Abstract

The objective of this thesis is to evolve Compositional Pattern Producing Networks Stanley [2007] capable of creating the 2-Dimensional shape of an eye. The author uses the fast, cluster-enabled C++ Mathisen [2007] implementation of Neuroevolution of Augmented Technologies (NEAT)Stanley [2004] called Neatzsche to evolve these networks.

The author creates his own simulation environment capable of representing the needed properties to evolve a model eye. That environment is a simplified abstraction of the real world, representing components such as different cells, light sources and light.

The resulting shapes from runs in different sized environments with settings are then shown and analysed. The analysis consists of both an evaluation of the different shapes, but also of the performance of the tools used.

Preface

The following thesis was writted in the academic year 2007/2008 under the advisor Professor Keith L. Downing. It was done at and for the University of Technology and Science (NTNU) in Trondheim, Norway, but also during my 3 month stay at the Evolutionary Complexity Research Group (EPLEX) at the University of Central Florida (UCF) hosted by Dr. Kenneth O. Stanley.

Acknowledgements

I wish to extend my thanks to my advisor Professor Keith L. Downing for finding my idea interesting and taking on the responsibility of advising me on it, his insight and advice on my specific problems and for enabling my stay and research opportunities in the EPLEX group at UCF. I would also like to thank Bjørn Magnus Mathisen, the creator of Neatzsche, for his technical support in regards to Neatzsche, but also his general input and helpful discussions on my various ideas, problems, solutions and subsequent results. I would also like to thank the Dr. Kenneth O. Stanely and the guys at EPLEX lab for their insight into CPPNs: J.T. Folsom-Kovarik, Jason J. Gauci, Philip Verbancsics, David D'Ambrosio and Joel Lehman. I would like to mention Richard Dawkins and his documentary the Blind Watchmaker as an inspiration for my idea to evolve the shape of an eye. Last, but not least I would like to thank my girlfriend for her patience while I was away in the United States for 3 months, thank you Lene Mellum.

Contents

List of Figures

Chapter 1

Introduction

To suppose that the eye with all its inimitable contrivances for adjusting the focus to different distances, for admitting different amounts of light, and for the correction of spherical and chromatic aberration, could have been formed by natural selection, seems, I freely confess, absurd in the highest degree.

– Charles Darwin

1.1 Evolution of the Eye

The evolution of the eye has long been a highly controversial topic, since the sheer complexity of it's structure makes it hard to intuitively imagine how it came to be. Darwin himself anticipated this when he published his theory of evolution. He wrote: "that the eye...could have been form by natural selection seems, I freely confess, absurd in the highest possible degree". Since then however, we have made many breakthroughs in our understanding of evolution and solved many of it's mysteries. This progress also includes the eye. One of those areas is confirming that the eye and it's complexities can indeed be explained by evolution in an acceptable time frame. Nilsson and Pelger 1994 showed it to be plausible that a fully functional mammal eye could pessimistically be evolved within 100 000 generations. I feel that had

Charles Darwin seen today's data, he would haven't have thought it so absurd anymore.

1.2 Previous Work

There has not been much work when it comes to simulating evolution of eye development, but there is a few worth mentioning.

1.2.1 Evolvable Hardware (Real world application)

Lukas Lichtensteiger and Ralf Salomon a robot with 16 movable light sensors and evolutionary algorithms to evolve the angle and position of the lights sensors in a simulated environment and then validating them on a real world obstacle avoidance scenario Lichtensteiger and Eggenberger [1999].

The robots sensors are a fixed rack of the 16 sensor mounted vertically on the bot and the obstacles are vertical light emitting rods. The avoidance method is a predefined mathematical function which calculates the distance to the obstacle based on speed and angle, so it works independently of how the sensors are spread out, and the point is to cover as much area as possible with the sensors.

The genetic encoding in this problem is basically the angle of each of the sensors. This is a fairly small domain, because you only have 16 individual angles to change. The evaluation of the individuals are done by measuring how successful the eyebot is at avoiding it's obstacles. It's also prudent to notice that since the evaluation is done in a real life environment it is prone to noise.

The results from this experiment showed that at least in this domain, artificial evolution solved the problem within 100-200 generations.

1.2.2 A theoretical paper from biology

As noted earlier, Nilsson and Pelger 1994 showed that you could evolve an eye in a relatively short time even when using pessimistic assumptions. Their

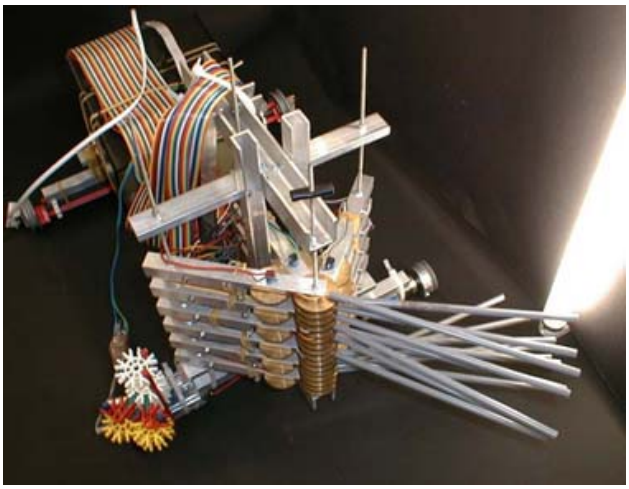


Figure 1.1: This figure show the sensor rack used on the eyebot. There are 16 tubes which at the bottom contain a light sensitive sensor. This gives it a very limited field of view.

model sequence is made in such a way that every change in it, results in an increase in spatial information the eye can detect. So they're basically using a guided hill-climbing search to achieve what they perceive as the optimal shape. Then they are calculating the number of generations needed for all those steps to complete. In figure 1.2 you can see how the eye starts out as a line of cells which then starts to sink in, creating the rounded shape we now have.

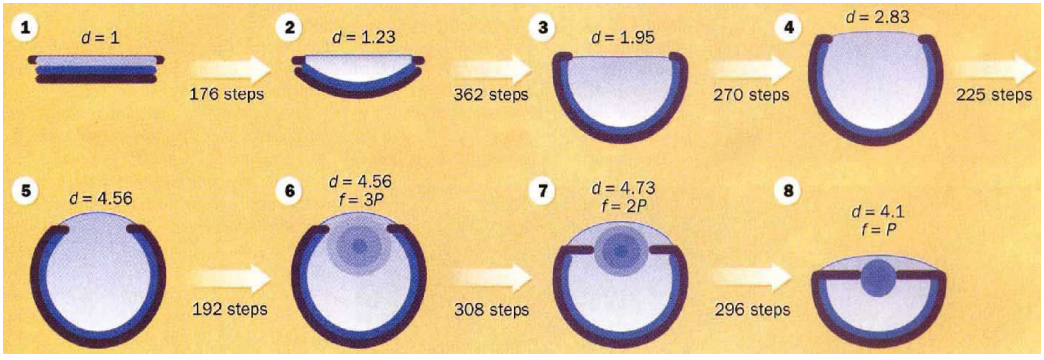


Figure 1.2: This shows the steps of the eye's evolution. All the steps increases the spatial information it can detect (borrowed from 1994).

They conclude with the ability for an eye to evolve within 363 992 generations, given the smallest possible rate of change in the structure each generation.

This of course only accounts for the eye itself, since the eye is useless on its own, without the proper neural processing abilities. The actual time in a specific species has to be somewhat longer, taking into account that both neural processing and eye function has to evolve simultaneously. However, given that neurons grow and "train" on their after their creation, it's very possible that a shape like the eye, in which the photoreceptors mirror the world outside, it's easy to connect that into the brain in a meaningful fashion.

Also, it's also very likely that the eye evolved much faster. Since genes are an indirect encoding, it's possible that a small change in the genome can have done the jump from flat area to circular cavity in much larger jumps.

1.3 Motivation

My motivation is to add to the discussion about the ability of the evolutionary theory to explain how the eye in it's current form can have come to be by natural means. I want to use what I perceive as the best genetic algorithm, the NEAT algorithm, to do this. Specifically the fast paralleled C++ NEAT implementation named Neatzsche Mathisen [2007] created by Bjørn Magnus Mathisen.

1.4 Working Hypothesis

Hypothesis I:*A CPPN evolved to create a shape in a 2D simulated environment, given a selection pressure that rewards high-information retrieval, will eventually evolve a shape similar to that of a cross-section of an eye.*

I will test this hypothesis by setting up a fitness function that will give a high fitness to models that has high information retrieval. I will also create my own simulated environment which capture the important aspects of the

world with regard to sight.

I will analyse the resulting shapes, and point to their similarity to an eye, both in shape and functionality. I will discuss my results with respect to this.

1.5 Overview of Thesis

This thesis consists of 7 chapters, chapter 1 introduces my problem and talks about some previous work on similar and relevant problems. Chapter 2 will give some background info on my methods used and why I chose the ones i did. Chapter 3 introduces some of my problems and consideration in regard to my thesis. Chapter 4 details my methods used, their specific settings and application. Chapter 5 will present my different runs and their results. Chapter 6 will give an analysis of my different results. Chapter 7 discusses the results and evaluates how my working hypothesis hold up.

Chapter 2

Background

This chapter will give an introduction too the various methods used in this document and give a justification for choosing one of them, based on which one is most suited for the current problem.

2.1 Genetic Algorithms

A Genetic Algorithm (GA) Mitchell [1996], Goldberg [1989], Jong [1975], Holland [1975] is a method for searching a domain, that uses the basic concepts of Darwinian evolution. This means that it has familiar abstractions such as

- Individual - describes a specific individual in nature
- Development - describes the developmental process of an individual
- Population - a collection of individuals
- Mating - where two individuals reproduce
- Mutation - a process where an individuals genes are altered randomly
- Genotype - the representation of an individual
- Phenotype - the physical manifestation of the genotype (the result of development of the genotype)

- Generation - a lifetime of one population
- Selection - the metaphor describing who gets to reproduce and who doesn't
- Fitness - a measure that describes an individuals ability to be selected and survive
- Selection Pressure - the effects of evolution "letting" the fittest survive and the poorer one die off. This is interpreted as a pressure to be better.

The genotype is the part of the individual that the evolutionary process changes. This genetic encoding is how you define your search space. It can range from an array of bits to an XML document. If you have an array of bits, your search space is effectively constrained by the length of the array. This genotype is then developed to a phenotype by the process called development. In the example of the bit-string you can interpret it as an integer, and the resulting integer is then the phenotype. The XML document can be interpreted in many ways, one of them can be to use it to create an Artificial Neural Network.

The population is the collection of all the individuals, where most of the time, all the individuals are different. Some individuals will then be closer to the solution than others, this is decided by the evaluator. The evaluator uses some kind fitness function to give the individual a fitness value. Usually this is described as a numeric value that scales fairly evenly from low to high (bad to good). An example of this would be if we wanted to evolve something as simple as a big number, we could just use the number itself as the fitness function (this would be the simplest function possible $f(x) = x$). Other fitness functions may be more advanced, from using arithmetic to interpreting a reading from the real world sensor.

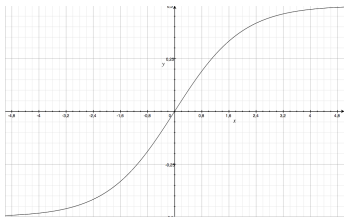
Between each generation the fittest individuals are generally statistically selected to live on and reproduce. The reproduction can be both sexual and asexual. And asexual reproduction means that you mutate the individual and put it into the next generation. The way mutation is implemented depends

on the genotype, in the bit-string example you can just flip a random number of bits, while on the XML-document you could interpret the document and change a value that it represents (like a number). In sexual reproduction, called crossover, you select two individuals to be the "parents" and then create a new copy which catches the traits from both "parents". Crossover is harder to implement, especially if you have a complicated domain. This is because the point of crossover isn't to randomly mutate, but the make an offspring that has qualities from both parents. Knowing how to keep individual traits is the key to crossover. In the bit-string a crossover could be as simple as taking parts of the string from parent A and the the other half from parent B. In the XML-document, you could do it low-level like that or use some kind of domain knowledge to select how to do it.

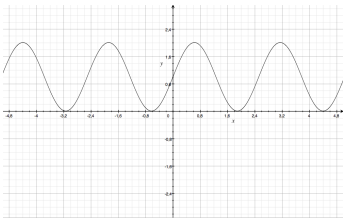
2.2 Compositional Pattern Producing Networks

Compositional Pattern Producing Networks (CPPNs) are a type of connected, directed graph very similar to the well known artificial neural networks (ANNs). However, the ANNs are abstractions of the connectivity in the brain, while the CPPNs are established as an abstraction for development Stanley [2007]. The topology of a CPPN is identical to an ANN; it consists of nodes, sitting in different layers of the network, which can have weighted links to other nodes. The nodes themselves however can have different activation functions from ANNs. While the ANNs use sigmoidal activation functions, CPPNs can have one of many activation functions, including but not limited to gaussian, sine, parabola and sigmoid (see figure 2.1).

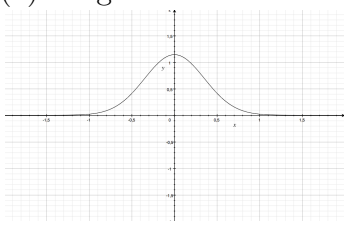
The CPPN is used in a way similar to an ANN as well, where it gets it's input through and array of input nodes, then passes that input on to the hidden layers via its connections. Each of the nodes takes the values coming into them and feeding them into the transfer function, the result of that function is then the output of that node. This output is then sent to the nodes connected to that node, and weighted according the the connections weight. This goes on until the it reaches the output nodes. The final output is then read out of one or many output nodes. You may have links from any



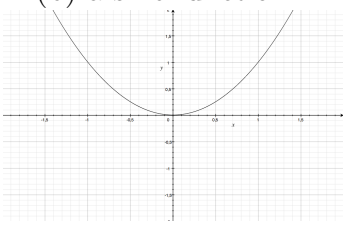
(a) a sigmoidal function.



(b) a sine function.



(c) a gaussian function.



(d) a parabola function.

Figure 2.1: These figures show the graphing of transfer functions. The figure in (a) shows a sigmoidal function, (b) shows a sine function, (c) shows a gaussian function and (d) shows a parabola function.

layer to any other layer, including recurrent links going "backwards" in the network.

"Learning" in the CPPN, as in the ANN, is mainly achieved by changing the weights or adding and removing links. A typical method for changing the links is what is called back-propagation. This is a method which based on the error of the output (the distance from the outputted value, to the expected value). It can given that error change all the weights but a small value from the end and back up the network, to minimize that error. If this is done over many iterations, and with a small change each time, the network will eventually converge on something that works (gives the expected output). There are many limitations with this method. One is that it requires a previous knowledge about exactly what it needs to output. This is not necessarily the case in all domains, like in my domain where I want to get a seemingly "random" result, and then check if it's any good. Another is that you need to decide on a topology in advance, and then train the weights for that topology. Since I want to discover both weights and topology, this is not applicable.

Furthermore, since the Transfer function in a CPPN can be one of multiple types, unlike the ANN, the topology of the net is more important. This is because in an ANN every node is a sigmoidal function, so a net with 3 hidden nodes has the same topology as another net with 3 hidden nodes. As mentioned above, in the CPPN these hidden nodes can contain different transfer functions. This means that if it's n possible transfer functions, there is 3^n possible permutations of that network. This is why the CPPN also has a larger search space.

A CPPN works in the same way as an ANN when it comes to application also, you enter numbers into the input nodes and then these number is modified by the links weights and go as input into the hidden nodes, which then apply their activation function on the incoming number. This goes on all the way down to the output node (or nodes) from which you read you final output and then interpret it.

2.2.1 Use of CPPNs

As I've already said, the CPPN is basically a composition of functions, however it's hard to visualize what that means. A good visual example of a CPPN, and specifically evolution of CPPNs are the picbreeder app Secretan and Stanley [2008]. Picbreeder is a web-app available from www.picbreeder.org that allows you to choose pictures to evolve for as many generations as you like. The point is that the pictures you will choose will create a new population of similar, but different pictures. The picture will increase in complexity as it goes on. Picbreeder works in much the same way as my own app works, by creating networks that given an x and an y coordinates outputs a value for that point, however picbreeder interprets that output as an gray scale value. This gives picbreeder the ability to create pictures. See figure 2.2.

figure 2.3 and figure 2.4 are two example pictures that I created with that app. Remember they are just compositions of the functions allowed in the described above in section 2.2.

Another application of CPPNs is to create ANNs. This is achieved by having a predefined "substrate" (a kind of predefined drawing board) on which

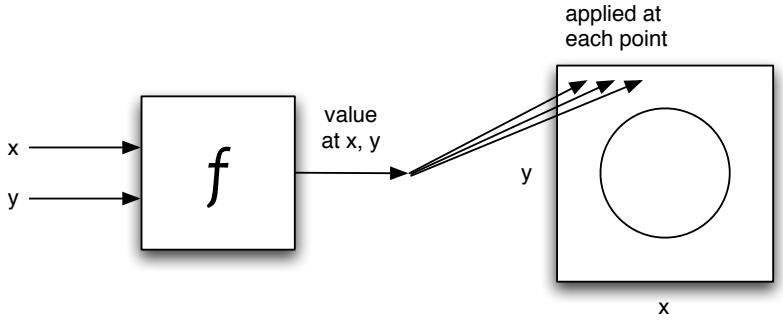


Figure 2.2: Here you can see a description of how CPPNs work. by getting the x and y coordinate as an input, and then deciding what to draw on that coordinate. This specific example results in a circle when all the coordinates have been queried. Recreated from Stanley [2007].

Figure 2.3: This image, created with the picbreeder application, looks like that of a strange little guy with headphones.

you can interpret the nodes of the substrate as the nodes of the ANN and then give your CPPN the x1, y1, x2 and y2 coordinates (which would describe the link from node 1 to node 2) and output the weight of that link. This is done by David. B. D'Ambrosio and Kenneth O. Stanley with a system they call HyperNEAT D'Ambrosio and Stanley [2007]. This methods creates ANNs that have geometric regularities, which can exploit the same geometric regularities as in the real world.

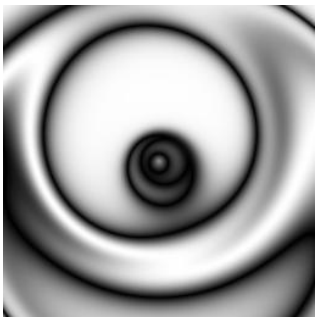


Figure 2.4: This image, created with the picbreeder app, looks like a close-up of an eye-ball.

2.3 Evolving CPPNs

0

Because of CPPNs topological and functional similarity too ANNs, the methods used to evolve ANNs can easily be adopted to create CPPNs instead there are many methods available to choose from Angeline et al. [Jan 1994] David W. Opitz [1997] Frederic Gruau [1996] Pujol and Poli [1997] Stanley and Miikkulainen [2001] Yao and Liu [1996] Zhang and Muhlenbein [1993]. Given that CPPN as an abstraction of development, one would want to use a method that maintains the essential properties of evolution in nature. One important property is gradual complexification, and currently the only algorithm to support this in both mutation and crossover is the NEAT algorithm Stanley [2004].

2.4 NeuroEvolution of Augmenting Topologies (NEAT)

NeuroEvolution of Augmenting Topologies (NEAT) is an approach to evolving ANNs that incorporates a specific set of methods to handle mutation and crossover, especially the concept of historical markings on the addition of links that makes it possible to preserve topology while doing crossover. It also has a concept called speciation, which groups similar individuals together

and allow them some time to refine as a group to find their niche.

2.4.1 Historical Markings

A historical marking is a unique number that is given to a link when it is created (see figure 2.5). This number makes it possible for it to align the correct links so that the crossover function can create a new network that contains the similarities from both of the parent networks. This feature of NEAT, combined with it's ability to add structure through mutation, enables it to complexify gradually, a feature which is important in real life evolution.

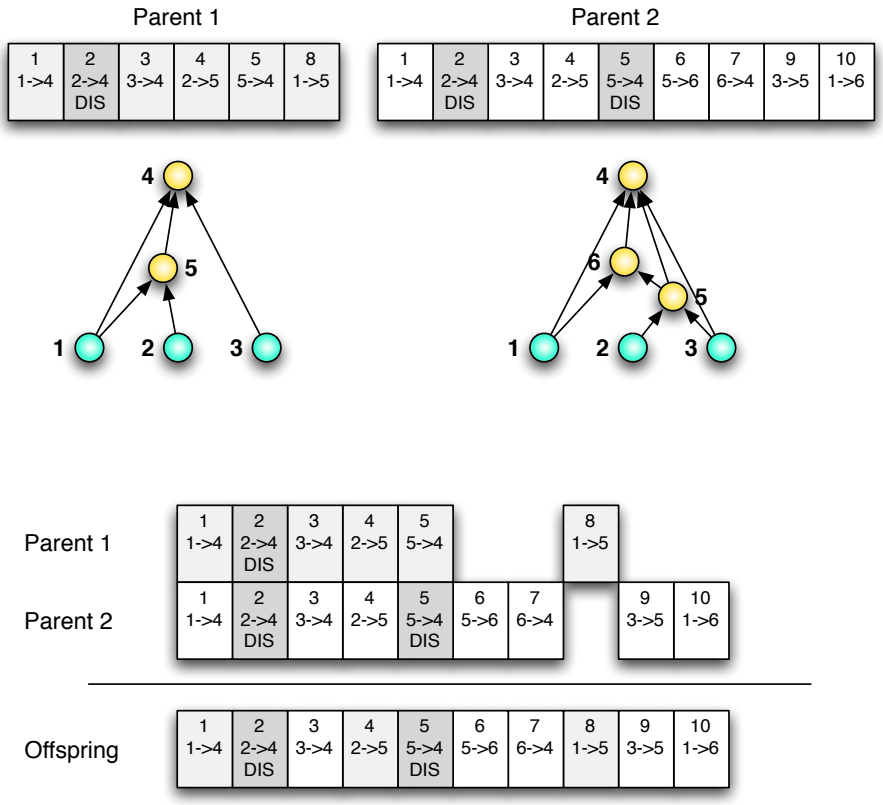


Figure 2.5: This figure shows two individuals (the parents) that 6 and 9 genes (links) respectively. But instead of just choosing genes at random, it aligns them based on the historical marking number and then and then chooses genes from each parent either randomly or by averaging the weights. Mathisen [2007]

2.4.2 Mutation

There are several different mutation methods in NEAT. The most common are:

- Adding a new link between two existing but unconnected nodes, as shown in figure 2.6.
- Taking an existing link and splitting it in two, adding a new node in between them as shown in figure 2.6. Here the original link could also be kept in addition to the new ones.
- Just changing the weight on an existing link.

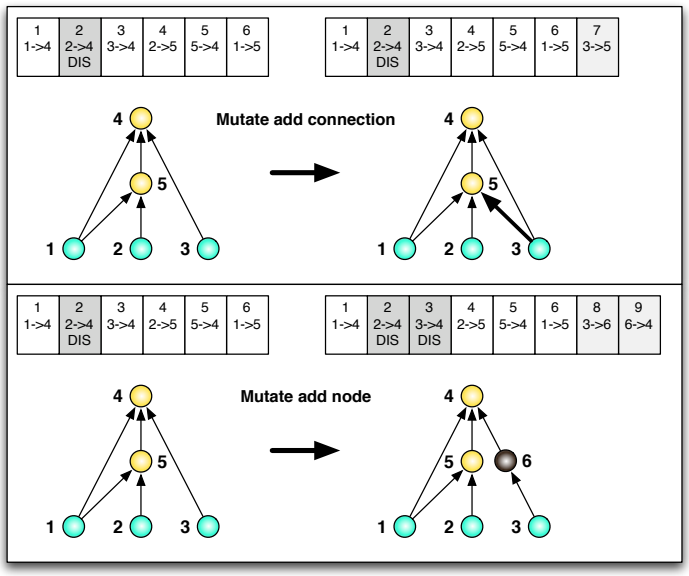


Figure 2.6: This figure depicts a NEAT genotype and the results when one mutates it by adding a connection and a node respectively. Based on a figure from Stanley [2004].

2.4.3 Speciation

NEAT also incorporates another concept from evolution, namely speciation. Speciation in NEAT means that you use a metric that calculates the ge-

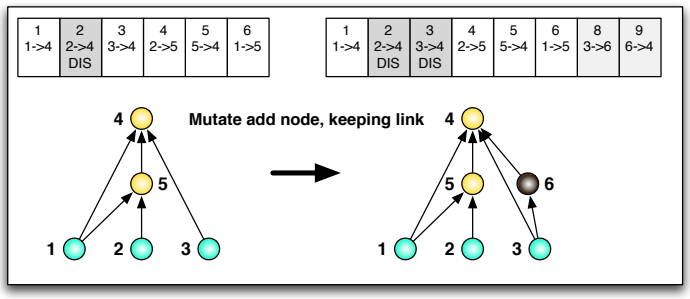


Figure 2.7: This figure depicts a NEAT genotype and the results when one mutates it by adding a node and keeping the link. Based in a figure from Stanley [2004].

netic distance to decide which individuals are structurally similar. Similar individuals are then grouped together, and share a fitness which is based on an average fitness. This helps these individuals traits to be protected for a preset amount of time, called the dropoff age. This gives the species time to evolve within itself to theoretically achieve the maximal fitness for that specific topology.

Chapter 3

Problem

3.1 Development

The process of developing a model of an eye poses many problems. First one has to have an acceptable abstraction of the environment in which the eye should develop. In the real world you have a intractable number of factors that affect the organism. Everything from different properties of light, to other external forces like heat and even radiation and internal forces like, blood flow to the eye and physical imperfections. All of these things are of course impractical to simulate accurately, and one has to settle on a set of useful abstractions.

Like in every application of GAs you have to have the correct selection pressure as noted in section 2.1. In this case you need to somehow figure out what drives the evolution of the eye in the real world, and try to re-create that in your fitness function.

The problem also includes the properties of the GA itself, where the different genetic operators need to be within certain ranges to be meaningful. These ranges vary from domain to domain, in the case of a CPPN these ranges include the mutation rate of the weights and the type and rate of the crossover operator. There is generally no automatic way of tuning those operators except perhaps applying a second genetic algorithm to search for them. This is generally very time consuming and impractical, and one usually

resorts to trail and error combined with intuition and insight into how the genetic operators affect the specific domain.

In this paper the genetic operators is tuned manually, and there is probably room in the final results to get better results by tuning them further.

3.2 Eye-specific Evolution

As noted above in the theoretical evolution paper 1.2.2, there is a likely evolutionary path along which the eye is thought to have evolved. That path starts out with a line of skin cells, a few of which are photosensitive. This patch of photosensitive cells' only function is to decide if there is light or not. It doesn't have any way to distinguish between direction or movement in any meaningful way. However this patch is thought to have sunk inwards, eventually creating a circular form behind the line, while getting a slit on top. This is shown in figure 1.2. It would be desirable for my algorithm to follow the same evolutionary path. It isn't however that likely that it will, since my algorithm doesn't necessarily start out with a "line" which is likely in nature, since that "line" represents a patch of the skin. There is also the possibility that an indentation might be a result of local interaction at that specific point, i.e. instead of the genes encoding that the skin is supposed to be a little curved, it encodes that it's not supposed to grow cells behind the skin, which indirectly creates the curve of the skin. While both methods are enable to encode that curved line, the method in which they both achieve them may be very different.

3.3 Selecting the right tools

There are many obstacles to capturing the essential properties of natural evolution. One important one however is the ability of the underlying system to actually represent what you want to end up with, this was mention in section 3.2. However, that is not the only problem, you may also how the problem of to much expressibility. A classic problem of all AI is that the

more expressive the representation is, the larger the search space is. The larger the search space is, the harder it is to find the solution, and ergo the more time you would need.

3.3.1 Tractability

Tractability in a modeling domain is highly dependent on the resolution of the model, and with the precision of the simulation. The more objects you include in your structure, the more objects you need to keep track of. In a domain where the relationship between the objects is crucial, the time taken to account for all those relationships grow exponentially. This also has an effect of the precision of the simulation, the more detail you include in each of the interactions, this also grows exponentially on top of number of objects.

In my specific implementation, the pruning of the rays of light take a lot of time. This is because for every ray I have to check every other object in the world, to see if they are blocking it or not. Given that I usually have 1000 individuals, that means that a small increase in time per individual will add up. When the time grows exponentially with the number of cells per individual, pruning time becomes an issue on larger sizes.

3.3.2 Biological Plausibility

For the results to have meaning other than validating the effectiveness of the underlying algorithm, the results should be interesting in their own right. This means that the system should be based on natural principles in a degree that the results from it is admissible in a biological discussion. My specific application tries to validate and re-create the theory that an eye could have evolved in a relatively short time, based on the simple theory of information retrieval and the simple and plausible idea that development can be approximated by simple functions. To achieve this I need an environment that captures enough of the features of the real world to be valid. This is both a problem of accuracy and resources. The more accuracy you add, the longer the run-time will be. It's also a question of how accurate you are able to make it. It's also a question of what is necessary and what is not.

Take for example an external force like heat, heat undoubtedly affects an eye somehow. However, it doesn't mean that it's crucial to the evolution of an eye to work optimally at different temperatures. It's plausible that it has some effect on it, but it's hardly crucial.

Chapter 4

Method

This chapter will try to detail the methods and environments underlying the system developed to explore the questions outlined in chapter 3.

4.1 Simulation Environment

My simulation environment or my world is where I test my network. As stated above I had to capture the essential properties of the world with respect to what's needed to evolve my final shape in much the same way as in the real world. To do so I decided I needed at least the following concepts:

Abstract Cell: All the objects in my world is what I call a cell, this does not necessarily mean the type of cell found in a body, but just means an object which has an x and an y coordinate. It also indirectly has a size, but this is not an individual value on each cell, but a value given indirectly by the rules of my world. It is however dynamically changeable in each instance of the world.

Light sources: I needed a type of object that emitted light, from which the eye needed to somehow discriminate. This source needs to be able to have multiple positions, and be at a distance from the eye. If I allow it to be everywhere, or allow the eye to be close to it, my eye might evolve to encapsulate the light sources to "capture" the light from it.

I decided to make a unique object which I called a lights source (LS)

which I allowed to be along a vertical column to the left side of my eye's available space. I have a configurable distance which I call the LS distance. This distance is an open space between the sources and where the eye is allowed to develop which ensures that the eye can't "cheat" and encapsulate the light sources as mentioned above (However, I can set it to 0 to confirm that it is indeed able to exploit these "cheats" and make exotic shapes which gets a high fitness).

Light Sensitive Cell: I also needed an object which equates to a real worlds eye photoreceptors located on the retina. I call those objects Light Sensitive Cells (LSCs). These cells are only allowed to be drawn in the space to the right of LSs and the LS distance.

Opaque Cell: There needed to be a type of cell that light could not penetrate, this could be seen as the kind of cell in the iris (which makes out the slit in the eye) and the skin cells around the eye that makes sure light doesn't come in any other way than through the eye's slit. I call this type of cell an Opaque Cell (OPC).

Light Vector: A light vector is basically the line from a LS to a LSC. It represents the ray of light from that LS, hitting the LSC. This ray has no thickness or other properties, and it is just represented mathematically as start and end coordinates.

The types of objects in my environment can be seen in figure 4.1. The Green objects represent the Lights Sensitive Cells (LSCs), the black objects are the Opaque Cells (OPCs) and the blue object along the left margin is the Light Sources (LSs).

In figure 4.2 you can see that I create all the possible light vectors in an image, and then prune (remove) the ones that are being blocked. The blocked vectors are shown as red, while the non-blocked are shown as yellow. For the rest of this document, I will not draw the blocked (red) vectors because of the amount of them usually present in the images.

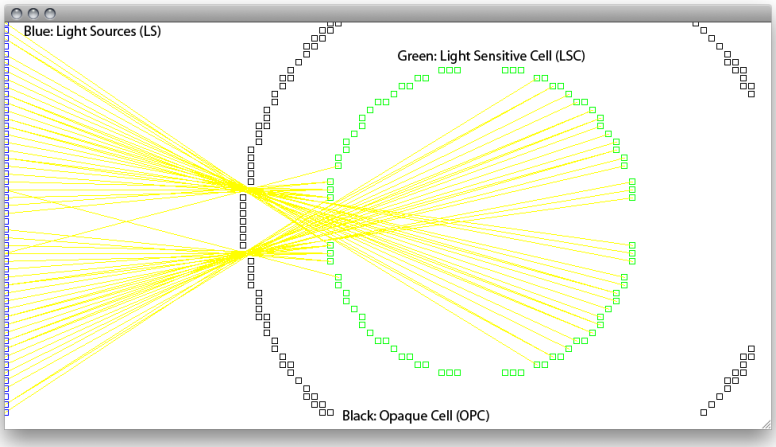


Figure 4.1: This is an example of my simulated environment. The green objects represent the Light Sensitive Cells (LCSs), the black objects are Opaque Cells (OPCs) and the blue objects all the way to the left are Light Sources (LSs). The yellow lines represent non-blocked light sources.

4.2 The Physics of the World

In addition to the objects in my world, I also need a set of rules to govern how the different things interact. These rules should also mimic on a high level of abstraction how the real world works. Again, I do not consider exotic effects like, the bending of light or the refraction of air, but try to keep it as simple as possible.

To decide if a LSC is hit by a given LS, I check to see if any other type of cell (LSC or OPC) is blocking it. This is done by seeing if the perpendicular line from OPC to the light vector exists and is closer than a given number (the size of the cells). If it is, I consider the light vector to be blocked and remove it from the list. However if no cell in the world is blocking it, it is allowed to stay. When I do this to every vector I have an environment where the light vectors represent the rays of light that hit a photoreceptor in the eye.

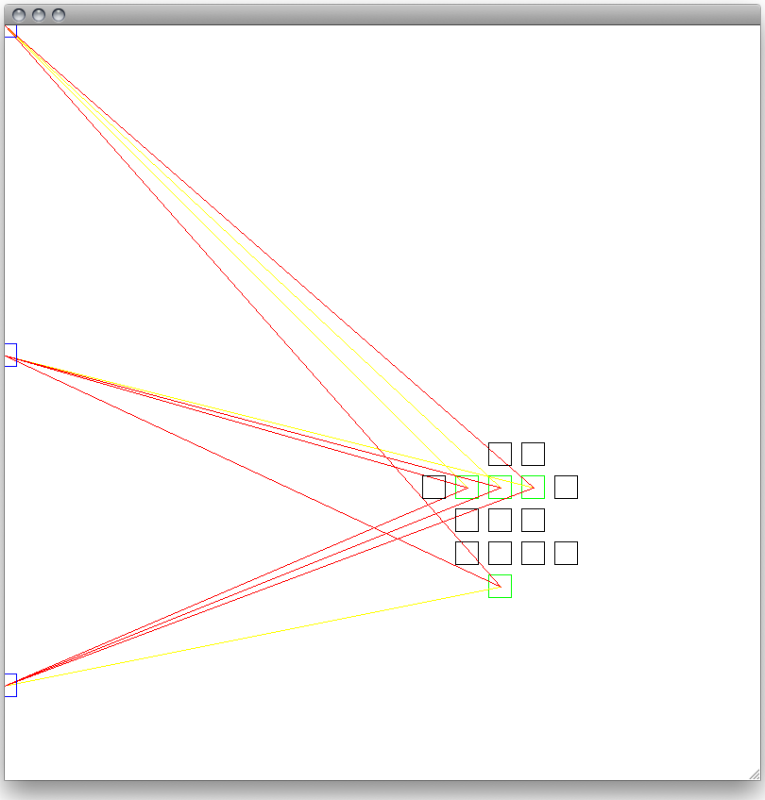


Figure 4.2: This is a simpler structure that shows how I create all the light vectors, and then prune the ones that are blocked by something. The red lines are blocked light vectors.

4.3 Creating the World with the CPPN

The world and it's objects are created by the CPPN which I evolve. The CPPN, as explained in chapter 2.2 is very similar in form and in use as an ANN. This means it has a set of input nodes in which the input is fed, and one or more output node from which the output is read. As input I give the x and y coordinates in my world, and in addition I give it the euclidean distance from center.

As the output I chose to use more than one output, and instead I went with 2, one which tells me if it wants to create an LSC and one that tells me to create an OPC. My output nodes always has a sigmoidal transfer-function,

but it's adjusted so as to give a value between -0.5 and 0.5. I then interpret it to be firing when, the value is between -0.15 and 0.15. If both is within that range, the one closest to zero is the one that is selected to be drawn. If none are within the range, that is interpreted as if no one is firing, and subsequently nothing is drawn. This is then applied to all the x's and y's which is limited by a predefined value of how big my eye's available space is (i.e. xmax=40 ymax=40).

4.4 Fitness Function

As said in section 2.1 the fitness function is a metric that tries to rate how well the eye performs. That means that it needs to say something about the eye's ability to gather information from the environment. In this case this means that, when a LS is "firing", the eye should somehow know from where, or in other words, which LS that is. The only way to know something about that is if it consistently hits the same LSC. This means that there is inherent information in the shape. However the eye does not know what to do with that information, nor is it in the scope of my assignment to use this information for anything.

Basically what would be optimal for me is a one to one relationship between every LS and an LSC. That means that for every LS there is one LSC that is by that, and only that LS. That LSC could then theoretically identify that LS whenever it was active. I therefor need to reward one to one relationships. Conversely, I need to punish LSCs that are hit by more than one LS, the more hits the worse it is as a source of information.

My fitness function has a theoretical value between 0 and 1, where 0 is the worst and 1 is the best. The optimal number 1 is never reached though, since I have a few additions that penalize specific behaviors that, while scaling, always pulls the fitness down a little. These specific metrics will be described below. Given these additions, the optimal number is between 0.7 and 0.8.

Specifically I try to achieve this with a two-part function. The first parts checks every LSC that is hit by an LS. If that LSC is only hit by one LS (which has to be the current one), it gets a value of 1, if it's hit by more

the value decreases with the number of other LSs that hit it. This number is then averaged.

Algorithm 1 Fitness algorithm element 1

1: **for** $i = 0$ to $getSize(LS)$ **do**
2: **for** $j = 0$ to $getSize(getHitLSCs(LS_i))$ **do**
3: $E_2 \leftarrow E_2 + (1/getNumberOfHits(getHitLSCs(LS_i)_j))$
4: **end for**
5: $E_1 \leftarrow E_1 + (1 - (getCurrentError(LS_i)/getMaxError(LS))) * (E_2/getNumHits)$
6: **end for**

Secondarily I also create a number of the LSC, where I get the number of average hits for all the LSCS on the form)1/hits)/numLSCs. This further hurts structures where a lot of LSCs are hit by a lot of LSs.

Algorithm 2 Fitness algorithm element 2

1: **for** $i = 0$ to $getSize(LSC)$ **do**
2: $S \leftarrow S + (1/getNumberOfHits(LSC_i))$
3: **end for**
4: $E_3 \leftarrow 1 - (S/getSize(LS))$

I also incorporate a different aspect into my fitness-function; in the real world, it is more expensive to grow a bigger structure. If that extra structure also does not serve any useful function that is a waste of resources. To combat this I also penalize larger structures, meaning the more cells you have, the bigger your penalty is. This should help reduce size, while still keeping a larger structure if it adds significantly to the information retrieval.

Algorithm 3 Favoring Less Cells

1: $F \leftarrow 1 - getTotalNumberOfCells()/(getXMax() * getYMax())$

Keeping with the biologically inspired theme of my fitness function, I also penalize structures growing very broad. In the real world, a structure usually grows outwards from a central starting point. For my eye to suddenly spawn helpful additions far from the originating central point isn't very plausible, since it would need something to grow from. This is indeed possible with

CPPNs when you interpret an output as "nothing" and I interpret something not firing within my parameters as empty space. A point can be made that the empty space is just transparent cells that exist within my model though, but that isn't a very common solution in nature. I therefor have a metric making cells existing far from the center pulling the total fitness down. This should also help keeping the size down but still allowing for helpful additions straying from the center. It would keep the bulk of the structure close to the center and mostly connected.

Algorithm 4 Favoring Closer to Center

1: **for** $i = 0$ to $getSize(LSC)$ **do**
2: $S \leftarrow S + sqrt(pow(getX(LSC_i) - getCenterX(), 2) * pow(getY(LSC_i) - getCenterY, 2))$
3: **end for**
4: **for** $i = 0$ to $getSize(OPC)$ **do**
5: $S \leftarrow S + sqrt(pow(getX(OPC_i) - getCenterX(), 2) * pow(getY(OPC_i) - getCenterY, 2))$
6: **end for**
7: $S \leftarrow S/(getSize(LSC) + getSize(OPC))$

The different parts of my fitness function is always normalised to a value between 0 and 1, this is not necessarily needed, but it makes it easy to see what values the different penalties may scale it with i.e. a penalty that has the potential of being between 1 and 0.5, has the ability at worst halve the original fitness, if this is to much, I may change it so that the worst case value is only between 0.75 and 1. It also adds another level of checks to my different metrics, since if the value is either less than 0 or higher than 1, I can deduce that something is wrong with the function.

4.5 NEAT Specifics

The NEAT system as described in section 2.4 is originally created to evolve ANNs, but it is easily extended to create CPPNs instead. It basically involves modifying it to use more transfer functions than the sigmoidal function usually used. The Neatzsche implementation of NEAT already has support for

sine, parabola and gaussian, and enabling them is as simple as setting a value in the settings file. I also added support for cosine and the identity function. This gives me a complete set of the generally accepted CPPN functions, but adding different and more exotic functions should be both simple and interesting to try.

Enabling the system to work with my simulated environment, and having the system create it worked by creating my own instance of the Evaluator class, which has it's evaluate function applied to each of the individuals in the population. This evaluation function creates my world from the phenotype (the CPPN) by taking all the coordinates as input (along with some other useful information like xmax, ymax and distance from center) and then interpreting the output as either create a LSC, create a OPC or do nothing. After this the fitness function is run on that instance of the world, and each individuals fitness is set.

Chapter 5

Results

This chapter will outline the results, under various conditions and settings. I will start out by showing some of my early and interesting results which I got while developing and testing my application. These will help explain some of my choices for the fitness function as well as show that those changes might be beneficial.

I will then go on to show results on different sized worlds and other different settings. This will vary from worlds as small as 20x30 to worlds as big as 120x200, which is also run on a cluster.

Lastly I will also check another property of the CPPN which is it's ability to scale to higher resolutions because of it not storing an image, but a functional representation of an image (much like an SVG image).

5.1 Early results

During my development process, while I was trying out different fitness functions, CPPN-inputs and simulation settings it did find some interesting results.

As seen in figure 5.1, it creates a novel mode to get a good fitness, by dividing the light sources up by walls, making sure that if any of those LSC cells are hit, it can be sure from which light source it is. I figured that this was cheating though, since in the real world, you are not allowed to come

close to most light sources with your eye, but it is still an interesting result.

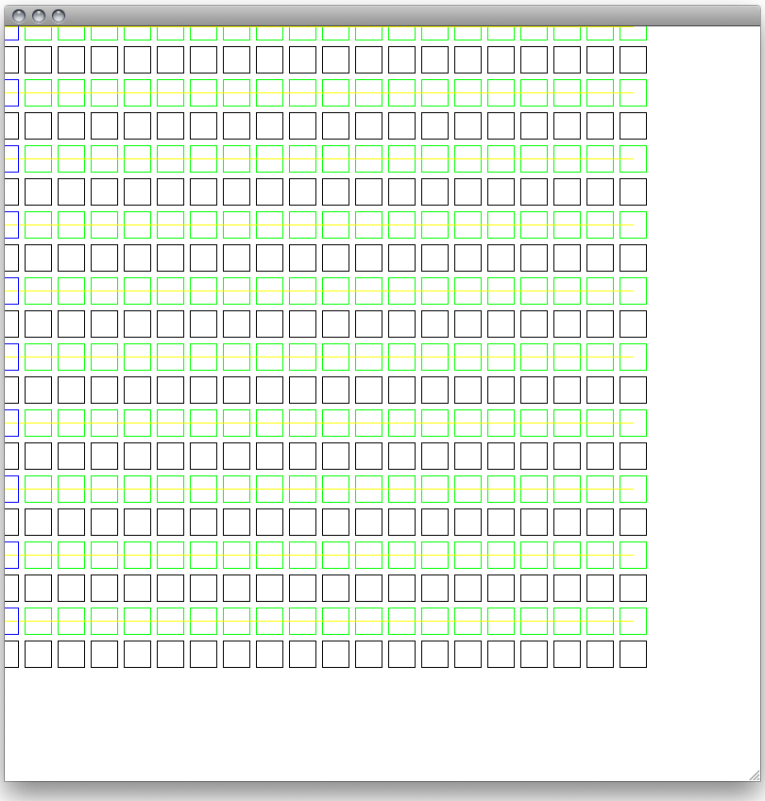


Figure 5.1: This shows what was back then an optimal solution, with as close to 1 to 1 as you could possibly come. In this early fitness function my LSCs didn't block light (only the OPCs did that), and my optimal number to hit was LSC/LS.

5.2 20x30 with 5 light sources

5.2.1 Experiment setup

This setup is a 20x30 world with only 5 light sources. It was run with a population of 1000 for 1500 generations. Light sources distance is 10 and cell size is 0.7.

5.2.2 Experiment result

The resulting structure is shown in figure 5.2 and the corresponding graph is shown in figure 5.3.

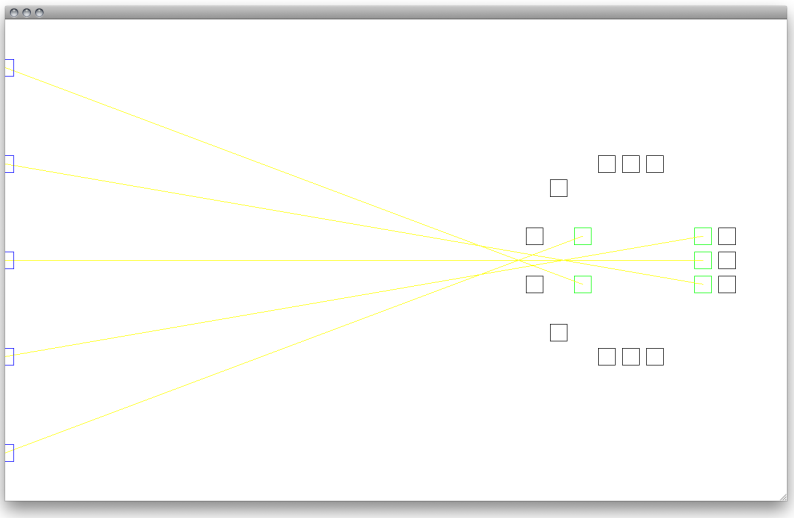


Figure 5.2: The visualisation of my 20x30 world run with 5 light sources. It round structure with all light vectors passing through the central slit.

5.3 20x30 with 20 light sources

5.3.1 Experiment setup

This setup is a 20x30 world with the max possible amount of light sources. It was run with a population of 1000 for 1500 generations. Light sources distance is 10 and cell size is 0.7.

5.3.2 Experiment result

The final structure is shown in figure 5.4 and the graph is shown in figure 5.5

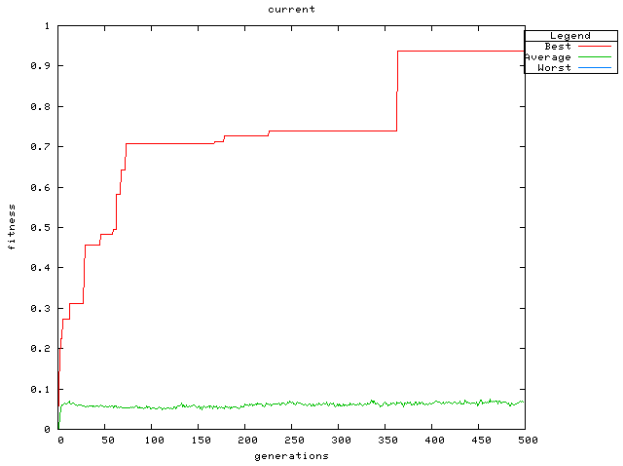


Figure 5.3: The graph of my 20x30 world with 5 light sources shows the best fitness going up to about 0.7 fairly fast, and then jumping up to about 0.9 at around generation 800.

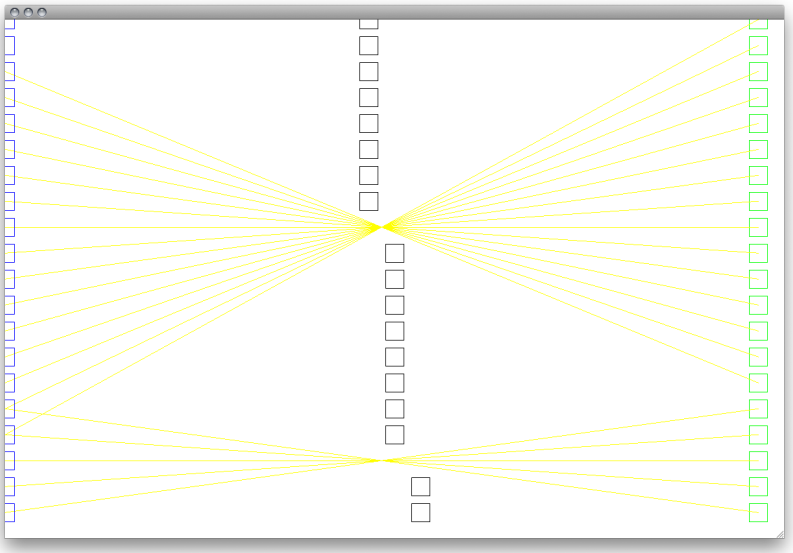


Figure 5.4: This shows the best solution in a 20x30 world with 20 light sources. The solution is a very good solution, but uses a 2 slit solution and two light sources hitting multiple LSCs.

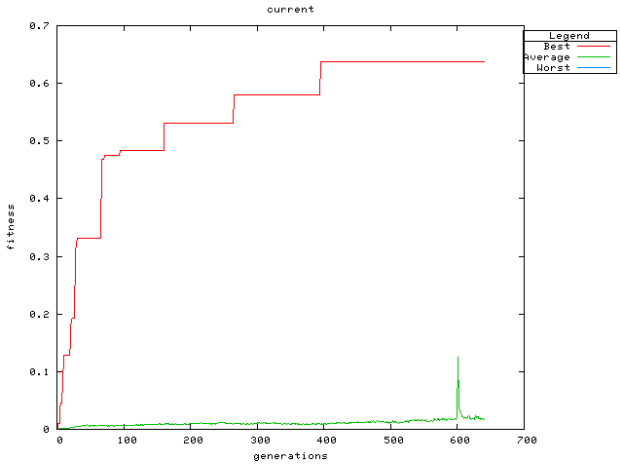


Figure 5.5: The graph of the 20x30 world with 20 light sources shows a steady progression up to a final fitness of 0.63.

5.4 70x90 world

The 70x90 world is a fairly large environment for my algorithm to evaluate, it takes about 2 hours for about 200 generations. This is a fairly short run, but because of the long run time and the almost optimal fitness I decided to stop it there.

5.4.1 Experiment setup

This setup runs, as stated above, for 200 generations. The population size is 1000. I use the maximal possible number of light sources, 70 and an light source distance of 35. The cell size is 0.7.

5.4.2 Experiment result

The resulting structure is shown in figure 5.6 and the graph is shown in figure 5.7.

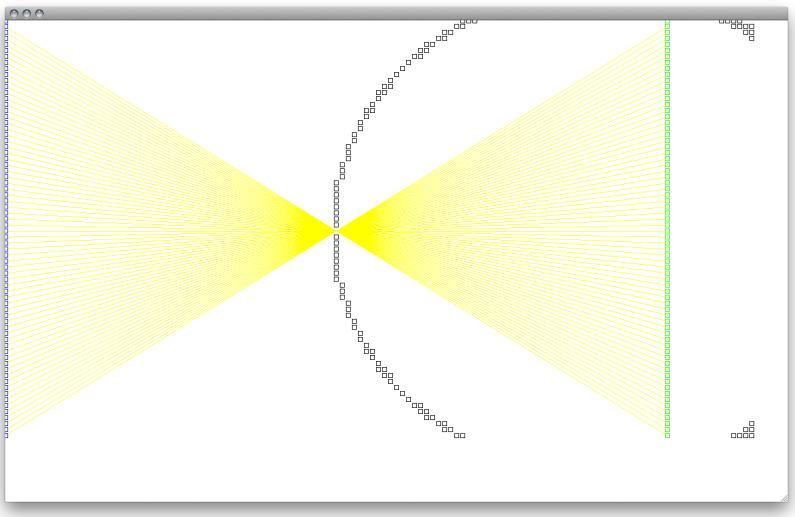


Figure 5.6: This shows the best solution in my large 70x90 world. The solution is almost optimal, take a few artifacts.

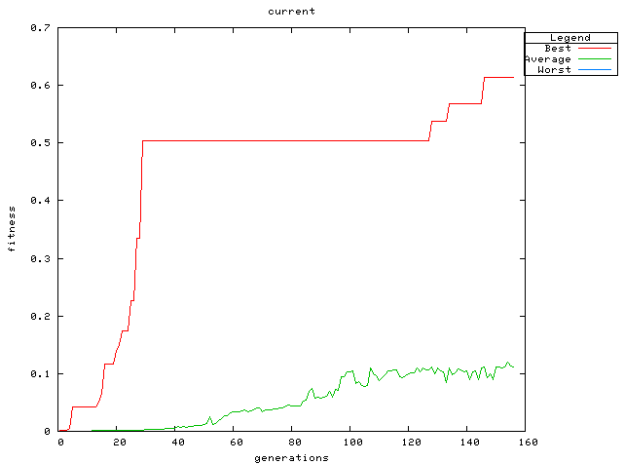


Figure 5.7: The graph of my 70x90 solution. This run stops at 200 generations at a fitness 0.614.

5.5 120x200 with 60 light sources

5.5.1 Experiment setup

This setup is run on very large 120x200 world, with 60 light sources. It was run with a population of 1000 for 800 generations. Light source distance is 50 and cell size is now 0.65 to give it a little more angle because of the long distances. This entire run took about 16 hours on 15 nodes on the norgrid cluster.

5.5.2 Experiment result

The final structure can be seen in figure 5.8 and the the fitness graph can be seen in figure 5.7.

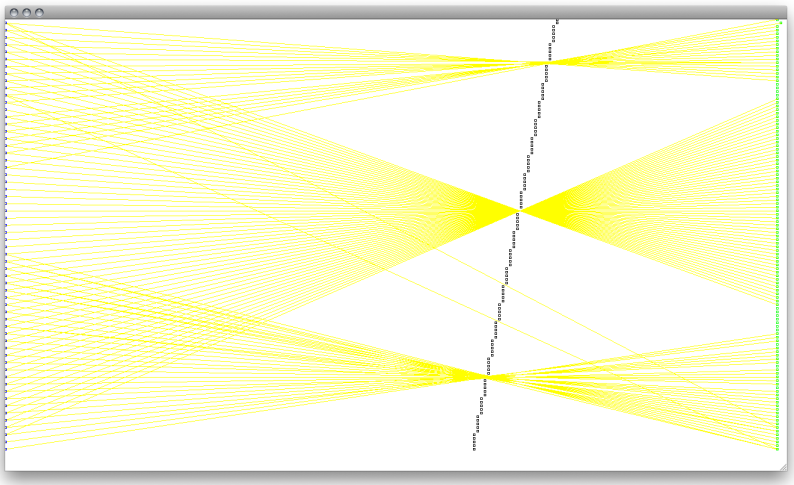


Figure 5.8: This shows the best solution in my large 120x200 world with 60 light sources. The solution isn't very elegant, but it achieves almost 0.7

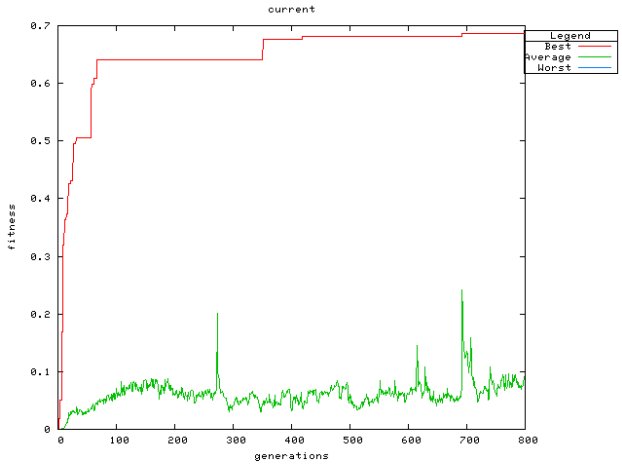


Figure 5.9: The graph of the 120x200 world with 60 light sources, this one finds a solution fairly early in the run, and marginally improves on it.

5.6 120x200 with 120 light sources

5.6.1 Experiment setup

In this experiment I run the very large 120x200 world with 120 light sources. This is my longest run so far and it took about 18 hours to complete all 800 generations on 15 nodes on the norgrid cluster. The population size used was 100. Light source distance was 50 and the cell size 0.65.

5.6.2 Experiment result

The structure is seen in figure 5.10 and the corresponding graph in figure 5.11.

5.7 10 runs on 50x75 with 10 light sources

5.7.1 Experiment setup

This is a special case where I run multiple runs on the same environment with the same settings. This was done to get a better overview of the possible

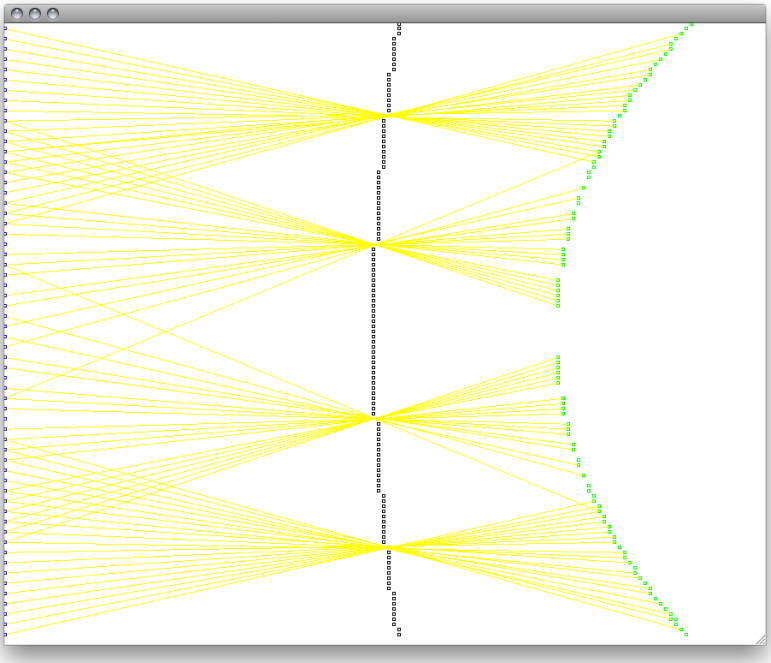


Figure 5.10: This shows a solution in my large 120x200 world with 120 light sources. This is not one of the better ones with a fitness under the 0.5 mark.

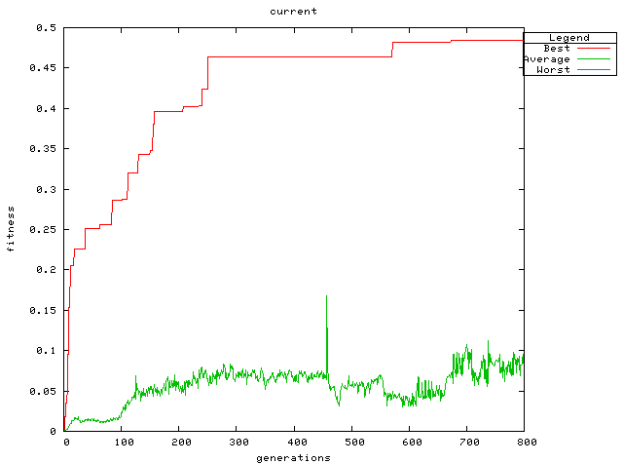


Figure 5.11: The graph of the 120x200 world with 120 light sources. This run finds small incremental improvements over the first 300 generations, and then seems to have problem finding better solutions on the path it has taken.

outcomes, as well as the different paths it's taking. The 50x75 with 10 light sources environment was chosen as a compromise that is fairly complex and still tractable. I chose to use only 10 light sources, because that takes down the run-time quite a while, while still getting fairly interesting results. The run took 10 hours on 15 nodes. In the interest of space, I won't be showing all the structures and graphs only the interesting and different ones.

5.7.2 Experiment result

In this run I of course got 10 different structures, some better than others. The first interesting one is this circular shaped one (figure 5.12), which has a very specific solution, but actually found the solution late in the run as can be seen in the graph in figure 5.13.

Another interesting structure is the one seen in figure 5.14. It's shape, with both a rounded "front" with a slit in the middle and a rounded back, looks very much like an eye in nature. It's fitness of 0.76 also says that it's function is fairly good.

The structure in figure 5.15, is similar to that of 5.14, however it has slightly higher fitness.

In figure 5.16 you can see a kind of hybrid structure, which uses a small circular LSC setup and a long vertical slit solution.

5.8 Scaling the world

As mentioned and explained earlier, the CPPN has the ability to scale to any resolution. To see how this feature manifests itself in my specific domain I will try scale up one of my more general solutions and see what happens. I will scale the 70x90 structure (with 70 light sources) described in section 5.4 and shown in figure 5.6.

I will scale it 10 times up to 700x900 with 700 light sources. As seen in figure 5.17, the new structure is a higher resolution version of the old structure, except from the slit, which is now turned into two slits, fairly close to the center. This will be discussed further in section 6.

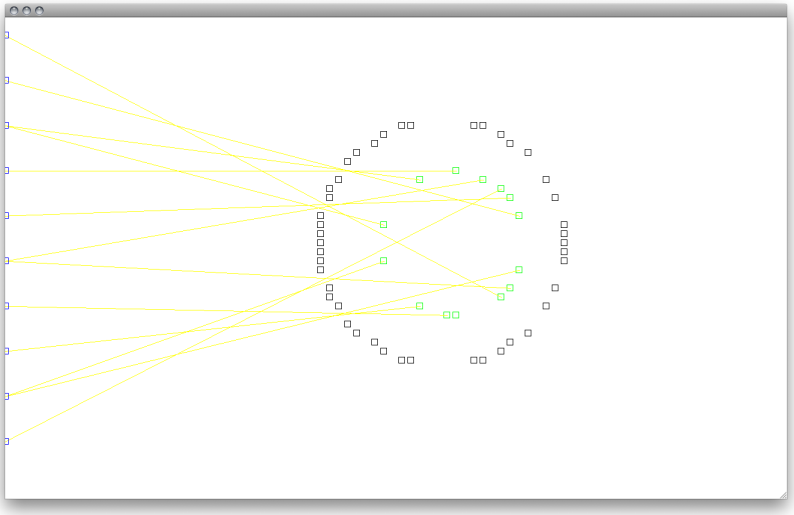


Figure 5.12: This one is kind of interesting, since while having a high fitness (0.83), it actually got the high fitness very late in the run.

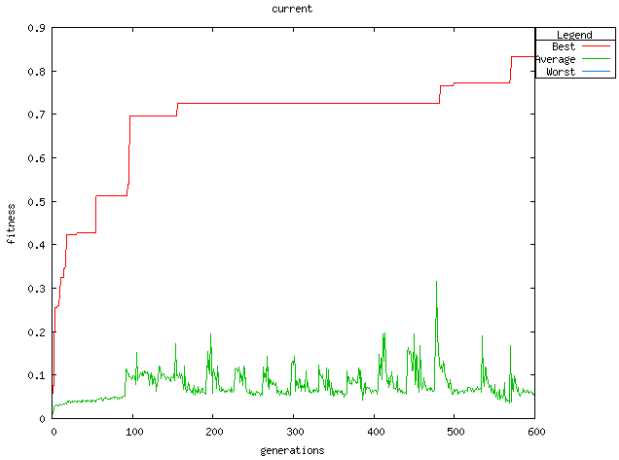


Figure 5.13: As can be seen in this graph, this is one of those runs that flat out early, but then suddenly finds something interesting and starts going up again.

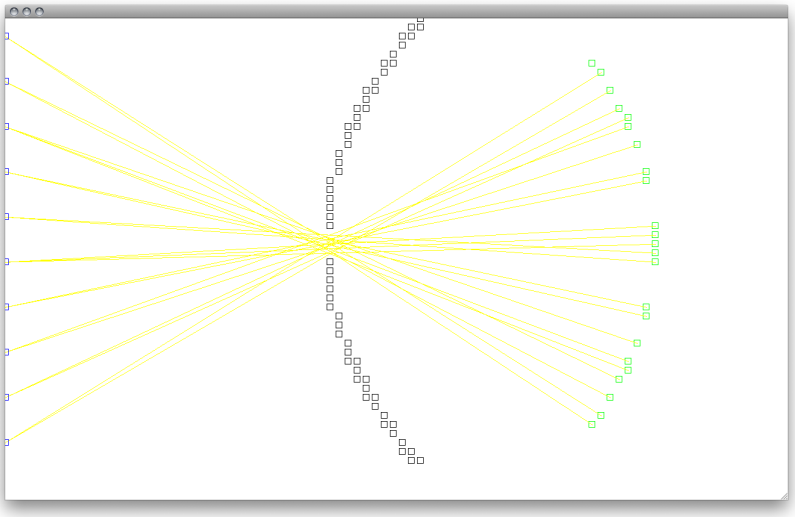


Figure 5.14: Here is another solution that is interesting. Especially because of the shape is very similar to an actual eye. The fitness is 0.76.

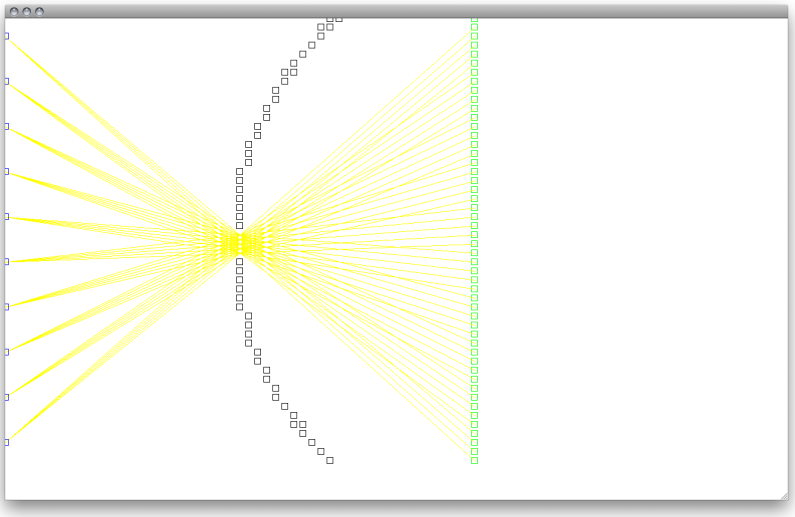


Figure 5.15: Very similar to that of figure 5.14, but with a slightly higher fitness (0.76) .

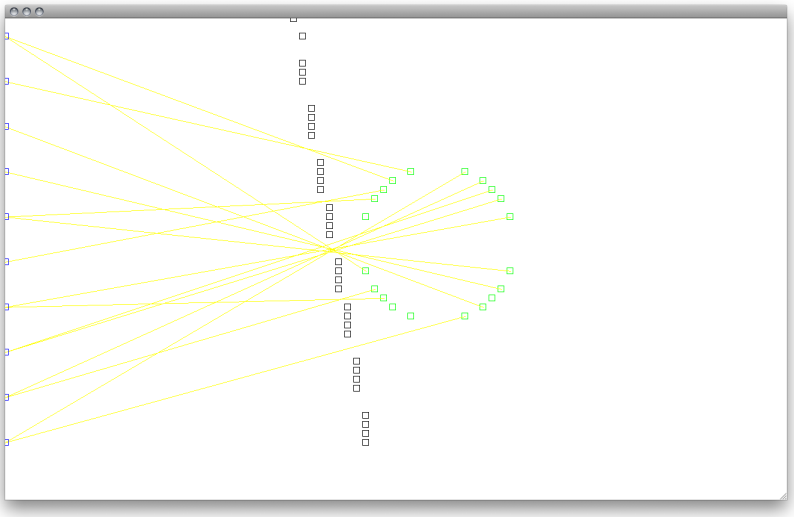


Figure 5.16: This structure looks like a kind of hybrid structure that uses both a small circular LSC setup and a vertical line as the slit solution. The fitness is 0.7.

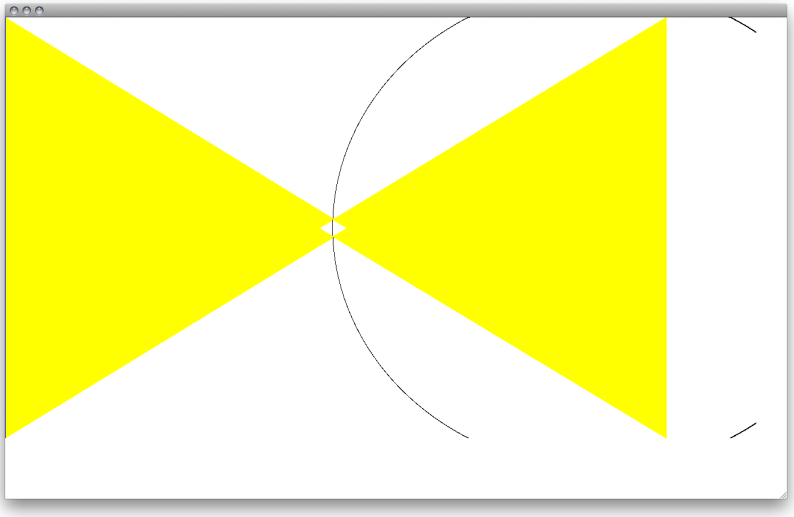


Figure 5.17: This figure shows the 70x90 structure scaled 10 times to 700x900. It still contains the same basic structure.

Chapter 6

Analysis of Results

In this section I will present comments and analysis related to the results presented in the previous chapter.

6.1 Performance

The performance of my system seems to be very good. The more LSCs I have however, the slower the pruning will go. The evaluation on the other hand seems to be always fairly fast, but it does increase with the number of light vectors. The running of the network itself to create the objects are negligible while it's small, but starts to add to the time at around generation 1000 (when run with a population of 1000). At this time the complexification process has come so far that the number of node may reach 100-150, which is a very large network, and the number of links may reach 1000.

The performance of the CPPN in itself is especially easy to test with the scaling experiment. On the original 70x90 world, it takes roughly 2 seconds to draw the entire world (3600 points to calculate), while at the 700x900 (630 000 points) it takes about 3 minutes. That gives about 0,0005 seconds per point, which is very fast and very scalable.

6.2 20x30 with 5 light sources

This result is actually very interesting since given it's fitness of almost 0.94 (see figure 5.3) is very close to being an optimal solution. The solution itself though is a very specific solution to that specific setup. If tested on more light sources or a different setup of the light sources the fitness would probably drop. It does however show that the algorithm does a good job at searching the space for solutions. It's also interesting to note that even though it's a very specific solution, adapted to a small amount of light sources, it uses many of the properties I would expect to see. As seen in figure 5.2, it uses a minimal amount of cells (and I do believe that the excess cells, i.e. the OPC all the way to the back, would eventually be removed had I let it run for more generations), and it's fairly small, with most of the cells spread out in a circular fashion around the center.

If we analyse the graph at figure 5.3, we see that it quickly does a lot of improving the first 100 generations. And ends up at around 0.7. Then it's fairly little innovation for about 300 generations before it has a major breakthrough and stops at the final fitness. There is little extra information in the median and the minimum fitness, the medium fitness is fairly stable at around 0.6 for the entire run, while the minimum fitness is too low to even show up on the graph. This means that there is always at least on of the population that is completely useless. This is usually a bad sign when it comes to the health of your algorithm. In this case though, given the complexity of both the representation and the domain, it is acceptable that at least on individual is useless. One could imagine that i.e. an individual with no LSC, or an individual with all the LSC out of reach (being blocked by a row or block of OPC) is very likely to happen.

6.3 20x30 with 20 light sources

With more added light sources, it doesn't seem to be able to find a solution that has the same circular shape concentrated towards the center, as in the 5 LS experiment. It does however find a fairly optimal solution (as seen in

figure 5.4), which uses two slits near the center of the horizontal plane. The central line of OPC does seem to either be a tilted line, or part of large circle. It's interesting to note that even though two LSs aren't hitting anything in this structure, two other LSs are hitting two LSCs, so there aren't any LSCs that aren't hit. This might be seen as redundancy, it's good to have a backup LS if something stops working.

The fitness graph shown in figure 5.5, shows a steady progression towards the final fitness value. Again, both the average and minimum fitness values doesn't tell us much of interest, however there is a sudden spike in the average fitness at around generation 400. This is usually because of a low-fitness species dying off and the average fitness goes up again until a new species fills the empty species slot.

6.4 70x90 with 70 light sources

The 70x90 world has a very simple and elegant solution as seen in figure 5.6. It consists of a circle of OPCs surrounding a line of LSCs. It uses one slit, which is exactly in the middle between the LS-line and the LSC-line. This makes this nearly a symmetrical figure, except from the circle surrounding it.

The graph of the fitness for this structure, seen in figure 5.7, is a little more interesting that the other graphs seen earlier. While the max fitness follows the same shape as the other graphs, the average fitness is different in that it goes up later than the others and that it is a little higher.

6.5 120x200 with 60 light sources

This solution (figure 5.8) on the other hand is not very elegant compared to the one in section 5.5. This solution uses 3 slits, spread out on a tilted line. Since there is double the number of LSCs than that of the LSs you'd expect about half of the LSs to hit more than one LSC, and this indeed happens.

The fitness graph shown in figure 5.9 looks like the rest of my fitness

graphs. The fitness goes up fairly fast and at around 100 generations it has a solution with good fitness. The rest of the generations has marginal gains (about 0.03 up in about 700 generations). The average fitness is behaving as usual too, it goes up to 0.06 or so fairly fast and then fluctuates a little around that mark. The spikes can again be attributed to a low-fitness species dying which makes room for more of the high-fitness individuals. However since the species size is preset to a specific goal number, a new species soon fills up the space left by the dead species.

6.6 120x200 with 120 light sources

The structure seen in figure 5.10 is another innovative solution, as it uses four slits and two rounded structures. The row of OPCs seem to be part of a huge circle, so large that it almost appears straight at first glance. The sub-structure consisting of LSCs on the other hand, seems to be a smaller circle, and might even look like its not completely circular but perhaps elliptic. While the fitness of this structure isn't optimal, the solution is indeed interesting and even quite esthetically pleasing.

The graph in figure 5.11 shows that the fitness slowly improves over the first 300 generations. This is much slower than my earlier results, and might indicate that the path it took early in this run wasn't very suited for this domain. It might also be that the choices made early a run defines how the final solution will look. It's interesting to note that the average fitness is very low until it hits the 100generation mark. This might indicate that it only had low-fitness individuals to work with early on, and instead of searching broader for newer individuals it improved on those.

6.7 10 runs on 50x75 with 10 lights sources

This run was interesting in that you can see the difference in results of 10 runs with the exact same settings. That lets you see how a different start and path can change the final result. It's interesting to note that, even though there

are differences, it seems to be prominent 2 solutions it comes up with. Those are to use two smaller circles as seen in figure 5.12 and then there's to have the larger circle with a structure of light sources behind it as seen in figure 5.14 and 5.15. I've seen structures similar to these in all the earlier results too, but this experiment shows that they aren't really that setting-specific. They are able to evolve both structure-types given the same initial settings. This is interesting and shows that chance is important. It's interesting to note that given the 10 runs, there are 4 of each, and two that are hybrids like the one in figure 5.16.

The maximum fitness found by each of the runs are also very good. None were below 0.5, and the average is just under 0.7. This tells us that it's hard for this algorithm to get stuck at low local optimum, which is very good.

6.8 Scaling Experiment

As seen in section 5.8, the CPPNs do indeed scale as expected. What happens is that even though you turn up the resolution, the structure stays almost the same. All the shapes get more detail i.e. the circle look more circular and is not as jagged as before. The structure does change a little however, as can be seen in the slit, which is now two slits. This might have many reasons though, like the function responsible for creating the slit is dependant on some static value from the inputs, or just be an that the lower resolution introduced some rounding errors resulting in the absence of the cells.

Chapter 7

Discussion

This chapter will present the findings from the experiments in the perspective of the motivation and working hypothesis presented in sections 1.3 and 1.4.

7.1 The Results

The results show that the CPPNs found by my algorithm, both have the ability to and indeed does, represent very complex and useful structures which have properties which solves my problem efficiently. It's also plausible that, it may have evolved fairly complex structures that may have been useful in other domains, but which I never saw since they weren't useful for my problem.

It is possible that parts of my fitness function may sometimes have biased the results in unfortunate ways. Specifically, I have a feeling that the use of the number of LSCs in various parts of the fitness function to i.e. see how many of them the light sources should hit, biases it towards not drawing ones that aren't hit at all. This in turn biases the structure to having exactly the same amount of LSCs as the the number of LS. This prohibits it from finding more generalized solutions which scale better with more or less light sources.

When it comes to the good and general solutions, they all seem to be very simple when it comes to structure. Take for example figure 5.6, it's basically a circle of OPC and with a line of LSCs inside it. This doesn't need a very

large CPPN to represent (the CPPN making this has 8 hidden nodes and about 100 links), still I've seen examples on networks with over 100 nodes and over 1000 links. This tells me that the CPPN should easily be able to create much more complex structures. Perhaps if given a more complex world model, i.e. where the LSCs are given an angle, the CPPN would be able to represent a structure that accounts for this i.e. by angling the LSCs towards the slit.

Also, given my light sources are fixed for each run, it is easy for the algorithm to prefer a solution specific the the number of sources as mentioned in section 6.2. More general solutions might have been found on those with a lower number of light sources if the light sources had been randomly distributed along the line.

Another issue I see is that since the light sources are placed along a straight line, it would prefer to place the LSCs along a straight line also, and effectively make a structure that mirrors the LSs around a central line with a thin slit in the center. This is seen most of the solutions to the larger worlds, but not as much in the smaller ones, I would think that is because of the angle plays a smaller part when the distances are shorter and a slightly wrong angle won't get that much of an error when it hits the LCSs.

The scaling example is very interesting, the structure is the same, except from the slit which is now transformed from 1 central slit into two slits close to the center. I am however pretty sure that this is because the CPPN actually do represent two slits, but in the original structure, the size of the objects spanned the area of both slits. By that I mean that when it comes to the center it gets as output "no object" because the the at that resolution both slits *and* the OPCs in between them exist at the same point at the lower resolution. This means that the structure actually contains data that is not only visible at that resolution, but actually isn't the optimal for this structure, but still it works out well.

I find it interesting that my algorithm seems to always end up using a limited set of tactics when it creates it's structures as mentioned it section 6.7. It shows that given the limitation and biases of my algorithm and simulation at least, there is just a few viable structures to go for. That all of them uses

a slit solution at least shows the potency of that tactic.

Another interesting thing about the algorithm is that in most cases, it actually finds a solution that is very good within 1000 generations. That is, a solution that get a fitness function over 0.5 and usually closer to 0.7. This at tells me that the domain and the tools are a good match. It also hints at Neatzsches problem-solving ability.

7.2 Evaluation of Working Hypothesis

Hypothesis I:*A CPPN evolved to create a shape in a 2D simulated environment, given a selection pressure that rewards high-information retrieval, will eventually evolve a shape similar to that of a cross-section of an eye.*

I think my hypothesis held up pretty well, given the assumptions taken by my environment and my fitness function. Many of my structures had shapes similar to that of an eye, in that they consisted of a small round structure of LSCs encased by OPCs. All of my high fitness structures all used some kind of slit to channel the light through. Some solutions used more than one though, but the underlying principle was the same. This might seem obvious, but it's interesting that the algorithm chooses this solution with no prior knowledge.

I would conclude that the selection pressure exerted by my fitness function does indeed hold some of the properties needed to generate an eye-like structure. I would also think that improving on it and making it even more accurate would further add to the results.

7.3 Future Work

Given the encouraging results of the experiments, I think future work on this domain is warranted. There are several parts of the experiments that could be improved to give both better results and add to the biological plausibility of the method used. Specifically the environment could be simulated more

realistically, either by adding features to the current simulation, or by adapting a more advanced environment to the problem. A good example could be to adapt an advanced ray-tracer as the environment.

Also, instead of a row of light sources as input, one could give as input a picture or a pattern, so that each pixel would represent a light source with a specific value or color. Then you would have the output nodes connected to a function which compares the output nodes to the input nodes and is tolerant to error. You would run the output nodes via a neural network which you would also have to evolve to get the correct mapping between the output nodes and input nodes. This would be a much larger search space, because this would effectively be co-evolution of both the structure and the mapping. It would however be analogous with how we think the eye evolved and it would be an indirect method to deciding how much information it is able to extract from a structure, as apposed to mine, which directly tries to decide how much information there is.

Another idea for future work would be to expand the environment to 3 dimensions. Given the current implementation this is pretty straight forward. The environment would need to contain another coordinate, and the pruning function would need to take this dimension into account when pruning blocked light vectors. The CPPN is even more trivial, it would just need a third dimension as input, for it to draw a 3d model. However the biggest problem, and the reason the problem wasn't extended to this 3d environment is the need to visualize this 3d environment, both for debugging and display purposes. It would also have added on the computational tractability of the problem, at least for my time constraints and available hardware.

There could also be possible that my interpretation of the network could be done differently. An idea would be to have just one node that is the draw or don't draw, and then have 2 other nodes that says is it an LSC or an OPC. It's hard to know in advance which is easiest for the network to learn. Other methods could also be used.

Of course, as with any implementation of genetic algorithms, it's almost always possible to improve the results by tuning the various genetic algorithms. Especially in NEAT and specifically the Neatzsche implementation,

there is a lot of tunable operators that might or might not have a measurable impact on the final result (see A).

When it comes to the point made in chapter 6 about the minimum fitness being effectively 0 all the time, it might have been an idea to have the fitness find some more useful traits to reward, because I have a feeling that even though there were a lot of bad ones, I think that they got too close a fitness. They could get between 0.0000001 and 0.000001 and both would be bad. However, compared to a good one at 0.6 or so, they have basically no chance of surviving. This is might be unfortunate though, given that the second worst one might be very much better than the absolute worst one.

Bibliography

1994. *A Pessimistic Estimate of the Time Required for an Eye to Evolve*, volume 256, 1994. The Royal Society. URL `http://www.jstor.org/stable/49593`.

P.J. Angeline, G.M. Saunders, and J.B. Pollack. An evolutionary algorithm that constructs recurrent neural networks. *Neural Networks, IEEE Transactions on*, 5(1):54–65, Jan 1994. ISSN 1045-9227. doi: 10.1109/72.265960.

David B. D'Ambrosio and Kenneth O. Stanley. A novel generative encoding for exploiting neural network sensor and output geometry. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2007). New York, NY: ACM*, 2007.

et al. David W. Opitz. Connectionist theory refinement: Genetically searching the space of network topologies. *Journal of Artificial Intelligence Research*, 6:177–209, 1997.

Larry Pyeatt Frederic Gruau, Darrell Whitley. A comparison between cellular encoding and direct encoding for genetic neural networks. *Genetic Programming 1996: Proceedings of the First Annual Conference*, 1996.

D. E. Goldberg. *Genetic algorithms in search, optimization and machine learning.* Reading: Addison-Wesley, 1989, 1989.

J. H. Holland. *Adaptation in natural and artificial systems. an introductory analysis with applications to biology, control and artificial intelligence.* Ann Arbor: University of Michigan Press, 1975, 1975.

Kenneth Alan De Jong. *An analysis of the behavior of a class of genetic adaptive systems.* PhD thesis, University of Michigan, 1975.

L. Lichtensteiger and P. Eggenberger. Evolving the morphology of a compound eye on a robot, 1999. URL `citeseer.ist.psu.edu/lichtensteiger99evolving.html`.

Bjørn Magnus Mathisen. Evolving a roving-eye for go revisited. Master's thesis, NTNU, 2007.

Melanie Mitchell. *An introduction to genetic algorithms.* MIT Press, Cambridge, MA, USA, 1996. ISBN 0-262-13316-4.

J. C. F. Pujol and R. Poli. Evolution of the topology and the weights of neural networks using genetic programming with a dual representation. Technical report, School of Computer Science, The University of Birmingham, Birmingham B15 2TT, UK., 1997.

Jimmy Secretan and Kenneth O. Stanley. Picbreeder: Evolving pictures collaboratively online. In *Proceedings of the Computer Human Interaction Conference (CHI 2008). New York*, 2008.

Kenneth Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. Technical Report AI2001-290, UTCS, 2001. URL `http://www.cs.utexas.edu/users/nn/pages/publications/abstracts.html#stanley.utcstr01.ps.gz`.

Kenneth O. Stanley. *Efficient Evolution of Neural Networks Through Complexification.* PhD thesis, The University of Texas at Austin, 2004. URL `http://nn.cs.utexas.edu/\downloads/papers/stanley.phd04.pdf`.

Kenneth O. Stanley. Compositional pattern producing networks: A novel abstraction of development. *Genetic Programming and Evolvable Machines Special Issue on Developmental Systems*, 2007. URL `http://eplex.cs.ucf.edu/papers/stanley_gpem07.pdf`.

X. Yao and Y. Liu. Towards designing artificial neural networks by evolution. *Applied Mathematics and Computation*, 91(1):83–90, 1996.

B.-T. Zhang and H. Muhlenbein. Evolving optimal neural networks using genetic algorithms with occam's razor. *Complex Systems,*, 7:199–220, 1993.

Appendix A

Appendix

A.1 Parameters

The actual parameter word used in the settings file is type faced in italic. Each of the settings uses a real number as its value, unless its a boolean setting, in which case it is marked as such and the values 0 and 1 apply.

- **Disjoint genes coefficient**
  disjoint_coeff
  This parameter lets you decide how much disjoint genes is considered when calculating the genetic distance between two genomes.

- **Excessive genes coefficient**
  excess_coeff
  This parameter lets you decide how much excessive genes is considered when calculating the genetic distance between two genomes.

- **Mutational difference coefficient**
  mutdiff_coeff
  This parameter lets you decide how much the mutational difference, that is the difference in weights, (for each of the aligned genes) is considered when calculating the genetic distance between two genomes.

- **Weight mutation rate**
  weightmutation_rate

The rate of genes(weights) in the genome which should be mutated, given that the genome already is selected for weight mutation.

- **Severe mutation rate**
  severe_mutation_prob
  A rate for each genome, which greatly increases weight mutation rate, this is fixed to 0.5 in the original version of NEAT, which means that half of the time a genome is mutated, it is a severe mutation.

- **Compatibility threshold**
  compat_threshold
  This threshold lets you decide how genetically close two genomes need to be given the formulae for compatibility. [1]

- **Drop off age**
  dropoff_age
  When the number of generations in which the species has not improved increases beyond this number, the overlap is used as a "agedebt", to adjust the fitness of the species down.

- **Age significance**
  age_significance
  This is a coefficient to the fitness of each phenotype in a species who's age is above 10.

- **Survival threshold**
  survival_thresh
  This is the percentage of phenotypes to survive within a species, if set to 0.2, the top 20% phenotypes will survive and possibly be selected to have offspring.

- **Species target**
  species_target

[1] This parameter should be considered as a starting point if you also specify species target, in which case the evolutionary algorithm gradually shifts this value in either direction to try to achieve the right number of species.

This tells the evolutionary algorithm to adjust the compatibility threshold so that the number of species will converge to this target number. As this algorithm just increases and decreases the threshold by 0.3, the number of species will oscillate around the target, but the average number of species with regards to generations will be quite close to this target number.

- **Species target size**
  species_target_size
  This does the same as species target, in a more consistent way to the experimenter, as the size of the species is what affect the other processes dependent on the species. This parameter is only used if the above parameter "Species target" is not set.

- **Mutate only probability**
  mutate_only_prob
  This is the probability that the offspring is a straight clone rather than a result of a crossover.

- **Mutate weights probability**
  mutate_link_weights_prob
  This is the probability that the weights of the offspring will be mutated.

- **Mutate toggle enable probability**
  mutate_toggle_enable_prob
  This is the probability that one mutates the genome via picking a gene and toggling whether its enabled or not.

- **Mutate gene re-enable probability**
  mutate_gene_reenable_prob
  This is the probability that one of the disabled genes of the offspring will be re-enabled.

- **Mutate add node probability**
  mutate_add_node_prob

The probability that the mutation picks a link, splits it and ads a new node in between.[2]

- **Mutate add link probability**
  mutate_add_link_prob
  The probability that the mutation adds a link between two nodes that doesn't have a link[3].

- **Interspecies mating rate**
  interspecies_mate_rate
  The rate of which a crossover operation is between two phenotypes from different species.

- **Mate multipoint probability**
  mate_multipoint_prob
  The probability that a crossover should be a multipoint crossover, deciding for each gene which parent it should be from.

- **Mate multipoint average probability**
  mate_multipoint_avg_prob
  The probability that a crossover should be like a mulitpoint crossover only that the gene's(or link weights) are averaged between the two parents(unless the gene in question is disjoint or excess).

- **Mate singlepoint probability**
  mate_singlepoint_prob
  The probability that the crossover chooses one point in the genes to use for crossover point

- **Mate only probability**
  mate_only_prob
  The probability that the resulting phenotype from a crossover is not mutated.

[2]See keeplink option, as this alters the effect of this parameter.
[3]See newlink tries as this alters the behaviour of this algorithm.

- **Newlink tries**
  newlink_tries
  This number determines how many times the algorithm for adding a link should try to find two nodes that doesn't have a link between them, this is necessary as there could be the case that this isn't any two nodes without a link between them.

- **Spawn recurrence probability**
  spawn_recur_prob
  The probability that during the random spawning of the initial generation, a given recurrent link will be created.

- **Cold gaussian weight mutate**(boolean)
  coldgaussian_weight_mutate
  This parameter turns on or off cold gaussian weight mutation[4].

- **Recurrent connection probability**
  recurrent_conn_prob
  This is the probability that the link will be added if the link in question a recurrent link.

- **Weight mutation power**
  mutate_link_weights_power
  This sets the maximum weight mutation.

- **Number of innovations**
  number_of_innovations
  This number sets how many innovations the NEAT algorithm should remember.

- **Keep link while adding node**(boolean)
  keeplink
  If set, this parameter will change the algorithm for adding nodes so

[4]Cold gaussian implies that the mutation doesn't use the original weight as a starting point, see the code for more detail

that it keeps the original link enabled after creating the two new links and adding the node in between.

- **Number of Lightsources**
  number_of_lightsources
  This number gives the number of light sources to draw for this run.

- **Size of X axis**
  xmax
  The size of my x axis.

- **Size of Y axis**
  ymax
  The size of my y axis.

- **Size of cells**
  cellsize
  The size of the cells. This size changes how much space a cell is able to block. A number between 0 and 1 makes sense, it can be larger, but that gives overlapping cells.

- **Distance from Light sources**
  ls_distance
  The distance between the light sources column and the space where my model is allowed to draw.

A.1.1 Fixed Parameters

Some of the parameters were fixed across experiments as they wasn't affected by the difference in task domain.

A.1.2 Experiment Parameters

This subsection will give an overview over which parameters were fixed for all my experiments.

Setting	Value
mutational diff coefficient	0.4
compatibility threshold	3.0
drop off age	20
species target size	60
Mutate link probability	0.85
Add node probability	0.03
Add link probability	0.04
Inter-species mating probability	0.001
Recurrence	Enabled
Weight mutation power	0.4

A.1.3 Experiment Syntax

To run the application run the followin command:
./run-localeye.sh ”spawn 5 2 0 0.5 500” 15 1000 10 settings/settings-eye default 0

The important tunables are as follows:
./run-localeye.sh ”spawn 5 2 <number og hidden nodes to start with> 0.5 <population size>” <number of nodes> <number of generations> <number of runs> settings/settings-eye default 0

To view a genome use the following syntax:
./lightsim-viewer 0 results/20086113000-4324 500 1300 800 20 0.75 0

Where the important tunables are:
./lightsim-viewer 0 <path to results> 500 <width> <height> <scale> 0.75 <show pruned>

A.2 Source code

The source code containing the implementation is released under the GPLv2 (see `http://www.fsf.org/licensing/licenses/info/GPLv2.html`) license from the Free software foundation. This source-code can be found at the following location: `http://neatzsche.generation.no`. The changes and experimental data pertaining to my thesis in not yet in a release tagged version of Neatzsche, and is found in the SVN (Subversion) repository.