NTNU – Trondheim
Norwegian University of
Science and Technology

Can Genome Information be used to Guide Evolutionary Search?

Håkon Hjelde Wold

Master of Science in Computer Science
Submission date: June 2013
Supervisor: Gunnar Tufte, IDI
Co-supervisor: Stefano Nichele, IDI

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

1. Abstract

Uniform cellular automata have been evolved as phenotypes from zygotes using an extensive rule table as the genotype. This is used to simulate complex systems which could impact future hardware development and programming. This work falls within the field of Evolutionary and Developmental Systems (EvoDevo). Genome parameters are used in a genetic algorithm to try to reduce the number of generations needed to find a genome with a given complexity. Lambda parameter has been used inside the fitness function and produced promising results. Lambda has also been used to discard genomes before they are developed with poor results. Transition parameters are shown to be similar to lambda in predicting the trajectory length of a developing phenotype, but have yet to produce the same results. The genome usage has been used to control mutation with good results. The results of the work have provided more insight into how genome parameters work and what to not do when using them.

2. Sammendrag

Det har blitt utviklet uniforme cellulære automater som fenotyoper fra zygoter ved hjelp av regeltabeller brukt som genotyper. Dette brukes for å simulere komplekse systemer som kan påvirke utviklingen av framtidens maskinvareutvikling og programmering. Dette arbeidet faller innenfor feltet evulosjonære og utviklingsmessige systemer (EvoDevo). Genomparametre brukes i en genetisk algoritme for å forsøke å redusere antallet generasjoner som trengs for å finne et genom med en gitt kompleksitet. Lambda-parameteren har blitt brukt inne i fitnessfunksjonen med lovende resultater. Lambda har også blitt brukt til å forkaste genomer før de blir utviklet med dårlige resultater. Overgangs-parametre er vist å være lik lambda for å forutse utviklingslengden til en fenotype, men kan enda ikke vise til tilsvarende resultater. Genombruken er blitt brukt til å kontrollere mutasjon med gode resultater. Resultatene av arbeidet har gitt mer innsikt om hvordan genomparametre virker og hva man ikke skal gjøre når man bruker dem.

Table of Contents

3. List of figures

4. List of tables

5. Introduction

This project is part of the research done at the Department of Computer and Information Science at the Norwegian University of Science and Technology. It continues the work done in my autumn project [21].

The goal of the project is to examine if genetic parameters can be used to improve the performance of a genetic algorithm. Langton's lambda parameter was the focus of the previous project, and is examined thoroughly in this project also. Additional parameters are also examined and compared to the lambda parameter. The way the genome is used when developing a phenotype is exploited to propose new parameters and control mutation in the genetic algorithm.

This work is relevant to the field of artificial life and the study of complex systems. In this field cellular automata can be used to simulate complex systems. There exists large variations of cellular automata, but here the type used is a grid of cells that can have a discrete number of states. Each cell can only communicate with its closest neighbors, and have a limited number of discrete cell states. To determine the behavior of one cell a set of rules are defined. These rules only depend on the closest neighbors of the cell and the cell itself. The cellular automaton is updated in discrete time steps where all cells update their own state to create new global states.

The behavior of cellular automata can become very complex, and it has been proven that some rules are universal [28]. Cellular automata can therefore possibly be used in a new type of hardware. Today's hardware has the limitation of separated memory and CPU which means that a lot of time is spent by CPU waiting for data from memory. Cellular automata can be used in a system that exploits locality and vast parallelism. This system would be a grid of calculation cores containing memory. They would only communicate with their closest neighbors and so eliminate the problem of waiting for data and at the same time achieve great performance due to the parallel execution.

This design is very hard to program due to the complex interactions between cells. The solution seems to be some kind of adaptive or dynamic development performed by computers. People only need to specify what the machine should do and then use an evolutionary algorithm to program it.

In this project the evolutionary algorithm used is a genetic algorithm which is used to evolve rules for cellular automata. This draws inspiration from evolution in nature where the rules are genotypes and the running cellular automaton is a developing phenotype. The concept of using genome parameters is that one can to some extent predict how a phenotype will develop based on its genome composition, and thus help guide the genetic algorithm in the right direction. The search space is vast and phenotypes take a long time to develop, so it is impossible to do a complete search. Genetic algorithms perform a random heuristic search

in this vast space; genetic parameters can help limit the search space, reducing the time it takes to find a good solution.

The first task in the project is to use lambda to improve performance of the genetic algorithm. The next tasks will examine how the genome affects the development of a phenotype to discover new ways to use it. When more is known about the genome the information will be used to find new parameters and examine how they perform compared to lambda and a standard genetic algorithm.

6. Theory

6.1. Complex systems

Complex systems are found all around us; they are researched and created in a wide range of scientific fields. Biologists seek to explain living organisms by studying phenotypes and how they develop from genotypes [8]. Economists study how businesses interact and develop in a large or small environment [9]. Computer scientists simulate numerous systems using computer hardware and software [10]. Psychologists try to explain the behavior of the human mind [11].

The scientific field of complex systems is relatively new and attempts to find general principles spanning multiple disciplines. As a result of this, fields like economy and neurobiology can suddenly be unified in a single discipline. Our economy consists of people exchanging money, which is in some ways similar to the way our brain consists of neurons exchanging electrical impulses.

When something is complex it consists of parts that are interconnected or interwoven. This means parts don't just exist together individually in isolation, they interact in different ways to make the system act as a whole. In such circumstances it is impossible to look at individual parts, one must look at the way they interact as well, and each part should be viewed in the context of the system in which it is residing. Computer simulations make it much easier to study large and complex systems. A cellular automaton is an example of this.

Several simple parts can form a complex system, this is called emergent complexity. One example is atoms; they can be viewed as simple, because they are just protons and neutrons in the nucleus and an electron cloud surrounding it. But atoms interact to make molecules as simple as water and as complex as DNA.

The function of the DNA is not simply understood by understanding the properties of each atom, this is called emergent behavior. To explain it one need to look at the interactions between the atoms in the molecule and see how those interactions leads to the behavior of the molecule as a closed system. [2]

6.2. Self-organization

A set of initially independent parts that interact only locally can spontaneously show a globally coherent pattern. This is called self-organization, and it is found everywhere in our society and in nature.

One simple example from nature is magnetization. Atoms in a substance have magnetic fields pointing in individual directions because of their spins. Normally their spins are in different directions because the magnetic fields repel each other.

In some metals atoms are freer to move around. In such a piece of metal some of the spins can therefore accidentally align. If this happens they will exert a combined force that could align more atoms which leads to an even bigger combined force. This creates a ripple effect that spreads through the metal until it is magnetized, creating a unified magnetic field strong enough to be noticed outside the metal. This is illustrated by Figure 1 where a set of spins are first randomly aligned before a fluctuation creates a ripple effect, causing all spins to align using only local interactions.

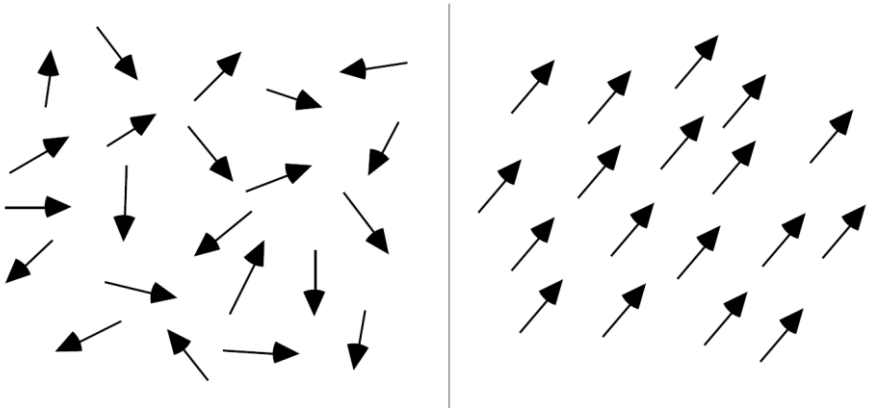

Figure 1: Magnetization, arrows representing magnetic fields [6].

The direction of the spins of the different atoms can be viewed as a set of states. The system as a whole, the piece of metal, has a global state which is the union of these states. When some of the atoms start aligning the ripple effect cause the system to naturally move towards a global magnetized state where all the spins are aligned. The state where the whole piece of iron is magnetized is called an attractor.

This is not the only attractor; the global field could ultimately be oriented in any direction. The initial alignment of spins is determined by fluctuations, this determines what path the system will take. Because the fluctuations could be impossible to predict accurately, probability is the best tool to find out what attractor the system most likely will move towards.

More formally an attractor is a state space from which a system can enter but not leave, and it has no smaller sub spaces. An attractor can contain only one single state, or many. The former is called equilibrium, or a point attractor, while the latter can make a loop of states or a cyclic attractor which is a self-organizing structure.

However, between the initial state and the attractor there is a set of states the system goes through. This is called the trajectory of the system, and it could be just as important as the attractor to research if one want to understand the system.

A way to identify attractors and the trajectories towards them is to use a fitness landscape. It will be a graph with peaks and valleys where the trajectory will follow the path of steepest decent towards an attractor. To make fitness landscape a fitness function assigns a

numerical value to every state of the system to determine its fitness. A valley will then be a local attractor; this means it could trap the system in a "less optimal" state if the local attractor is not the deepest valley, also called the global optimum. For the system to find the deepest valley a degree of noise should be added so the system can climb out of valleys and find potentially deeper ones.

Self-organizing systems are stable, and may repair themselves if they get damaged. The problem with them is that it is hard to find practical applications, due to their unpredictable development and decentralized control. [6]

6.3. Biology

Every cell in every living organism on earth has a set of genes that describes how the organism is developing. Using the DNA and cell division a single cell, a zygote, can develop into an advanced multicellular organism like a human.

All the information about the different cells function and placement is stored in the DNA. But not necessarily directly, it is the way the cells interact with each other and the environment together with the information they have from the genes that determines the fully developed organism [7].

The genes store the genotype of the organism, and the fully developed organism is called the phenotype. Developing from a zygote to a phenotype needs to be done using only local interactions and the DNA, this process is self-organizing.

Natural selection is the principle that weaker phenotypes (less fit) die, and the stronger ones (more fit) live, being able to carry their genotypes to new generations. Sexual recombination of phenotypes together with mutation helps evolve the species. Over time evolution makes the phenotype more fit by "improving" the genotype. These principles have inspired scientists and are being exploited in software development and experimentations. Examples are evolutionary algorithms and cellular computing. [12][3]

6.4. Cellular computing

Cellular computing consists of three principles; simplicity, vast parallelism and locality [1]. A cellular computing system consists of a vast amount of very simple computing units called cells that can only communicate with a few other cells. The communication between cells are purely local, this leads to a decentralized control.

An example is using a vast amount of very simple processing cores organized in a grid. They would only communicate with their 4 closest neighbors to compute their states once per clock cycle. The raw computing power and scalability offered by such a system would far exceed what we see today in our traditional von Neumann architectures. Today's systems

have reached a point where the bottleneck is the time it takes to communicate and locate data from memory. Using a cellular computing system, locating data from memory will no longer be a concern because all communication happens locally between cells.

The cellular computation model can be defined in different ways depending on what tasks it is meant to solve. A cell can contain discrete or continuous states. Its behavior is defined by its own state and its neighbors' values. To determine the behavior one can use a function, a program, a set of rules or an exhaustive list of neighboring values and resulting states. A cell can be mobile or not, if it is not the cells can be organized in a grid or a graph. Cells can have an environment to work in such as a chemical map that provides extra input and output possibilities for the cells. The way cells are updated can be done discretely or continuous, synchronously and asynchronously. And finally; cells, their behavior and connections can be different from each other or all can be the same.

There are several operational and behavioral aspects that are very important to cellular computing. First is how to program it. It may be difficult to program it directly, so an adaptive approach is more likely to succeed. Then there is the issue of I/O which needs to be defined, how input is written and how it is read for the cellular model that is used.

Finding local interaction rules to solve global problems is another challenge, the definition of emergent computation is also still lacking. To simplify these problems it is possible to look for a hierarchical decomposition, this is something that is present in both natural and artificial systems. A more practical issue is how to implement a cellular computing model. It could be done with transistors as processors are made today, or with molecular computing or quantum technology.

Robustness and scalability is together with the possibility of vast parallel computation possibly the biggest advantages to cellular computing. Scalability is possibly very hard to achieve, but because of the decentralized control it could be unlimited. The robustness also stems from the decentralized control, a fault will be more contained and depending on the CA it could be able to repair itself, or degrade gracefully, and not stop working completely. [1]

6.5. Cellular automata

A cellular automaton is an Idealized Cellular machine; it produces complex behavior using simple rules and could prove useful in achieving cellular computing. It is an n-dimensional grid of cells which are evenly spaced and with a given number of neighbors. Each cell can have 2 or more states and 2 or more neighbors and has a simple behavior based on its neighbors and state. The cells have discrete states and are updated in discrete time intervals. The simplest example of a cellular automaton is one with a 1 dimensional grid or line of cells that each has two neighbors and just two states. This is called an elementary cellular automaton [13].

Figure 2 displays an example of rule 30, which is a famous elementary cellular automaton. It is famous because the patterns it produces are seemingly random, and it has been used for random number generators, like Wolfram Mathematica [29].

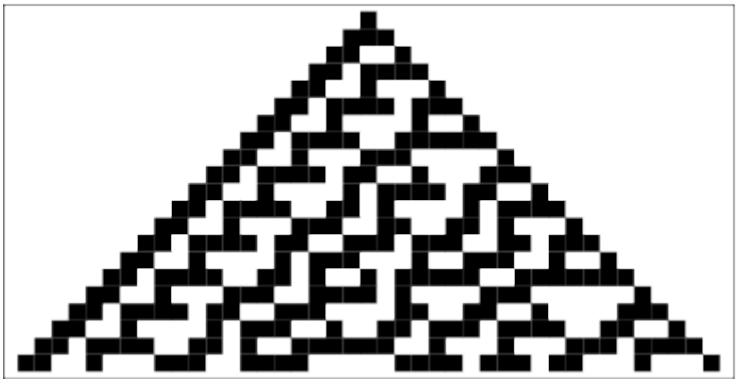

Figure 2: First 20 generations of rule 30, an elementary CA. [13]

What is interesting about cellular automata is the complex behavior that emerges from simple rules. They can make complex systems from simple parts, just like atoms come together to make complex molecules. Even more similar to cellular automata is the way the genome in the form of DNA gives rise to a phenotype through complex behavior and interactions. DNA can hardly be called simple, but similarities can be found between the development of a cellular automaton and the growth of an organism. The rules for each cell in a cellular automaton can be viewed as its genome and the final state that emerges from that genome as a phenotype, or an organism. Inspired by biology, evolutionary algorithms might prove useful as a way to program cellular automata to do useful tasks.

When dealing with cellular automata it is important to consider what to do with the edges of the grid in which they are developing. The grid cannot be infinite, so there will be edges. One could make all the cells at the edges have constant state neighbors. Another way is to give the cells on the edges different rules. The edges could also be connected to their opposites, so that the neighbor of the rightmost cell(s) is the leftmost cell(s) and top to bottom as well.

A cell communicates within a finite local region within the system. It is defined by a neighborhood template consisting of N neighbors and each cell can have K number of states. Each cell has a finite input alphabet with size A and F possible transitions. Then we have:

$$A = F = K^N$$

To define if a cellular automaton can do anything useful is difficult, and different genomes can give rise to many similar behaviors or phenotypes. Stephen Wolfram has tried to classify different types of one dimensional cellular automata based on their behavior, and this list seems obligatory for every paper dealing with cellular automata [16].

6.5.1. Class 1

Almost all initial configurations lead to another fixed configuration after a transient period of time. This is independent of the initial configuration, and almost no patterns emerge, and the process is irreversible.

6.5.2. Class 2

Almost all the initial configurations lead to another fixed configuration or a temporarily periodic cycle of configurations after a transient period. This is dependent on the initial configuration and is not reversible.

6.5.3. Class 3

Almost all initial configurations lead to chaotic behavior after a transient period. The evolution process of these cellular automata is reversible, and though it is chaotic it is not random.

6.5.4. Class 4

This class is not rigorously defined, but it is the only class with non-trivial automata. The other classes are trivial because they either have fixed dynamics or chaotic dynamics. Class 3 is trivial because it is reversible and not random. This class of automata sometimes results in complex localized behaviors that sometimes is long lived and is irreversible. Wolfram proposed these automata could be capable of universal computation.

6.6. Environment

The genome alone do not decide how a living organism in nature will grow, it is a complex process that includes the genome, the interactions between cells and the environment in which it grows. According to Lewtontin we have an incomplete view of genes as the blueprint for an organism and we fail to see the importance of other factors in the development of it [8].

An approach used in cellular computing is therefore to add an environment for the organism to grow in. One example is the French Flag Organism by Miller. It uses chemicals together with a cellular automaton to make an organism that resembles a French flag that is highly resilient to damage and repairs itself quickly. [7] The interesting part is that the chemicals are the environment and it may help stabilize the cellular automaton. A different example is more specific, it is finding a solution to the traveling salesman problem by having a form of cellular computation work in the graph with nodes corresponding to cities.

A cellular automaton could be evolved to fit in its environment to solve a task, or the environment could help the cellular automaton to evolve in a more general way. There is not too much research to be found on this subject, but it might prove interesting as a way to speed up evolution or guide it.

6.7. Lambda parameter

Genome parameters are used to estimate the behavior of an emergent phenotype based directly on the genome. It is hard to find such parameters because of the complexity of cellular automata. The genome is not a direct mapping of the phenotype; it just describes the behaviors developing it.

Langton has tried to find one such genome parameter; a relation between the behavior of cellular automata and a parameter λ [5]. In this work he found that the basic functions required for computation is most likely to be found in class 4, between the ordered and disordered dynamics, the edge of chaos.

The λ parameter says something about the degree of randomness in the cellular automata. It is simple to calculate from an exhaustive rule table where every neighborhood configuration and the resulting states are listed.

Let C be the number of all possible neighborhood configurations (which is the same as state transitions in the rule table). Choose one arbitrary state s from the possible states, n will then be the number of transitions to this state in the rule table. Then lambda is found like this:

$$\lambda = \frac{C - n}{S}$$

If n = C all transitions in the rule table will result in the state s and λ = 0.0. This coincides with class 1 of cellular automata. If n = 0 there will be no transitions to state s and λ = 1.0. When λ = 1.0 – 1/S , with S being the number of states for one cell, there is an equal amount of transitions to each state in the rule table. This coincides with class 3 of cellular automata. This means that class 4 of cellular automata, the edge of chaos lies somewhere between λ = 0.0 and λ = 1.0 – 1/S. [5]

The maximum lambda value could be used to limit the search space for a suitable cellular automaton. For 3 states the search space is limited by 1/3. However for larger state spaces it is limited by a smaller amount.

6.8. Usage of the lambda parameter

The developed cellular automaton can be stable, as in having reached a single state that is not changing, a point attractor, or it can be self-reorganizing with many states, a cyclic attractor. The work of Tufte and Nichele shows many tests where they calculated the lambda value of genomes and analyzed the complexities of the resulting phenotypes [15]. The phenotype was evolved from a zygote in a 4x4 and a 5x5 grid of cells. They used a 2-dimensional cellular automaton where each cell could have 3 states and 4 neighbors. For different lambda values there have been shown patterns in average complexity of the evolved phenotype. The complexity was quantified by the length of the attractors and

trajectories of the resulting phenotype. This has shown that the lambda parameter could be a good indication on how a system will develop with regard to complexity. One could therefore use a lambda parameter to help predict how complex a cellular automaton will be given its genotype, and also help evolve systems with desired complexities faster. Figure 3 is the result of an experiment measuring trajectory lengths in a 4 by 4 grid for phenotypes developed from genomes with certain lambda values. It shows a clear increase in genomes with long trajectories as lambda get closer to 1.0 – 1/number of states.

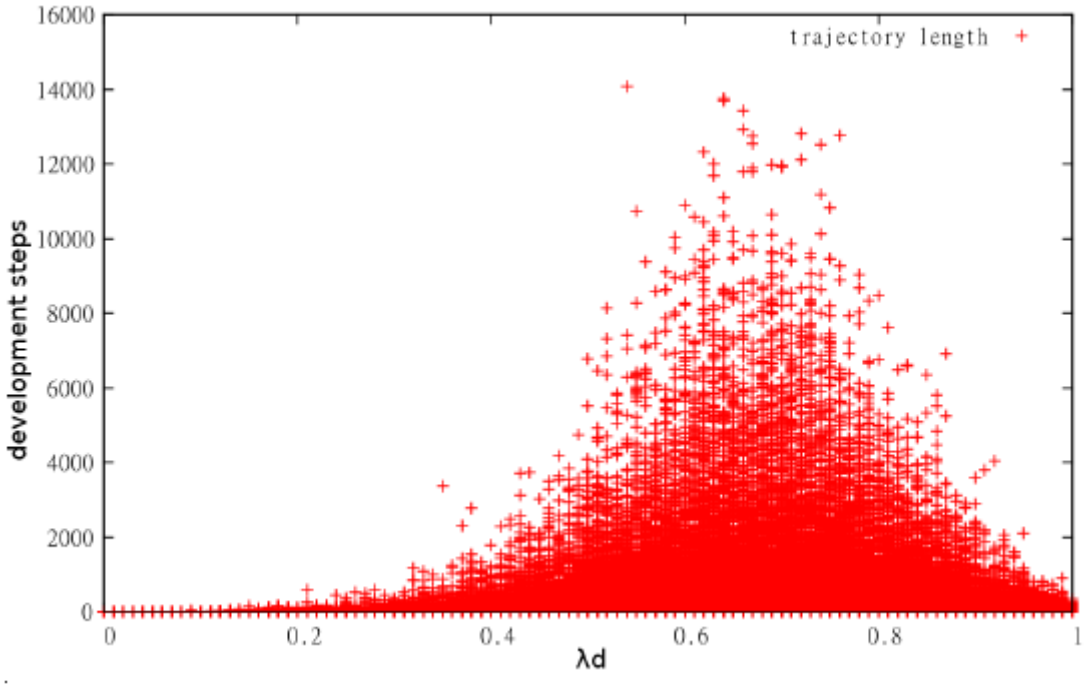

Figure 3: Plot of trajectory length and lambda [15].

6.9. Genetic algorithms

Genetic algorithms are a type of evolutionary algorithms, drawing inspiration from natural selection. They use a population of individuals which are each assigned a fitness value. Each individual has its own genotype, which could be anything, for example a string or an integer array. The genotype is used by the fitness function to develop a phenotype, on which a fitness value is measured. This is done once for each generation, and only the best individuals in a population will move on to the next one. Genetic operators are used on the best individuals to produce offspring for the next generation.

Genetic algorithms can be used to find approximate solutions to a wide range of problems, including ones we do not fully understand. Cellular automata are such a problem. They are very hard to understand, especially because changing only a small part of the genome can change their behavior completely.

Assuming the individuals can be represented as bit strings, a genetic algorithm generally has these steps:

1. Create a set of N random individuals
2. Find a fitness value for each individual. This can be done in many possible ways, but the fitness must be a numerical value.
3. If some goal is reached the algorithm is now finished.
4. Repeat for n new individuals:
   4.1. Choose two of the individuals with the best fitness as parents.
   4.2. Do a crossover by choosing a random position in the bit string for the parents. Then create two children by placing a part from each parent in the first child and the other parts from each parent in the other child.
   4.3. Add the children to the new population
5. Add the N-n best individuals from the current population to the new population
6. Go to step 2.

There exist some techniques to improve this basic genetic algorithm:

Mutate the individuals with a low probability. A good value is suggested to be 1/L for a bit-string of length L [3]. Figure 4 illustrates this; one bit is flipped in the top genome resulting in the bottom genome.

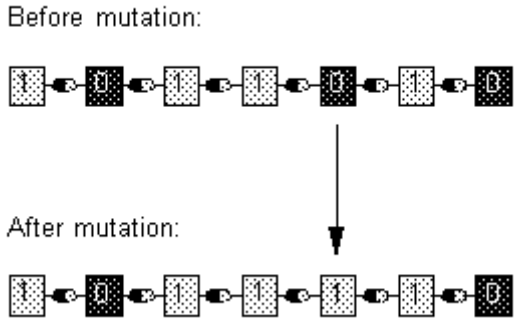

Figure 4: Mutation of a gene in a genome [18].

Roulette wheel selection gives different probabilities to the individuals to be picked for the next generation, not simply the best lives and the worst dies. The ones with the best fitness get a high probability to be picked, and the worst a low probability. In Figure 5 A is the most fit genome and G is the worst fit one. A random number selected between 0 and F has a higher probability of selecting A.

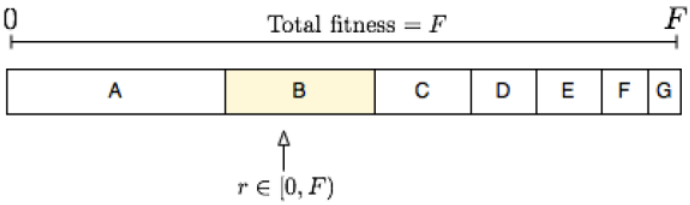

Figure 5: Roulette wheel selection. [3].

Introducing a crossover rate means to add a probability for the crossover operation to happen. The individuals can possibly go on to the next generation without any crossover. A good crossover rate is suggested to be 0.7 [14]. Figure 6 is an example of doing crossover. This uses a fixed point at the center of the genome. The bars illustrate bit/byte strings where one part of each parent is placed in each child.

Figure 6: Crossover [3].

These techniques are inspired by the introduction of noise in self-organizing systems. The improvements add randomness to the algorithm. The genetic diversity increases as poor individuals has a chance to survive to a new generation and good individuals have the possibility to be crossed over with bad ones. Viewing this from a fitness landscape perspective one could say that the techniques adds noise that helps the search climb out of local maxima and improve on already good solutions.

6.10. Complexity measurement and prediction

To have a clear definition of complexity is important in developmental systems. An approach taken by Hornby in defining complexity in developing systems is to refer to our “interest and the ability to produce designs”. To define complexity according to what we want to produce. This is described in [19] as:

“His [Hornby’s] notion of complexity implicitly encapsulates characteristics which directly contribute to the fitness of the generated organism”

This means that if we want cellular automata to perform universal computation, we should find complexity characteristics that emphasize this. One example is the complexity of complex structures forming in class 4 cellular automata. How can the definition of complexity in cellular automata contribute to growth towards such structures? A way to make use of a better definition of complexity is to use genomic parameters. They can use the genome to predict the complexity of the emergent phenotype; lambda is an example of this.

6.11. Hardness for genetic algorithms

To improve the performance of a genetic algorithm it is first important to understand what makes a problem hard.

Terry Jones and Stephanie Forrest have examined the correlation between evolutionary algorithms and heuristic state spaces**Error! Reference source not found.**. They argue that genetic algorithms can be viewed as randomized heuristics rather than an algorithm. The heuristic search space is similar to the fitness landscape of a genetic algorithm if viewed as a directed graph. The heuristic function is then also similar to the fitness function.

It is hard to define what a fitness landscape is. The peaks and valleys are defined by the fitness function. But these peaks can exist in a variety of landscapes, or in a multidimensional fitness landscape. Further, the notion of peaks and valleys are relative to some point which implies the existence of a neighborhood. This neighborhood is defined by genetic operators. One crossover operation leads to a new individual which is the neighbor of its two parents connected in a directed graph. The same goes for mutation, but this is an entirely different fitness landscape.

The use of Hamming distance between two genotypes can be used to estimate the distance between them in the fitness landscapes. It will operate across fitness landscapes, in terms of operations needed to change the genome string, but so does the GA.

In [20] they calculated hamming distance between genomes and a known global optimum. It was shown that the correlation between hamming distance and the fitness gives an indication of how hard a problem is.

This is similar to how heuristics is better if they estimate the cost of reaching the goal more accurately. In the same way a fitness function is better if its output value is indicative to how many genetic operations are needed to reach the global optima.

6.12. Earlier work

The work this project is based on can be found in the previous report [21]. There the use of lambda parameter to improve a genetic algorithm was investigated. The goal was to find cellular automata rules that produced systems with a certain number of development steps, or trajectory lengths. Lambda was used both inside the fitness function weighted 50 – 50 with the trajectory length and outside the fitness function to discard unwanted genomes before they were developed.

The plot shows the result of searching for individuals with trajectory length 25000. Fitness is the inverse of the trajectory length on the Y-axis and generation is on the X-axis. The setup for the experiment is similar to what will be used in this project. A 3 state uniform cellular automaton using von Neumann neighborhood and with a 4 x 4 grid and wrap around was used.

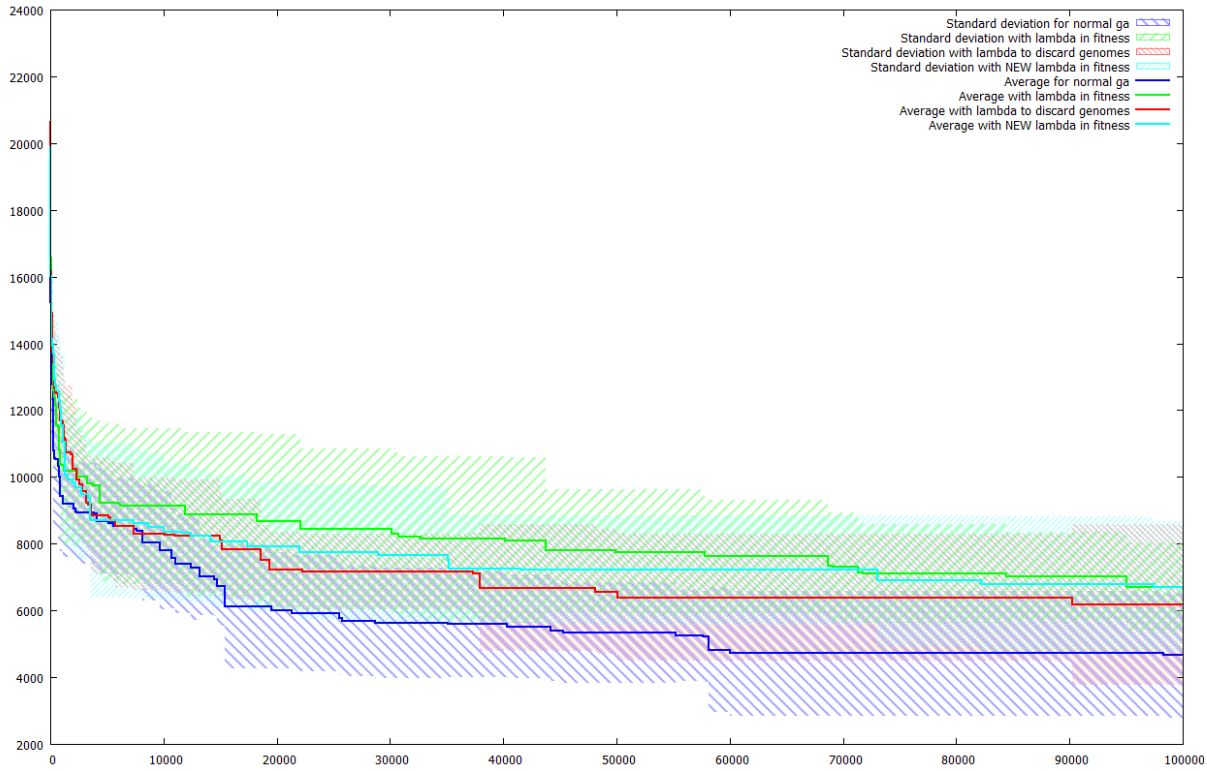

Figure 7: Fitness plot from previous project, complexity test 25 000

The results gave no clear indication that the techniques improved the algorithm. The average results measured from 10 runs of the genetic algorithm proved to be too inaccurate. There also seemed to be a number of issues with both the genetic algorithm and the extensions.

The method with lambda in fitness used a lambda value relative to the best performing genome of the present generation. This seems to give a too high emphasis on this genome when performing roulette wheel selection, meaning the best genome got picked too frequently. This may have resulted in a very homogenous population unable to evolve past

local maxima. One proposed solution was using a lambda value relative to the best genome of the previous generation. Also the weight given to lambda in the fitness function could be decreased.

Lambda to discard controlled genomes after the genetic operators had been applied. If a genome string had a lambda value too different from the best genome of the present population it would not be added to the new population. The limit at which a genome gets discarded may have been too small. It is possible that genomes that are more different helps drive the evolution forward in terms of creating noise to escape local maxima.

The genetic algorithm itself had an elitism of only 2, where 8 new genomes were introduced each generation. The mutation rate of 1/L also may have been too low. The reason these choices were not altered previously was that the initial tuning of the genetic algorithm was believed to not affect the improvements provided by lambda parameter.

7. Experimental setup

In this chapter the design and implementation of the experimental setup is described. The focus is on the cellular automata engine, the genetic algorithm and the genome parameters. Details of tuning and implementation for each experiment is not described here, this is left up to each experiment.

The setup was re-implemented in Java programming language early in development. The previous project used C++ [21]. The reason for the change is that the system is highly memory bound and it accesses random locations in memory frequently. This means it is hard to exploit locality and leads to a high number of cache misses, which would be very time consuming to fix in C++. Java has automatic memory management and limits this problem [22].

The re-implementation worked very well, in fact experiments that previously took 2.5 hours to run only takes 15 minutes with the Java implementation. This made it possible to have average plots with more runs of the genetic algorithm.

The first half of the experiments was run on an experimental virtual server hosted by openstack.idi.ntnu.no. The performance provided were however weaker than what my laptop could do. The last half was run on my laptop, because it provided better performance with Intel Core 2 Duo 2GHz and 3GB of ram.

The code was written in java using Netbeans 7.0.1 and compiled using standard jdk 1.6 with no additional libraries. The results of the experiments were written to text files and Gnuplot were used to plot the contents of the text files. Scripts were used to customize the plots and examine simple relations between transition parameters.

7.1. Cellular Automata

Uniform cellular automata are used to simulate the development of complex systems. A cellular automata engine class has been implemented to run the simulation. It runs as discrete time steps where all cells are updated simultaneously; the current state is read and the next state is written. It is highly customizable, but this project has only used one configuration.

The algorithm used for the cellular automata engine:

1. Set initial state
2. For each cell each time the cellular automata is stepped
   2.1. Find the neighborhood cell values.
   2.2. Use neighborhood to calculate the new cell value index
   2.3. Read new cell value from rule array
   2.4. Write the new cell value to the next state

For simplicity the engine uses an integer array to store the cellular automaton. The border conditions and size is set before initialization. For wrap around the opposite edges of the cellular automata both vertically and horizontally will be treated as neighbors. If wrap around is not used an edge state have to be defined.

The possible neighborhoods are the 2 and 3 dimensional von Neumann and Moore neighborhoods and the 3 cell one dimensional neighborhood. The engine uses the neighborhood definitions to distinguish between different dimension cellular automata. The neighborhoods created will always be one dimensional integer arrays, and treated that way. The difference is only the way cells are selected for the neighborhood.

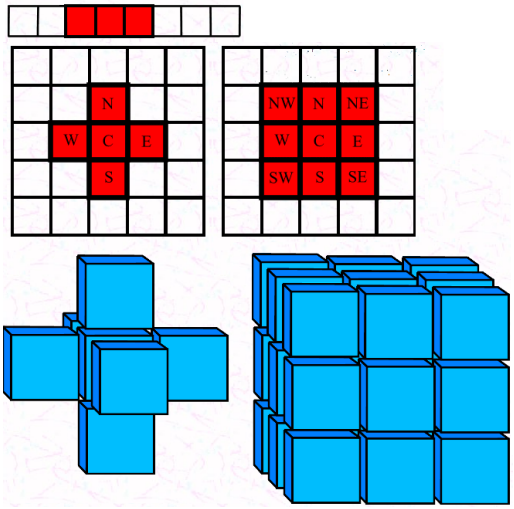

Figure 8: The possible neighborhoods of the cellular automata engine [17]

An extensive rule table is used to define cell behavior. The center cell is always found at the middle position in the neighborhood. For each possible neighborhood the resulting state for the center cell is defined. These resulting states are stored in an integer array and the index of each state is calculated from the neighborhood in the getGeneIndex method below.

```
private int getGeneIndex(int[] neighborhoodConfiguration) {
        int index = 0;
        for(int i = 0; i < neighborhoodSize;i++)
        {
            if(neighborhoodConfiguration[i] != 0)index +=
                  neighborhoodConfiguration[i] * indexHelper[i];
        }
        return index;
    }
```

The engine has later been extended to keep statistics like measuring genome usage. These modifications are described together with their experiments.

7.2. Genetic Algorithm

The genetic algorithm has been designed to develop cellular automata with given attractor or trajectory lengths, not to increase past a threshold value. The rule array used in the cellular automata engine is used as the genome and the developing cellular automaton is the phenotype. The fitness calculations take up the most time during execution. The algorithm therefore runs one thread for each fitness calculation.

The algorithm will be tested with improvements based on genome parameters, so the basic version is very simple. It uses a crossover and mutation rate together with roulette wheel selection to drive the evolution. The extensions added later will not change the existing algorithm, they are implemented in isolation and can be turned on and off easily.

The fitness value is the target trajectory or attractor length subtracted with the actual length. This means that the algorithm converges on 0, and this leads to small changes in some parts of the algorithm.

The basic genetic algorithm:

1. Create initial population of size n
2. Calculate fitness by developing phenotypes
3. Check if goal is reached, if not continue
4. Elitist selection; the k best individuals is sent to the new population
5. Repeat until new population is filled with n unique individuals
    5.1. Use roulette selection to find 2 parents
    5.2. With a probability do crossover to create 2 children from the 2 parents, if not the parents becomes the children directly
    5.3. Mutate children with a probability
    5.4. Add children to the new population if they are unique
6. Return to 2

Individuals are stored as a separate data structure called Genome Data. It contains the genome, fitness, parameter values and statistics. This makes it easy to add functionality and avoid calculating fitness more than once.

The fitness is calculated by stepping the cellular automata engine with a given genome and standard initial state. A hash map is used to store and search all states. The state itself is the key and the time it appeared as the value.

The algorithm used in the fitness function:

1. If fitness is already calculated, exit
2. Initialize cellular automata engine with genome
3. Initialize state counter to 0
4. Initialize cellular automaton state to standard start state

5. Repeat until a state appears twice
    5.1. Put cellular automaton state and state counter into hash map
    5.2. Step cellular automata engine to get a new state
    5.3. Increment state counter
    5.4. Search hash map for new state
6. Return goal trajectory/attractor length minus state counter as fitness

To perform roulette wheel selection the individuals are first sorted so that the one with the best fitness is first. The fitness is then adjusted by subtracting the worst fitness of the population from the actual fitness of each genome. The individuals are then normalized so the population gets a total fitness of 1.

Roulette wheel selection then generates a random decimal number between 1 and 0. The number are then subtracted the finesses one by one in descending order until it reaches 0. The last individual that was used is then selected.

Crossover uses roulette wheel selection to choose two parents. The genomes are then split at the middle and recombined to create two children.

Mutation is performed by traversing the entire genome once and for each gene a decimal number between 1 and 0 is selected. If it is smaller than the mutation rate that gene is incremented and wrapped back to 0 when too high. This means if a gene in a genome is 0 and it is mutated it will become 1, next time it mutates it will become 2.

The initial population can be created using random genomes or empty ones. A random population is created by setting every gene in a genome to a random, legal, value and controlling that the genome is unique. An empty population is genomes with all genes initialized to 0 (the dead state).

7.3. Genome parameters and statistics

The way genome parameters have been used is explained more closely for each experiment, but here is a rough explanation of their place in the algorithm.

Lambda parameter is used inside the fitness function as part of the fitness value and outside to discard genomes. Inside the fitness function lambda has been calculated relative to the best genome of the previous generation and also relative to the highest theoretical value.

Old lambda in fitness version used previously calculated the best lambda relative to the best one of the current generation. The way it was added to the fitness function was that first all the lambda values of a population was normalized to 1 as for the fitness explained earlier. This value for each genome was added to the fitness so the roulette wheel selection chose a random value between 0 and 2 instead. The new way to calculate lambda in fitness simply

adds a calculated lambda value to the fitness in the fitness function. This is explained in 9.15.

When used to discard genomes that are to be put into the new population lambda is checked to be within a given limit from the best genome of last generation. This is shown below.

```
if(useLambdaToDiscard) {
      getLambda(childA);
      if(lambdaLimitFromBest <
            Math.abs(childA.getLambda() - bestLambda)) continue;//discard

      getLambda(childB);
      if(lambdaLimitFromBest <
            Math.abs(childA.getLambda() - bestLambda)) continue;//discard
}
```

The usage of the genome has been measured and used to control mutation rate dynamically. The genome usage is how big part of the genome is used in a developing phenotype. In other words this means counting how many different neighborhood configurations appear in the execution of a cellular automaton and dividing it on possible neighborhoods (number of CA steps multiplied with number of CA cells). This is done in the cellular automata engine as shown below.

```
private int getGeneKeepStatistics(int[] neighborhoodConfiguration) {
        int index = getGeneIndex(neighborhoodConfiguration);
        totalUsage++;

        genomeUsageStatistics[index]++;

        return genome[index];
}
```

Different transition parameters were investigated and used the same way as lambda. A transition is what the center cell of a neighborhood changes between. If a neighborhood configuration contains a center cell that is 0 and result in a center cell with value 1, this is a growth transition between 0 and 1. A different growth transition is 0 to 2.

To use them in genetic parameters knowledge about what gene in the genome represents which kind of center cell in its corresponding neighborhood configuration is needed. The corresponding center cell for a gene with a given index is found using integer values in this way:

$$center\ cell = \frac{gene\ index}{number\ of\ states^{(\frac{neighborhood\ size}{2}-1)}}\ \%\ number\ of\ states$$

The transitions can be sorted in 4 sub-groups; growth, differentiation, death and none-transitions. Growth is when a cell state changes from 0, while death is a transition to 0.

None-transitions are cells that do not change and differentiation are changes not to or from 0. These transitions have been examined by Gunnar Tufte [23].

For a 3 state cellular automaton we have these transitions:

	Transition type	Transition (center cell to result)
1	None	0 to 0
2	None	1 to 1
3	None	2 to 2
4	Growth	0 to 1
5	Growth	0 to 2
6	Death	1 to 0
7	Death	2 to 0
8	Differentiation	1 to 2
0	Differentiation	2 to 1

Table 1: Transitions for a 3 state cellular automaton

8. Hypothesis and purpose of the work

This explains the hypothesis that genome parameters can improve the performance of the genetic algorithm and explains briefly about the reasoning behind the work that is done.

Evolutionary techniques and more specifically Genetic algorithms are widely used to evolve cellular automata [3] [4] [24] [23] [25]. They are well suited for this purpose because the unpredictable outcomes of changing individual rules in the rule table make cellular automata hard to understand and develop manually. The mapping of genotype to phenotype is natural for both cellular automata and genetic algorithms as well.

The focus is on trajectory length used as a measure of complexity. This is simple to measure and unambiguous. All parts of a developing phenotype will be contained within the trajectory. Though it is simple to measure it takes time, as every step of the developing phenotype needs to be analyzed and stored. The main reason for using trajectory length is because this work is in context of other work done by Stefano Nichele and Gunnar Tufte [15]. Langton has also shown that trajectory length is related to emergent complexity [5].

Sometimes it can take a very long time to evolve cellular automata, and time becomes an issue. It is therefore great potential in reducing the time it takes for a genetic algorithm to converge on a solution.

Genetic parameters could be suitable for this purpose. Lambda parameter has been shown to have some relation to trajectory and attractor length [15]. This property can be used to provide the algorithm with an indication as to what genomes are good to evolve.

Lambda will be used to discard genomes. This is done by only letting genomes that are within a limit of difference from the best genome of the previous population. This should keep genomes more similar, in terms of behavior and also in terms of genome distance. This should prevent the algorithm spending time to develop phenotypes that most likely do not have the desired properties.

Lambda will also be used inside the fitness function and returned as part of the fitness value. This will make the fitness function closer related to the genome, resulting in a fitness better indicating the edit distance between genomes. As described in 5.11 this can reduce the hardness of the problem for a genetic algorithm and reduce the time it takes to find genomes.

A Mutation rate that is high to begin with and decreases over time is shown to perform very well in genetic algorithms [26]. An adaptive approach to this has also been shown to be successful [27]. Genome usage rate could therefore be a useful way to regulate mutation rate. As genome usage is low initially when the initial population is empty genomes and, as later experiments show, the genome usage is only low in the beginning of the genetic algorithm.

Transition parameters are investigated as they provide similar information as lambda. The use will therefore also be similar. The question is whether or not they are able to perform better compared to lambda parameter. They have different levels of information about the genome. They can be combined in one parameter to see how this information level affects the genetic algorithm.

It is interesting to learn more about cellular automata by studying their genomes. This will help making genetic algorithms that can develop better genotypes faster. Using genome parameters is a direct way of exploiting this knowledge. Genome parameters will be examined and used inside the fitness function to contribute to fitness. They will also be used outside the fitness function to control mutation and to discard genomes.

The goal is to show that genome parameters can improve the genetic algorithm. Some questions that needs answering: How does lambda affect the search from within and outside the fitness function? How can other genome data compare to lambda in terms of performance? Does the configuration of the genetic algorithm affect the performance when using genome parameters? What properties of the genome data are useful for improving the genetic algorithm? The experiments attempt to answer these questions and more as they arise.

9. Experiments

Here the experiments performed are presented chronologically, which makes it easier to argue the choices made for each experiment. The tests to determine how well the genetic algorithm is performing has also evolved through the process, but the same type of data has been collected for almost all experiments. Mainly what has been evaluated is the fitness of the best individual per generation and how fast a solution is found on average.

All experiments are based on developing uniform 2 dimensional cellular automata. To be able to study systems with multiple cell types, and not just binary systems, 3 cell states are used so there is 2 states of living cells and one dead state.

Cellular automata configuration for all experiments:

Neighborhood type:	Von Neumann (size: 5)
Dimensions:	2
Grid edge handling:	Wrap around
Number of states:	3
Grid size:	4 by 4 (=16 cells)
Rule table (genome) size:	243 (3 states , 5 neighborhood)

The size of the cellular automaton used is 4 by 4 giving a total of 16 cells. The reason for the small grid size is that larger grids have higher trajectory lengths. This leads to phenotypes taking longer to develop, which is the most time consuming part of the genetic algorithm already.

5 by 5 grids are realistic, but they do not have any properties 4 by 4 also have. 6 by 6 grids will provide 4 individual neighborhoods, and therefore much more complex genomes. However, they are not plausible in C++ because of the time spent and not plausible in Java because the memory needed is too high to even know how long time developing one phenotype will take.

Every cellular automaton is initialized with a single cell of state 1 and the rest 0. The placement of the cell does not matter because wrap around is used.

The experiments test the genetic algorithm by running it for maximum 100 000 generations for a number of at least 20 runs and calculates an average of these runs. The last project ran it for 10 runs [21] due to the difficult memory management of C++ making 20 runs implausible. This was shown to be very imprecise, and 20 runs provide a big improvement.

The genetic algorithm has been run with both an empty population and a random one. It always searches for a given trajectory length, not attractor length. This is because the trajectory length is easier to validate and contains the attractor and the trajectory before reaching it.

In the descriptions of the experiments the search for a specific trajectory length by the genetic algorithm will be called a test on that trajectory length. When specifying number of runs this is for each individual test, not a total.

9.1. Experiment 1

The first experiment aimed to find flaws in the genetic algorithm used in the previous project [21]. The algorithm has not been used with an empty (initial) population before, and it was believed trying this would reveal weaknesses better than using a random population. This is because empty populations provide a constant starting point that can lead to a better average result.

Genetic algorithm configuration:

Mutation rate:	1/L (L is length of genome)
Crossover rate:	0.70
Initial population:	Empty
Population size:	10
Elitism:	2
New genomes per generation:	8

Experiment setup:

GA versions tested:	Standard, lambda in fitness, lambda to discard
Complexities tested:	1 000, 5 000, 10 000, 15 000, 20 000, 25 000
Number of runs:	20

The search for trajectory length 1000 is presented in Figure 7; it is an average plot of 20 runs. Fitness of the best individual is Y-axis and generation is on the X-axis. Fitness is inversely proportionate to the trajectory length, meaning this graph shows how far from the goal the best individual is. So a fitness of 0 is trajectory length 1000 which means the goal is reached.

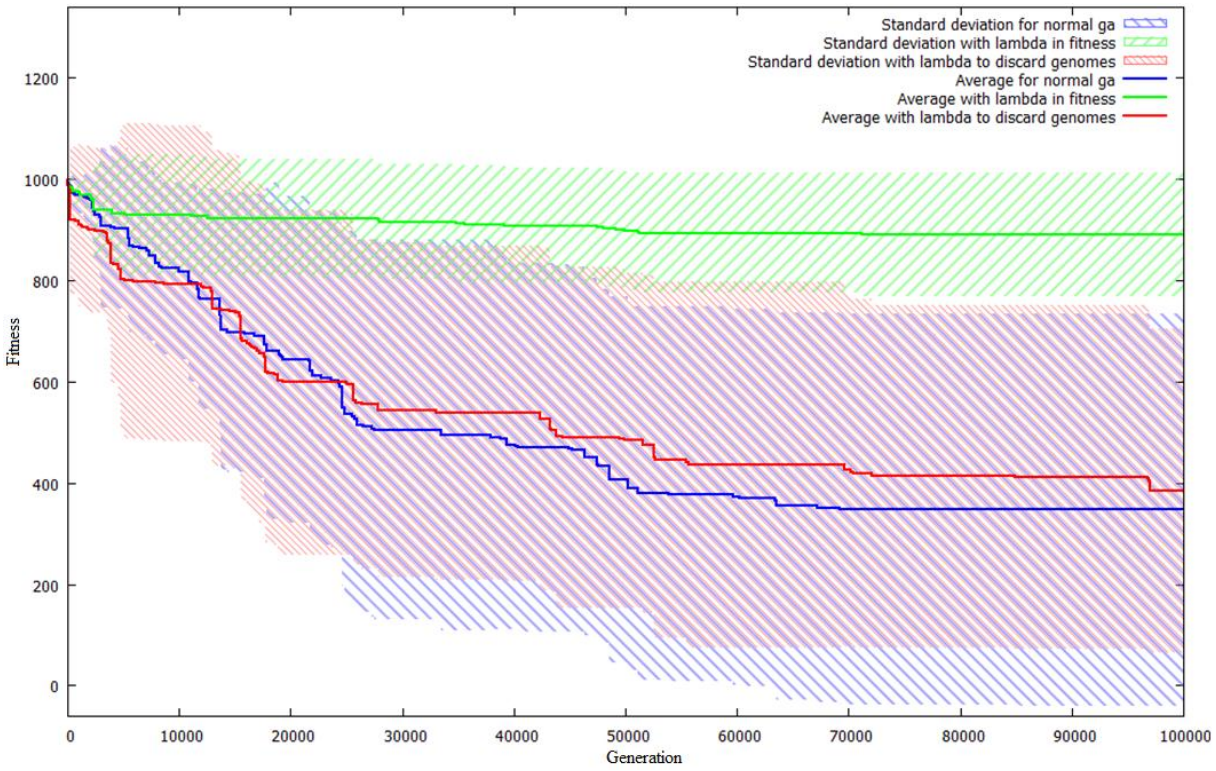

Figure 9: Trajectory length 1000, Experiment 1

Analysis:

Clearly something is wrong; the algorithm is struggling to find genomes with a trajectory length of 1000. The same result is repeated for all searches, no averages surpassed trajectory length 700.

Looking at the genetic algorithm configuration there are two values that may be too low. One is the mutation rate and the other is the elitism. More genomes need to be preserved between generations to make the population more diverse. Also when initializing with an empty population mutation adds the initial variation a random population already has, and thus making it very important initially, but less important as time goes on.

Lambda in fitness has poor performance compared to the two other versions. This is probably an indication that something is wrong, probably in the way lambda is added to the fitness.

9.2. Experiment 2

The issues from the former experiments were addressed in this experiment. Since tuning the genetic algorithm on parameters is outside the scope of this project only a few values were changed and the experiment has been very simple. The goal was to make the algorithm good enough, so later results will be comparable to algorithms used generally.

The genetic algorithm is the same as before, just a few variables have been changed. It is important to keep the number of new genomes per generation the same, so it is comparable to earlier tests. Increasing elitism size is therefore the only way to increase population size.

3 new configurations have been tested:

- Configuration 1 increases mutation rate to 2%
- Configuration 2 increases the elitism to 16 and population to 24.
- Configuration 3 combines 1 and 2.

Experiment setup:

GA versions tested:	Standard
Complexities tested:	1 000, 25 000
Number of runs:	10

The average plot for complexity test 25000 is presented with all configurations and the normal genetic algorithm as a reference. The plot configuration is the same as before.

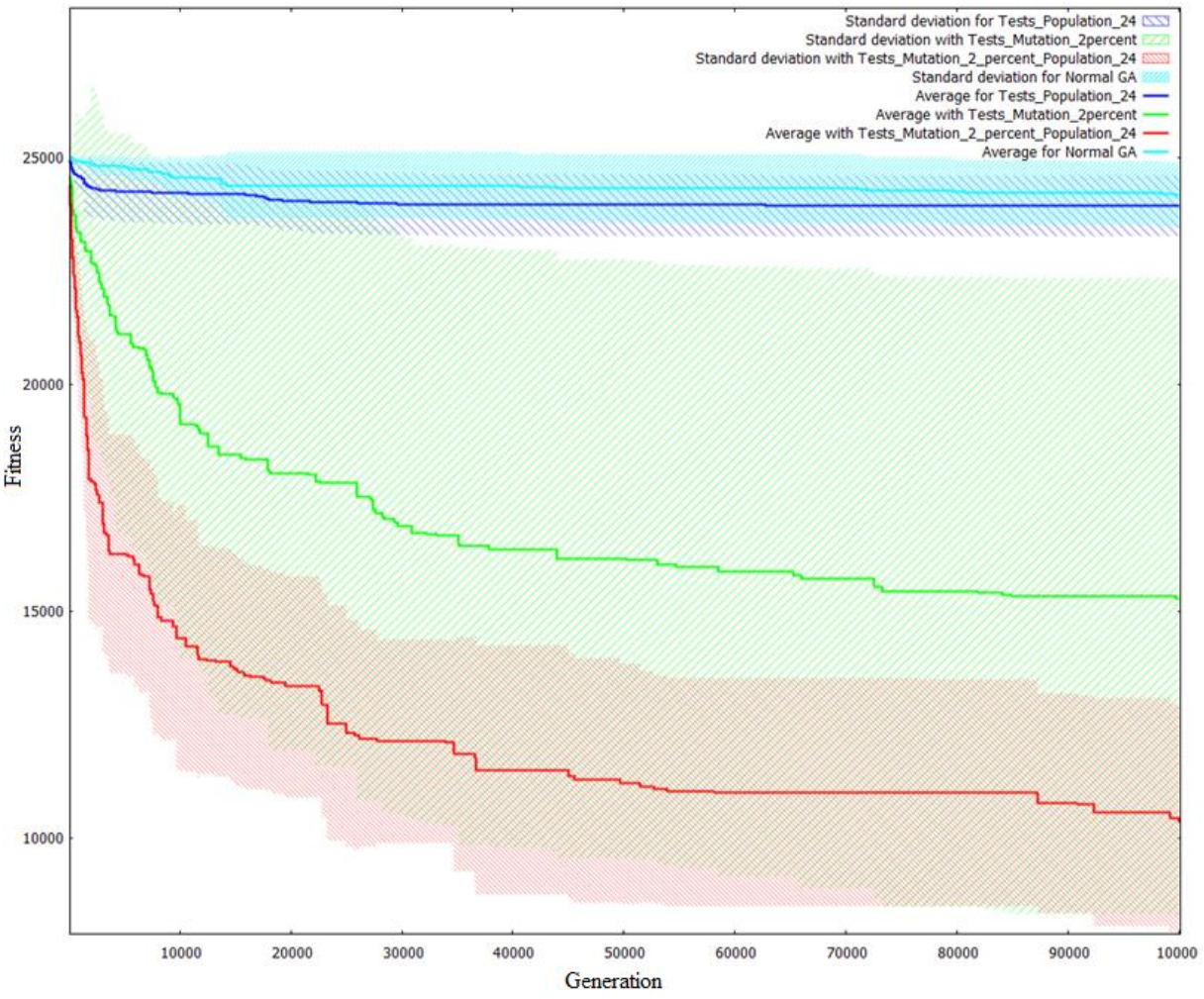

Figure 10: Complexity 25000, Experiment 2

Analysis:

This test shows clear differences between the versions despite the low number of runs. The most important change is to increase mutation rate, which is not surprising. Increasing population size alone does not improve performance by much. However, it improves the performance dramatically in combination with increased mutation rate compared to mutation rate alone. The choice is obviously Configuration 3, it converges fastest and the average result is also higher trajectory lengths.

For complexity test 1000 the results were much the same. The big difference was that configuration 1 could get stuck early resulting in a relatively high average. This is probably due to the lack of diversity in the population because of the low elitism/population size, causing the algorithm to get stuck in local maxima.

The result of this experiment is a better configuration for the genetic algorithm that makes it comparable to other implementations in general.

9.3. Experiment 3

This experiment recorded lambda and the genome usage together with the fitness of the genomes. The hope is to see some relation in the way the different values are evolving through a run of the genetic algorithm. Genome usage is interesting because as individuals get longer trajectories it is more likely that a bigger part of their genome is used. Perhaps this can be used the same way as lambda, since lambda also is expected to increase towards 0.66 as trajectory lengths increases.

Genetic algorithm configuration:

Mutation rate:	0.02
Crossover rate:	0.70
Initial population:	Empty
Population size:	24
Elitism:	16
New genomes per generation:	8

Experiment setup:

GA versions tested:	Standard
Complexities tested:	1 000, 5 000, 10 000, 15 000, 20 000, 25 000
Number of runs:	20

The plots show 20 individual runs for complexity test 25 000. Generation is still on the X-axis, and genome usage, lambda and fitness is on the Y-axis.

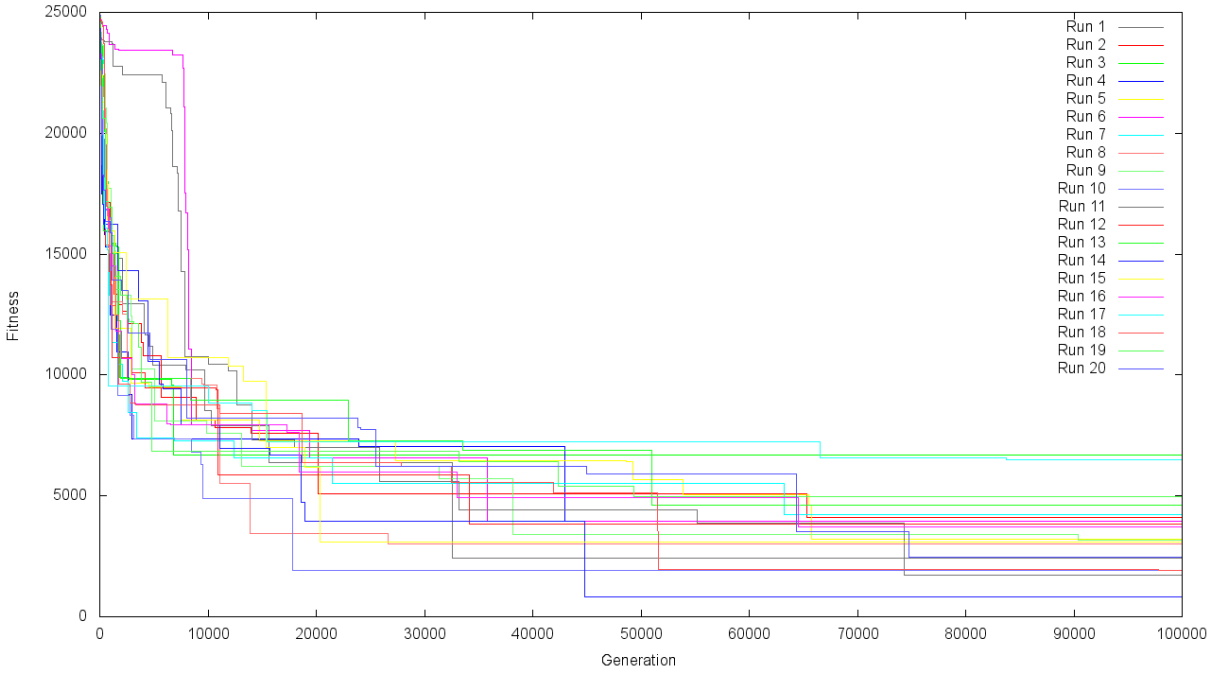

Figure 11: Fitness, Complexity 25000, Experiment 3

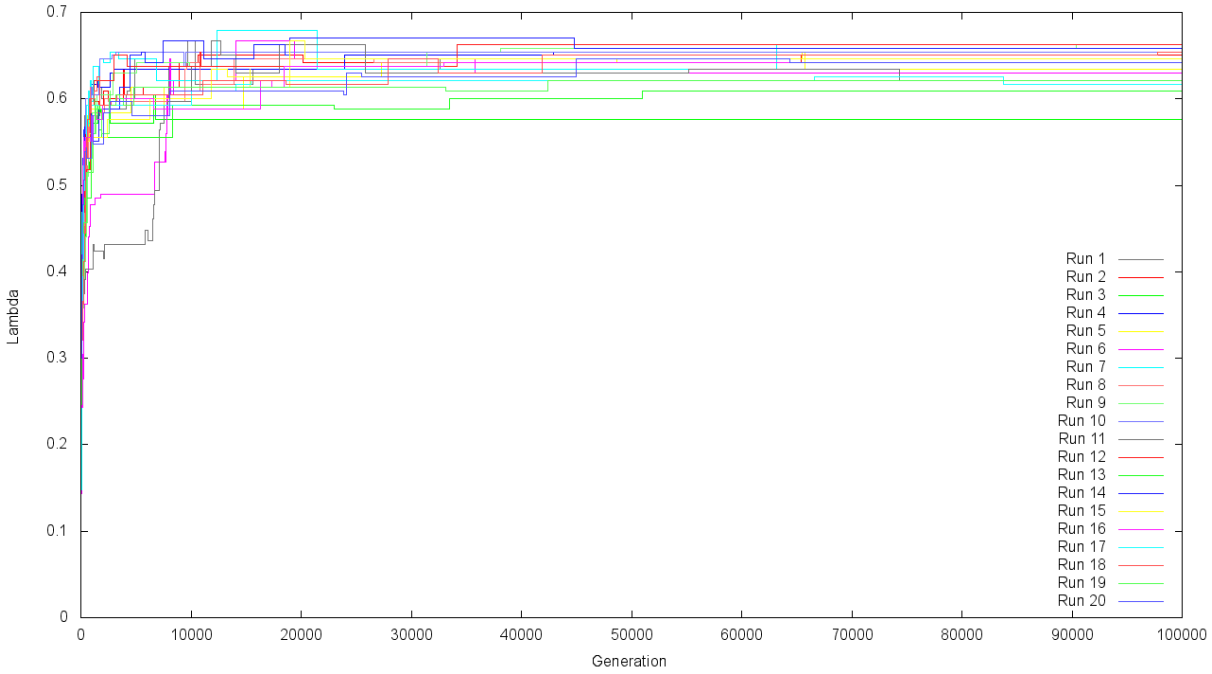

Figure 12: Lambda, Complexity 25000, Experiment 3

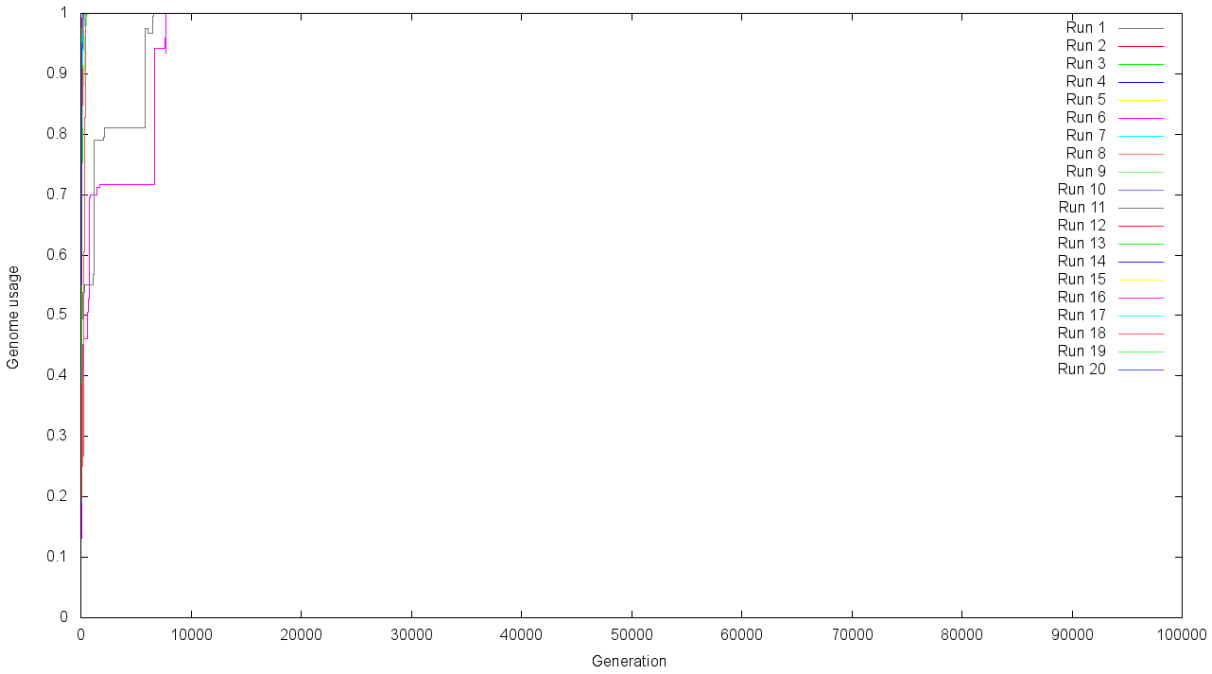

Figure 13: Genome Usage, Complexity 25000, Experiment 3

Analysis:

Plots of individual runs of all complexities were analyzed. But complexity test 25 000 provides the best example where two searches got stuck in a local maximum. The gray and purple runs got stuck until just before generation 10 000. This event is easy to identify in the plots of lambda and genome usage also. This suggests that there is some relation between lambda, fitness and genome usage.

Genome usage increases very quickly to 1, which may make it unsuited to be used the same way as lambda. But maybe it is usable to control mutation rate dynamically. While developing the phenotype, it is possible to analyze what parts of the genome are triggered. When mutating a genome these parts are the only ones that will affect the development of the phenotype. Therefore it could be possible to concentrate mutation here and quickly exploit the potential of the genome, this will be tested later.

Lambda increases to around 0.66 and it takes more generations to do so, compared to genome usage. This makes lambda more suited than genome usage, but still lambda seems to increase very fast. The algorithm may not get a lot of help from lambda past generation 10 000. However, the population as a whole may still benefit from using lambda.

9.4. Experiment 4

This experiment recorded the lambda and genome usage of the other genetic algorithm versions. This would hopefully show how the extensions perform compared to the standard genetic algorithm. The plots could also give a clue to why they perform as they do.

The genetic algorithm configuration is the same as for the previous experiment. The experiment setup is also the same, but lambda in fitness and lambda to discard is tested and compared to the standard version. Lambda to discard discards new individuals with lambda more different than 0.1 compared to the best individual of the previous generation.

The plots show the average of each set of runs for standard GA (blue), lambda in fitness (red) and lambda to discard (green). The strong colored line is the average and the two faint colored lines above and below it is the best and worst run. The colored field is two standard deviations in total.

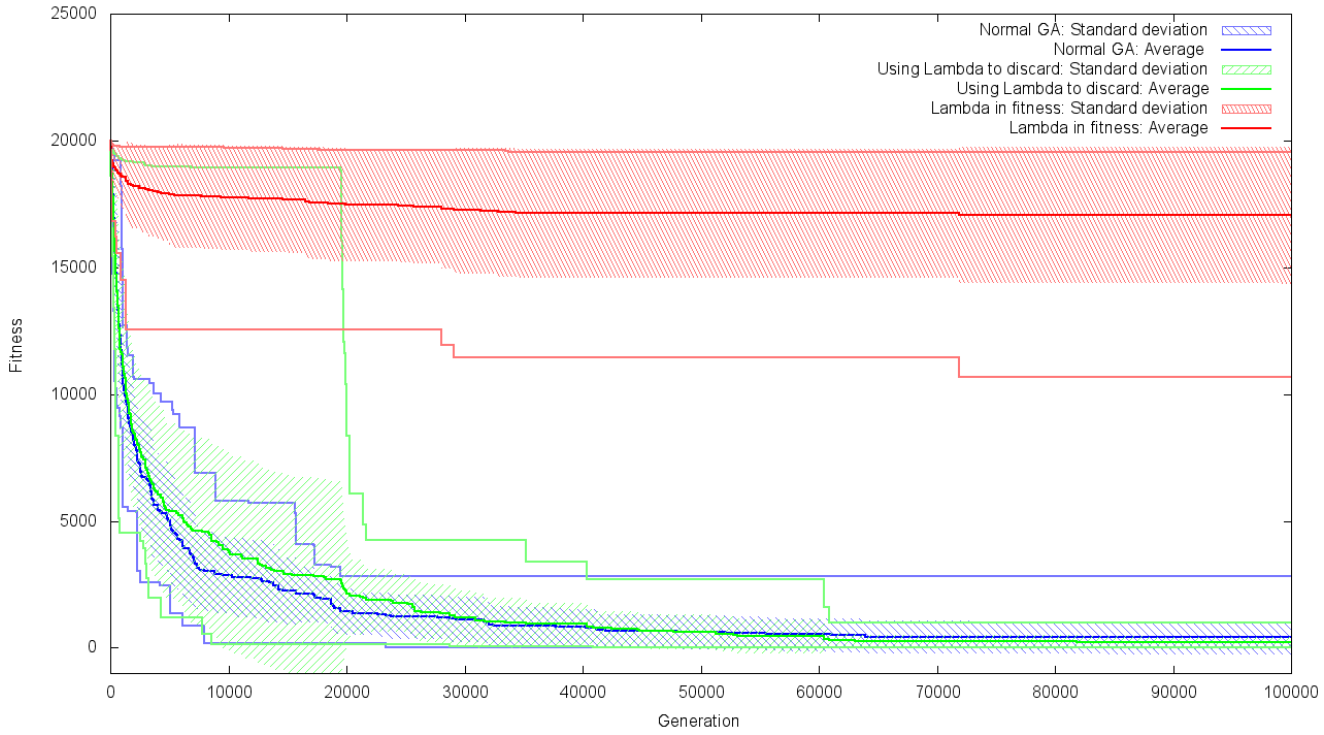

Figure 14: Fitness, Complexity 20 000, Experiment 4

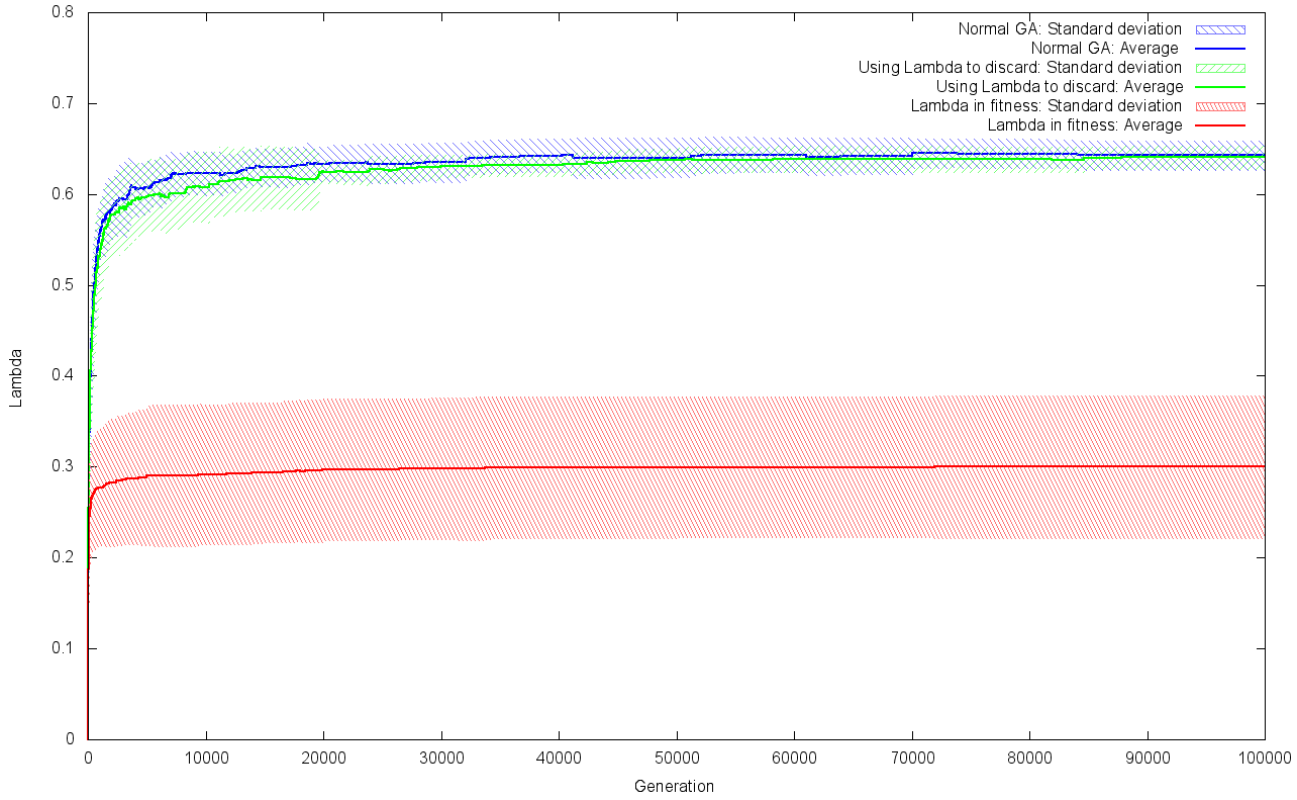

Figure 15: Lambda, Complexity 20 000, Experiment 4

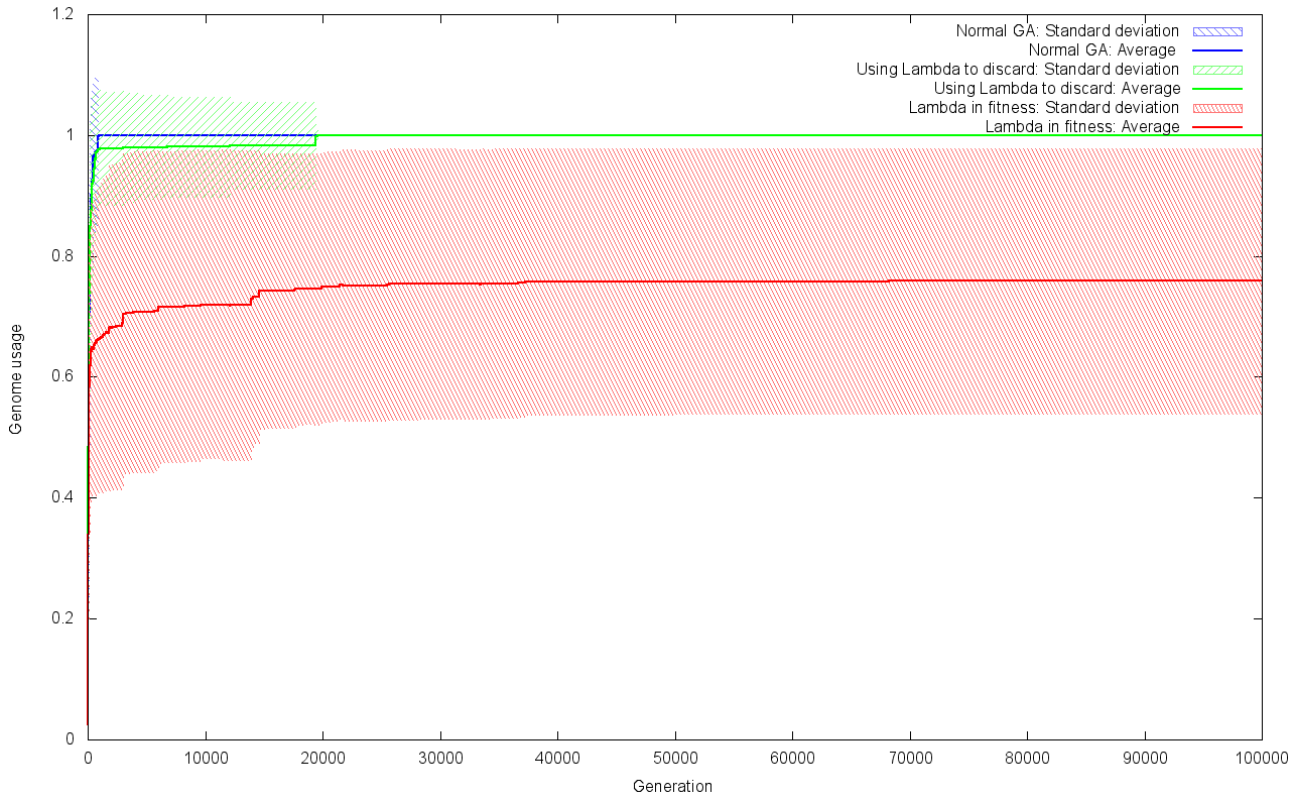

Figure 16: Genome Usage, Complexity 20 000, Experiment 4

Analysis:

Still it is obvious that lambda in fitness is inferior to the other methods. This is also shown in both the lambda and genome usage graphs. This suggests that something is wrong; the individuals cannot even exploit their whole genome after 100 000 generations. The reason is not clear from these results.

Lambda to discard performs more or less identically to the standard genetic algorithm on average. The lambda to discard limit of 0.1 seems too high, a difference, good or bad, should be noticed on the plots. The problem was that if the limit is too low the algorithm gets trapped in an eternal loop. It may be possible to tune lambda to discard to see an effect.

Mutation rate is related to how much genomes change per generation and therefore the discard limit could also be related to this. A good mutation rate should therefore be found first and then the discard limit should be tuned with it.

9.5. Experiment 5

Lambda, genome usage and trajectory length seems to be related from the results in the previous experiments. The experiments recording lambda and genome usage consists of a total of 360 genetic algorithm runs. This should provide a good base to calculate a correlation and establish the relations more formally.

The Pearson correlation coefficient was calculated for each run and the average of 360 runs were found. Also the worst and best correlating runs are shown. The correlation does not distinguish which variable is first and last, so only 3 combinations needed to be tried.

The results of all 360 runs:

Components	Average correlation	Best correlation	Worst Correlation
Genome usage and fitness	-0.67	-0.99	-0.45
Lambda and fitness	-0.83	-0.99	-0.14
Lambda and Genome usage	0.78	0.99	0.2

Table 2: Correlation coefficients, Experiment 5

The negative values are due to the fitness being inversely proportionate to trajectory length. The sign does not matter; it is the correlation coefficient that matters.

Genome usage quickly grows to 1 and the trajectory length keeps increasing after genome usage reaches maximum. Therefore the correlation for the first 1000 generations was measured. This is when fitness grows the most, and where genome usage could be useful to control mutation.

The results of the 1000 first generations of all runs:

Components	Average correlation	Best correlation	Worst correlation
Genome usage and fitness	-0.79	-0.99	-0.39
Lambda and Genome usage	0.91	0.99	0.63

Table 3: Correlation coefficients 1000 first generations, Experiment 5

Analysis:

Lambda correlates highly with both genome usage and trajectory length. Genome usage also has a good correlation with fitness. Especially the first 1000 generations genome usage has a high correlation with fitness and lambda.

This supports what have been shown before, lambda is indicative to the trajectory length of an individual. Genome usage can possibly be useful early in development to control mutation.

The results also suggest that the search follows some curve related to the plot of Nichele and Tufte seen in Figure 3: Plot of trajectory length and lambda [15].Lambda tells something about the expected upper bound of trajectory length that is possible to find. The genetic algorithm always pushes the top of this limit and is forced towards higher lambda values.

9.6. Experiment 6

This experiment attempted to find out how much time is spent recalculating the fitness of individuals. The way the genetic algorithm works, makes this only possible if the exact same genome is created in two different generations.

The way it is measured is that a hash map is used in the same way as for the fitness calculation to store and search for genomes. The keys are genomes and the values are the count of how often they have been calculated. For each generation the sum of all recalculations are recorded.

The purpose was to see if it could be used as a complementing measure of performance, because this number is hidden in previous data. If the number of recalculations is high, specific measures should be taken to reduce them, and maybe genome parameters can be used.

Experiment setup:

GA versions tested:	Standard
Complexities tested:	1 000, 10 000, 25 000
Number of runs:	50

The plot of complexity test 25 000 shows the cumulative count of genomes that have gotten their fitness calculated more than one time on the y axis. The x axis is the generations.

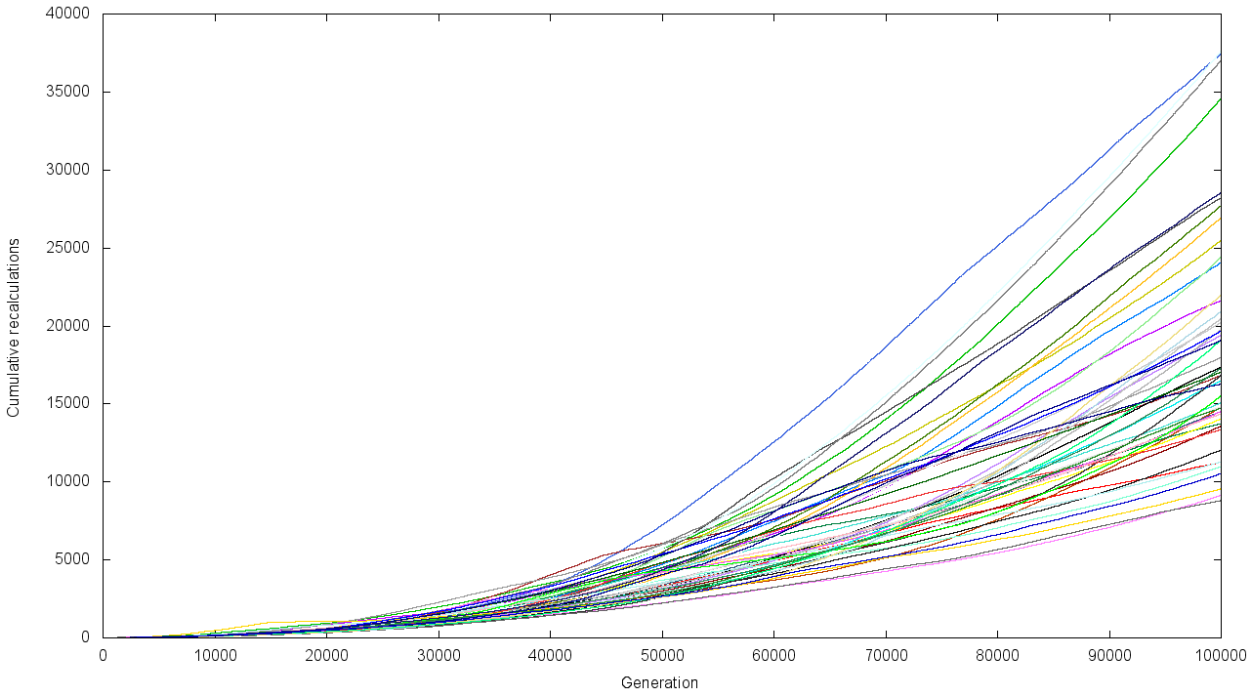

Figure 17: Cumulative reused genomes per generation, Experiment 6

Analysis:

The graphs look similar for all complexities, though the scale relies on number of generations rather than the complexity searched for. There are some redundant calculations done, and the growth is exponential.

This suggests that running a genetic algorithm for too long may not be very effective. The alternative is restarting when efficiency measured by number of recalculations per round drops below a certain point. This is however outside the scope of the project.

Worst case is about 37 000 recalculations after 100 000 generations. For each generation 8 new individuals are calculated. This means that 37 000 of 800 000 individuals have already existed, which is 4.6% recalculations. The best case for complexity test 25 000 is about 9000 recalculations which gives 1.1% recalculations. These are average numbers of a whole run,

and the last half is much worse compared to the first. The recalculation rate is not very high, and it is probably not useful to aim to decrease number of recalculations at this time.

9.7. Experiment 7

The goal of this experiment was finding a good mutation rate to use with lambda to discard. At the same time lambda would be analyzed to see what effect the mutation rate has on the lambda development. Given the previous experiments, lambda should converge faster the same way as fitness.

Experiment setup:

GA versions tested:	Standard
Complexities tested:	15 000
Number of runs:	20
Mutation rates tested:	0.24 to 0.01 with step 0.01

The genetic algorithm configuration is the same as before, but the mutation rate was varied between 0.24 and 0.01. The steps were 0.01, so there was a total of 24 runs.

The plot shows all 24 runs in the same configuration as before with lambda and trajectory length on two different plots.

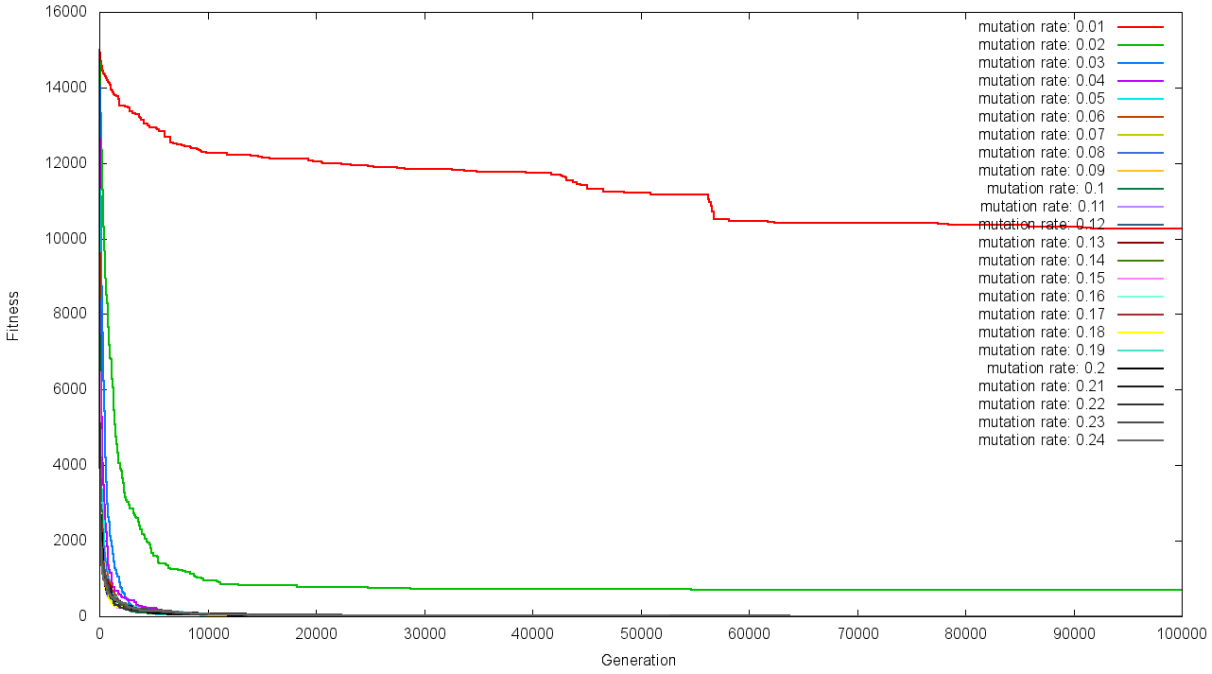

Figure 18: Fitness, Complexity 15 000, Experiment 7

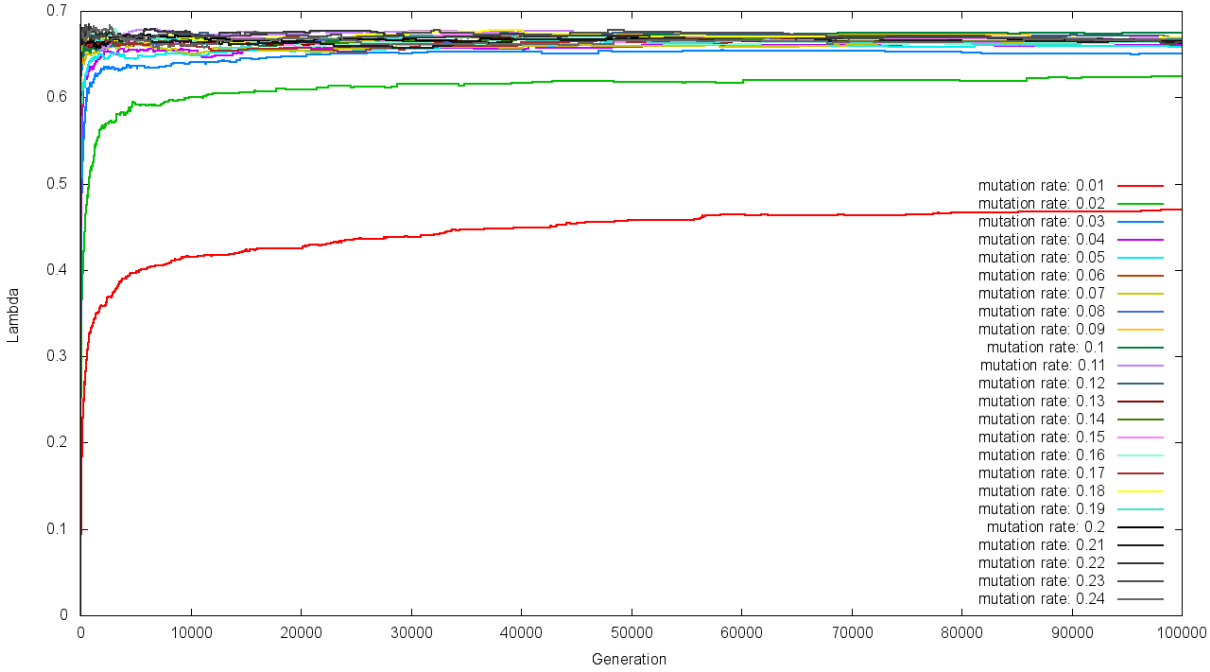

Figure 19: Lambda, Complexity 15 000, Experiment 7

Analysis:

The most interesting result is that the mutation rate used up until now, 0.02, performs noticeably worse compared to higher ones. A mutation rate of 0.03 and 0.04 is closer to the rest but still it is easy to see that the plots for them converge slower than the higher ones.

The mutation rate should not be too high, because one needs to control the development of the GA so it performs better than random. These results suggest that the genetic algorithm currently performs worse than that. Increasing the mutation rate to 0.05 will make the performance much better, without making the mutation rate too high.

The effect on lambda value is expected, and further illustrates its relation to trajectory length in the genetic algorithm. Versions with a low average trajectory length also have a low average lambda. Higher mutation rate makes genomes change faster, and gene types become more randomly distributed within the genome. This supports the idea about lambda; that more evenly distributed genomes produce more complex phenotypes.

9.8. Experiment 8

Now that a good mutation rate has been established, the genetic algorithm has been configured with 5% mutation rate. For this experiment the discard limit in lambda to discard has been tuned. The goal has been to see if this makes this version perform better than the standard genetic algorithm. It was though that the mutation rate and the discard limit should be similar, because the mutation rate decides how much the genome changes.

Discard limits up to 0.1 was tested first, but the results was not promising, so higher values were tried. Last experiment used complexity test 15 000 which was a bit low, which made it hard to read. This test therefore uses a higher complexity.

The test had to be configured with a way to detect eternal loops that happens when the discard value was too low to allow any individuals through to the new generation. The choice was to set a hard limit of 100 000 discards before stopping the algorithm at that complexity. This will make it clearly visible in the plots if the algorithm gets stuck often.

Experiment setup:

GA versions tested:	Lambda to discard
Complexities tested:	25 000
Number of runs:	20
Discard limits tested:	0.02, 0.03, 0.04, 0.05, 0.1, 0.15, 0.25, 0.35, 0.55 and 1.0 for reference

The plots show the fitness per generation, and the standard deviation (Y-axis) per generation (X-axis) for the corresponding fitness plots.

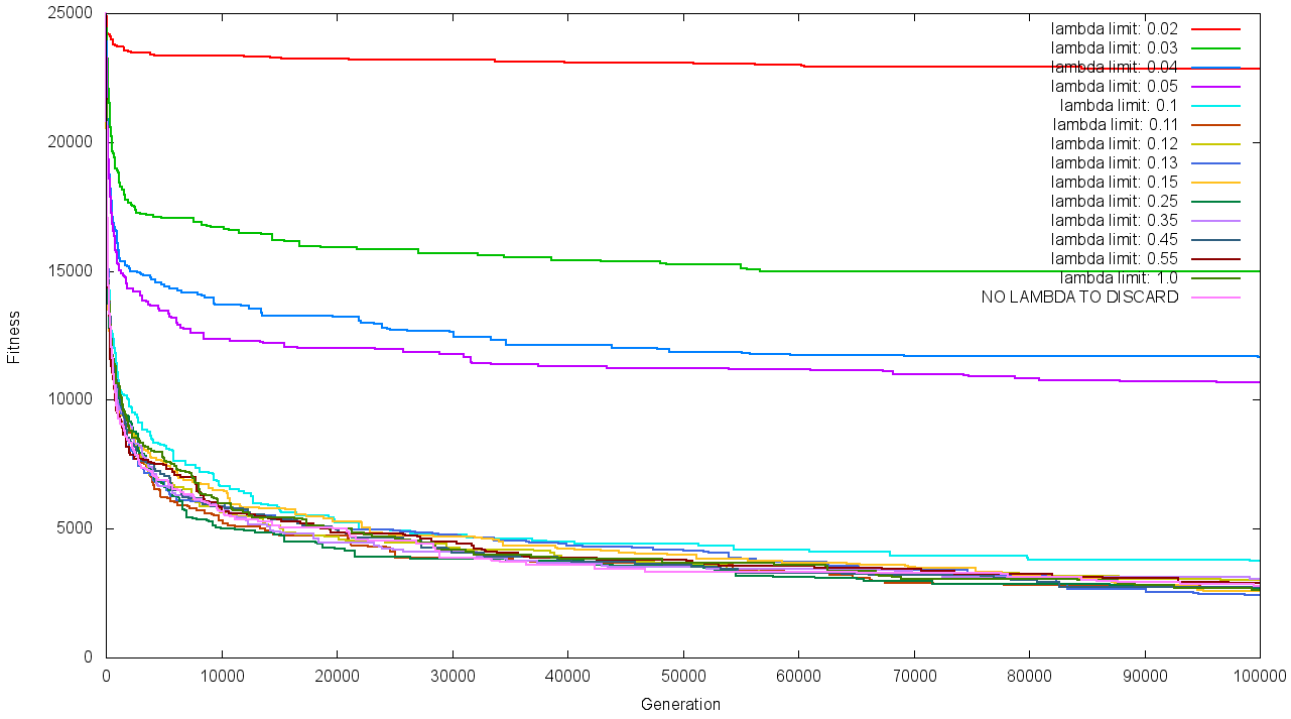

Figure 20: Fitness, Complexity 25 000, Experiment 8

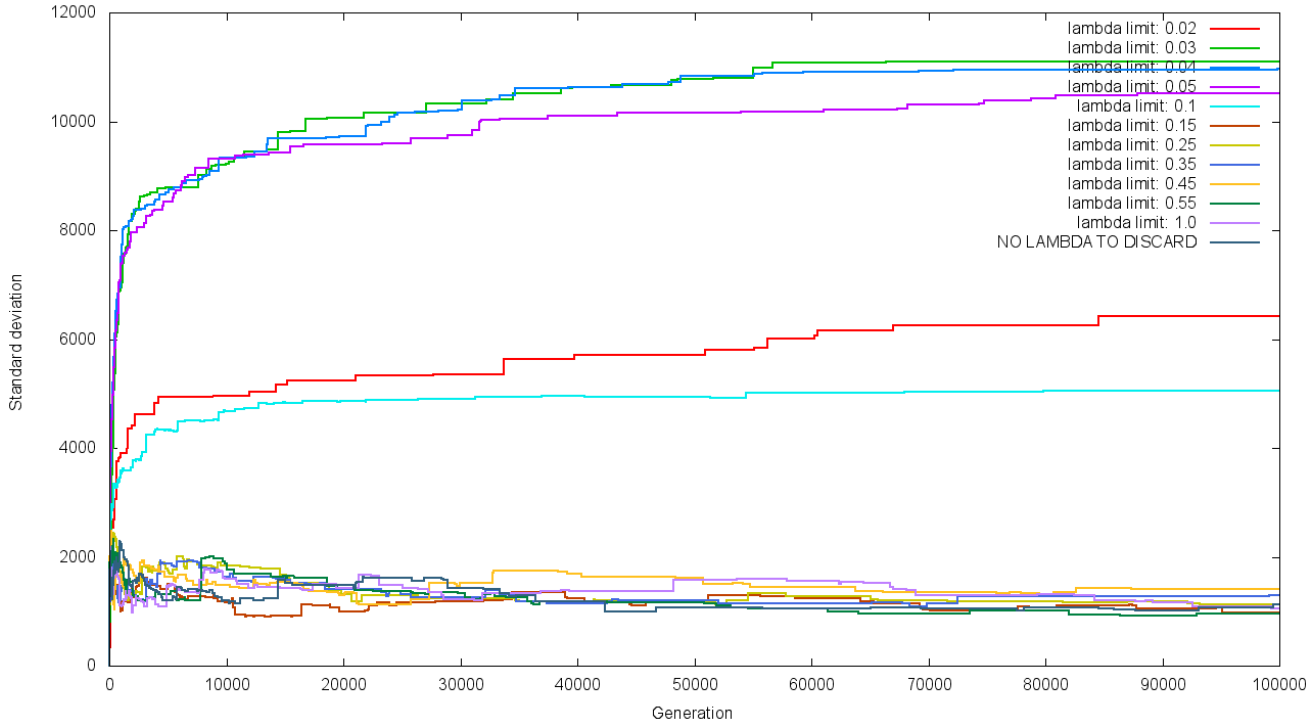

Figure 21: Standard deviation, Complexity 25 000, Experiment 8

Analysis:

The performance of the genetic algorithm get better the larger the discard limit is. When reaching 0.1 the difference in performance gets marginally different from the standard genetic algorithm, and most likely this is because fewer genomes gets discarded. This suggests that discarding genomes does nothing positive for the performance.

The standard deviation is also higher for lower limits, even for 0.1, but for higher limits it is very low. It is lower for 0.02 because the results are more consistently bad. And it is very high for 0.03, 0.04 and 0.05 because the results are either good or very bad. This suggests that lambda to discard destabilizes the algorithm and give less consistent results.

The discard limit and mutation rate is not as closely related as first assumed. There are more factors to be counted in. First of all the crossover happens 70% of the time making the genomes very different from each parent. Also the mutation rate changes the genome, which may or may not change lambda much, depending on which genes are changed.

To discard genomes seems to only be hurting the search by limiting the width of the search and creating a less diverse population. That or entering an infinite loop, makes the results of each run vary greatly.

9.9. Experiment 9

Because of the poor performance of lambda to discard, this experiment aims to find the reason for this. The difference between the lambda value of the individuals and the best lambda value, the value used to discard genomes, was recorded. This will show how many genomes would be discarded per run and also reveal the best discard limit.

To do this the limit was put into an array of counters with granularity of 0.01, which persisted over several runs.

Experiment setup:

GA versions tested:	Standard
Complexities tested:	25 000
Number of runs:	10
Mutation rates tested:	0.05, 0.1, 0.25

The plot shows boxes for each 0.01 step of lambda limits of newly created individuals. The values are accumulated in 10 runs of the genetic algorithm. The lambda limits is on the X-axis and the accumulated count of individuals is on the Y-axis.

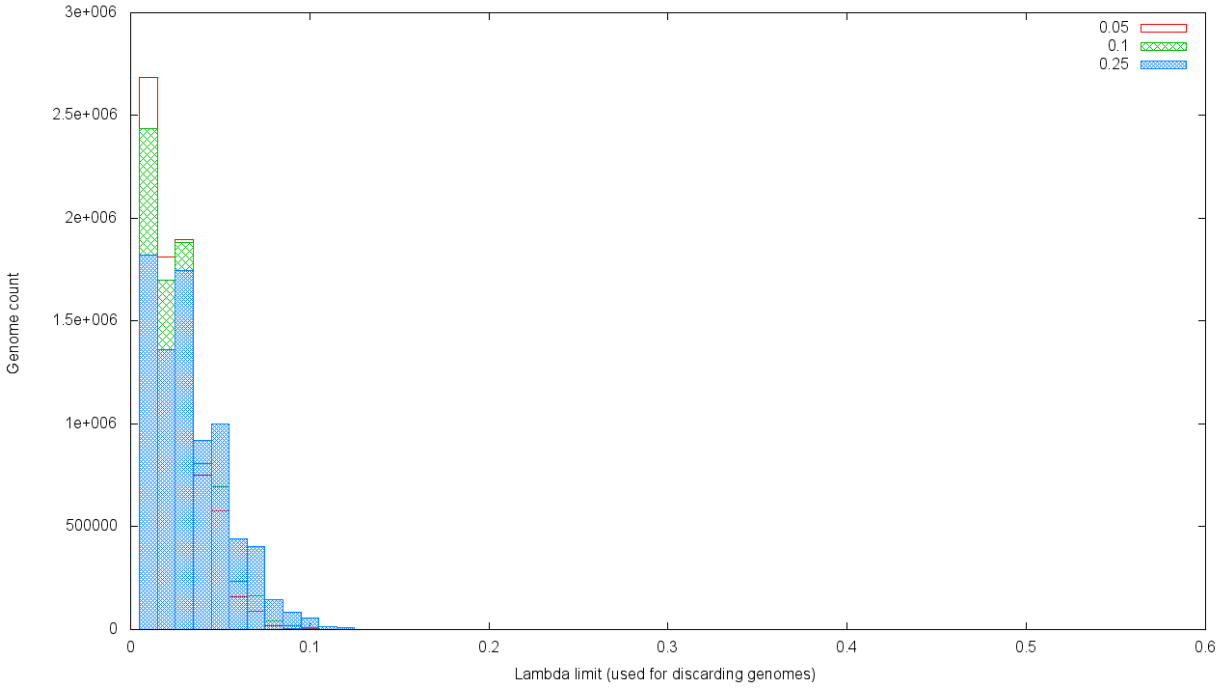

Figure 22: Accumulated lambda differences, Experiment 9

To get a better sense of scale the 0.1 point represents 58127 for mutation rate 0.25(blue). For 0.05 this was 3068 and it is not visible.

Analysis:

The mutation rate used previously was 0.05, and the discard limit where the performance of the algorithm came close to the standard genetic algorithm was 0.1. In 10 runs 3068 genomes were found with this limit.

In total 3940 genomes were found to have difference of 0.1 or bigger compared to the best lambda value. In average this is 394 per run. This means that with a discard limit of 0.1 394 of 800 000 individuals were discarded which is about 0.05%. This is so small that the effects are negligible.

It is now possible to conclude that this version of lambda to discard cannot improve the genetic algorithm. There may be other ways to use it, like using parents to calculate relative lambda instead of the best genome of the previous generation. But it seems unlikely that this will have any positive effect as it will still be limiting the search in the same way.

9.10. Experiment 10

In this experiment the old version of lambda in fitness was tested. The weight between lambda and fitness value have always been 50-50 up until now. The goal was to see if reducing the emphasis on lambda value would result in an algorithm performing better than the standard genetic algorithm.

The ratios given are how much lambda counts in the fitness function. Trajectory length then has the inverse ratio of 1 – lambda ratio.

Experiment setup:

GA versions tested:	Lambda in fitness
Complexities tested:	25 000
Number of runs:	20
Lambda ratios tested:	0.5, 0.4, 0.3, 0.2, 0.1

The plot shows the lambda in fitness algorithm compared to the standard one.

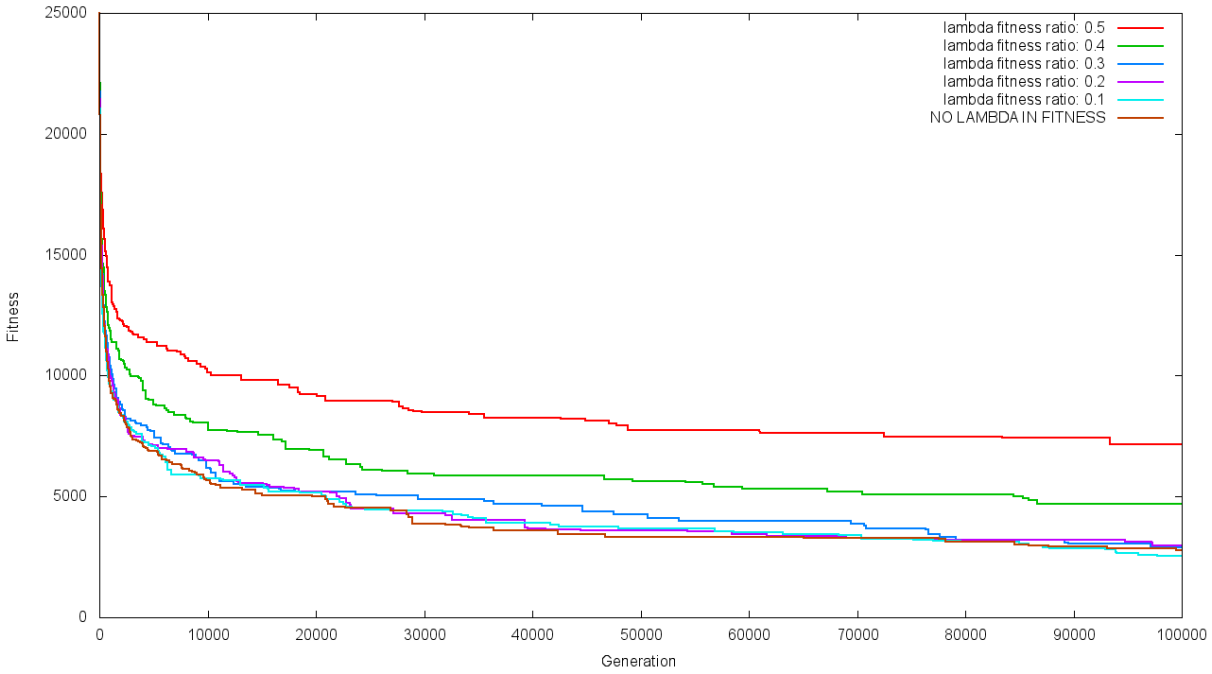

Figure 23: Fitness, Complexity 25 000, Experiment 10

Analysis:

Lambda in fitness performs much better than before with the new mutation rate, but it is never better than the standard genetic algorithm. In the same way as lambda to discard it closes in on the standard version as the lambda emphasis gets smaller. This version does not improve on the genetic algorithm.

9.11. Experiment 11

To shorten the time for experiments a different approach has been tried. A higher amount of runs searching for shorter trajectory lengths provides more precise results, and uses less time. This approach also allows to measure performance as the average generations it may take the genetic algorithm to find genomes.

This is useful because even if the algorithm finds high complexities fast, it may be slow at finding the desired properties. In these experiments this is illustrated by finding an exact trajectory length.

The genetic algorithm is configured the same way as before. Except now both random and empty populations were tested. Due to the high amount of runs the initial population does not affect the average as much.

Experiment setup:

GA versions tested:	Standard, Lambda in fitness , Lambda to discard
Complexities tested:	100, 1 000
Number of runs:	1000
Discard limit:	0.1
Lambda ratio in fitness:	0.3, 0.1

The plot shows a zoomed in view of the average of 1000 runs at complexity test 1000.

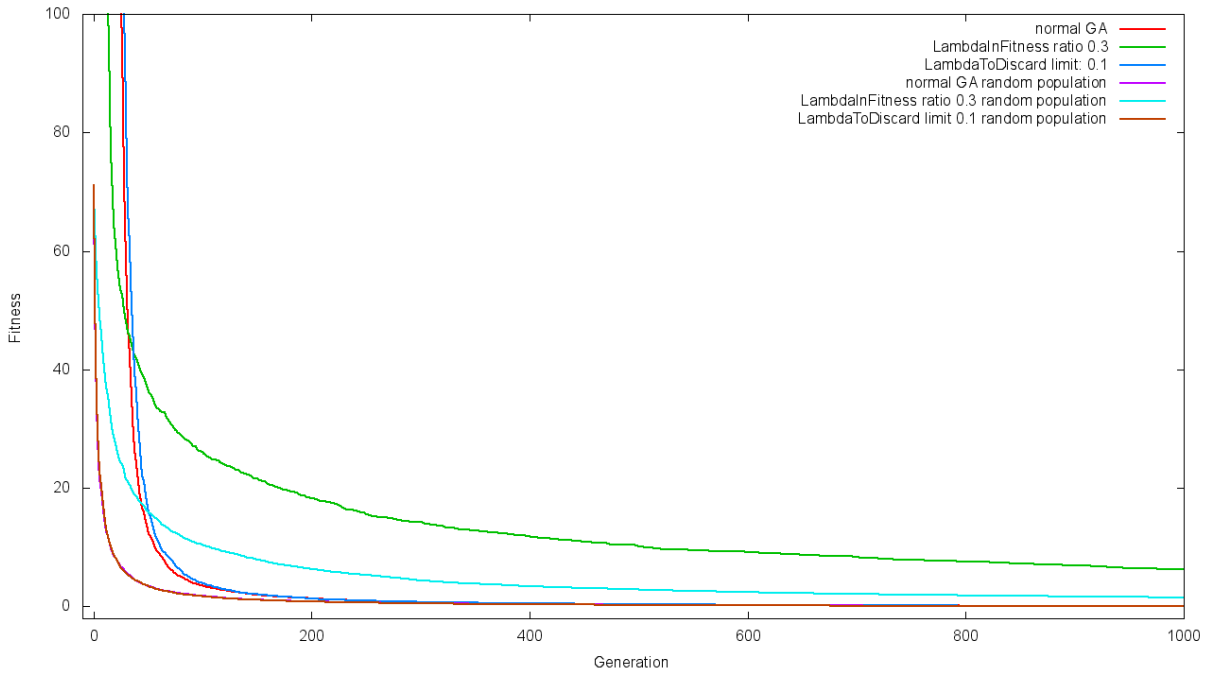

Figure 24: Fitness, Complexity 1000, Experiment 11

Average generation the algorithms found genome of trajectory length 100.

GA version	Random population	Empty Population
Standard	112	42
Lambda to discard	104	45
Lambda in fitness	806	4066

Table 4: Average generation genome of traj len 100 found

Average generation the algorithms found genome of trajectory length 1000.

GA version	Random population	Empty Population
Standard	371	433
Lambda to discard	358	443
Lambda in fitness	3347	16077

Table 5: Average generation genome of traj len 1000 found

Analysis:

The results confirm what have been shown in the 25000 complexity tests. Lambda in fitness clearly underperforms here as well. An extra test where the ratio is 0.1 was tried, but showed no improvements past the standard genetic algorithms. Lambda to discard however shows a small improvement for random populations. It does not converge any faster than the standard version but it finds genomes slightly earlier on average. This difference is small, but needs to be examined further.

Random population performs better than empty population, which was expected. But for complexity test 100 it performs worse. This is probably because an empty population is closer to 100 than a random population will be on average.

9.12. Experiment 12

After the last experiment showed lambda to discard showed a slight improvement compared to the standard genetic algorithm this had to be tested further. These experiments runs several tests with different discard limits to see if lambda to discard can consistently perform better than the standard genetic algorithm. The tests were done on a random initial population.

One difference is made here, and that is that lambda to discard continues running if an infinite loop is detected. It simply gives up and lets the genome through even if it is outside the limit. The idea was that this may provide all the benefits of lambda to discard and ignore the limit if it does not help the algorithm (like in the beginning).

Experiment setup:

GA versions tested:	Lambda to discard
Complexities tested:	100
Number of runs:	1000
Discard limits tested:	0.02, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08, 0.09, 0.1, 0.11, 0.12

The plots did not provide interesting knowledge but a table with the average generations genomes were found is presented:

Discard ratio	Random population
Standard GA	112
0.02	105
0.03	103
0.04	104
0.05	104
0.06	111
0.07	109

0.08	108
0.09	104
0.1	104
0.11	105
0.12	107

Table 6: Average generation genome found tested on different discard limits

Analysis:

The tuning could not get any improvements, and the results are very close to each other. The performance is marginally better than standard version, but too close to say it is an improvement. It seems that for limits smaller than 0.06 the limit is too small to let any genomes through to the next generation. The new mechanism of just giving up then shows here because the performance of 0.05 is suddenly better.

9.13. Experiment 13

In this experiment genome usage statistics have been used to control mutation dynamically. This mutation can only be applied to genomes that has not undergone crossover, because data from the last fitness test is needed. The crossover rate is 0.70 which means that on average only 30% of new genomes are mutated this way

The mutations of the genomes are controlled using the gathered genome usage data. Only the parts of the genome that has been used in the last run will be mutated. The mutation rate for this part will be 1 – genome usage rate. The effect of this is that when there are few genes in a genome to mutate the chance of mutating each one is very high, but when the number of used genes in the genome increases the chance of mutating individual genes decreases. When all of the genome is used, no extra mutation is happening.

Experiment setup:

GA versions tested:	Genome usage
Complexities tested:	25000, 1000, 100
Number of runs:	20 for 25000, 1000 for others

The plot is zoomed in and show comparisons to the normal GA with random and empty initial populations for complexity test 1000. For the 25 000 test the plots were very similar and not very interesting.

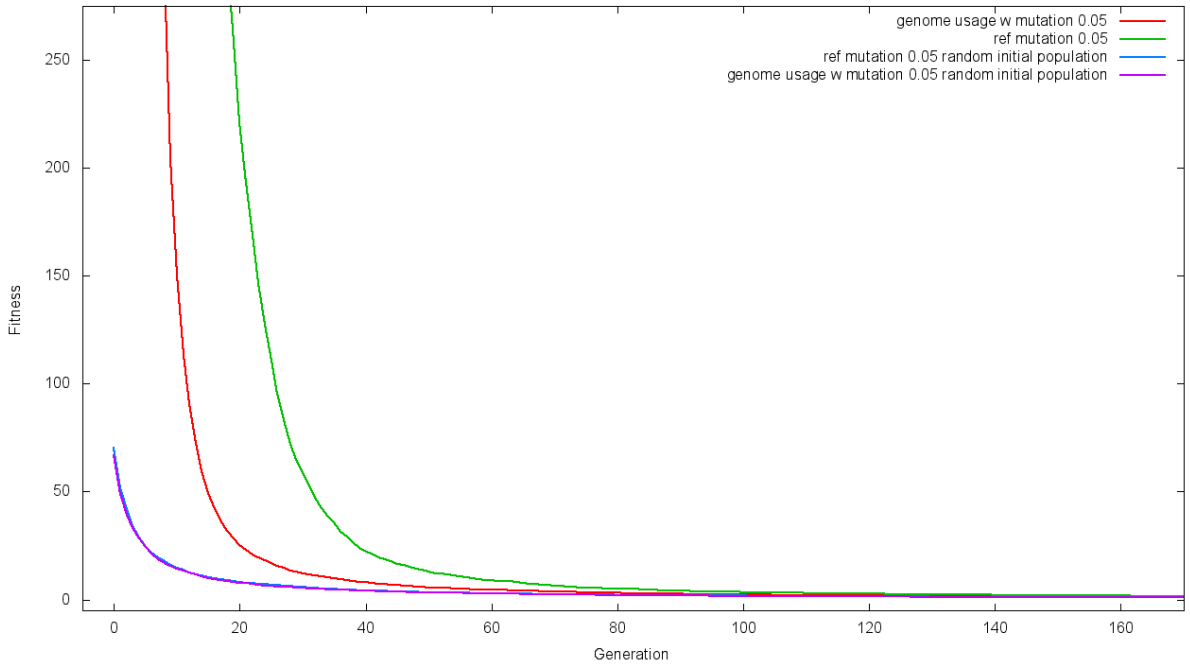

Figure 25: Zoomed in fitness, Complexity 1000, Experiment 13

Average generation the algorithms found genome of trajectory length 100.

GA version	Random population	Empty Population
Standard	112	42
Genome usage	127	50

Table 7: Average generation genome of traj len 100 found

Average generation the algorithms found genome of trajectory length 1000.

GA version	Random population	Empty Population
Standard	371	433
Genome usage	340	397

Table 8: Average generation genome of traj len 1000 found

Analysis:

The plot shows that genome usage makes the algorithm converge much faster than the standard genetic algorithm. The tables show that it performs better than standard algorithm for complexity test 1000 and worse for complexity test 100.

Random population also performs in the same way, better on 1000 and worse on 100. They seem to be related in the way that both provide variation to the population in early stages. Yet they are not the same, because it seems that genome usage improves performance even when population is random. This may be because a random genome does not guarantee that the whole genome is used, but mutating it with genome usage does.

9.14. Experiment 14

After the last experiment it was speculated on how the genome usage algorithm performs with different mutation rates. Mainly it was believed that the mutation rate could be lower and the algorithm would perform better. This way the claim that a too high mutation rate is bad could also be tested. The experiment was split into two parts; the high mutation rates and low mutation rates.

Experiment setup:

GA versions tested:	Standard, Genome usage
Complexities tested:	5000
Number of runs:	50
Mutation rates tested:	0.05, 0.1, 0.25, 0.01, 0.02, 0.03, 0.04

The plot shows the low mutation rates zoomed in, only empty population was tested.

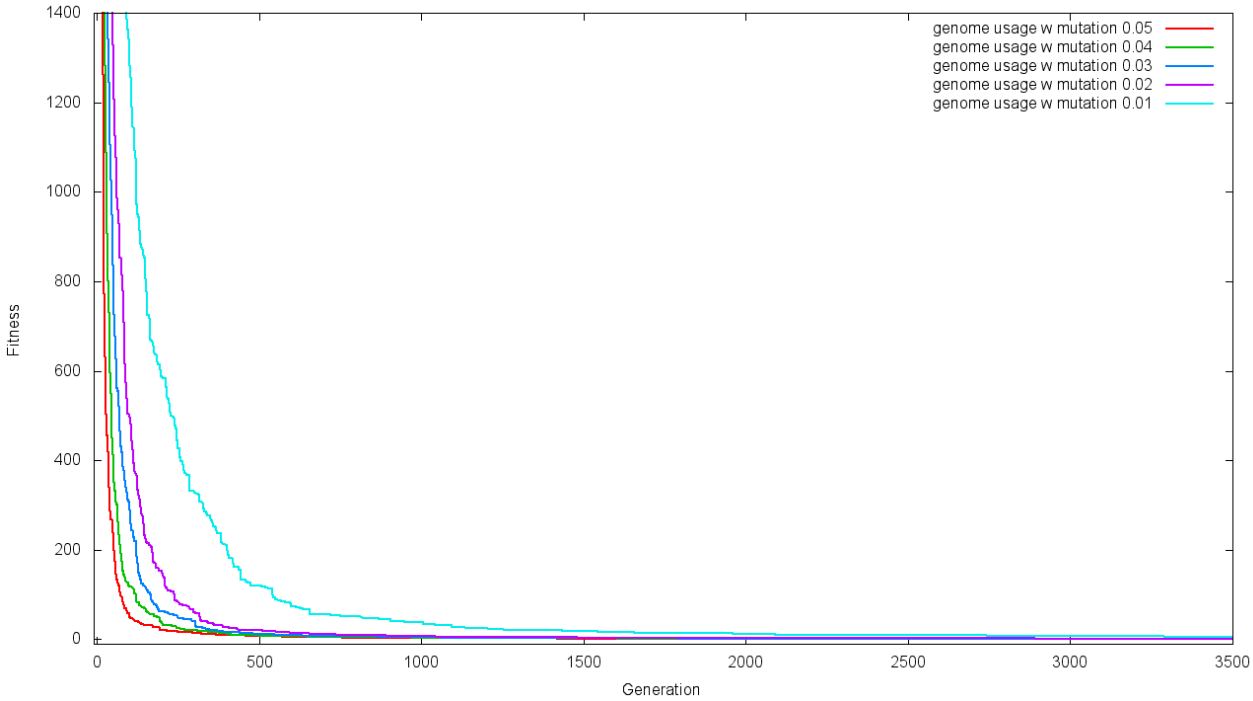

Figure 26: Fitness, Complexity 5 000, Experiment 14

Mutation rate	Genome usage algorithm	Standard GA
0.05(standard)	4866	6267
0.1	5682	5841
0.25	7228	6072

Table 9: Average generation the algorithm found genomes, higher mutation rates.

Mutation rate	Genome usage algorithm
0.05(standard)	4866
0.04	4777
0.03	5623
0.02	7402
0.01	14559

Table 10: Average generation that the algorithm found genomes, lower mutation rates.

Analysis:

The genome usage improves the genetic algorithm when mutation rate is around 0.05 and below. Higher mutation rates make it harder to find genomes even if the search converges faster. Genome usage makes the search converge fast, but does not make it harder to find genomes. It improves on lower mutation rates versions of the genetic algorithm, but the best performance is found around 0.05.

9.15. Experiment 15

At first this experiment tried to test the old lambda in fitness with using lambda value directly, simply trying to get it as high as possible. During this work it was discovered that the old lambda in fitness did not provide consistent fitness values for roulette wheel selection. Same fitness value and lambda value would be combined to different values depending on the rest of the population.

To change this lambda in fitness was simplified; lambda and fitness were to be combined inside the fitness function before preparing the population the same way as before.

Now lambda in fitness cannot be given a definitive ratio, only a variable to tune the weight. A few variations of this lambda in fitness were tested to find the one which provides an advantage over standard genetic algorithm. They use two different types of relative lambda values, high and relative to best, and two types of punishments, relative and absolute.

These 4 versions were tested:

- HiLambda is the lambda value where the highest trajectory lengths are found
- bestLambda is the lambda value relative to the lambda of the previously best individual

Close to best, relative:

$$Combined\ Fitness = Fitness + Fitness \times Abs(Lambda - bestLambda) \times ratio$$

Close to best, absolute:

$$Combined\ Fitness = Fitness + TargetComplexity \times Abs(Lambda - bestLambda) \times ratio$$

High, relative

$$Combined\ Fitness = Fitness + Fitness \times \frac{Abs(HiLambda - lambda)}{HiLambda} \times ratio$$

High, absolute:

$$Combined\ Fitness = Fitness + TargetComplexity \times \frac{Abs(HiLambda - lambda)}{HiLambda} \times ratio$$

The challenge is finding a fitting ratio which will be unique for each version.

First a series of complexity test 1000 and 100 runs were done to roughly determine the best ratios of each version. Then a series of complexity tests 1000 and 1000 runs were done to find a good specific ratio. The best results of the last tests for random and empty populations are shown in the table below. The plot shows the same results.

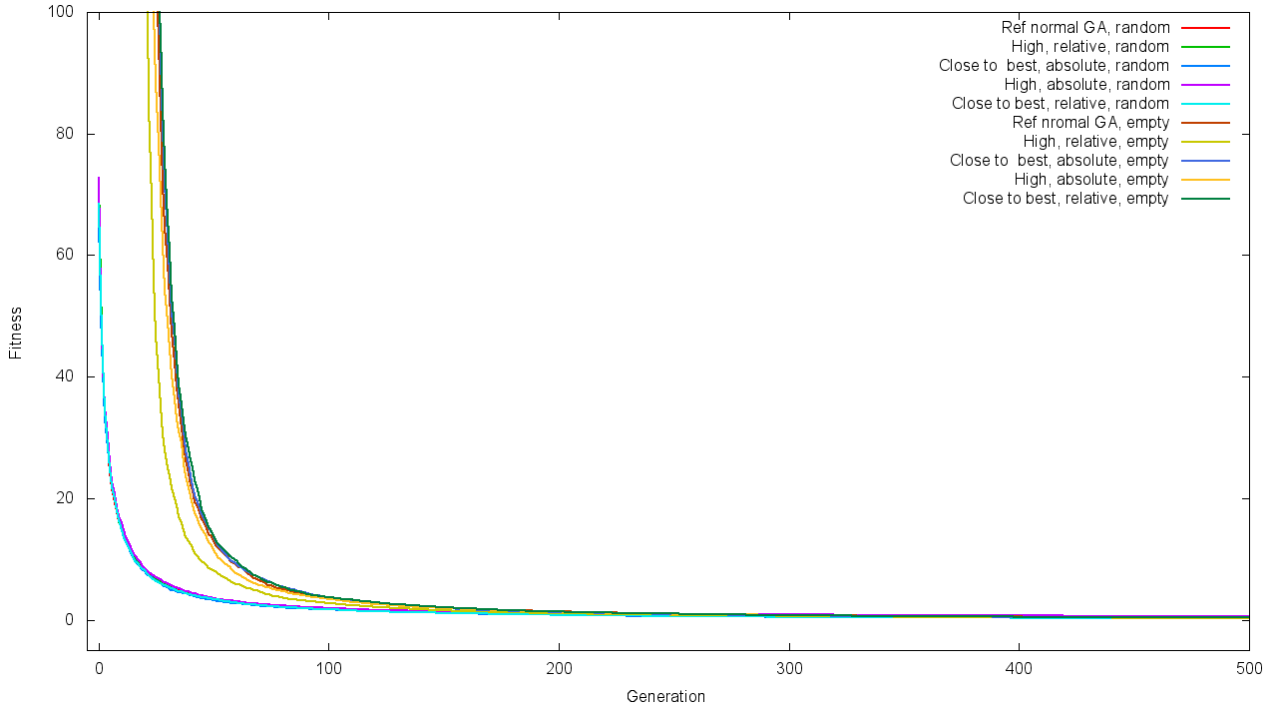

Figure 27: Fitness, Complexity 1000, Experiemnt 15

Lambda in fitness type	Random population	Empty Population
Reference normal GA	372	433
Close to best, relative	339	423
Close to best, absolute	341	418
High, relative	351	384
High, absolute	350	432

Table 11: Average generation genomes were found, best results, 1000 runs complexity test 1000

Analysis:

Looking at the table the results are very similar. Random population has a bigger variation and is probably not affected by using lambda at all. This is because the lambda value is initially close to the highest possible, and there is no lambda development. Lambda can probably therefore be used only in empty populations.

For empty population the high, relative version distinguishes itself. It both converges slightly faster than the others and finds genomes faster on average. This version will therefore be examined further and used as the new lambda in fitness.

9.16. Experiment 16

The goal of this experiment was to show that the new lambda in fitness is better than the standard genetic algorithm for higher complexities. The first step was tuning the lambda to

trajectory length ratio to test it on optimal settings. Then run it on a series of complexity tests and compare it to the Standard genetic algorithm.

Experiment setup:

GA versions tested:	Standard, Lambda in fitness
Complexities tested:	1000, 5000, 10000, 15000, 20000, 25000
Number of runs:	1000 for 1000 and 20 for others
Rates tested:	10, 15, 20, 25, 30, 40, 50, 100

The tuning run complexity test 1000 for 1000 runs with ratio at increments of 10. The criterions set was that the average generation a genome was found should be as low as possible, and the plots should converge as fast as possible.

When the optimal value of 20 was found 15 and 25 were tested to see if there was any difference. There was no difference and the ratio used in the end was 20. The average generation a genome was found for ratio 20 was 365.

Below are the average plots of complexity tests 15000 and 20000. Standard genetic algorithm with empty and random initial population is provided for comparison.

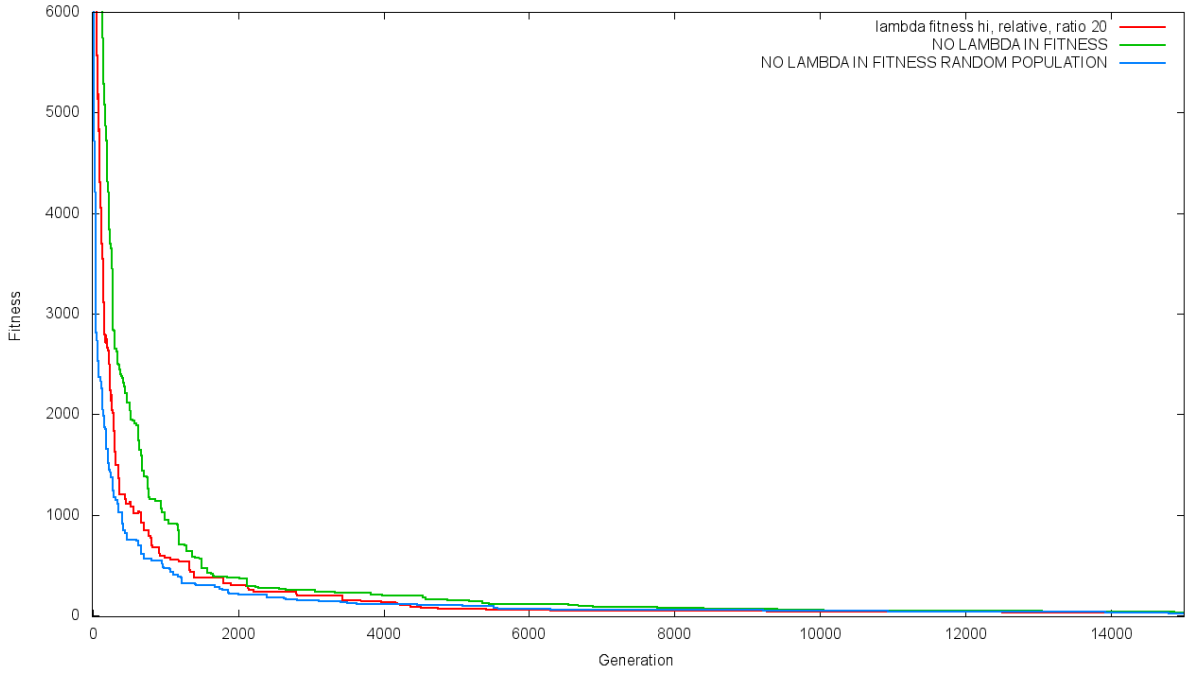

Figure 28: Fitness, Complexity 15 000, Experiment 16

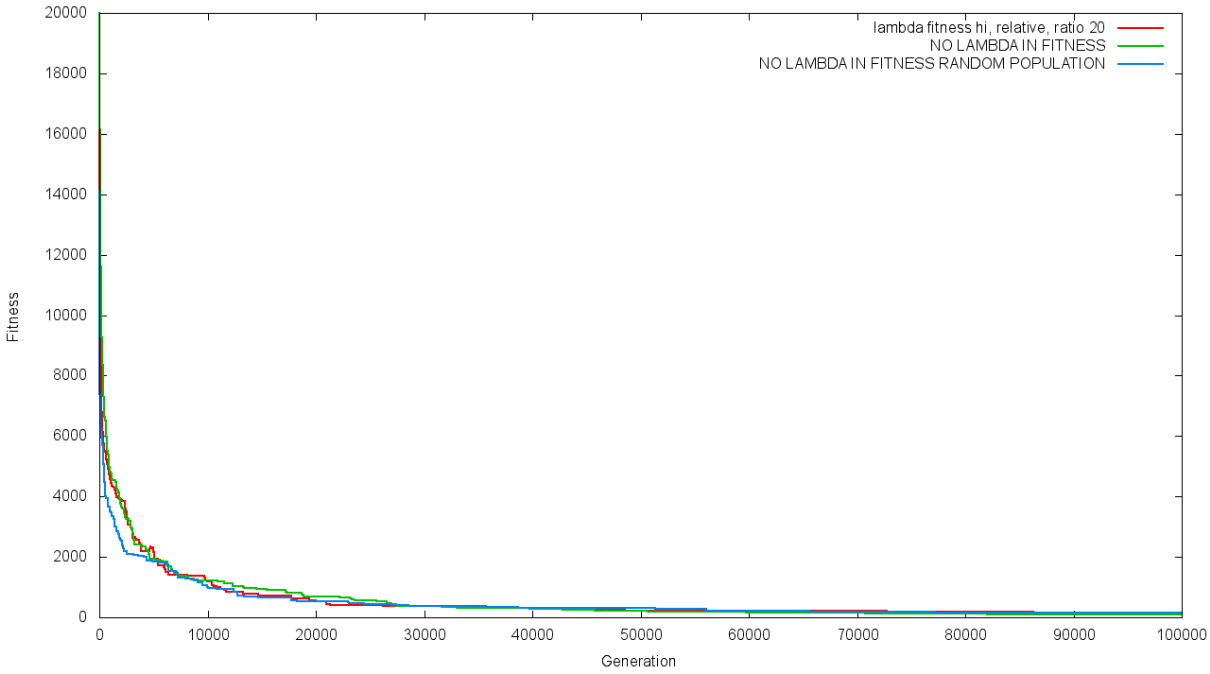

Figure 29: Fitness, Complexity 20 000, Experiment 16

Analysis:

The average generation the genomes were found for lambda in fitness is better than for Standard genetic algorithm with random initial population. However in all the plots the random population converges faster. This may not be surprising as the random population gives a big head start.

The plots get more and more similar the closer one get to 25 000 (a solution is never found). For complexity test 15 000 the standard genetic algorithm converges slowest and lambda in fitness is in the middle. This is what all the tests up until then looked like. But for the tests 20 000 and 25 000 they start to be indistinguishable.

This experiment shows that lambda in fitness definitely provides an improvement over standard genetic algorithm. It is however unclear what happens when searching for very high complexities.

9.17. Experiment 17

This experiment aimed to gain knowledge about the developing phenotype in terms of growth differentiation and death transitions (GDD). The usages of the different transitions were measured by counting the number of times the rules were used for each step of the cellular automaton. The hope was that this knowledge could contribute to a new genome parameter, and shed light on how to use genome parameters in general.

Experiment setup:

GA versions tested:	No GA, only recording developing CA stats
Complexities tested:	100, 1000, 5000, 10000
Number of runs:	1

The genetic algorithm was not used in this experiment. Only the GDD stats were recorded for each development step and the value for each development step has been plotted together with the cumulative value. The plots show the development of a genome with complexity 100.

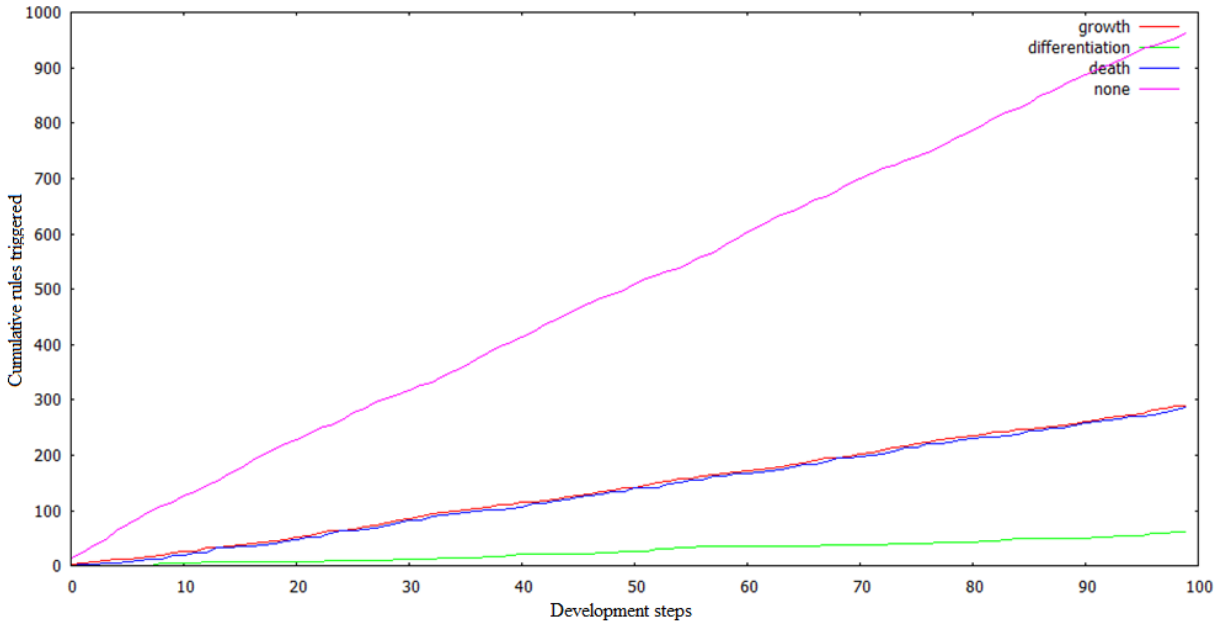

Figure 30: Cumulative GDD for development of phenotype with traj len 100

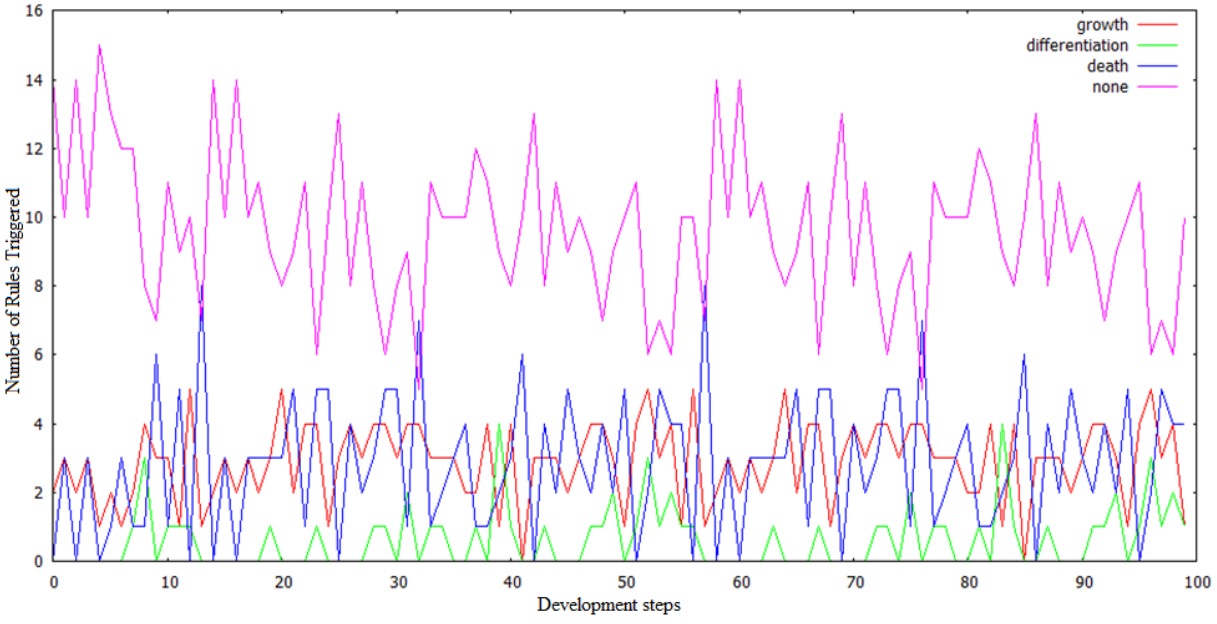

Figure 31: GDD stats for development of phenotype with traj len 100

Analysis:

The cumulative plots all looked similar to the one for 100. The step by step plot cannot be for much more steps than 100, because the lines become indistinguishable.

The slopes of the cumulative plots were compared to each other across different trajectory length phenotypes, but no clear trends were found. However, the slope is the ratio of which each transition type is used, and is an interesting measurement to be used with the GA.

Another property of the cumulative plots is that growth and death transitions always have the same slope. This suggests that they need to be balanced. None-transition is used more than the rest because it contains 3 sub transitions instead of 2 as for the others. Differentiation transition is mostly random. Some times higher than growth and death and sometimes lower.

The step by step plot seems to have a repeating pattern. The flat are at 37 and 80 is the same. This is even though the trajectory length is confirmed to be 100. The reason for this was because the cellular automaton uses wrap around, and the phenotype moves in a certain direction. This means that the same pattern of cell states shows up at two different places in two different states in the cellular automaton.

9.18. Experiment 18

For this experiment 100 000 random individuals were created. Their fitness was calculated and plotted using GDD stats and sub transition stats. It is different from the last test in that GDD stats are calculated from the genome, and not measured in a developing phenotype, making this a test on genome parameters. The transition stats were calculated by counting the number of transitions in the genome and dividing them on the length of the genome to create a transition rate.

The goal was to see how well they are suited as genome parameters. The plot is done the same way as Figure 3: Plot of trajectory length and lambda [15]. except this population has not been seeded in any way to avoid bias to any of the transition types.

Experiment setup:

GA versions tested:	No GA, only fitness function used
Complexities tested:	Random complexities
Number of runs:	100 000

The plots have trajectory length on the Y-axis and the parameter on the X-axis. It shows the GDD stats and lambda, and one sub transition.

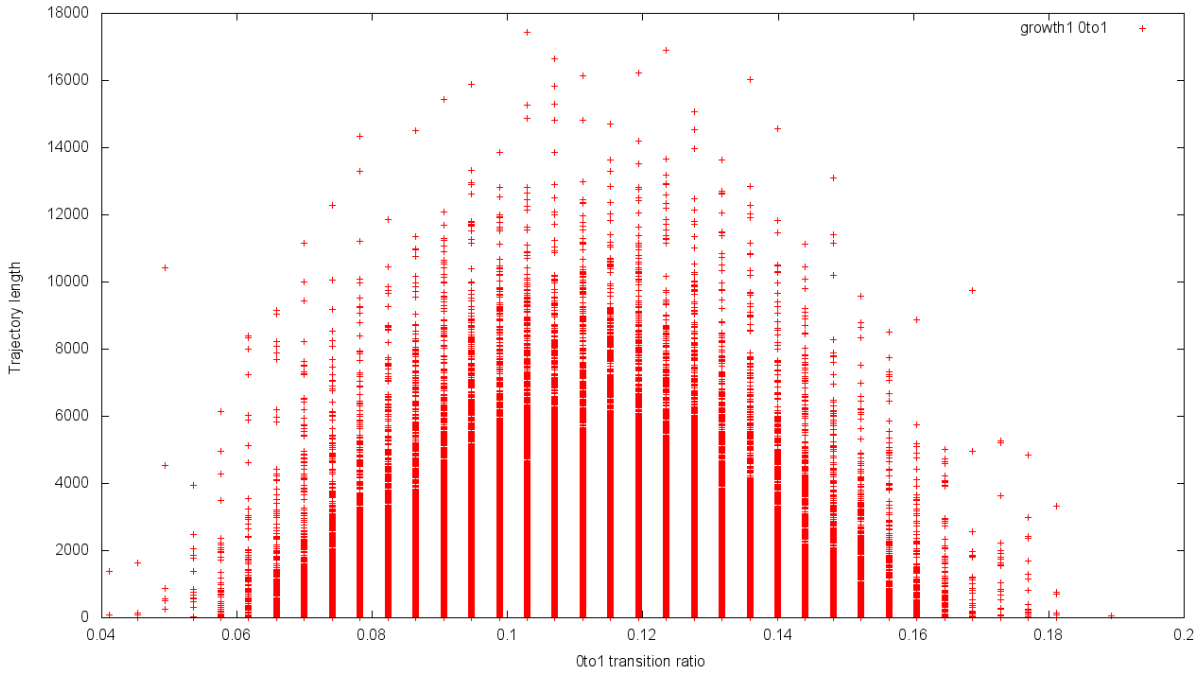

Figure 32: Sub transition of growth, center cell change from 0 to 1

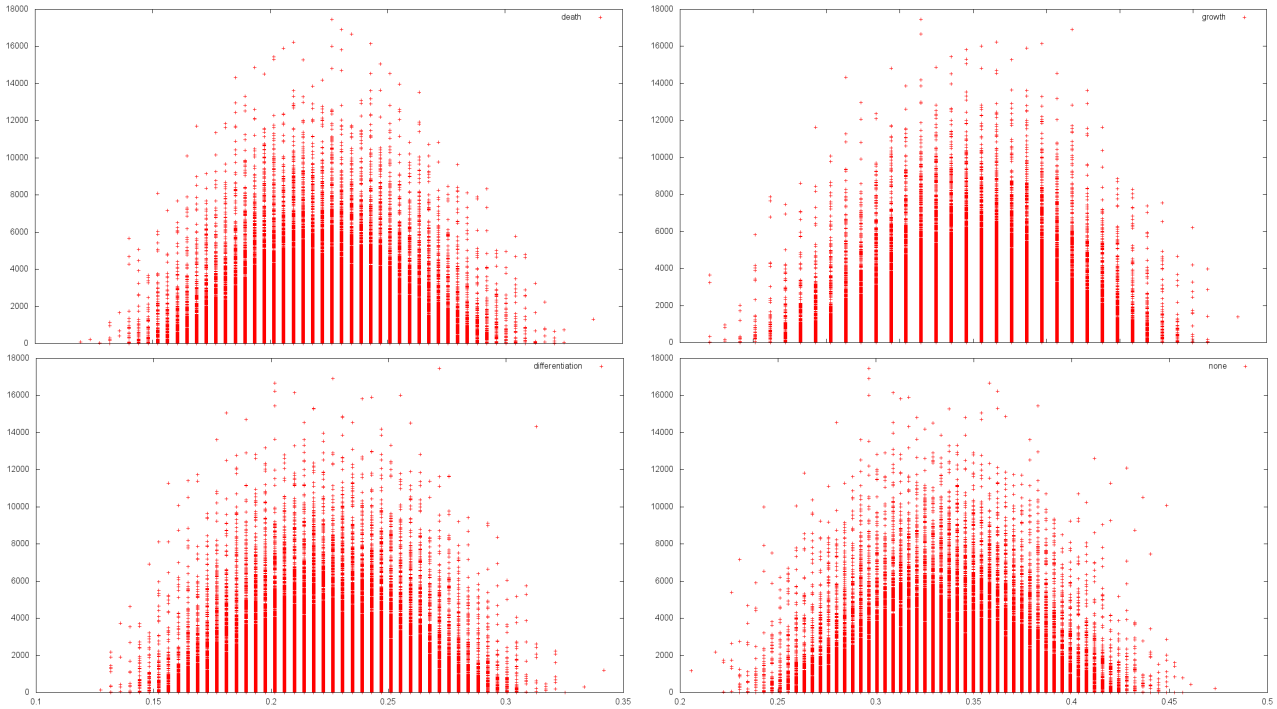

Figure 33: From top left: Death, Growth, Differentiation, None.

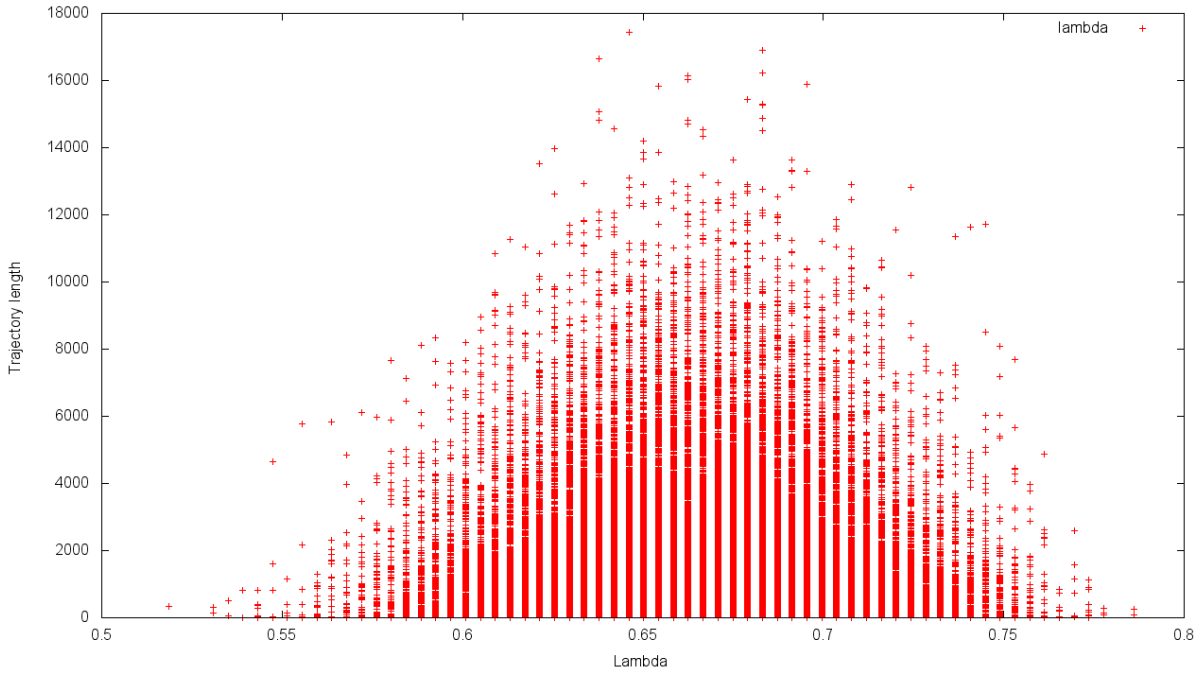

Figure 34: Lambda

Analysis:

All the plots looks very similar, they have a different scale on the X-axis as the parameters are measured as different sized parts of the whole, but this will not matter if they are used as parameters. They are also all very similar to the lambda plot, which means that they probably can be used in the same way.

They all tell something about the variation in the genome on different levels. The sub transitions are more specific and the none-transitions and lambda are the least specific. If this can help the genetic algorithm or not needs to be investigated further. The transitions could definitely be usable as genome parameters.

9.19. Experiment 19

The previous two experiments treated the same genome properties in two different ways. In 9.17 the usage of the different GDD stats in a developing phenotype were measured, and in 9.18 the GDD stats was calculated from the genome.

This experiment aimed to measure the GDD stats and plot them the same way as in 9.18. This would also make it easier to find Relations and dependencies among the different stats. The main goal was to check if growth and death transitions really need to be balanced.

Experiment setup:

GA versions tested: No GA, only fitness function used

Complexities tested: Random complexities
Number of runs: 100 000

The plots are presented for differentiation transition and one where X-axis is the difference between growth transition rate and death transition rate. Several combinations of the different transition stats were tested, but all looked similar to the differentiation plot.

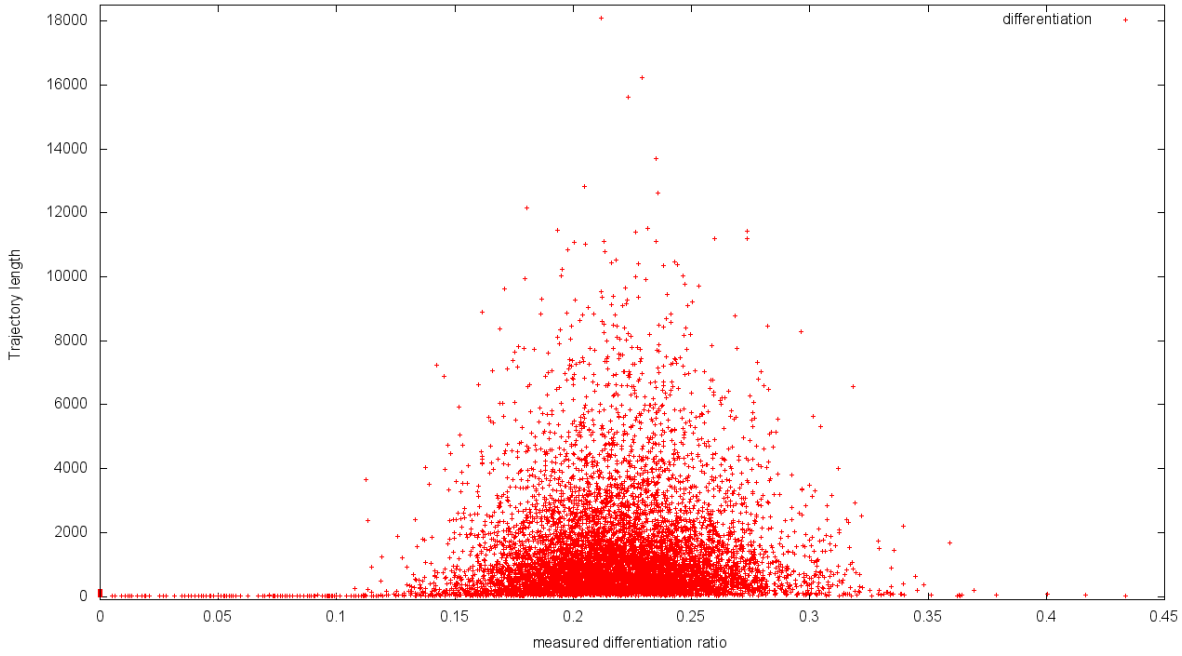

Figure 35: Differentiation transition rate

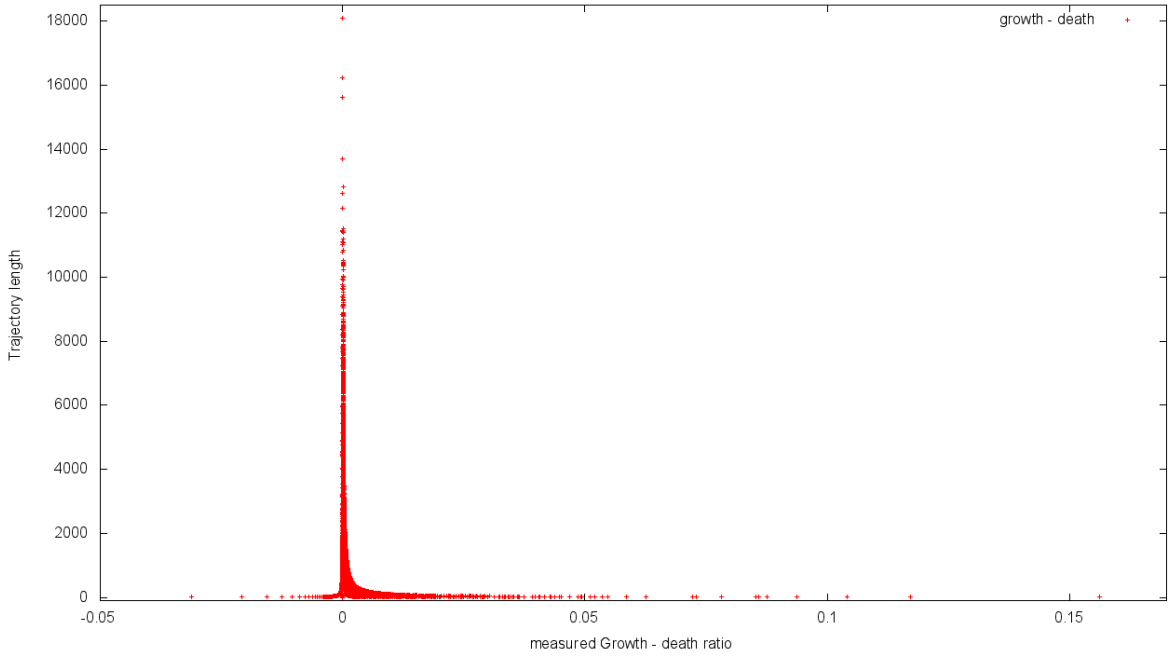

Figure 36: Growth – Death

Analysis:

The measured usage of differentiation transition rules in the cellular automata provides a more accurate prediction of where to find good genomes compared to the one calculated from the genome. From 0 to 0.1 most genomes have very low trajectory lengths while at 0.22 the best genomes are found.

This makes it much easier to know where the good genomes are as one can disregard genomes at certain values. The plots in general looked the same for all GDD stats. All had areas with bad genomes and the good genomes were concentrated around 0.22 for growth, death and differentiation, and around 0.33 for none-transitions.

Growth and death usage seems to have to be balanced; good genomes are only found when the difference between them is close to 0. This makes sense because if they are not balanced the developing phenotype will move towards a point attractor where all cells are of one type, or one type is not present.

9.20. Experiment 20

This experiment uses the knowledge gathered about measuring usage of different parts of the genome to see if it can improve on the genetic algorithm. There are a lot of possibilities, but the time only allowed for a few variations to be tested.

The tests focused on using the average generations because this have proven to be a good measurement to how good the algorithm performs also on higher complexities. It also takes less time allowing testing more variations.

Experiment setup:

GA versions tested:	New versions
Complexities tested:	1000
Number of runs:	1000

The different versions are presented below together with their results at the end. The values presented are used the same way as lambda is used in the new version, as seen in 9.15.

Measured Lambda

The plots of the measured GDD stats seemed to suggest that they were more precise than calculating them from the genome. This version seeks to find out if this precision is an advantage to the genetic algorithm.

This version measures the lambda value in the same way that GDD stats are collected from a developing phenotype. The number of times a quiescent state is used during development of the phenotype is used to calculate a measured lambda value.

$$Measured\ lambda = \frac{totalGenomeUsage - quiecentStateUsage}{totalGenomeUsage}$$

Death Parameter

Simply uses the number of death transitions in the genome and calculate a death parameter the same way as lambda. Death parameter is dependent on growth, and contains only 2 sub transitions; the question is how this will compare to lambda parameter.

$$Death\ Parameter = \frac{genomeLength - deathTransitions}{genomeLength}$$

Single Transition Parameter

Death parameter contains two sub transitions, this version aims to find out how a single specific parameter compares to lambda. This parameter uses only the transition rules from 0 to 1 in the genome. Only neighborhood configurations with a center cell 0 that results in 1 will be counted.

$$Single\ Transition\ Parameter = \frac{genomeLength - 0to1Transitions}{genomeLength}$$

Measured Average

Measured average uses the average difference of growth, death and differentiation transitions from 0.22. 0.22 is used because there are a total of 9 possible transitions for 3 state cellular automata and each have 2 sub-transitions (2/9 ≈ 0.22). This was tested because they incorporate 6 sub transitions into one parameter which is interesting to compare to lambda which only contains 3 (all transitions to the quiescent state).

$$Average\ Parameter = \frac{\text{abs}\left(\text{growth ratio} - \frac{2}{9}\right) + \text{abs}\left(\text{death ratio} - \frac{2}{9}\right) + abs(diff\ ratio - \frac{2}{9})}{3}$$

Measured Growth minus Death

This parameter uses growth minus death to filter out bad genomes in the fitness function. It basically said that if genomes are outside a given limit the fitness gets very high. If it is inside the growth – death transitions were used the same way as measured lambda.

Below are the results of the average generation genomes were found for each version. The plot shows Trajectory length on Y-axis and the measured average on X-axis. Together with an example of filtering out genomes using measured growth – death.

GA version	Generation genomes found
Reference normal GA	433
Reference lambda in fitness	365
Measured lambda	364
Death parameter	397
Single Transition Parameter	436
Measured average	423
Measured Growth minus Death	425

Table 12: Average generation genomes were found, best results, 1000 runs complexity test 1000

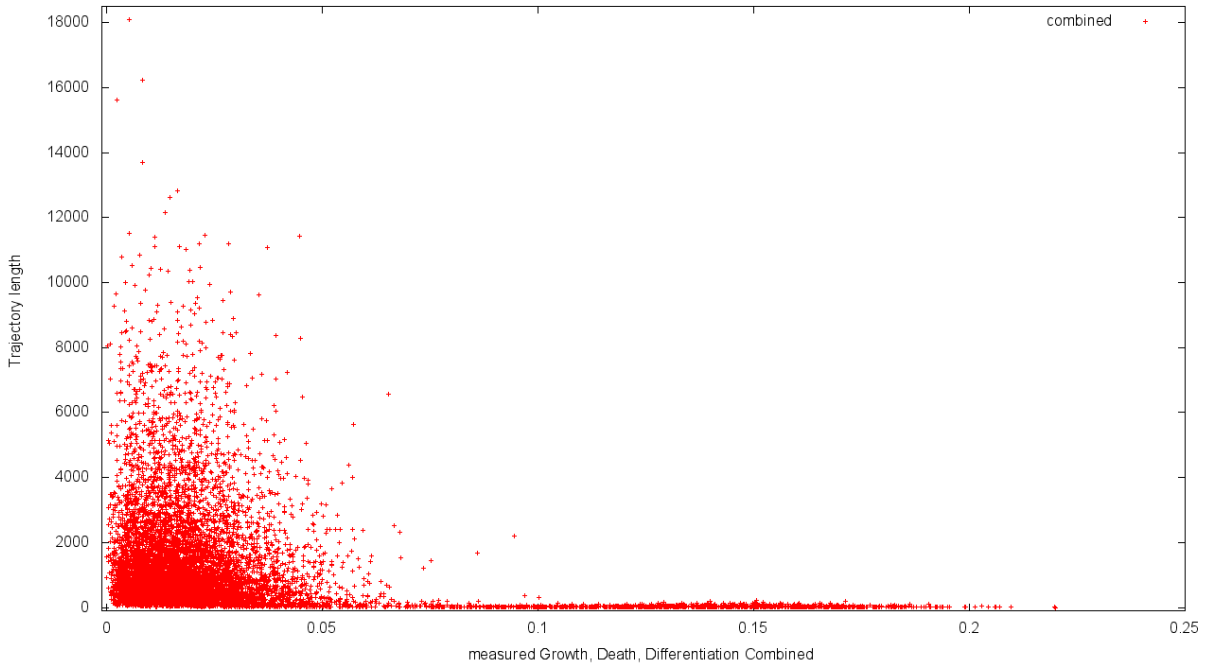

Figure 37: Measured average plot

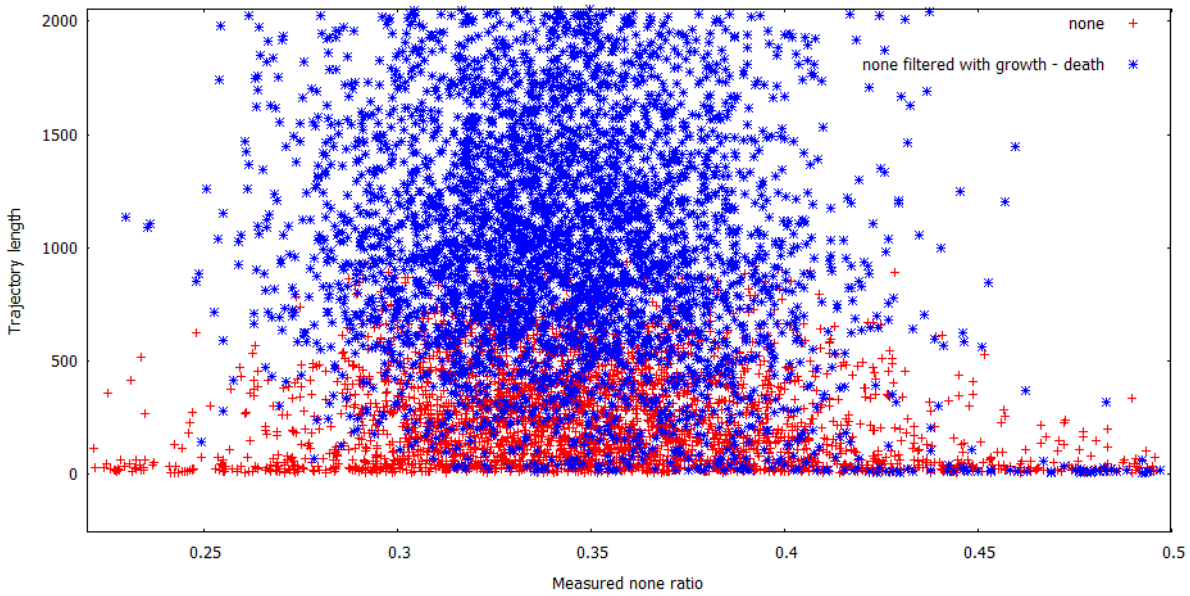

Figure 38: Filtering out genomes with growth –death (red are unfiltered, blue are after filtration)

Analysis:

None of the new versions performed better than lambda in fitness. Measured lambda was interesting because it performs exactly the same compared to lambda in fitness. This means that the precision of the parameter in predicting the exact fitness is not as important as knowing what area it is probable to find good individuals.

Death parameter was the only one close to lambda in fitness. This is probably because it is so similar, and it may be possible to tune it somehow. However the single transition parameter performed badly despite its similarity to lambda in fitness. It may have something to do with the fact that it does not cover all aspects of the genome.

The measured average and Measured Growth minus Death did not do well. There probably exists much better way of using these parameters. Bur unfortunately there is not enough time for that.

The plots presented show an example of the last two versions, and suggests why they should work. The measured average has a clear area where the best genomes are found. It also has many low scoring genomes far away from the highest point. Using growth minus death as filter removes a lot of the lower complexity genomes which should better indicate where to find the good genomes.

10. Discussion and further work

In this project the effects of lambda and other genome parameters on a genetic algorithm have been studied. Lambda in fitness has been able to improve the genetic algorithm for empty populations. Genome usage shows promise in controlling mutation and transition parameters may prove useful even if the results in the last experiment were not good.

This section discusses the overall results of the experiments and suggests further work. The main portions are discussed; Lambda parameter, genome usage and transition parameters. In the end general properties of genome parameters in relation to the genetic algorithm are discussed.

10.1. Lambda

How to use lambda to improve the genetic algorithm was the first task in this project. Lambda to discard showed the most promise in the beginning and it was attempted to tune it to see if the performance could improve over standard genetic algorithm. It turns out lambda to discard can only be tuned up to the performance of the standard genetic algorithm but not further. This happens when discard limit is 0.1 which only discards about 0.05% of the genomes through one run of the genetic algorithm. So this means the fewer genomes are discarded the better the algorithm performs.

Discarding genomes seems to narrow down the search, making the population less diverse. This in turn makes it harder for the genetic algorithm to climb out of local maxima. The main result is that the algorithm becomes more unstable, either finding good genomes or getting stuck early at low trajectory lengths.

One issue is that lambda to discard prevents variation; a better approach could be to use it to add variation to the population. There are a number of other ways to exploit lambda outside the fitness function. This project aimed to investigate other genome parameters as well, so they were not tried, but here are some suggestions:

- Discard children based on parents instead of the best individual of the previous generation.
- Instead of creating two children to go on to the next population it is possible to create 10 children by crossover and mutation from two parents. Lambda will then be used to select the best one(s) to go on to the next generation.
- Lambda could be used to ensure that a certain number of individuals become more different from the best individual of the last population, rather than limit them.
- Lambda could be used to control the mutation rate dynamically.

Lambda in fitness has shown good promise for empty populations. The results are in fact comparable to that of a standard genetic algorithm initialized with a random population. During the tests it also seemed that lambda affect a randomly initialized population less than an empty one. Lambda increases fast when complexities get higher, and at very high complexities it do not seem to change the behavior of the genetic algorithm at all. This is the same for a random initial population. Searching for trajectory 25 000 removes the differences between lambda in fitness and standard genetic algorithm with both empty and random population.

Lambda in fitness needs to follow a path of increasing lambda values to be useful. Starting with a random population will make the population initialize on an average lambda 0.66, diminishing the effect. However, there is probably an advantage to it anyway as genomes with potential (a high lambda) may still have a low fitness, but using lambda together with fitness will send them to the next generation anyway. This promotes variation in the population which seems to be very important.

It is clear that lambda in fitness finds fit genomes faster than the standard genetic algorithm. Lambda only tells where potentially high genomes are found, poor genomes are still found at lambda value 0.66. Leading the population to this area increases the probability of finding individuals with high fitness. It is likely this is why the fitness increases faster if the genetic algorithm is forced towards this area.

The version of lambda in fitness chosen was based on forcing the lambda value as high as possible and reducing the penalty relatively to how complex the individual is. The reason this version works better than the others is maybe because it is important to get to a high lambda value quickly, but when lambda is at 0.66 it is not really useful anymore, so it is good that the emphasis on it decreases.

Lambda parameter shows very roughly how far from a solution an individual is. Supported by the explanation in 6.11, this by itself explains why lambda in fitness can improve the genetic algorithm. It relates the genome to the developing phenotype in the fitness function.

The work that may be important to do on lambda in fitness is seeing how well it works for cellular automata with more states. The last experiment showed that a transition parameter that only checks 1/9 of the genome (the single transition parameter) did not perform well. As the cellular automata gets fewer states lambda may have a lower effect. For a 9 state cellular automata lambda may be as little useful as the single transition parameter is for a 3 state cellular automata. In that case a GDD parameter may be more useful as they span an equally big part of the genome independent of the number of states.

10.2. Genome usage

The experiment on genome usage in mutation showed some promise. But the genome usage was also investigated when measuring the GDD usage and lambda in the developing phenotype in experiment 58, 62 and 64.

The performance on both random and empty population was slightly better than the standard genetic algorithm. This difference was very small however and it may be that the same results could be achieved only controlling the mutation rate dynamically.

Recording genome usage statistics for each single gene in a genome provides interesting possibilities however. One can look at individual genes and see if they are used. This way one could know exactly which genes to mutate to see an effect. It may also be possible to predict what the mutations will do.

One can start with an empty genome, and then only mutate the part that is triggered during development of the phenotype. This will create a new genome which, while developing, will probably have a larger part triggered. Comparing two steps like these can provide interesting knowledge about individual genes roles in a phenotype. This can also be used to roll back a mutation and try over if the results are not desired.

It could be possible to treat the development of a genome almost like a heuristic search. Each state is a point in a directed graph, and the possible paths from one point to another are defined by what genes are mutated at each step. The problem may be that the growth in complexity may be very high for each step, and may mean that a random search is better.

The other way of measuring genome usage with an emphasis on genome parameters did not prove very useful. The result of measuring a parameter is more precise than calculating it from the genome. When using measured lambda it was shown that the performance compared to normal lambda in fitness was exactly the same. The precision do not seem to matter so much, this may be because they both lead genomes to the same area of lambda 0.66. Genome usage does not seem to be able to improve on genome parameters in this way.

It was shown that death transitions and growth transitions need to be balanced, but it was hard to exploit. The value needs to be measured so it cannot be used to discard genomes. It may exist a better way of using it in fitness, for example together with lambda.

10.3. Transition parameters

The transition parameters are alternatives to lambda in fitness and can be used much in the same way. The last experiment was the only time they could be tested, and they failed to perform equally to lambda.

However they created a notion of sub transitions, and many genome parameters can be built from them. Death parameter, for example, has two sub transitions of 1 to 0 and 2 to 0 when the cellular automaton has 3 states. Lambda parameter also consists of sub transitions. If the quiescent state is 0 the sub transitions are 0 to 0, 1 to 0 and 2 to 0. Lambda then has 3 sub transitions. As can be seen from the last experiment, lambda in fitness performed better than death parameter which performed better than single transition parameter.

This suggests that more sub transitions used in a genome parameter leads to a better fitness function, but it ignores the measured average which incorporated 6 sub transitions. The way this has been done may not have been correct and it differs from the other parameters in that it combines transitions that are not necessarily related.

Sub transitions may be used in similar ways to the measured average to design genome parameters. It becomes more relevant for cellular automata with more states as the number of sub transitions increase. It could even be used with a developmental model with the fitness function searching for trajectory length 1000 and use the average generation genomes were found after 1000 runs.

Before the parameters were chosen a plot like Figure 3 was created. It was analyzed and measured average looked like it would have good properties. Because it did not it seems that what these graphs look like do not always dictate if the parameter will be good. It may be an idea to look into how to interpret these kind of plots more closely.

The inherent properties in the different transitions also need to be considered. The growth and death transitions should be balanced in the developing phenotype. This is not true for growth and differentiation for example. More research into these properties may also be useful.

One example is two transitions of to1 and from1. To1 consists of the sub transitions 0 to 1 and 2 to 1 and from1 consists of the transitions 1 to 0 and 1 to 2. To1 and from1 will probably also have to be balanced in the same way as growth and death. In that case the same will be true for to2 and from2. These are properties that may be able to tell more accurately what genomes are good and not. They may also be strongly related so if growth and death is balanced so will the others.

The last property of the transition parameters was using growth minus death and filter out the genomes where growth and death are too different. This in combination with another genome parameter could be used to discard many of the lower complexity genomes. From the last experiment this strategy did not work very well, but this did not use a different genome parameter.

10.4. Genome Parameters and the Genetic Algorithm

Genome parameters ability to help the genetic algorithm seems to be dependent on their ability to both drive mutation and drive the development in the right direction. The genetic algorithm needs more variation, and attempting to limit variation in favor of going in the right direction seems to have a bad influence. The experiments with lambda to discard show this.

Increasing the mutation rate of the genetic algorithm helped a great deal compared to the earlier versions. Increasing past 2% mutation also made it a lot more stable. The result is that the difference between the standard algorithm and lambda in fitness may be smaller than it would have been at lower mutation rates.

During the experiments it has also become clear that a genome needs to be balanced. This is the theory behind lambda value. The genomes become more random as they reach lambda 0.66(for 3 state CA), which is also the area with the longest trajectories. The same can be said for all sub transitions, where the longest trajectories are found around 0.22 (1/9 for 9 sub transitions).

This project only tested very specific cellular automata with size 4 by 4, 3 states and a 5 cell neighborhood. It is unclear how lambda would perform with a larger number of states, or with a bigger grid. This would take much more time, and would make it hard to test more ways to exploit the genome. The lambdas effect at a higher number of states was explained earlier. Larger grids or neighborhoods will not affect how much of the genome lambda covers, so this will probably not affect lambda in fitness.

Genome usage however may actually become more useful as the genome gets bigger. In a 9 cell neighborhood for example the genome is much larger and it may take longer to exploit all of it, especially if there are more states. With genome usage mutation is targeted exactly where needed and may help develop good individuals much faster initially.

In the end the tests started to rely more and more on the test setup with 1000 runs at complexity 1000. This is because these results were very consistent, especially for empty populations. The versions that performed well at 1000 runs of complexity test 1000 performed well in general. However after complexity test 20 000 the differences began to disappear.

Nothing seems to help the algorithm a lot at complexity test 25 000 initially some are better, but in the end the results are too similar. Maybe 1000 runs at this complexity would be accurate, but that simply takes too long time.

11. Conclusion

The experiments have shown that using lambda parameter in the fitness function can improve a genetic algorithm. Other parameters like the transition parameters have not performed as good, but the plots for the parameters looks the same as for lambda parameter; this suggests that at some level they should be able to provide the same advantages. The transition parameters present a way to design genetic parameters. Lambda and GDD parameters all consists of sub transitions that can be combined in many different ways. The number of sub transitions used for the genome parameter may affect how useful it is, because more sub transitions incorporate more information about the diversity of the genome.

Using lambda to discard genomes has been shown to have little or no benefit for the genetic algorithm. The results pointed to how important the width of the search is. Limiting the search even by a small fraction decreases the performance of the genetic algorithm.

Genome usage presents an interesting idea of mutating only the part of the genome that was used when developing the phenotype. It has been used to control mutation dynamically with good results, though not necessarily just because of the technique, but also because of a higher mutation rate. Genome usage can be used to gain great control of the genome search.

The tests done show that running a complexity test of 1000 for a 1000 runs provide an accurate picture of the behavior of the algorithm all the way up to complexity test 20 000. The average generation that genomes were found is accurate and the plot looks the same as for higher complexities. For complexity test 20 000 and up the differences are very small, and it becomes harder to assess the performance.

Genome parameters can improve the genetic algorithm, but this project has only scratched the surface.

12. References

[1] Moshe Sipper, “The Emergence of Cellular Computing”, Computer, 1999

[2] http://necsi.edu/publications/dcs/DCSchapter0.pdf 10.12.2012

[3] Stefano Nichele, “Trajectories and attractor basins as a behavioral description and evaluation criteria for artificial EvoDevo systems”, 2009

[4] Stefano Nichele, “Discrete Dynamics of Cellular Machines: Specification and Interpretation”, GECCO '11 Proceedings of the 13th annual conference companion on Genetic and evolutionary computation Pages 767 – 770, 2011

[5] C. Langton. “Computation at the Edge of Chaos: Phase Transitions and Emergent Computation”, Physica D Volume 42 Issue 1 – 3 Pages 12 – 37, 1990

[6] Francis Heylighen, “The science of selforganization and adaptivity”, The Encyclopedia of Life Support Systems (EOLSS), 2001

[7] Julian Francis Miller, “Evolving a Self-Repairing, Self-Regulating, French Flag Organism”, Proceedings of Genetic and Evolutionary Computation Conference (GECCO), Springer LNCS3102, 2004

[8] Richard Lewtonin, “The triple helix Gene, organism and environment”, 2002

[9] W Brian Arthur, Steven N. Durlauf, David A. Lane, “The economy as an evolving complex system II”, 1997

[10] Peter J. Denning, “Is Computer Science Science?”, Communications of the ACM - Transforming China Volume 48 Issue 4 Pages 27 - 31, 2005

[11] Herbert A. Simon, “Invariants of human behavior”, Annual Review of Psychology, 41, Pages 1-19, 1990

[12] C. Langton, “Studying artificial life with cellular automata”, Physica D: Nonlinear Phenomena Vol. 22, No. 1-3: Pages 120-149, 1986

[13] http://mathworld.wolfram.com/ElementaryCellularAutomaton.html 17.11.2012

[14] http://www.ai-junkie.com/ga/intro/gat2.html 19.11.2012

[15] Gunnar Tufte, Stefano Nichele, “On the Correlations Between Developmental Diversity and Genomic Composition”, GECCO '11 Proceedings of the 13th annual conference on Genetic and evolutionary computation Pages 1507 – 1514, 2011

[16] Stephen Wolfram, “Universality and complexity in cellular automata”, Physica D: Nonlinear Phenomena, Vol. 10, No. 1-2 Pages 1 – 35, 1984

[17] http://cell-auto.com/neighbourhood/ 28.11.12

[18] http://java.icmc.usp.br/dilvan/thesis.phd/genetic4.gif 28.11.2012

[19] Taras Kowaliw,” Measures of Complexity for Artificial Embryogeny”, GECCO 08 Pages 843 – 850, 2008

[20] T. Jones, S. Forrest, “Fitness Distance Correlation as a Measure of Problem Difficulty for Genetic Algorithms”, Proceedings of the 6th International Conference on Genetic Algorithms Pages 184 – 192, 1995

[21] H. H. Wold, “Using genome parameters to improve performance in genetic algorithms”, 2012

[22] http://www.oracle.com/technetwork/java/javase/memorymanagement-whitepaper-150215.pdf 9.5.13

[23] Gunnar Tufte, “Gene Regulation Mechanisms introduced in the Evaluation Criteria for a Hardware Celluar Development System”, Adaptive Hardware and Systems, 2006. AHS 2006. First NASA/ESA Conference on Adaptive Hardware and Systems Pages 137 – 144, 2006

[24] A. Chavoya, Y. Duthen, “Using a Genetic Algorithm to Evolve Cellular Automata for 2D/3D Computational Development”, GECCO '06 Proceedings of the 8th annual conference on Genetic and evolutionary computation Pages 231 – 232, 2006

[25] M. Mitchel, J. P. Crutchfield, R. Das, “Evolving Cellular Automata with Genetic Algorithms: A Review of Recent Work”, In Proceedings of the First International Conference on Evolutionary Computation and Its Applications (EvCA'96). Russian Academy of Sciences, 1996

[26] T. C. Fogarty, “Varying the probability of mutation in genetic algorithms”, Proceedings of the Third International Conference on Genetic Algorithms Pages 104 – 109, 1989

[27] W. Lin, W. Lee, T. Hong, ”Adapting Crossover and Mutation Rates in Genetic Algorithms”, Journal of information science and engineering Pages 889 – 904, 2003

[28] http://mathworld.wolfram.com/UniversalCellularAutomaton.html 24.5.2013

[29] http://mathworld.wolfram.com/ElementaryCellularAutomaton.html 31.05.2013