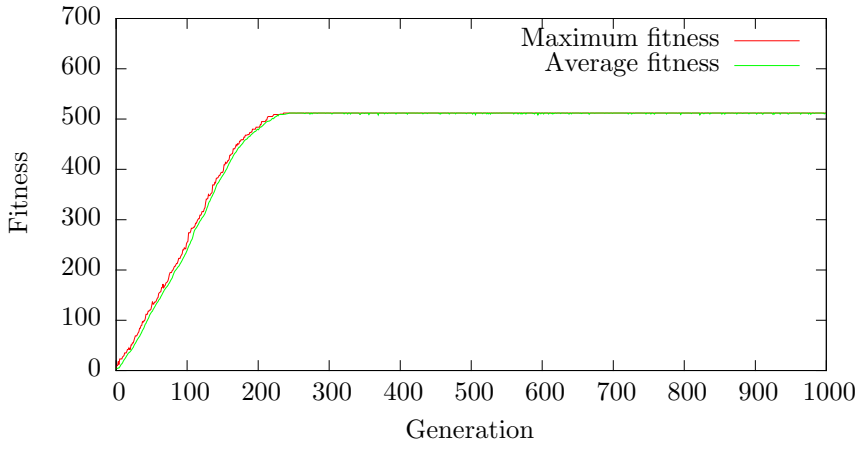


Figure 4.17: Maximum and average fitness with fit_{gl}

Results

Figure 4.17 shows the maximum and average fitness for the first 1000 generations using fit_{gl}. We see that we are able to reach the maximum genome size in less than 300 generations. This shows that the GA is able to search a simple fitness space effectively. We also see that the average fitness in the population is very close to the maximum fitness.

Figure 4.18 shows the maximum fitness with fit_{ls-img}. The experiment was run using $gen_{3,k}$ and $k \in \{2, 3, 4\}$, however for clarity the figure shows only the best for each goal. For the checkerboard this was $k = 3$ and for the smiley $k = 4$. We see that the GA is able to find a perfect genome for the checkerboard goal. In the smiley case the GA is not that successful. It ends up at a maximum fitness of 58. Figure 4.21a shows the resulting image.

Figure 4.19 shows the maximum fitness with fit_{ca-img} when using the smiley as the goal image. We see it never reaches the optimal fitness of 64. The best genome ends up with a fitness of 62. Figure 4.21b shows the image this genome produces. We see that with the exception of the monobrow this is equal to the goal image in Figure 4.16.

Figure 4.20 shows the maximum fitness with the checkerboard goal. As we can see a maximum fitness is reached in less than 200 generations with both 2, 3 and 4 cell types.

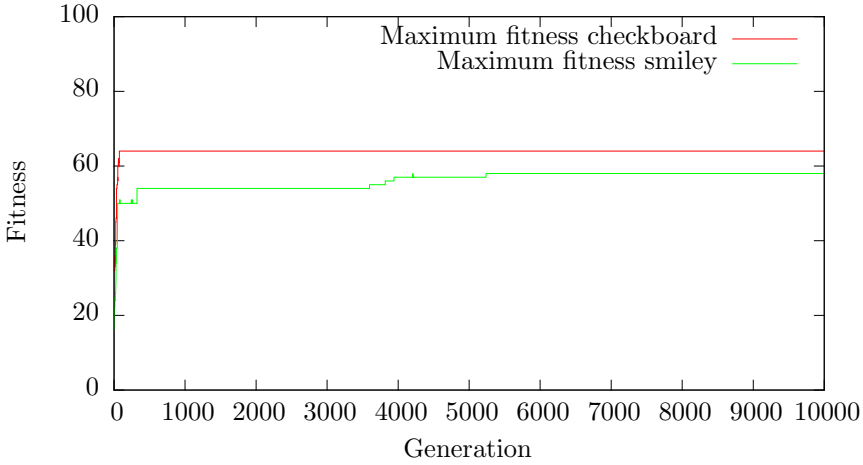


Figure 4.18: Maximum fitness with fit_{ls-img}

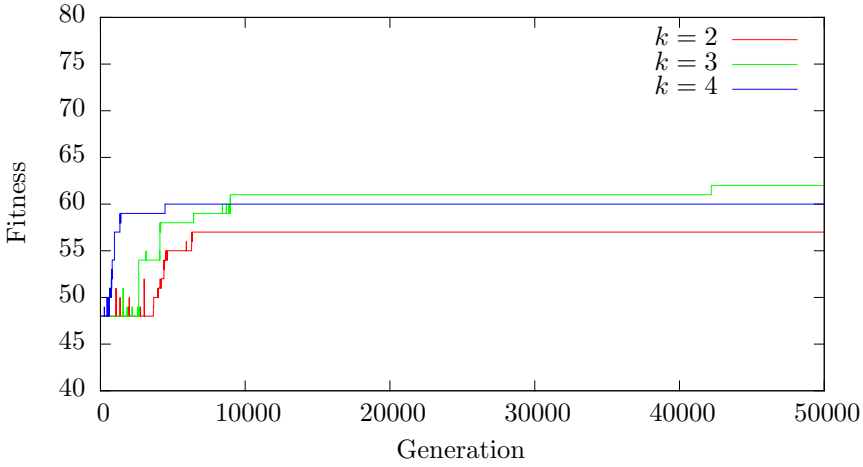


Figure 4.19: Maximum fitness with smiley goal using fit_{ca-img}

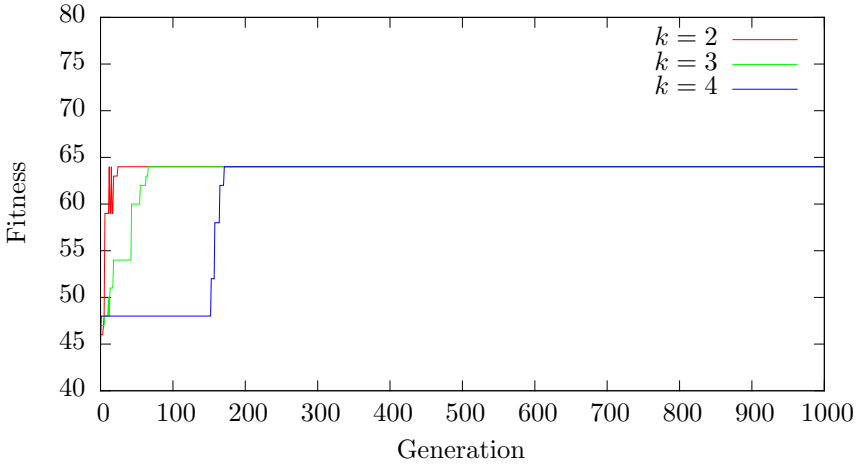
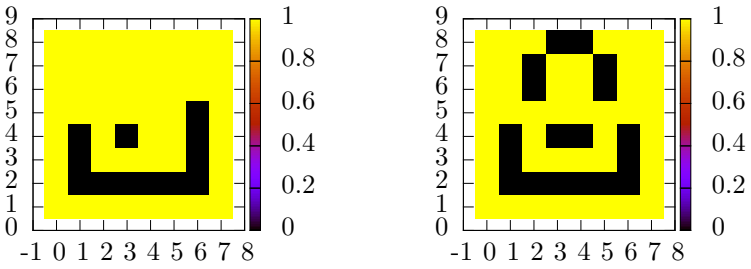


Figure 4.20: Maximum fitness with checkerboard goal using fit_{ca-img}



(a) fit_{ls-img} (b) fit_{ca-img}

Figure 4.21: Best smileys evolved

4.3.3 Experiment 9: Even or odd number of bits

Experiment 8 showed that the GA works great on simpler fitness function. The problem thus seems to be the fitness function used in experiment 7, fit_+. To calculate fit_+ the CA is run with four different square waves. Because of only four possible input values the genome-space will naturally be divided into plateaus where the genomes on the plateaus gives the correct value for 0, 1, 2, 3 or 4 of the inputs. To move between plateaus the GA has to randomly search until it finds the edge and it can move to the other plateau. This experiment will try a different problem, where a more gradual evolution is possible. The GA will try to evolve a CA that is able to differentiate between inputs depending on whether they have an even or odd number of bits set. The experiment uses $gen_{3,k}$, $k \in \{2, 3, 4\}$ and ρ_t.

Fitness function

The fitness function fit_{eo} is calculated by running the CA with a constant input for 32 iterations. This gives the CA 16 iterations to settle, before running the last 16 output values through a DFT. Then the average frequency $\overline{F}$ of the spectrum is found. The input either has an even or an odd number of bits set, and $\overline{F}$ is added to the even or odd cluster. Lastly the Dunn Index is used to determine how well clustered the values are. To make the fitness function faster to calculate it will only test every 11th input from 0 to 255. This comes out to a total of 24 inputs, 12 with an even number of bits and 12 with an odd number.

Results

Figure 4.22 shows the maximum fitness of the population and Figure 4.23 shows the frequency averages on all 256 inputs when using the best solution found. The inputs in Figure 4.23 are numbered by converting the 8-bit input vector to a number by simply interpreting it as an 8-bit unsigned integer. As we can see the GA is unable to find a good solution. The best solution is able to cluster the even inputs slightly tighter than the odd inputs, but it is still very scattered.

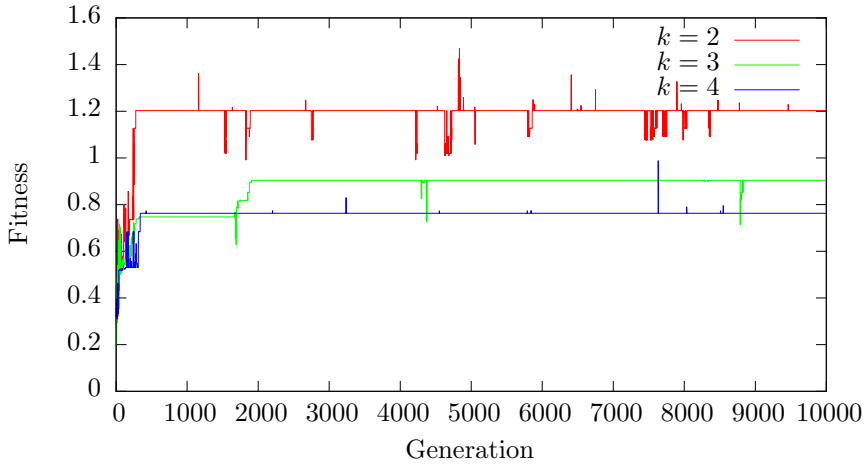


Figure 4.22: Maximum fitness with fit_{eo}

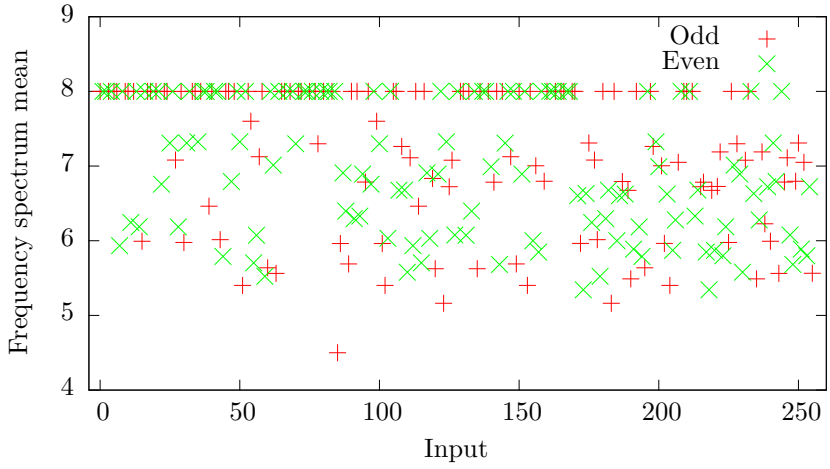


Figure 4.23: Average output frequency of all input values for the best genome found with fit_{eo}

4.3.4 Discussion

Unlike the 1+4 ES, the GA seems to demand a "nice" fitness function. It was unable to find good solutions with fit_+, while 1+4 found several. This could be, as hypothesized above, because of the lack of resolution in fit_+. We saw that the GA performed very well with more well behaved fitness functions where it was possible to improve the fitness in small increments. This is worth having in mind when we define the fitness functions for the remaining experiments.

4.4 The Majority Problem

The GA was not able to find a perfect solution in experiment 9. This section contains experiment that tries to use the GA to find a CA able to solve a simpler problem; the majority problem. We would like the CA to be able to decide if the input consists of mostly 1s or 0s. We will try two different ways of interpreting the output, one based on DFT and another simpler mapping. The GA will have a crossover rate of 0.5 and mutation rate of 0.5. Because our CA is 8×8, the input has 8 bits. This means inputs with 4 bits set have neither the minority or majority of bits set, therefore 70 of the 256 possible inputs have undefined output values. All the experiments in this section, with the exception of the last one, use the $gen_{3,k}$ genome representation with $k \in \{2, 3, 4\}$ along with the ρ_t turtle graphics mapping.

4.4.1 Experiment 10: DFT and Dunn Index

In this experiment we will use a DFT to interpret the output and a fitness function based on the Dunn Index.

Fitness function

The fitness function is very similar to the fit_{eo} function used in experiment 9, we just redefine the two input groups to inputs with minority of bits set and majority of bits set. The CA is run for 32 iterations, the 16 last output values are run through a DFT and the average frequency $\overline{F}$ is added to the correct cluster. The clusters are then evaluated with the Dunn Index to find how well they are clustered. We will call this fitness function fit_{m-di}.

Results

Figure 4.24 shows the maximum fitness. As we can see a good result is only obtained when 2 cell types are used. We also see that the evolution seems to jump from a bad to a good fitness in one generation, suggesting that few, if any, solutions are able to partially solve the problem. However, this might not necessarily be true and highlights one of the problems with the Dunn index. Remember that it is defined as

$$DI = \min_{1 \leq i \leq m} \left\{ \min_{1 \leq j \leq m, j \neq i} \left\{ \frac{\delta(C_i, C_j)}{\max_{1 \leq k \leq m} \Delta_k} \right\} \right\}$$

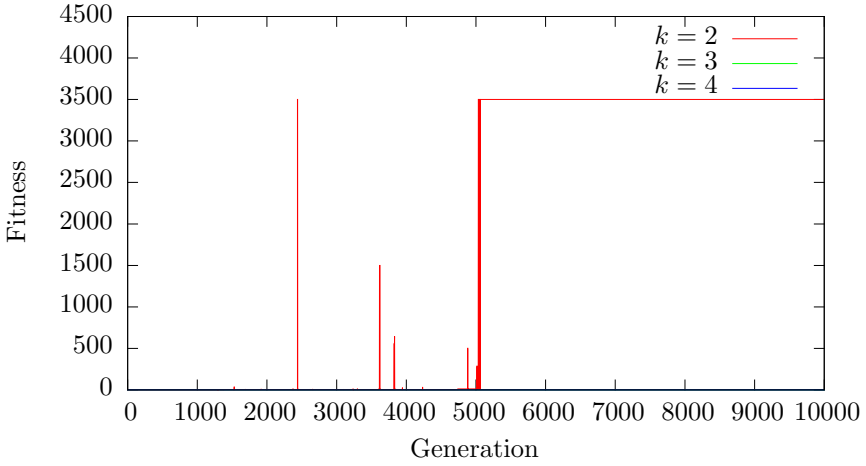


Figure 4.24: Maximum fitness with fit_{m-di}

When Δ_k, the maximum distance between two points in a cluster, goes towards 0, DI goes towards infinity. This gives an artificially high fitness to solutions with very tight clusters, and could result in huge differences in fitness between solutions with similar performance. Note that the actual implementation adds 0.001 to Δ_k to avoid division by 0. This explains the fitness peaking at around 3500 (division by 0.001 is the same as multiplication by 1000).

Figure 4.25 shows the average frequency of the output on all the possible inputs for the best genome found. As we can see this CA is able to solve the problem perfectly.

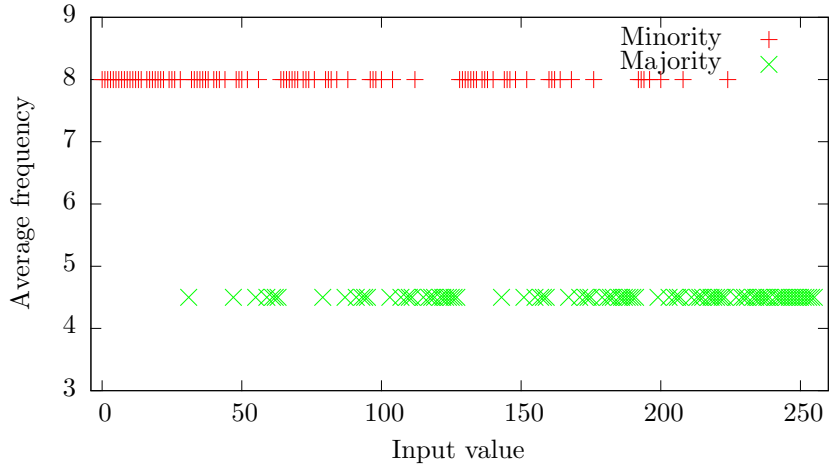


Figure 4.25: Average output frequency of all input values for the best genome found with fit_{m-di}

4.4.2 Experiment 11: Simple output mapping

Instead of using an DFT to map the output to an average frequency, it is possible to interpret the output directly. For this experiment only the last output value from the CA is inspected. If the input has a majority of bits set the output should be all 1s, and for minority inputs the output should be all 0s.

Fitness function

The fitness will be calculated by running the CA for 32 iterations on each input, and then counting the number of inputs where the last output value of CA is correct. As before the CA is 8 cells wide, meaning 70 of the 256 inputs are neither in the minority or majority. This gives us 186 valid inputs, and an optimal fitness of 186. This fitness function is named fit_{simple}.

Results

Figure 4.26 shows how the population evolves over time. We see it starts out with a maximum fitness at 93, which is most likely a CA that always returns just 0s or 1s. After about 1400 generations we see progress with 2 cell types. Around the 5000 generation mark 3 cell types are also able to make progress. However, the GA is not able to find a solution with an optimal fitness.

The long delay before any progress is achieved suggests that the GA has to do a random search before finding a genome able to give a better solution than always returning 0s or 1s. However, once such a genome is found, the evolution progresses quickly. Figure 4.27 shows the outputs of the best genome found.

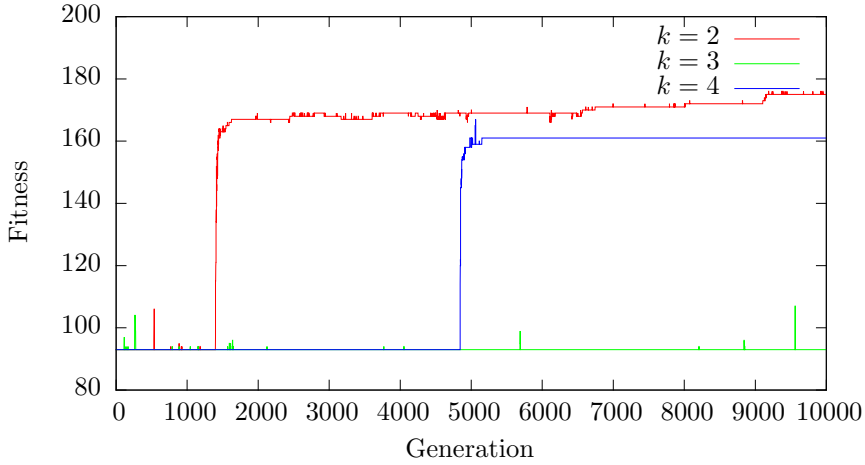


Figure 4.26: Maximum fitness with fit_{simple}

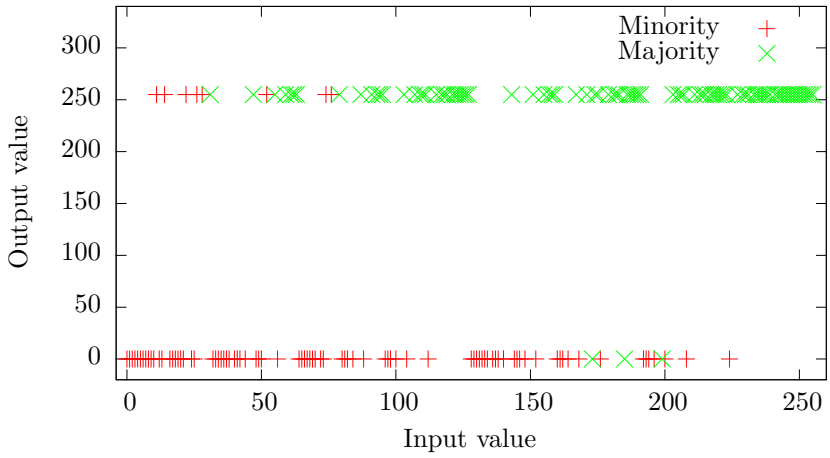


Figure 4.27: Output values for the best genome found with fit_{simple}

4.4.3 Experiment 12: New clustering measure

Inspired by fit_{simple}'s performance once the random search part is done, we define a simpler clustering measure. Instead of using the Dunn Index to measure how clustered the spectrum averages are, we instead look at how many values are in the wrong cluster. This should give us a fitness function with a good resolution, and unlike the Dunn Index the fitness will not get artificially high as the clusters tighten.

Fitness function

Let c_{maj} be the cluster of values for majority inputs and c_{min} the cluster for minority inputs. These clusters are obtained the same as in experiment 10: the CA is run for 32 iterations, a DFT is run on the last 16 output values and the average frequency $\overline{F}$ is added to the correct cluster. We then find the boundaries of the two clusters and count the number of values in the opposite cluster that are outside these boundaries. More formally it could be written as

$$fit_{cc} = \sum_{i \in C,\, j \in C,\, a \in j} outside_i(a)$$

where

$$outside_i(a) = \begin{cases} 1 & \text{if } a \text{ is outside cluster } i \\ 0 & \text{otherwise} \end{cases}$$

and C is the set of clusters. In our case $C = \{c_{maj}, c_{min}\}$.

Results

Figure 4.28 shows the maximum fitness for 1000 generations. As we can see the maximum fitness is reached in less than 100 generations with 2 cell types and less than 400 generations with 3 cell types. We also see that the progress is fairly smooth, with the exception of the jump from a fitness of around 150 to the maximum of 186. If we compare it to the experiment with the Dunn Index based fitness function evolution is both faster and with less random searches. Figure 4.29 shows the output values of the best genome. If we compare it to Figure 4.25 we see one of the drawbacks of this new fitness function. Once the two clusters are completely separated the fitness can not be improved by increasing the separation, and we end up with clusters that almost touch.

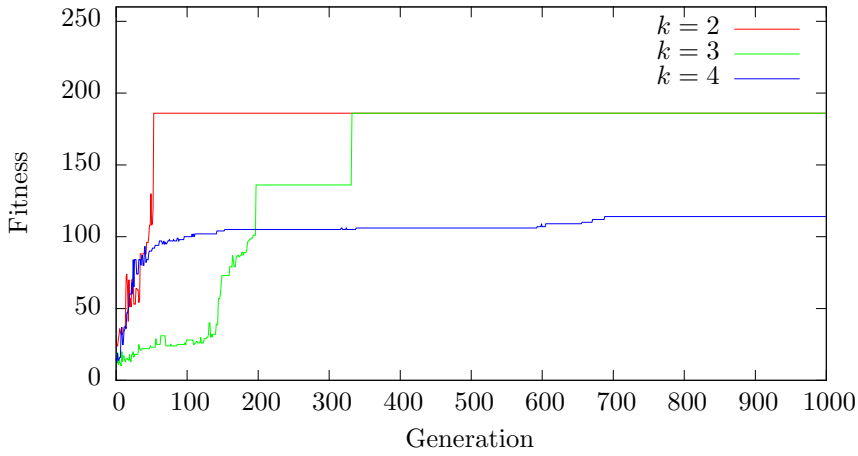


Figure 4.28: Maximum fitness with fit_{cc}

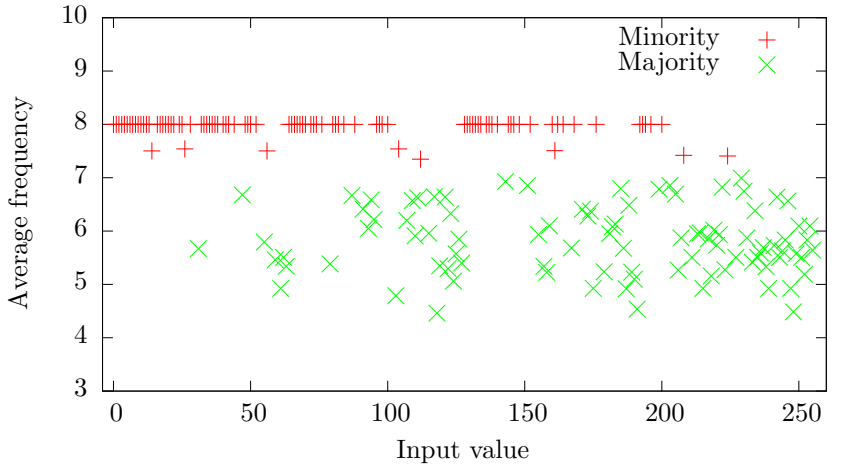


Figure 4.29: Average output frequency of all input values for best genome found with fit_{cc}

4.4.4 Experiment 13: An extra class

As mentioned earlier the input has 8 bits, meaning input with 4 bits set does not fall in either the majority or minority class. In this experiment we will try to evolve a CA that is able to differentiate between majority and minority as above, but also between the inputs with an equal number of bits set and not set.

Fitness function

The simple DFT fitness function discussed in the previous section, fit_{cc}, is used. The only difference is that a cluster for the middle values are added to set of clusters C. When a new cluster is added the maximum fitness change. We now have 256 possible input values, and they can all be outside two clusters that are not their own. That gives us an optimal fitness of $256 \cdot 2 = 512$.

Results

Figure 4.30 shows the maximum fitness. We see that with 2 cell types it is almost a linear increase in fitness until it hits a roof at around 470. The cause of the roof is unknown. It could be some computational limit of this kind of CA, or the search room could suddenly have changed characteristics around that point, making the GA unable to improve further.

Figure 4.31 shows the output values for the best genome. We see that the three clusters are mostly separated.

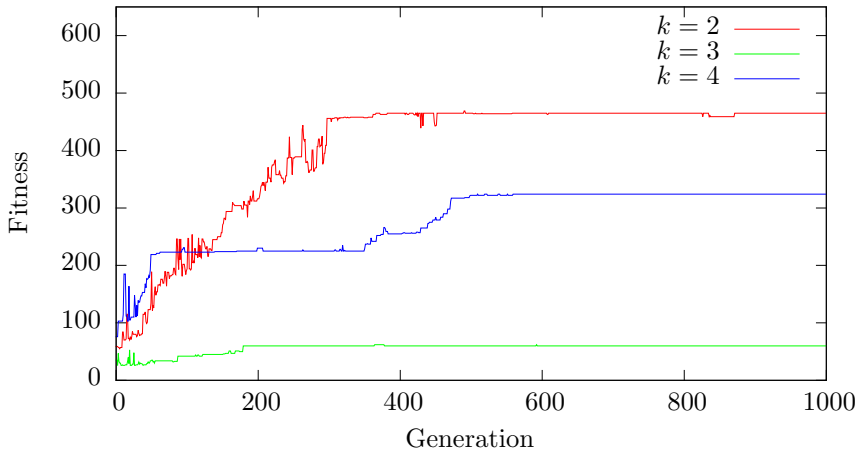


Figure 4.30: Maximum fitness found with fit_{cc} and an extra class

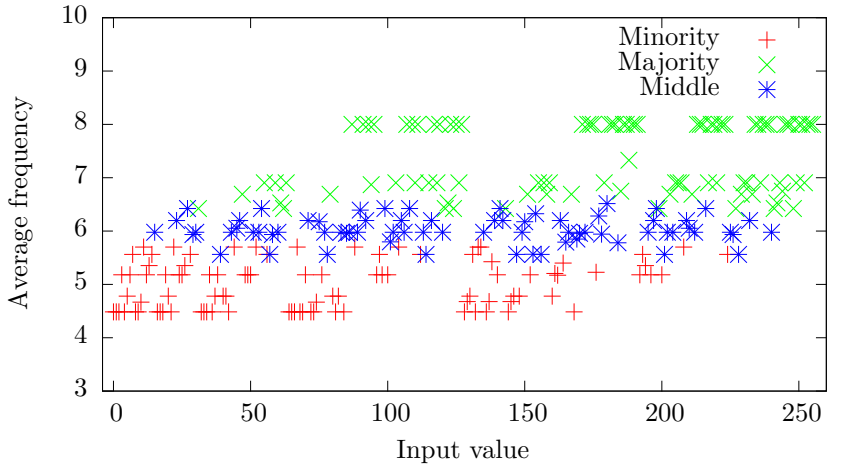


Figure 4.31: Average output frequency of all input values for best genome found with fit_{cc} and an extra class

4.4.5 Experiment 14: Combining Dunn Index and New Clustering Measure

As we saw in experiment 12 the new clustering measure had good evolvability, but was unable to further refine the solutions once the clusters were completely separated. An idea is to combine this clustering measure with the Dunn Index. The evolvability of the new clustering measure would allow a separation to be found, and then the Dunn Index could help refine this clustering.

Fitness function

The fitness function is defined as $fit_{cc-di} = fit_{m-di} + fit_{cc}$.

Results

Figure 4.32 shows the maximum fitness with 2 and 3 cell types zoomed in at the 180-200 interval (4 cell types are ignored, since it did not get a high enough fitness). With fit_{cc} 186 was the highest possible fitness. Here we see that GA is able to improve this further. The result of this increased separation can be seen in Figure 4.33. Compared to Figure 4.29 we see that the clusters are much better separated.

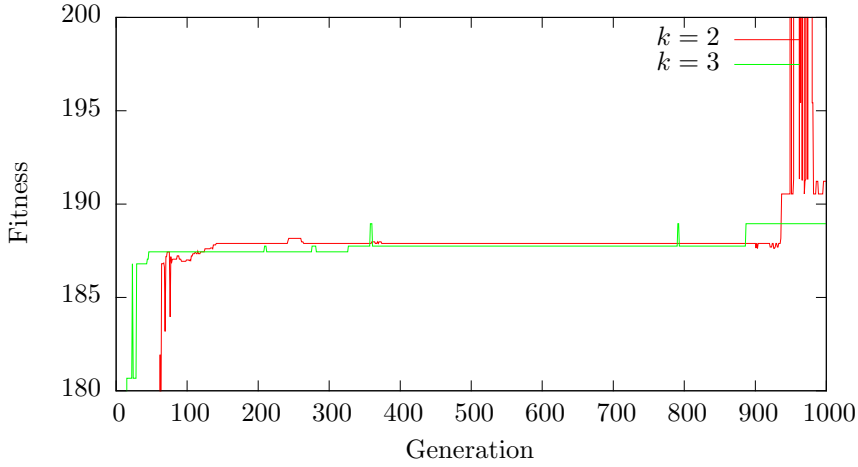


Figure 4.32: Maximum fitness with fit_{cc-di}

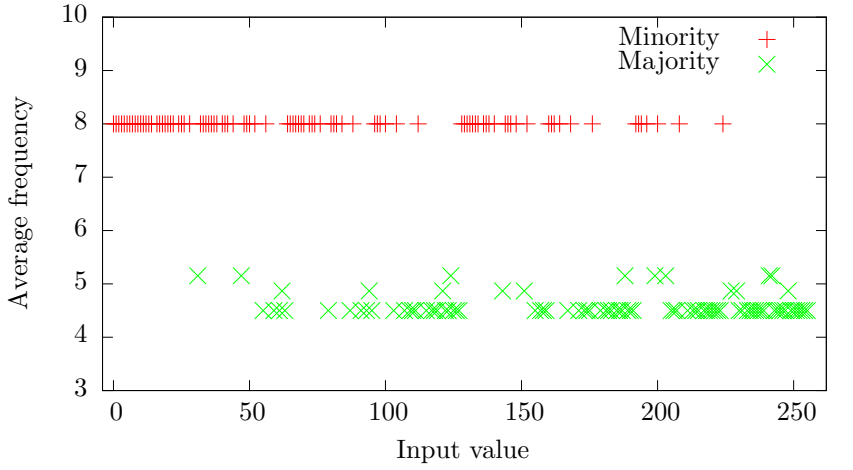


Figure 4.33: Average output frequency of all input values for best genome found with fit_{cc-di}

4.4.6 Experiment 15: Without L-system

The earlier experiments used an L-system to transform the genotype into the phenotype. In this experiment we will instead try the constant mapping used in experiment 5. We use the $gen_{2,k}$ genome representation with $k \in \{1, 2, 3, 4\}$ along with the μ_{su} mapping.

Fitness function

fit_{cc} is used as the fitness function.

Results

Figure 4.34 shows the maximum fitness. Compared to Figure 4.28 we see that they behave similarly, except for the slower evolution of 2 cell types. This could just be random behavior though, as the GA seems to be sensitive to the initial population. The final fitness with 4 cell types also end up at a lower value than when we used an L-system, and the progress seems to be rougher, containing longer stretches of no progress followed by jumps. We also see that a uniform CA ($k = 1$) is able to solve the problem perfectly.

Figure 4.35 shows one of the perfect solutions found.

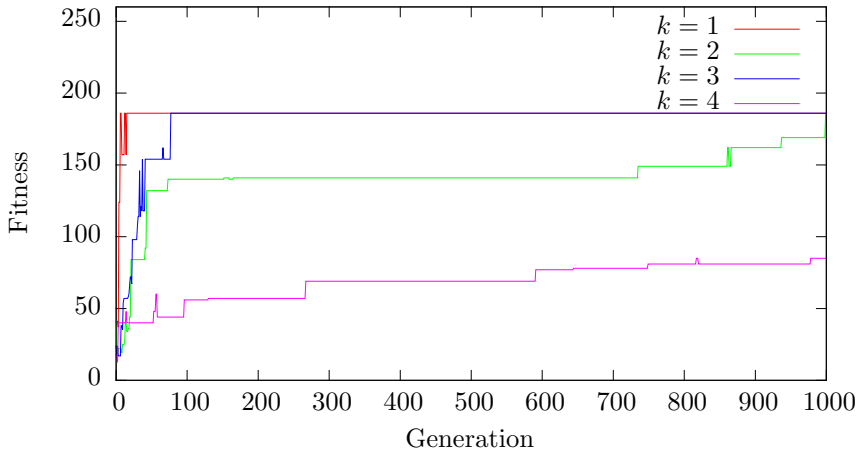


Figure 4.34: Maximum fitness with fit_{cc} and without L-systems

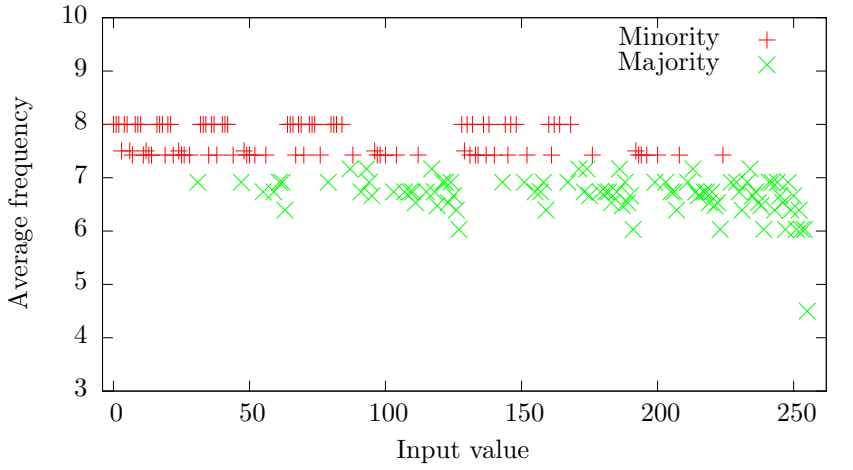


Figure 4.35: Average output frequency of all input values for best genome found with fit_{cc} and without L-systems

4.4.7 Discussion

We have seen that the system can evolve CAs able to solve the majority problem perfectly, and when the problem was extended to include another class of inputs it found a near perfect solution. In addition the performance seem to depend heavily on the fitness function. The best results are achieved when using fitness functions that allow a gradual increase in fitness.

We also see that performance seem to fall as the number of cell types increases. This is unlike experiment 5 and 6 where we compared uniform and semi-uniform CAs. In those experiments some of the best solutions had 3 or 4 cell types. If we look at the last experiment in this section we see that the system is able to evolve a uniform CA that solves the problem perfectly. In experiment 5 and 6 no such uniform CA could be found. It is possible that the majority problem is too simple. Since it can be solved by a uniform CA, increasing the genome size by adding more cell types will only result in an unnecessarily large genome space, making searching harder.

Chapter 5

Conclusion

The experiments show that using an L-system as a developmental stage between the genotype and the phenotype sometimes result in improved fitness, but it seems it could also slow down evolution. An example of improved fitness can be seen in the results for experiment 6. If we compare Figure 4.10 with Figure 4.14 there seems to be a clear advantage to a developmental mapping based on L-systems compared to a constant mapping. There also seems to be a positive correlation between the number of cell types and the performance. If we compare the figures to Figure 4.8 we also see that semi-uniform CAs outperform uniform CAs on that particular problem. However, when applied to a different problem the findings change. If we look at Figure 4.28 from experiment 12, the correlation between the number of cell type and the performance seems to be negative. Comparing Figure 4.28 from experiment 12 with Figure 4.34 from experiment 15 the differences between a constant mapping and one based on L-systems also seem to be minimal. We also see that the system finds a uniform CA that is able to solve the problem perfectly. The reason for this apparent contradictory behavior could be that experiment 6 uses the 1+4 ES while experiment 12 and 15 uses a GA. A GA has more parameters, and it could be it is not tweaked properly. However, a more likely explanation for the divergent results is that the experiment showing positive results try to solve a harder problem. This is supported by the fact that no uniform CA able to solve the problem used in experiment 6 is found. This is in contrast to the problem used in experiment 12 and 15. The extra power of a semi-uniform CA is therefore not needed, and as more cell types are added to the genome it only slows down evolution by expanding the search space.

As for the CA itself, the input/output scheme worked very well. The evolutionary algorithms were able to find CAs that worked on both static (same input for the entire simulation) and dynamic input (square waves). Experiments 3 and 4 showed that evolving the configuration values was very beneficial for the performance. This was true for both uniform, semi-uniform and non-uniform CAs. Interpreting the output stream with a DFT also showed good results. If we look at Figure 4.28 and Figure 4.26 we see that a DFT outperforms a simpler output interpretation that only looks at the last value. The results also show that using a clustering algorithm to determine the quality of the output works very well. This removes the need for the designer to specify exactly what the output should look like and gives the system freedom to search for solutions the designer might not have thought about. The ease of this approach is especially evident in experiment 13 where an extra output value was added with almost no change to the fitness function.

5.1 Future Work

It should be noted that this thesis is not focused on GA optimization, but rather with looking at a special class of CAs and ways to evolve them to solve problems. It is very likely that the GA and the genetic operators used are suboptimal, and looking into ways to improve the GA performance would be a good starting point for future work. The crossover operator is particularly raw, and testing other crossover operators would be an interesting endeavor. Here one could draw inspiration from the one used by Gabriela Ochoa [18].

The experiments showed that the number of cell types affected performance in different ways for different problems. That means we can not conclude that a certain number of cell types are best, instead it has to be tweaked for every problem. Another approach would be to make the number of cell types part of the genome. That way the EA can adjust the genome size to the difficulty of the problem, creating uniform CAs for simple problems, and more complex semi-uniform CAs as the problems get harder.

Finally it would be interesting to try the system on harder problems. The majority problem turned out to be quite simple, being solvable by a uniform CA. It could also be worth trying to evolve feedback systems where the input stream is somehow affected by the output stream. This would really put the system to the test, and show if it is able to respond to input that change characteristics as it runs.

Bibliography

[1] M. Sipper, "The emergence of cellular computing," 1999.

[2] A. W. Burks, *Essays On Cellular Automata*. University of Illinois Press, 1970.

[3] M. Sipper, *Evolution of Parallel Cellular Machines: The Cellular Programming Approach*. Springer-Verlag, Heiderlberg, 1997.

[4] C. G. Langton, "Computation at the edge of chaos: Phase transitions and emergent computation," *Physica D: Nonlinear Phenomena*, vol. 42, 1990.

[5] S. Wolfram, "Universality and complexity in cellular automata," *Physica D*, vol. 10, no. 1-2, pp. 1–35, 1984.

[6] M. Mitchell, "Life and evolution in computers," *History and Philosophy of the Life Sciences*, vol. 23, 2001.

[7] S. Kumar and P. J. Bentley, eds., *On Growth, Form and Computers*. Elsevier Limited Oxford UK, 2003.

[8] P. Prusinkiewicz and A. Lindenmayer, *The algorithmic beauty of plants*. New York, 1990.

[9] A. L. Waage, "Discrete transformation of output in cellular automata," Master's thesis, Norwegian University of Science and Technology, 2012.

[10] O. H. Jahren, "Emergent behaviour in the frequency-power spectrum of discrete dynamic networks," Master's thesis, Norwegian University of Science and Technology, 2012.

[11] M. Land and R. K. Belew, "No Perfect Two-State Cellular Automata for Density Classification Exists," *Physical Review Letters*, vol. 74, pp. 5148–5150, June 1995.

[12] H.-P. Schwefel, *Evolutionsstrategie und numerische Optimierung*. PhD thesis, Technische Universität Berlin, 1975.

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

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

[15] D. E. Goldberg and K. Deb, "A comparative analysis of selection schemes used in genetic algorithms," in *Foundations of Genetic Algorithms*, pp. 69–93, Morgan Kaufmann, 1991.

[16] K. A. D. Jong, *Evolutionary Computation: A Unified Approach*. MIT Press, 2006.

[17] E. Kreyszig, *Advanced Engineering Mathematics*. John Wiley & Sons, Inc., 9th ed., 2006.

[18] G. Ochoa, "On genetic algorithms and lindenmayer systems," in *Parallel Problem Solving from Nature — PPSN V* (A. Eiben, T. Bäck, M. Schoenauer, and H.-P. Schwefel, eds.), vol. 1498 of *Lecture Notes in Computer Science*, pp. 335–344, Springer Berlin Heidelberg, 1998.