

Abstract

In this thesis the coding style of creatures from the artificial life platform Avida has been studied. This has been done using several different techniques ranging from qualitative approaches like graphic genome representation and by representing the genes as logical circuit diagrams, to measuring stylistic properties of the code in a quantitative fashion. This analysis has shown that the genome of Avida creatures are characterised by strong correlations between different genes, substantial redundancy in the genome and by a reuse of code. The analysis has also shown that the coding style depends on the environment in which the code evolved.

The evolutionary coding style was also compared to the coding style of human written programs, and the results show that the styles differ in several aspects. The difference in coding style is conjectured to come from the different selection pressures the codes are subject to. Finally a simple hill-climb algorithm is shown to produce a coding style that is similar to that of Avida creatures.

Acknowledgments

I would like to thank my supervisor Torbjörn Lundh for excellent guidance and help throughout my work, Nils Baas for answering questions regarding hierarchies and emergence in Avida and finally my wife Emelie for her support.

Contents

1	INTRODUCTION	1
2	ARTIFICIAL LIFE.....	2
2.1	GENERAL BACKGROUND	2
2.2	HISTORICAL BACKGROUND.....	3
3	AVIDA.....	6
3.1	INTRODUCTION	6
3.2	MAIN STRUCTURE	6
3.3	THE VIRTUAL COMPUTER.....	6
3.4	TIME SLICING	8
3.5	THE FITNESS LANDSCAPE	8
3.6	REPRODUCTION	9
3.7	MUTATIONS	9
4	GRAPHIC GENOME REPRESENTATION	11
4.1	INTRODUCTION	11
4.2	METHOD	11
4.3	DISCUSSION	13
5	GENE DEVELOPMENT.....	14
5.1	INTRODUCTION	14
5.2	GRAPHIC REPRESENTATION	14
5.3	EVOLUTION OF A GENE	15
6	STYLISTIC MEASURES ON AVIDA CREATURES	19
6.1	INTRODUCTION	19
6.2	CODES	19
6.3	STYLES OF A CODE.....	20
6.4	MEASURES.....	20
6.5	COMPARISON OF DIFFERENT STYLES.....	24
6.6	DISCUSSION	27
6.7	CONCLUSION	29
7	HIERARCHIES IN AVIDA	30
7.1	INTRODUCTION	30
7.2	STRUCTURAL LEVELS	30
7.3	EMERGENCE	32
7.4	EMERGENT PROPERTIES OF STYLISTIC MEASURES.....	33
8	GENE CORRELATIONS	34
8.1	INTRODUCTION	34
8.2	MUTATIONAL EFFECTS	34
8.3	COMPRESSION.....	36
8.4	DEVELOPMENT	37
8.5	CONCLUSION	37
9	COMPARISON TO HUMAN CODE.....	39
9.1	INTRODUCTION	39
9.2	THE HUMAN CODING STYLE	39
9.3	THE EVOLUTIONARY CODING STYLE.....	39
9.4	A TOP TO BOTTOM PERSPECTIVE	40
9.5	HUMAN FITNESS	40
9.6	LOCAL VS. GLOBAL PROPERTIES	40
9.7	A CASE STUDY	41
9.8	CONCLUSION	45

10	THE BLINDFOLDED PROGRAMMER	46
10.1	INTRODUCTION	46
10.2	AVIDA VS. THE BLINDFOLDED PROGRAMMER	47
10.3	IMPLEMENTATION OF THE BLINDFOLDED PROGRAMMER	47
10.4	THE CODING STYLE OF THE BLINDFOLDED PROGRAMMER	49
10.5	DISCUSSION	51
10.6	CONCLUSION	51
11	CONCLUSION	52
	BIBLIOGRAPHY	53
	APPENDIX A	54
	APPENDIX B	57
	APPENDIX C	60

1 Introduction

Since the discovery of DNA, biologists have been able to sequence the entire genomic sequence of several species, the greatest success being the sequencing of the human genome. Although the entire DNA sequence is known we still know very little about the functions coded in the DNA? The view that the DNA can be compared to the source code of a computer program may give some insight to the problem. Consider the following analogy [1]:

"DNA is machine code. Genes are assembler, proteins are higher-level languages like C, cells are like processes ... the analogy breaks down at the margins but offers useful insights."

It has been shown, using block entropy [2], that the DNA is closer to random code than human written computer code. This suggests that the DNA is coded in a different fashion compared to human computer code. One way to interpret this is to say that the DNA is coded in a different coding style compared to human written code.

The scope of this thesis is to investigate how evolved programs are coded, i.e. to search for an evolutionary coding style. This will be done using Avida [3], an Artificial Life platform developed by the Digital Life Lab at Caltech, in which digital creatures evolve, due to mutations and limited space. Note that Avida is not a simulation of carbon-based life, but real evolution in a virtual environment.

Using the notion of a stylistic measure, developed in [4], we will compare creatures that have evolved in different environments in Avida, in order to investigate how the environment influences the coding style. We will also compare evolved programs to their human written counterparts, and show that their styles differ. Further we will also look in detail at the genetic structure of evolved code and how the genes develop over time and finally the coding style of programs that are a product of a simple "hill-climb"-algorithm in the Avida fitness landscape will be investigated.

2 Artificial Life

2.1 General Background

The complexity and diversity of life on earth has probably astonished mankind since the early stages of human civilisation. The curiosity of man has resulted in a vast knowledge about everything that surrounds us. Through hard research we now understand the laws of physics and chemistry that explain the properties of the inanimate world, but still we don't have a fundamental understanding of life or the living state. What is the essence of life? What separates the living from the non-living? Is it possible to formulate a theory of the living state?

The understanding of the essence of life has been obstructed by the fundamental difference between the living and non-living. The properties of a non-living object can often be understood by disassembling the object and examining the parts instead of the whole object. In this way we can understand that a metal conducts electric current because the atoms that constitute the metal are arranged in such a manner that they share electrons, and that it's these electrons that transmit the current. For the living state such an investigation is impossible as in most cases the parts that constitute the living object don't have the property of being alive themselves. Instead it seems like the living state is the property of a collection of components. This makes the usually successful reductionist approach of science impossible, instead we must study life as an emergent property.

Now, as we cannot study an elementary living object, the best we can do is to study the simplest living object. If such an object can be found then maybe from the study of that system we can draw conclusion about the principles and essence of the living state and even state a theory of the life. One such minimal living object could be the RNA molecule, which has been suggested to be the ancestor to the prokaryotes in the early stages of evolution. But before we state a theory of the living state we must have a definition of what is living and what is not. This has been proven to be a quite difficult task. Although it's easy for us to separate the living from the non-living it's hard to state a satisfactory definition that separates the two. Many definitions have been used and they focus on either the interaction with the environment or the possibility to store and transmit information. In [5] Adami gives yet another definition:

“Life is a property of an ensemble of units that share information coded in a physical substrate and which, in the presence of noise, manages to keep its entropy significantly lower than the maximal entropy of the ensemble, on time scales exceeding the “natural” time scale of decay of the (information-bearing) substrate by many orders of magnitude.”

Note that this definition rather relies on the preservation of information than on interaction with the environment and transmission of the information. The information may be preserved by copying it, like in terrestrial life, but it's not a requirement for a living systems. The only requirement is that the information should decay slower than normally.

The frustration around these questions has together with the advances in technology spawned a new field of research, namely Artificial Life. This science is centred around the emulation, simulation and construction of living systems. The field of Artificial Life actually traces back to the 1930's, but the discipline was reborn at the Artificial Life conference in Los Alamos in 1988 [6]. The advances in technology made it possible to perform simulations of living systems that had the size and speed to make them valuable tools in the search for the understanding of life.

2.2 Historical background

2.2.1 The automaton approach

The first attempts to create Artificial Life traces back to the works of Turing and von Neumann. In 1936 Turing invented the famous Turing machine, which is an abstract automaton that could compute any number. We will not go into detail how a Turing machine is constructed, but it can be said to be the simplest construction of a computer. Turing even showed that any other Turing machine (or computer) can be simulated on a special Turing machine called a universal machine.

The idea of the universal Turing machine, that can simulate any other computer, inspired von Neumann to construct an automaton that could construct any other automaton. The constructing automaton A was thought to float around in an infinite supply of parts for the automaton to be constructed. When given a description I the constructing automaton would create an automaton according to that the description. Now if A was given a description of itself it would self-replicate. Von Neumann then tried to implement his self-replicating automaton on a two-dimensional Cellular Automata, but the construction was extremely complicated and was never finished.

The work in this direction stood still until Chris Langton started investigating the ideas of von Neumann in the 80's. In a paper from 1986 [7] Langton investigated the idea of artificial biomolecules interacting through an artificial chemistry using a two-dimensional Cellular Automaton. He randomly searched the rule space for an 8 state Cellular Automaton, and found that there exists a special class of rules that give rise to complex structures and cyclic behaviour on the grid. He even found structures that were capable of self-replication, precisely von Neumann's initial idea.

2.2.2 Coreworld

The appearance of computer viruses, small programs that infect ordinary programs and spread through self-replication, inspired researchers to take the field of Artificial Life into a new direction. The idea of using self-replicating code as a simulation of life was born.

At the same time a computer game called "Core wars", in which players write their own programs that compete for space in a virtual processor, appeared. Taking inspiration from both computer viruses and "Core Wars" Sten Rasmussen and his co-workers [7] constructed the first virtual environment in which self-replicating code competed for space. The platform

called Coreworld was a virtual computer core that was arranged in a circular fashion. The language used in the core was an assembler-like language (inspired from “Core Wars”) that held 10 instructions, each taking two arguments. The arguments could either be interpreted directly or as a pointer to another instruction or as a pointer to another pointer. The core was also subject to noise, so that when the `MOV` command, which copies instructions, was executed there was possibility the copy was flawed.

When self-replicating code was inserted into the core, it replicated for a while but died soon afterwards, because of the noisy environment. Instead the entire core evolved into a stable state containing cooperating program fragments.

2.2.3 Tierra

Inspired by the Coreworld platform the biologist Tom Ray began working on his own version of Coreworld, which he named Tierra [14]. The system architecture was similar to Coreworlds, using a circular memory and an assembler-like language. But Ray understood that it was the argument based language of Coreworld that made the programs so brittle. Instead he used a pattern-based addressing scheme without operands. In his construction every program received its own CPU, which means that the programs had separated computation, which wasn't the case in Coreworld. The programming language in Tierra held 32 instructions, each instruction coded with 5 bits, thus making each codon 5 bits long. These instructions acted on a virtual CPU that had one instruction pointer, four registers (AX, BX, CX, DX), one circular stack of length 10 and a number of flags. The instruction set also included so called no-ops (`nop0` and `nop1`), which did nothing when they were executed, but could be used for pattern matching. A pattern matching algorithm, using the commands `adrf` or `adrb`, could be used to find the complement of a label and returning the address. The memory was also subject to noise, which was implemented like point and copy mutations, which changed one instruction to another.

The virtual CPU also held a slicer queue and a reaper queue. The slicer queue was used to keep track of which instructions to execute and the reaper queue to keep track of which instructions to overwrite. When a program is born it is placed on the top of the slicer queue, and executes a fixed number of instructions and is then placed at the bottom of the queue. At the same time it is placed in the bottom of the reaper queue, in which it moves up when it ages.

The birth of a new program is done in the following way in Tierra: First the parent must allocate new memory for the child using the `mal` command, which allocates CX instructions at a location decided by the reaper queue. The size of the allocated space can be calculated using pattern matching. Then the instructions of the parent are copied using the `copy` command, which copies instructions from the parent to the allocated space. Finally a `divide` command removes the writing possibility for the parent and creates a new instruction pointer for the child.

What Ray did was to seed the world with a hand-written ancestor, which was 80 instructions long and had the capability to self-replicate. The program started to fill the memory and the mutations created diversity. When the memory was full the programs had to compete for the space, which resulted in evolution. The faster a program was at replicating the better chances it had to survive. The evolution in the tierran world produced programs that no one ever could have imagined: hosts, parasites, immune hosts, symbionts etc.

This was by far the most successful attempt to simulate the living state so far, the result was so satisfying that it was questionable to talk about it as a simulation. This was evolution and adaptation, the only difference was that it took place in a virtual environment.

3 Avida

3.1 Introduction

The huge success of Tierra inspired Chris Adami and his co-workers to create Avida [3]. The Avida platform is quite similar to Tierra, but has been modified in a way that it more resembles a real biological system.

The circular structure of Tierra was broken down, instead each program resides on a grid point, and their instructions are ordered in a circular fashion within every cell, so that if the instruction pointer runs off the program it simply starts at the beginning of that same program, much like the structure of bacteria and virus genomes. This introduces a locality in the environment, where a program only is in contact with its 8 eight closest neighbours. When a program replicates it allocates space at the end of its own genome, and at a successful divide the child is placed among one of the neighbouring cells. This locality forces information to spread in a diffusive process through the population, very different from Tierra where every program was in contact with everyone else through the reaper queue. There are also changes in the CPU structure and the instruction set.

As all experiments in this thesis will be performed in Avida 1.3 a detailed description of the architecture of the platform will be given.

3.2 Main structure

The creatures in Avida reside on a grid, which by default is 40 x 40 grid points and has a toroidal topology. Each creature has its own virtual CPU which executes its code, and makes it possible for the creature to replicate. As the number of grid points is finite, the creatures have to compete for space. The competition together with the fact that the creatures are subject to mutations leads to evolution where the programs that replicate the fastest will survive. The replication rate of a creature can increase if the creature manages certain computational tasks, which are rewarded by giving extra execution time.

3.3 The Virtual Computer

The virtual computer in Avida consists of the CPU, in which the creatures (programs) are executed and the instruction set, the instructions that the creatures are made up from. The CPU can be viewed as the hardware and the creatures as the software. Note that the virtual computer is constructed in such a manner that it's computationally universal, i.e. any partial recursive function can be calculated with the virtual computer.

will in most cases affect the BX register, but if it's followed by another no-op than `nop-B`, the register corresponding to that no-op will be affected. When used as templates `nop-B` is the complement to `nop-A`, `nop-C` to `nop-B` and `nop-A` to `nop-C`. For a more detail description of the instruction set please consult [3].

3.4 Time slicing

The number of instructions each creature is allowed to perform every update is decided by the time slicing method that is used and significantly influences the global behaviour of the population. The time slicer decides how much execution time a cell will get, disregarding computational rewards, which we will discuss later. The time given to each cell is proportional to its merit, which can be computed in several different ways depending on the setting. Below is a description of the size merit methods that will be used in the thesis:

1. *Merit is proportional to the minimum of executed and copied size.* This method is the default one in Avida, and makes it possible for creatures to increase their size without being punished, as long as they don't skip portions of code, as this would decrease the number of executed instructions
2. *Merit is proportional to copied size.* With this method the creatures can produce introns, but they still have to copy their code in a correct manner.
3. *Merit is independent of size.* This method gives the same time slice to all creatures independent of size, and therefore punishes longer genomes.

3.5 The Fitness landscape

In Avida the creatures have the possibility to increase their merit by taking numbers from the input buffer, manipulating them, and putting them in the output buffer. The functions that are rewarded are set by the user in the file `task_set`. The functions that are rewarded by default are 2- and 3-input boolean functions, like NOR, XOR, etc., but in order for the population to reach the boolean functions, simpler tasks must be rewarded on the way. First the execution of a `get` or a `put` command are rewarded, then the combination `get, get, put`. Further the capability to echo a number from input to output is rewarded. Note that this construction rewards the phenotype rather than a certain genotype, which leads to open-ended evolution. The rewards that the creatures trigger for performing a function is typically proportional to the number of `nand` instructions that are required, and are by default set to be multiplied with the merit. The number of times a function is rewarded can also be changed, and is by default set to three.

The fitness (3.1) of an creature is simply the replication rate of the creature. Theoretically this is given by the merit divided by the gestation time, the number instructions executed per generation.

$$\alpha = \frac{M}{t_g} \tag{3.1}$$

3.6 Reproduction

The reproduction in Avida is typically performed in the following four stages:

- Allocation of new memory at the end of the parent genome
- Copying the parents instructions to the allocated area
- Division of the parent and child genome
- Placement of the child in a neighbouring cell

The placement of the offspring can be done in the following ways in Avida:

- Choose randomly
- Choose eldest
- Choose maximum of age/merit
- Choose empty

The default method in Avida is to choose the eldest.

3.7 Mutations

Avida has both implicit and explicit mutations. The implicit mutations occur when an offspring is incorrectly copied but still viable. Examples of implicit mutations is when a section of code is copied multiple times into the offspring or when the size of the child is doubled. The other types of mutations, called explicit mutations, are controlled by the user by changing the rates or probabilities of them to occur, and changes an instruction into a randomly picked instruction. The types of explicit mutations that are implemented in Avida are:

- *Copy mutation*– may occur when a instruction is copied into the offspring
- *Point mutation* – may hit an creature at any time, and is therefore also named cosmic-ray mutation
- *Divide mutation* – mutates a random instruction in the parent each time a divide occurs
- *Divide insertion* – inserts a random instruction at a random site every time a divide occurs
- *Divide deletion* – deletes a random instruction at a random site every time a divide occurs

The mutation rates affect both the speed and characteristics of the evolution in Avida. If the mutation rates are too high the population won't be able to survive, and if the mutation rates are too low the evolution will proceed very slowly. The different types of mutation also give rise to qualitatively different behaviour. It has been shown [13] that copy mutations and point mutations give rise to different learning rates in the population.

The default settings in Avida are copy mutations at a rate of 0.005 per-creature and divide insertions and deletions set to 0.05 per-creature. This means that on average every 200 instruction will be copied incorrectly and every 20 divide will introduce or remove a random

instruction. These are the settings that will be used in the experiments unless something else is stated.

For further information about the structure of Avida 1.3, please consult [3] or [9].

4 Graphic genome representation

4.1 Introduction

The Avida creatures are programs in an assembler-like language, where each line of the code is an instruction which acts on the registers or the stack. They are relatively short but contain loops and jumps which can make their execution complicated. In order to study their genetical organisation we would like to represent their genomes in a more understandable manner than just printing the executed sequence, which can be quite hard to understand at times, although it can give interesting information about gene construction for example. Below is shown a short part of an execution sequence. The first column is the current instruction, the second is the instruction pointer, the following three are the registers, the next is the stack and the final column shows any output from the program.

Inst.	Line	AX	BX	CX	Stack	Output
nop-A	1	0	0	0	0	
add	2	0	0	0	0	
put	3	0	0	0	0	
get	4	0	0	210461440	0	
sub	5	0	-210461440	210461440	0	
put	6	0	-210461440	0	0	ECHO
nop-C	7	0	-210461440	0	0	
push	8	0	-210461440	0	-210461440	
dec	9	0	-210461441	0	-210461440	
get	10	0	-210461441	360241664	-210461440	
nand	11	0	-210461441	-292591617	-210461440	
nop-C	12	0	-210461441	-292591617	-210461440	
nand	13	0	503053056	-292591617	-210461440	
put	14	0	0	-292591617	-210461440	OR
pop	15	0	-210461440	-292591617	0	
add	16	0	-503053057	-292591617	0	
put	17	0	0	-292591617	0	NOR

4.2 Method

In order to get a better overview of the structure we decided to create a graphical representation of the genome in the following way. Represent every instruction (line of the program) by a node and order them in a circular fashion. Now connect those nodes that are neighbours in the execution sequence. If the program goes from a lower to a higher instruction pointer (IP), make the connection solid and if it's the other way around make the connection dotted. In this way we can distinguish between forward and backward jumps in the code. After the connections have been made the genes are marked. This is done by going from the output instruction backwards until one `get` instruction, for NOT and ECHO, or two `get`

instructions, for two-input functions, have been found. An example of this procedure can be seen in the fig 4.1.

Note that this representation will not give any information about in which order the sequence is executed, but that information can be retrieved from a plot that shows the instruction pointer as a function of the number instructions executed. From fig. 4.2 we can for example see that the creature first performs six backward jumps then goes into the copy loop and finally jumps forward and divides, note also that the child part of the genome is executed before a divide occurs. This is a common technique as it allows for double execution of a beneficial gene.

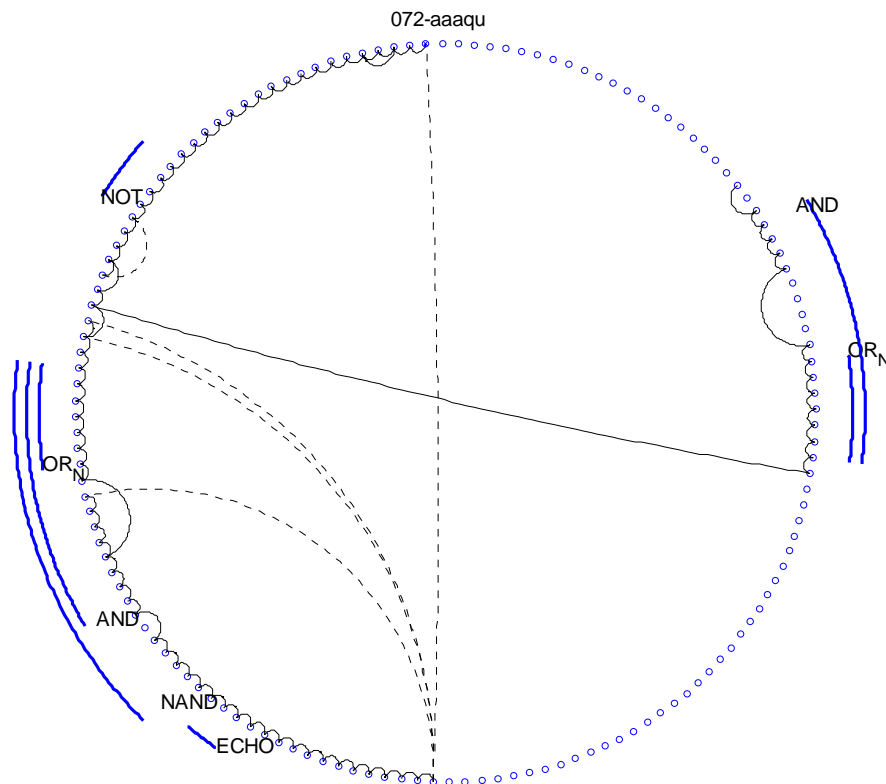


Figure 4.1 Graphic genome representation of the genome of a typical Avida creature. Each instruction is represented by a node, and the nodes are connected if they follow each other in the execution sequence. If the program goes from a lower to a higher IP the connection is solid and dotted if the IP decreases. The genes are represented by the arcs. Here one clearly sees the existence of introns and the tendency for genes to overlap.

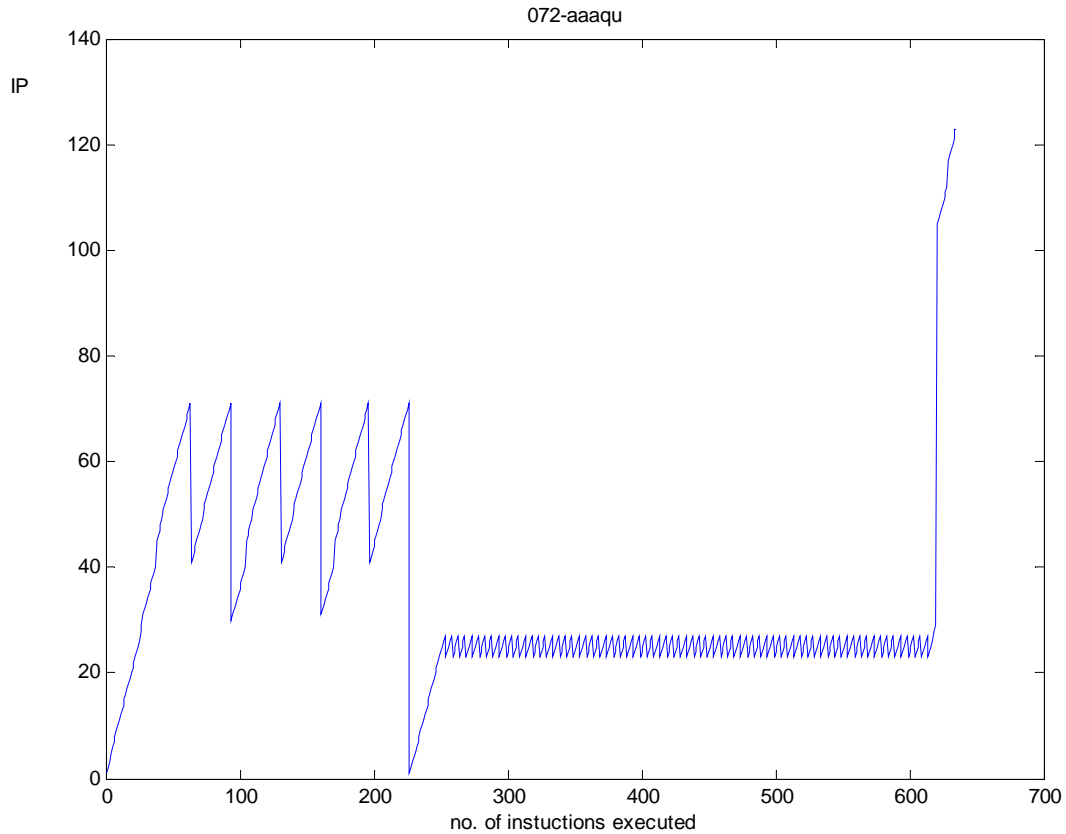


Figure 4.2 Shows the instruction pointer (IP) as a function of the number of executed instructions for the same creature as in fig. 4.1. The copy loop is visible as the oscillating part of the sequence between execution 250 and 600. The reason why the IP is larger than the creature size is because the creature executes the child part of the genome before a successful divide occurs.

4.3 Discussion

The graphic genome representation described here gives a good overview of the genetic structure of Avida creatures. Although it is a completely qualitative tool it can be used for preliminary analysis of genomes. If an interesting feature is detected, using the graphic representation, a quantitative measure can be devised to measure this feature on a large number of creatures.

5 Gene Development

5.1 Introduction

The digital organisms in Avida evolve in an environment that rewards creatures that can perform simple boolean functions like NAND¹ and OR. The instruction set holds a `nand` instruction, which serves as building block for the boolean functions, as the nand-gate is a basic gate, with which all boolean functions can be constructed. It has been observed [10] that the genes for different function overlap, and depend on the same instruction. The scope of this section is to investigate the structure of these overlapping genes and to look at their development over time.

5.2 Graphic representation

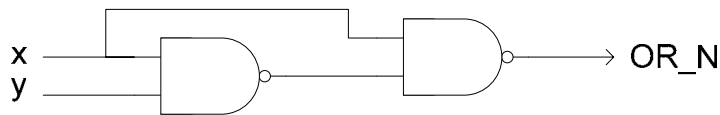
The program that constitutes the genome of a creature is often hard to analyse, not just because it isn't very well structured, but the assembler-like language that is used in Avida is a bit hard to penetrate. To be able to get a better view of the structure of the genes I decided to create some kind of graphic representation of them. This was done by representing them as a logical circuit diagram, which is commonly used for describing circuits in electronics. I will show an example below to show how the conversion from code to diagram is performed.

First extract the part of the genome that performs the function/functions, the gene. This is in most cases done by going from the `put` command backwards in the code until two `get` commands have been found. The code below is an extracted OR_N-gene.

```
get
get
nop-B
nand
nand
put
```

What this code does is the following: first it puts one input in the CX register, and then one in the BX register. It then performs $BX \bar{\wedge} CX$ and puts the result in BX, and then performs the same thing again. Finally it outputs the result. If we label the first input x and the second y , the resulting expression can be written $(x \bar{\wedge} y) \bar{\wedge} x$, which is equivalent to $x \vee \bar{y}$ (x OR_N y). Transferring the operations into a circuit diagram is now easy, and the result can be seen below, where the gates are nand-gates.

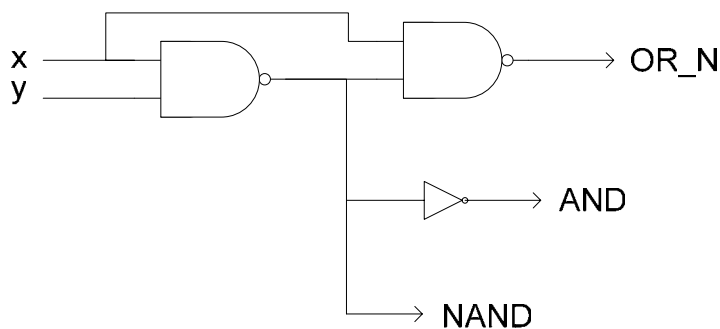
¹ When the capital "NAND" is used we mean the boolean function that is rewarded, when the "nand" is used we mean the instruction in the Avida assembler language and when "nand" is used we talk about the boolean function in general terms or about the logical gate.



This representation is very simple, and using it, it is easy to get an understanding of the structure of the gene. The example above is very simple, and often found in the early stages of evolution in Avida. The circuit below shows a more complex gene, where the order of the outputs is from left to right (NAND is the first output, then AND and finally OR_N). The small triangle represents an inverter. The reason why this symbol is used is because the creatures can invert a number in two ways. One is by using it as both inputs to a nand (as $x \bar{\wedge} x = \bar{x}$), and the other is to perform the following (assuming that a number x is in CX, and $BX = 0$).

sub
dec

The `sub` command subtracts CX from BX, which gives $-x$, and the `dec` command decreases BX by one, thus giving $-1-x$, which is just the inversion of x in the 32-bit representation used. When the first method is used it's drawn with a nand-gate, and if the other is used an inverter symbol is drawn.

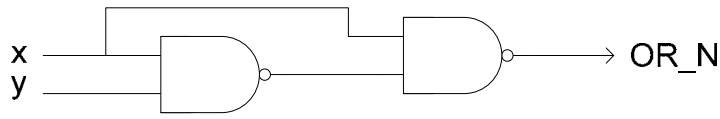


5.3 Evolution of a gene

We will now look at the evolution of a specific gene. The experiment was performed in the following way: One genotype was selected after a run that lasted 20 000 updates, which corresponds to approximately 6000 ancestral generations. The genealogy of the final genotype was created, and all genotypes were inspected, but only those where a change in the gene was found were kept. The specific gene of the genotypes was then converted into circuit diagrams. The square gate seen below, with a '+' in, is an add-gate, the output from it is simply the sum of the inputs, and corresponds to the `add` command in the instruction set.

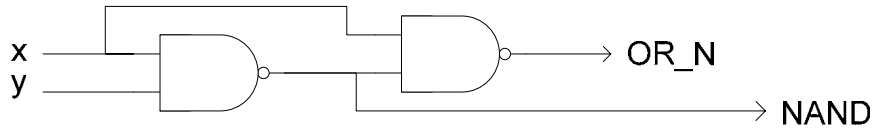
Update Output: 1148

Name: 045-aaacu



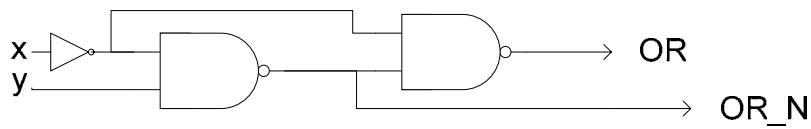
Update Output: 1337

Name: 045-aaady



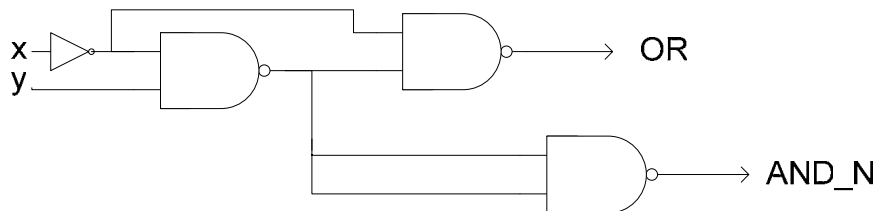
Update Output: 4421

Name: 102-aaaam



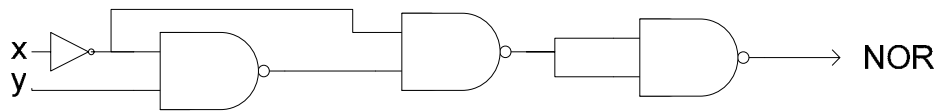
Update Output: 6033

Name: 122-aaaaa



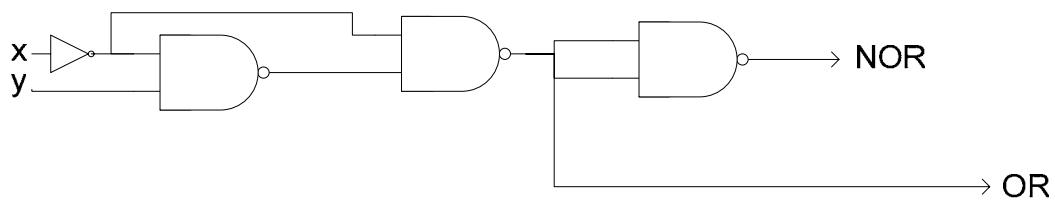
Update Output: 7598

Name: 154-aaaak

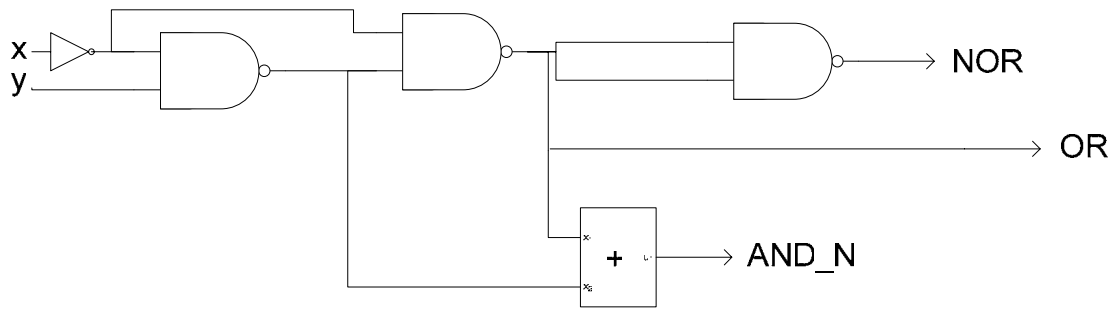


Update Output: 8455

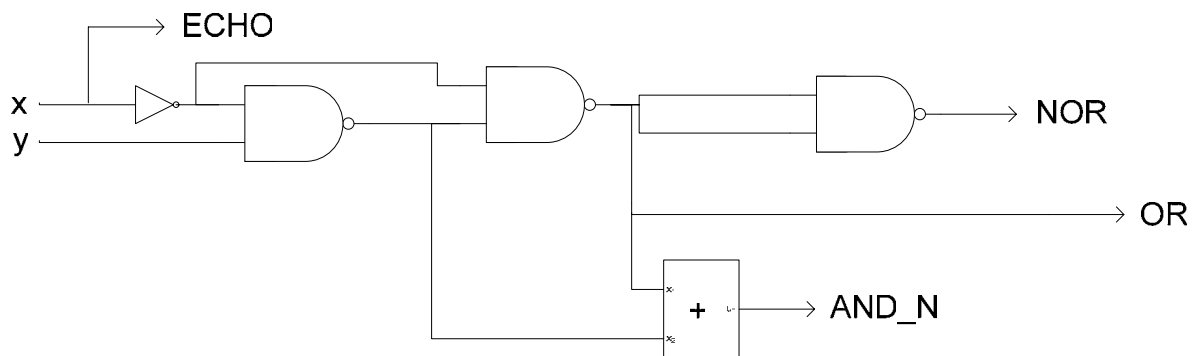
Name: 144-aaaaa



Update Output: 11643
Name: 144-aaadh



Update Output: 16187
Name: 152-aaae



In the above sequence we can observe the development from a gene that creates one output (OR_N) into a complex structure that manages four tasks (ECHO, AND_N, NOR, OR).

From the above we can also conclude that there are three ways in which the gene develops:

1. Adding an extra output node
2. Inserting a new gate (inversion or nand)
3. Rewiring the circuit

The above changes correspond to the following changes in the genome:

1. Inserting a put
2. Inserting a nand or inversion (sub, dec)
3. Adding/changing the label of an existing nand

An example of the first method can be found in the transition between 045-aaacu and 045-aaady, where an extra output node adds NAND to the gene. An example of the insert method can be found between 102-aaaam and 122-aaaaa, where an OR_N is transformed into an AND_N. An example of the final method can be found between 122-aaaaa and 154-aaaak, where an OR and AND_N are transformed into a NOR.

From this perspective we can view the development of a gene as the extension and rewiring of a logical circuit, note that a circuit with several outputs is very brittle as a mutation that hits the beginning of the circuit may destroy all outputs. The emergence of these highly correlated genes will be investigated further in chapter 8.

6 Stylistic measures on Avida creatures

6.1 Introduction

When a population evolves and adapts to an environment, information about what traits or chemical reactions that are beneficial in that environment are written into the genome. The environment of course influences what information is coded into the genome, but the scope of this section is to investigate how the information is coded. Does the environment affect the coding style of the genome? An environment that is very competitive and requires fast replication will intuitively enforce a restrictive coding that is very space effective and which punishes introns.

In this section we will investigate the different coding styles that the environment enforces on the genomes. We do this by examining Avida creatures that have evolved under different size merit methods and mutation rates, but with the same rewarded functions. We will do this using the notion of a stylistic measure that was introduced in [4].

6.2 Codes

The genome of a creature can be thought of as a code that is written in an alphabet consisting of a finite numbers of letters, which is read or interpreted by the CPU. The genomes found in nature are written with the four letters A, T, G and C, which corresponds to the base pairs in the DNA. In Avida the genomes consists of a combination of 24 different CPU instructions, which alter the state of the virtual CPU in some manner.

We will therefore define a **code** to be a finite string of letters taken from a finite alphabet A ,

$$Code = \{\alpha_i\}_{i=1}^k, \quad \text{where } \alpha_i \in A,$$

such that when the code is interpreted, the code will represent a well-defined function or process, $Code_j \rightarrow f_j$.

From the above definition it is clear that different codes can have the same function representation. Let us therefore define a class of codes C_f to be the set of all codes that perform the function f when they are interpreted. All codes that cannot be interpreted correctly are put in the error class C_ε , which can be viewed as the complement to all interpretable codes.

From a genetic point of view one can interpret the function of the code as the phenotype of a creature, and thus the code classes as classes of phenotypically equivalent creatures.

In Avida the interpretation is performed by running a creature through the virtual CPU for one generation. If a successful divide occurs the function that the code defines is simply the tasks that the creature performs during one generation. If a successful divide doesn't occur we put the creature in the error class C_ε .

6.3 Styles of a code

If we look at two codes from the same class we know that they perform the same function, i.e. they have the same phenotype, but their genotype may differ. As the evolutionary process is very sensitive to perturbations we cannot expect to find the same genotype if evolution occurred two times in the same environment. It's therefore interesting to study the style of the code, as this is more likely to be constant between two instances of the same process. We would therefore like to define the style of a code. This is done by introducing a measure (6.1) on the code class C_f , that maps each code to a point in $[0,1]$.

$$\mu_i : C_f \rightarrow [0,1] \tag{6.1}$$

Putting together several of these measures we can create a profile measure $\mu = (\mu_1, \mu_2, \mu_3, \dots)$, which serves as a fingerprint of the code.

6.4 Measures

As we are interested in distinguishing between different coding styles of creatures from Avida we have constructed five different measures that measure different properties of the genome. We will give a description of the different measures, but first a short description of the Functional Genomic Array will be given, as it is used in the calculation of some of the measures.

6.4.1 Functional Genomic Array (FGA)

The circular genome graphs, introduced in section 3, give a good graphic representation of the genome of a creature, but they don't give any information about which instructions are used for replication and about correlations on large distances in the genome. A good way to investigate these properties and the localisation of task genes is to produce a functional genomic array of the genome, a technique which was used in [10]. This is done by replacing each instruction, one at a time, by a neutral instruction (`nop-x`), which does nothing in all cases (it can't even be used as a label). For each instruction replaced we check if the creature can replicate and which tasks it can perform, by running it through a test CPU. This results in a two-dimensional array, with 0 and 1 as entries, which shows on which instructions the particular tasks depend. From this array it is possible to create measures on the genome, which reveal how correlated the genes are, the amount of redundant instructions and the fragility of the creature. Fig. 6.1 shows a FGA for a typical Avida creature. An instruction is coloured white in a task column if the task depends on that instruction and grey if not.

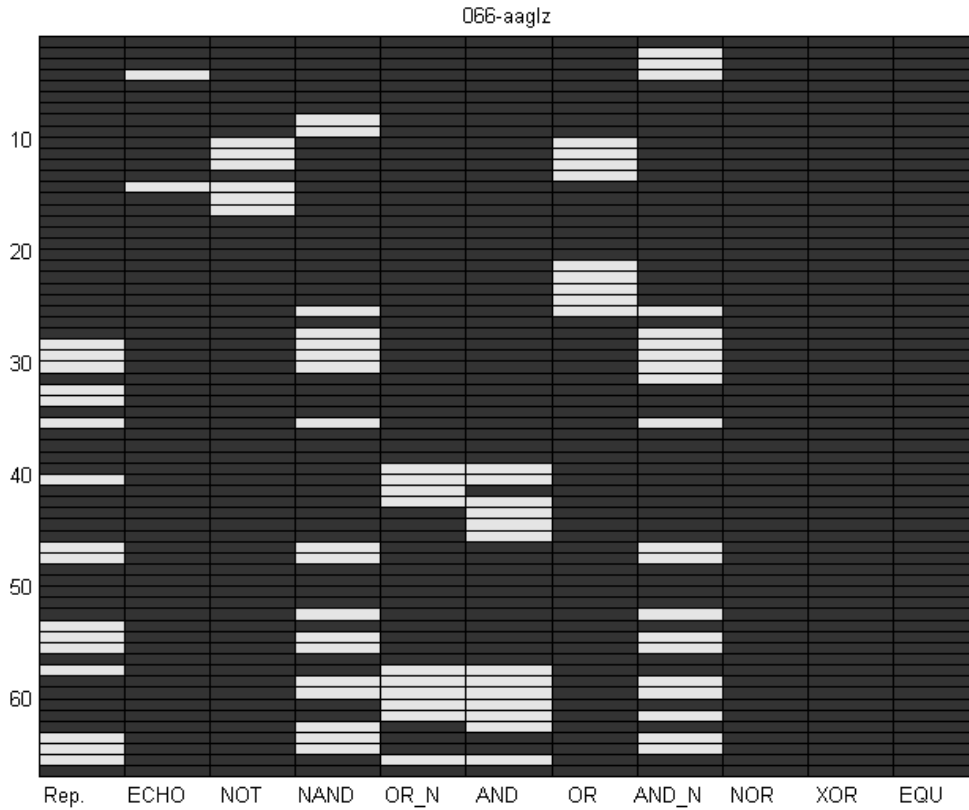


Figure 6.1 The Functional Genomic Array (FGA) of a typical Avida creature. This array shows on which instructions each task that the creature can perform including replication depends. An instruction is coloured white in a task column if the task depends on that instruction and grey if not.

6.4.2 Gene correlation

This measure shows how correlated different genes are and is constructed in the following manner. Sum the functional genomic array along the rows, and remove all zero entries in the resulting vector. Sum up all entries that are larger than one and divide by the number of tasks plus one (for replication) and by the number of non-zero entries in the vector. The reason why the single task instructions aren't counted is because the gene correlation shouldn't be large just because the creature only manages a few tasks. This construction of the gene correlation will give zero if the creature only can replicate, which is sensible as there can be no correlation between only one gene.

If we let \mathbf{T} be the row sum of the FGA, N be the number of tasks the creature manages plus one (for replication), L_E , the number of non-zero entries in \mathbf{T} (the number of essential instructions), then the gene correlation is given by:

$$g_c = \frac{\sum_i \mathbf{T}_i}{NL_E}, \quad \text{where the sum is runs over all indices for which } \mathbf{T}_i > 1$$

For the FGA in fig. 6.1 we have that $\mathbf{T} = (0,1,1,2,0,\dots)$, $N = 8$ and $L_E = 47$, which gives a gene correlation $g_c = 0.1995$.

The gene correlation is a number between 0 and 1, where 0 means that all tasks including replication are coded disjoint in the genome, and 1 means that they all depend on the same instructions. Note that this measure also can be interpreted as the compression of information in the genome.

The gene correlation normally increases in the evolutionary process as the population stores more and more information about the environment. This measure will be investigated in more detail in section 8.

6.4.3 Redundancy

This measure gives the fraction of instructions that don't affect the tasks and replication of a creature. As above the FGA is summed along the rows, but this time we count how many zero entries the vector holds, this number is then normalised by the length of the creature. For the creature in fig. 6.1 we have that the redundancy is $R = 19/66 = 0.2879$.

Under size merit method 4, which is the default method in Avida, the redundancy is rather high (see table 6.1). The reason for this is that this method gives merit proportional to the minimum of executed and copied size, which means that creatures may hold redundant instructions as long as they are executed. The redundant instructions play an important role in Avida as they serve as potential coding areas for new functions. The redundant instructions can be viewed as allocated memory in which new functions may be coded.

Note that the redundancy also can be interpreted as the probability of a copy/cosmic ray mutation being neutral.

6.4.4 Introns

This measure simply gives the fraction of instructions that never are executed when the creature is executed for one life-cycle. Under the standard size merit method 4, the fraction of introns is rather low (see table 6.1), as they are punished. Note that the introns are a subset of the redundant instructions.

6.4.5 Gene reuse

This measure reflects the reuse of code on a global scale, and is constructed in the following manner. Consider the gestation time of a creature, which can be written,

$$t_g = \frac{l_c L}{m} + l_w \quad (6.2)$$

where t_g is the gestation time, l_c is the length of the copy loop, L the length of the genome, m the number of instructions that are copied in every copy loop and l_w is the work length, the number of instructions, not involving copying itself, the creature executes every generation. These are the instructions where the creature performs self analysis and performs tasks.

In most creatures we have $l_c = 4$ and $m = 1$. The work length can be written,

$$l_w = t_g - \frac{l_c L}{m} \quad (6.3)$$

Now consider the quantity

$$\hat{L} = \frac{L - l_i}{l_w} \quad (6.5)$$

where l_i is the number of introns the genome holds. The reason why the number of introns is subtracted from the length is because we want to compare the work length with the number of unique instructions that are executed in the genome each generation. If the execution of a creature is completely straight without any introns we have

$$\hat{L} = L / (L - l_c) \approx 1 \quad (6.6)$$

and if the creature has got many loops, which gives a $l_w \gg L$, we get a \hat{L} close to zero. This implies that \hat{L} is a measure of the global reuse of code. Note that the gene reuse may go above one, as some nop's aren't executed but included in the length of a genome.

6.4.6 Fragility

This measure gives the number of instructions that are essential for replication, and is constructed from the FGA by simply summing the replication column. This measure isn't normalised with the length of the creature because the number of instructions needed to replicate are independent of the length of the creature, as the creatures use a copy loop. But as we want a measure to lie in $[0,1]$ this measure is normalised using the map (6.7).

$$f(x) = \frac{x}{x+c}, \quad c > 0 \quad (6.7)$$

From preliminary data we know that the number of instructions essential for replication in most cases lie between 5 and 50. The c in (6.7) is therefore optimised so that the image of the interval $f([5,50])$ is maximised. The optimisation is straight forward and gives the value $c = \sqrt{(50 \cdot 5)} \approx 15.8$, which gives $f(50) \approx 0.76$ and $f(5) \approx 0.24$.

The creature in fig. 6.1 has 20 instructions essential for reproduction, which gives a normalised fragility of $F = 0.5030$. The fragility is kept relatively low (see table 6.1) in creatures that have evolved in Avida. The reason for this is that it's beneficial to have low fragility, as it increases the chances of a viable offspring in an environment where copy mutations occur. A genotype that has a high fragility will go extinct as it will produce fewer viable offspring. In an environment that doesn't reward computational efforts it is only the fragility that is minimised [11], which results in a minimised genome length.

6.5 Comparison of different styles

6.5.1 Size Merit methods

The different size merit methods in Avida generate qualitatively different styles of coding, for example size merit method 1, which gives merit proportional to copied length, tends to produce introns and doesn't put any pressure on the lengths of the creatures, while size merit method 0 does quite the opposite, because merit is independent of size. The third size merit method, 4, takes the minimum of copied and executed size as merit.

The merit is a very important property as the fitness (6.8) is based on the merit, M , and the gestation time, t_g . This implies that the size merit method has a direct impact on how selection is performed in Avida.

$$\alpha = \frac{M}{t_g} \tag{6.8}$$

The question is if these differences in merit calculation can show in a stylistic analysis of the different methods. To investigate this we created three sets of creatures each containing approximately 60 creatures, from the three size merit methods 0, 1, 4, the full settings for these runs can be found in Appendix A.

The optimal comparison would be to compare creatures from the same code class (phenotype), but as evolution in Avida proceeds with different pace each run one has no guarantee that a certain phenotype has evolved after a fixed number of updates. One way to accomplish this would be to reward only those functions that we want the phenotype to perform and use a large number of fixed updates, but this approach also has its drawbacks. If the required phenotype appears early in evolution, then the code is optimised during the rest of the run as no new functions are rewarded. If a creature from the above run is compared to a creature from a run where the required phenotype appeared just before the maximal number of updates their coding style would certainly differ, as one creature has optimised its coding while the other has not. Instead we decided to compare creatures with approximately the same complexity. This was done by extracting a creature from the dominant genotype after 40 000 updates in each run and if it had reached a certain degree of complexity (it managed at least three boolean functions) it was kept for analysis.

Note that the environment in Avida can be decomposed into one part that describes which functions that are beneficial, the specification, and another part that describes the dynamics of the process, i.e. size merit method, death method and mutation rates, the rule set. In this experiment we only change the size merit method in the different sets of creatures, the rewarded tasks remained constant. Table 6.1 shows the average and standard deviation for each measure under the three size merit methods.

Size merit Method	Gene correlation	Redundancy	Introns	Gene reuse	Fragility
0	0.2655 (0.1221)	0.1666 (0.1102)	0.0249 (0.0496)	0.5594 (0.3615)	0.4860 (0.0402)
1	0.2647 (0.1161)	0.6443 (0.1836)	0.3581 (0.1767)	0.4559 (0.1930)	0.5639 (0.0706)
4	0.2288 (0.0990)	0.5230 (0.1905)	0.0708 (0.0911)	0.4271 (0.1497)	0.5565 (0.0705)

Table 6.1 The average and standard deviation of the stylistic measures for creatures from three different size merit methods.

Another way to analyse the data is to perform a principal component analysis on the data, which reduces the dimensionality to 2, and gives a better graphical representation. The principal component analysis generates a new set of variables which are linear combinations of the old ones. These new variables span an orthogonal basis in the space of the measurements. The first principal component is the axis on which the projections of the measurements have the largest variance. The second principal component is an axis perpendicular to the first on which the projections have the largest variance. This process is repeated until an orthogonal basis for the space has been created. The data is then plotted against the 1st and 2nd principal components. The result of the PCA can be seen in fig. 6.2.

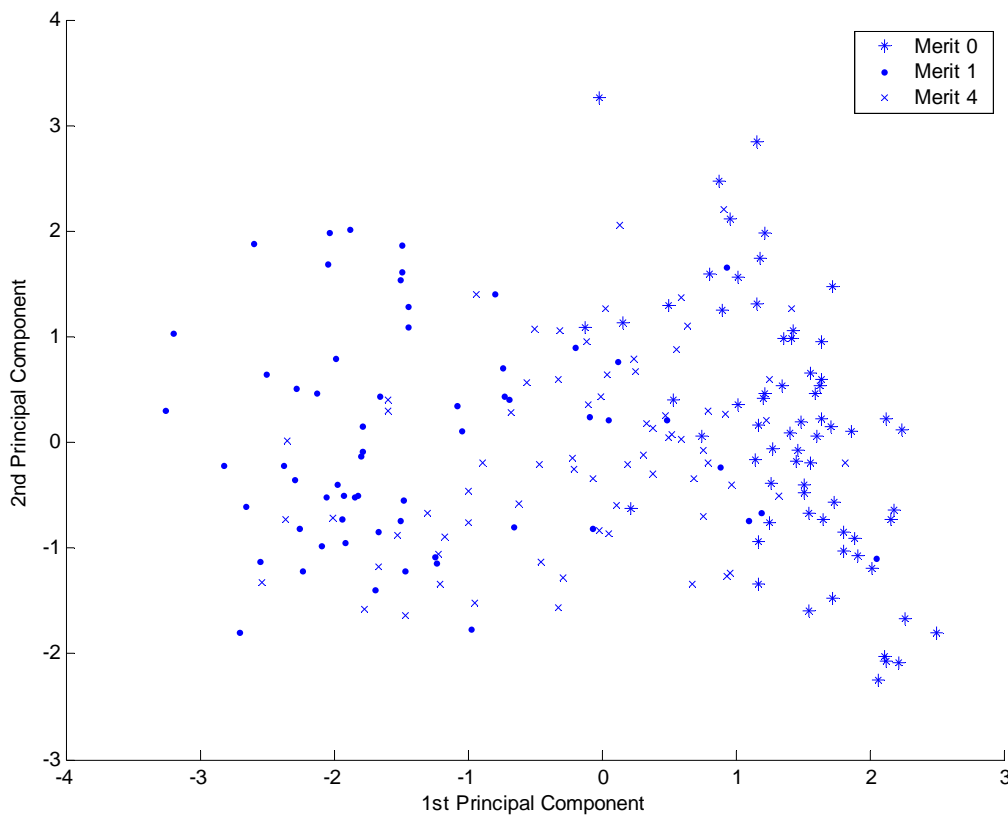


Figure 6.2 Principal component analysis of the profile measure of creatures that have evolved under different size merit methods.

The vectors that span the plane in fig. 6.2, the 1st and 2nd principal components, are given by:

$$P_1 = \begin{pmatrix} -0.0132 \\ -0.6263 \\ -0.5723 \\ 0.1700 \\ -0.5011 \end{pmatrix} \quad P_2 = \begin{pmatrix} 0.7861 \\ -0.1275 \\ 0.0156 \\ -0.5990 \\ -0.0825 \end{pmatrix}$$

6.5.2 Mutation rates

The mutation rates in Avida play an important role in the adaptive process. If the mutation rates are low evolution tends to proceed very slow as the fitness landscape is explored at a low pace and if they are high the population will have trouble sustaining information in the genomes.

To investigate how the mutation rates influence the coding style we created three sets of creatures that had evolved under different copy mutation probabilities each containing approximately 60 creatures. The point mutation rates were set to zero, the insert/delete mutation probabilities were kept constant at 0.05 per divide and the size merit method was set to 4 (the default value in Avida). The copy mutation probability was set to one low value ($P_c = 0.001$), one intermediate value ($P_c = 0.005$) and one high value ($P_c = 0.025$).

As in the experiment above the populations evolved for 40 000 updates after which the dominant genotype was extracted, but it was only kept for analysis if it managed three or more boolean functions. A detailed description of the settings can be found in Appendix A. The three sets of creatures were then analysed using the stylistic measures and the results can be found in table 6.2 and a PCA-plot in fig. 6.3.

P_c	Gene correlation	Redundancy	Introns	Gene reuse	Fragility
0.001	0.2263 (0.1093)	0.6076 (0.2006)	0.0487 (0.0883)	0.6275 (0.3237)	0.5877 (0.0811)
0.005	0.2288 (0.0990)	0.5230 (0.1905)	0.0708 (0.0911)	0.4271 (0.1497)	0.5565 (0.0705)
0.025	0.3037 (0.1372)	0.2847 (0.1558)	0.0372 (0.0985)	0.4964 (0.2690)	0.4872 (0.0465)

Table 6.2 The average and standard deviation of the stylistic measures for three sets of creatures that have evolved under different copy mutation rates.

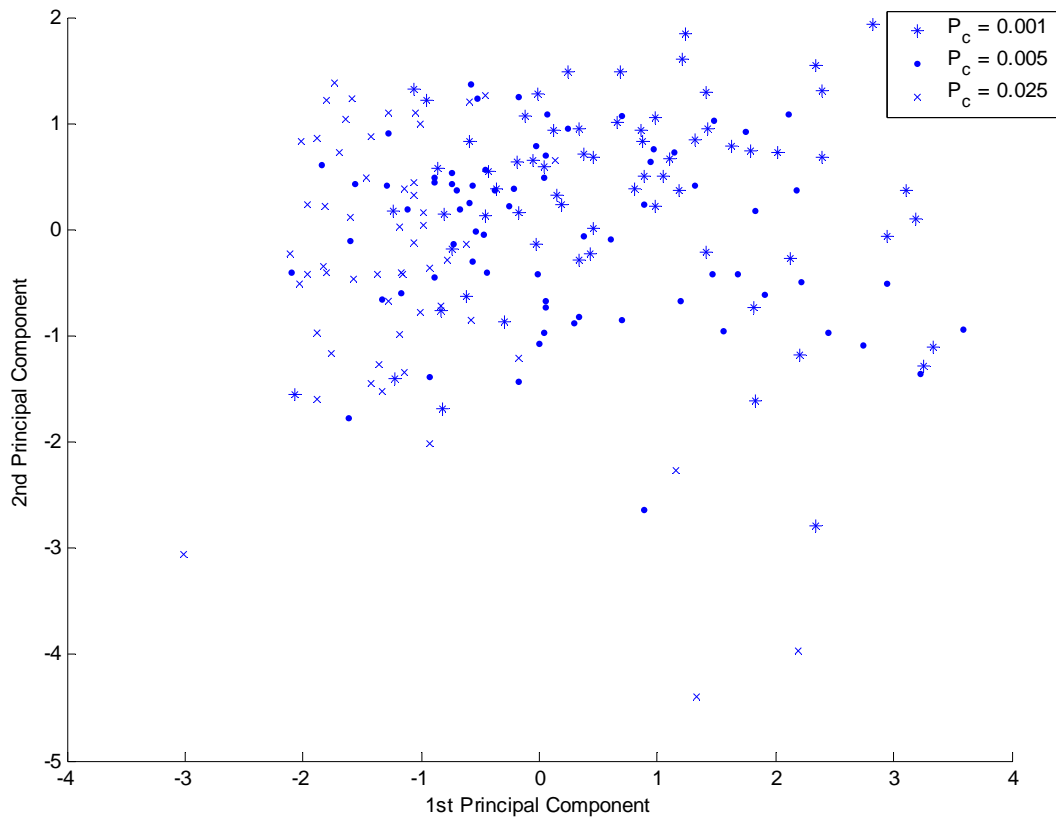


Figure 6.3 Principal component analysis of the profile measure of creatures that have evolved under different copy mutation probabilities.

The 1st and 2nd principal components are:

$$P_1 = \begin{pmatrix} -0.2087 \\ 0.6374 \\ 0.4462 \\ 0.2359 \\ 0.5435 \end{pmatrix} \quad P_2 = \begin{pmatrix} -0.7657 \\ -0.0693 \\ -0.5643 \\ 0.2695 \\ 0.1337 \end{pmatrix}$$

6.6 Discussion

6.6.1 Size Merit methods

In fig 6.2 one can see a separation between the different size merit methods, which clearly indicates that there are differences in their coding style enforced by the size merit method. One way of analysing these differences is to look at the composition of the 1st and 2nd principal component. The components with the larger weights show where the styles differs the most. This shows that the coding styles differ most with respect to the redundancy, intron and fragility measures. What can also be seen in the PCA plot is that the coding style within

each size merit methods varies quite much, a thing that also can be seen in the standard deviations in table 6.1. The reason for this is that the evolutionary process in Avida takes different paths each run due to the randomness in the process, and thus giving rise to a unique coding each run. The large standard deviations makes it impossible to draw any conclusions about how the gene correlation and gene reuse is affected by the size merit method, but the other measures show separation between the different size merit methods.

We will now discuss the coding style of each size merit method in more detail.

Method 0 – This method punishes long genomes as the merit is independent of size, this implies that the only way to gain length in this environment is to acquire task bonuses. This means that the redundancy and the fraction of introns will be very low, as the creatures cannot afford to have instructions that aren't beneficial. This results in a slow evolution compared to the two other size merit methods, as redundant instructions are potential areas for new functions. This also results in a low fragility, which simply is the result of the pressure on efficient coding.

Method 1 – In this method the merit is proportional to the copied size which gives the creatures the possibility to have introns and redundant instructions, which shows in the respective measures. The reason why the fragility is low is because there is no pressure on effective coding of the replication part of the genome.

Method 4 – This method can be said to lie somewhere in between the two above. The merit is calculated as the minimum of copied and executed size, which punishes introns but allows redundant instructions. This shows in the measures of redundancy, introns and fragility, where it lies between size merit method 0 and 1 and in fig. 6.2, where most creatures from this size merit method lie between those from 0 and 1.

6.6.2 Mutation rates

The principal component analysis plot in this case doesn't show the same separation between the coding styles as in the case with size merit method. The coding style of the creatures from the low and intermediate mutation probabilities seem clustered together, but the coding style of the high mutation probability show at least some separation from the other two. The 1st and 2nd principal components show that the largest differences between the coding styles lie in the redundancy, fragility and gene correlation measures. The standard deviations in this experiment are also large and again the standard deviation of the gene reuse measure is so large that no conclusion can be drawn from it.

The averages of the stylistic measures shows the same tendency as the PCA plot, the low and intermediate mutation probabilities give approximately the same values while the high probability differs in the gene correlation, redundancy and fragility measures.

The fragility decreases when the mutation probability increases. The reason for this is that a high mutation probability requires a more efficient coding of the self-replication. A creature that uses too many instructions for self-replication would be less likely to produce a viable offspring when the copy mutation probability is high.

The gene correlation on the other hand increases when the mutation probability increases. The high mutation probability forces the creatures to compress the information in the genome, in order to make it less likely to be struck by a deleterious mutation. This compression corresponds to that the genes share instructions which gives a higher gene correlation.

The reason why the redundancy decreases when the mutation probability increases is a puzzling result, to which we have no explanation yet.

6.7 Conclusion

The results of the experiment clearly show that settings in Avida produce different coding styles. Most of the differences that appear are intuitive and can be explained directly from for example how the merit is calculated. While some results, like the change in redundancy in the experiments with mutation probabilities, are a bit more puzzling and require further investigation to be explained.

What these experiments show is that different environmental settings affect different stylistic properties of the genome. The gene correlation doesn't seem to depend much on the size merit method but rather on the mutation probabilities and the fraction of introns seems independent of the mutation probabilities but depends strongly on the size merit method. The fragility measure on the other hand seems to depend on both mutation probabilities and size merit method. But the main result of this section is that we can distinguish between different coding styles from different environments using a stylistic profile measure.

One question that arises is if the coding style of a genome is independent of what is coded into the environment and thus only depends on how it is coded. To investigate this one would have to go beyond Avida, which only rewards boolean functions, and evolve populations in environment that reward vastly different functions.

7 Hierarchies in Avida

7.1 Introduction

The Avida architecture has a hierarchal structure with the smallest unit being a single instruction and the largest being the entire population. In this section we will investigate this structure in more detail and investigate emergent properties of the different levels in the hierarchy.

7.2 Structural levels

At the lowest level we have the instruction, which serves as a codon in the Avida chemistry. The instructions can be divided into categories: No-op's, Flow control operations, Biological operations etc., but by looking at a single instruction it's impossible to tell what the meaning of the instruction is, the instruction has to be put into a larger context in order to have any meaning.

This leads us to the next level which is the gene. Putting together two or more instructions we get a gene, an example of a gene is an I/O-gene, like a NAND-gene.

Genes in an Avida creature tend to overlap and share instruction, this motivates the next level which we call gene clusters. These are subsets of the code that contain two or more genes.

The next level is the entire creature, which is a self-replicating digital organism with metabolism. By identifying creatures with the same genomes we define a genotype. Creatures with similar, but not identical genomes, are labelled with as species, and at the highest level we have the entire population.

Below is a summary of the different structural levels in Avida.

1. Instruction
2. Gene
3. Gene cluster
4. Creature
5. Genotype
6. Specie
7. Population

There can also exist a level between creature and genotype, if we allow the CPU of a creature to move its instruction pointer to a neighbouring creature. If this is possible two creatures can form host-parasite or symbiotic pairs, which then would define a new structural level. Fig. 7.1 on the next page shows an example of the hierarchal structure of an Avida creature from the instruction level to the creature level.

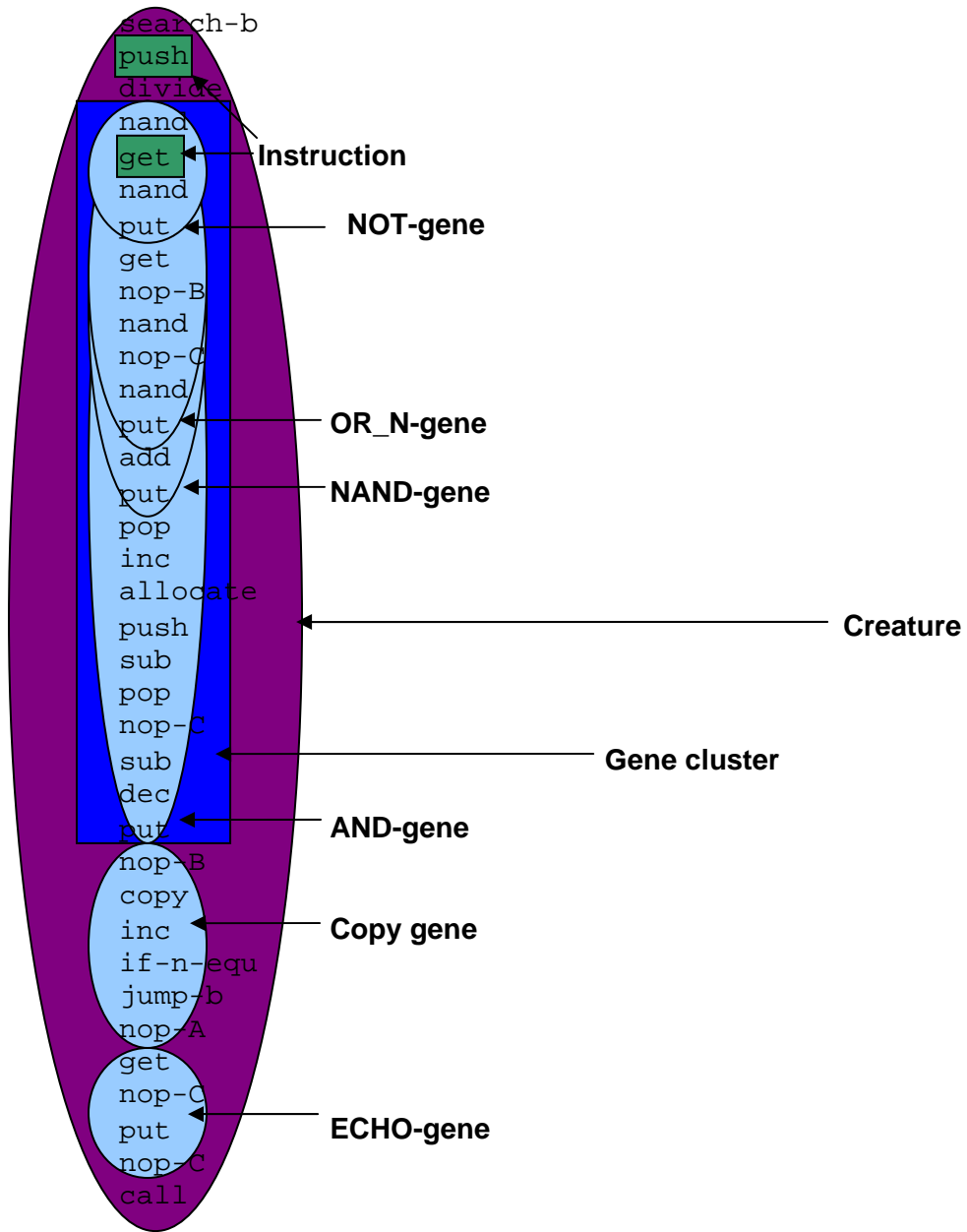


Figure 7.1 Shows the hierarchal structure of a typical Avida creature. The instructions are the lowest level in the hierarchy, they are the building blocks for the genes. The genes tend to overlap in gene clusters. The genes, gene clusters together form the entire creature.

7.3 Emergence

As mentioned the Avida architecture constitutes a hierarchal structure. It is therefore useful to use the notation introduced in [12] for describing the system.

Let S_i be a family of “agents”, let Obs^1 be observational mechanisms and let Int^1 be the interactions between the agents. The observation mechanism measures the properties of the agents to be used in the interactions. The interactions now generate a new kind of structure S^2 in a reaction:

$$S^2 = R(S_i, Obs^1, Int^1)$$

This leads to the following definition of emergence:

$$P \text{ is an emergent property of } S^2 \Leftrightarrow P \in Obs^2(S^2) \text{ and } P \notin Obs^2(S_i^1)$$

In this formalism we let S_i be the instruction set, Obs^1 is then the action of each instruction on the virtual CPU, and we set Int^1 to be the composition of several instruction. The reaction of the above gives the genes, S^2 . The genes also interact by composition, but they have other observables, namely the action of the entire gene on the CPU, that describe their action like for example performing a XOR or calculating the length. The next reaction produces gene clusters S^3 , they interact as before, by composition. The observables of the gene clusters are sequences of tasks the gene cluster produces when it’s executed. The reaction of the gene clusters gives the creature S^4 . At this level we see a big change in structure and new observables like fitness and gestation time occur.

A creature may hold redundant instructions, that aren’t part of any gene or gene cluster. In order to make them fit in this hierarchal structure we define them as genes that don’t have any function.

The creatures interact according to the rules in the architecture of Avida and further reaction yields genotypes, species and finally the entire population. From this we can conclude that Avida is a hyperstructure, as it’s a multi-level emergent structure.

Some of the levels in the hierarchy are inherent from the architecture of Avida. For example the creature level doesn’t emerge in the process, it is hard-coded in the architecture of Avida. The same goes for the instruction and population.

Other levels do emerge in the process, like the gene and gene cluster level. The genes emerge because certain functions are rewarded with extra execution time. The emergence of this level is not very surprising as the creatures are subject variation and if a gene that is rewarded occurs it will fix in the population, as a creature that holds the gene will replicate faster than creatures that don’t have the gene.

The emergence of the gene cluster level on the other hand can not be explained as easily, and emerges for a number of reasons, which will be studied in detail in section 8.

We now turn to the properties that emerge in this system. A striking example is that of fitness, which is a part of $Obs^4(S^4)$, but isn’t a part of $Obs^4(S_i^3)$, one cannot assign a fitness to only a subset of the code. This shows that fitness is an emergent property. In the same way we can show that the gestation time also is an emergent property.

Let us now define a property to be global, with respect to the creature level, if it is an emergent property of S^4 and local otherwise, and as we only are interested in properties of the creature and lower levels, we will disregard the higher structures for now.

Using this formalism to investigate the stylistic measures defined in section 6.

7.4 Emergent properties of stylistic measures

The stylistic measures were designed to be able to show the difference between different coding styles, but it is also of interest to know on what levels in the hierarchy these properties emerge.

Gene correlation – The gene correlation is constructed from the FGA. The calculation of the FGA uses the entire genome, which implies that the gene correlation is an observable on the creature level. The gene correlation can't be observed on a lower level as the FGA can't be constructed from a subset of the code. This implies that the gene correlation is an emergent property of S^4 and thus global.

Redundancy – this measure is also constructed from the FGA, which, by the same reasoning as above, makes it a global measure.

Introns – this measure is constructed from the execution sequence, which is produced when the creature is executed for one generation. As the process requires the entire creature this measure is an emergent property of S^4 and thus global.

Gene reuse – the gene reuse is constructed from the gestation time. The gestation time is an emergent property of S^4 , which implies that the gene reuse is an emergent property and thus a global measure.

Fragility – this measure is constructed from the FGA, and is therefore also a global measure.

From the above examination of the stylistic measures we see that all the measures used actually are global measures. The above examination also shows that a measure is local if it can be applied (and have meaning) to a subset of a creature and global otherwise. Note that the gene correlation, redundancy and intron measures could be applied to a subset of the code if we drop the aspect of self-replication. The programs written in the Avida language never terminate if no divide occurs, but if we let the program run for a large number of time steps and construct the FGA from that execution (without a replication column) the measures mentioned above could be constructed on a lower level, and thus be local measures. We shall make use of the difference between local and global properties in a later section.

8 Gene correlations

8.1 Introduction

The gene correlation (8.1) is a measure on a creature that was introduced in section 6. It measures how correlated the genes in a creature are, which also corresponds to how compressed the information in the genome is. Creatures that have evolved their code in Avida tend to have a high gene correlation. It was shown in section 7.4 that the gene correlation is a global measure on the creature level. If the gene correlation is high many functions share instructions, which is the definition of a gene cluster given in section 7. This implies that the gene correlation can be interpreted as a measure of gene clusters in the genome, which are emergent structures.

We will show that there exist three different mechanisms that reinforce the high gene correlation.

$$g_c = \frac{\sum_i \mathbf{T}_i}{NL_E}, \quad \text{where the sum runs over all indices for which } \mathbf{T}_i > 1, \quad (8.1)$$

\mathbf{T} is the row sum of the FGA, N is the number of tasks and L_E is the number of essential instructions

8.2 Mutational effects

The structure of the genome in a creature simply reflects the best way to store information in the genome with respect to the environment where the creature evolved. In Avida the optimisation of information storage creates a high correlation between different genes. The result is a genome, where a single mutation can knock-out several functions and reduce the fitness significantly, but the probability of a deleterious mutation is lowered with this structure, as the number of instructions where information is stored is lowered. One would expect that a structure like the above would be disadvantageous for a creature because of the high risk of a significant fitness drop, as it seems so fragile. But this is actually not the case. The reason for this is that a fitness drop is bad in all cases, no matter how small it is it will in most cases kill the creature eventually. Consider the following example:

Consider two genomes that are phenotypically equivalent, one of the creatures has a high gene correlation while the other a low correlation. Now assume that they both are surrounded by members of their own genotype on the grid. Further assume they are both struck by a deleterious mutation, which affects them in different ways because of their genetic structure. One of them will have its fitness reduced very much while the other only very little. How will they be able to compete with their neighbours after the mutations? Actually they will both go extinct very soon as they both have a fitness (reproduction rate) that is lower than the average in their neighbourhood.

But the example above is idealised as the neighbours of a creature very seldom only consists of creatures from its own genotype. We must therefore compare the fitness loss due to a

mutation against the variability of fitness in the population. If the fitness after the deleterious mutation is lower than the lowest fitness in the population then the mutant will go extinct. To examine this we calculated the relative fitness difference between the maximal and the minimum fitness within the population using (8.2) (only viable genotypes were considered). This quantity was then compared to the average relative fitness decrease (8.3) of a deleterious mutation of a typical creature. We assume that the creature loses a multiplicative bonus Δb , where $\Delta b \geq 1$. This implies that the new fitness, after the mutation, can be written $\alpha_m = \alpha/\Delta b$.

$$\Delta\alpha_p = \frac{\alpha_{\max} - \alpha_{\min}}{\alpha_{\max}} \quad (8.2)$$

$$\Delta\alpha_m = \frac{\alpha - \alpha_m}{\alpha} = \frac{\Delta b - 1}{\Delta b} \quad (8.3)$$

The relative fitness difference for the population was averaged over 10 runs each lasting 10 000 updates, where the maximum and minimum fitness was sampled every 100 updates and the result was $\Delta\alpha_p = 0.4080$ on average. In all runs size merit method 4 and the standard mutation rates were used, the task bonuses were set to the default values (Appendix A). Fig. 8.1 shows the relation between the two quantities, where $\Delta\alpha_m$ is plotted as a function of Δb .

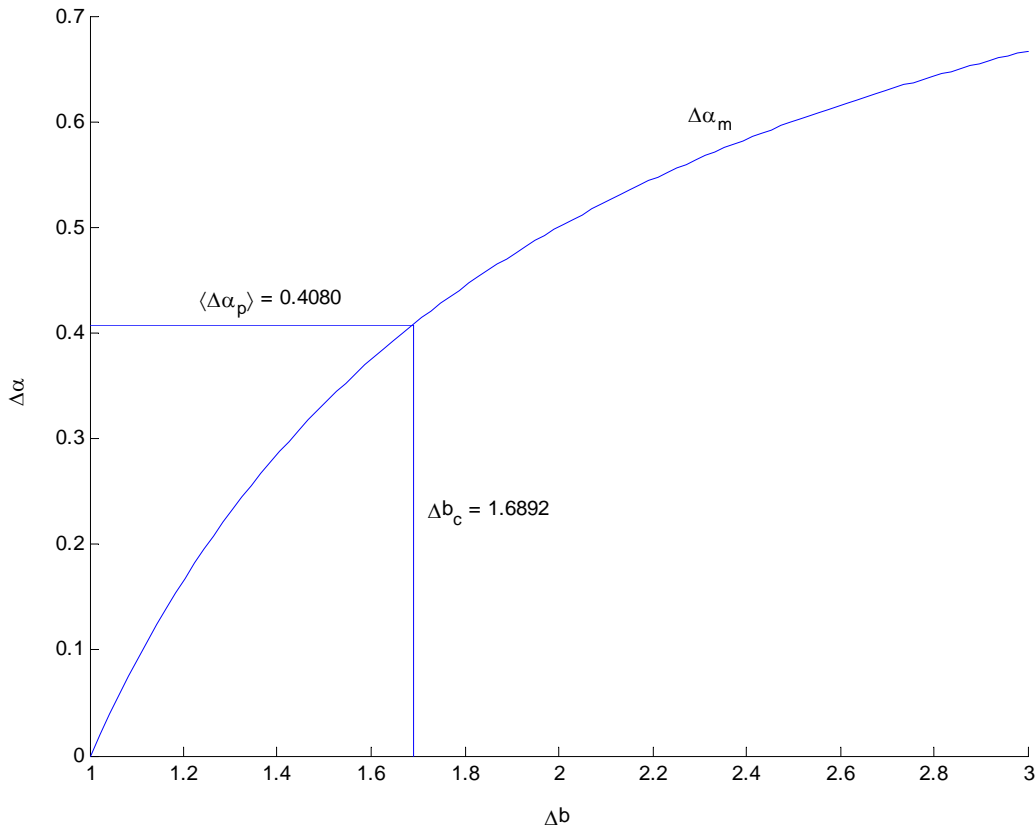


Figure 8.1 This plot shows the relative fitness difference of a mutant, due to a mutation that removed a multiplicative bonus of Δb , compared to the average relative fitness difference in the population. The plot shows that if a task that acquires a higher bonus than 1.7 is lost the mutant will go extinct.

From fig. 8.1 we see that if a creature loses a task that acquires a higher bonus than the critical bonus, $\Delta b_c \approx 1.7$, it will fall down below the lowest fitness and consequently will go extinct. The only functions rewarded by a bonus lower than 1.7 are ECHO, NOT and NAND, which means that if a deleterious mutation destroys a higher task it might as well destroy several functions, because the mutant will go extinct anyway. Note that the critical bonus is an overestimate as it is based on the highest fitness in the population. A creature that lies somewhere between the highest and lowest fitness will have a lower critical bonus as the relative fitness difference between that creature and the least fit creature will be smaller.

The above implies that it isn't disadvantageous to have a high gene correlation, as a single deleterious mutation will kill the creature although it only affects one function. Note that this effect is due to the strong selection pressure in Avida. If the variability in fitness would be larger, then a small fitness loss would be beneficial compared to a large one, which implies that an uncorrelated structure would be more beneficial.

8.3 Compression

The second mechanism that reinforces gene correlation is the fact that a high gene correlation compresses the information about which tasks that are beneficial, which is stored in the genome. The compression of this information reduces the number of instructions required to store the information. A gene correlation of 1 corresponds to maximal compression, while a gene correlation of 0 corresponds to no compression. If the information is compressed it has a higher chance of being transmitted correctly when the creature replicates, as the effective mutation probability is lowered. The fidelity, which is the probability of a creature to correctly transmit its code, is given by (8.4), where R is the probability of a copy mutation, P_i and P_d are the probability of an insert mutation and delete mutation and L is the length of the creature.

$$F = (1 - R)^L - P_i - P_d \quad (8.4)$$

But creatures in Avida have redundant instructions that don't take part in any of the functions that the creature performs and therefore don't affect the phenotype. These instructions don't have to be copied correctly in order for the offspring to have the same phenotype. We therefore introduce the effective fidelity, which only takes the instruction that perform functions, including self-replication, into account. It can be interpreted as the probability for a creature to correctly transmit at least its phenotype to the offspring. By at least we mean that the offspring can become a higher (collect a higher bonus) phenotype, because mutations in redundant instructions can create new functions.

The total length of a creature can be decomposed into two parts: the redundant and the essential instructions, $L = L_R + L_E$, by essential we mean instructions that are used in any function that the creature performs including self-replication. The effective fidelity is now simply defined as the standard fidelity (8.4) with L changed to L_E (8.5). This is actually an over estimate as an insert or delete mutation may be neutral or advantageous if it occurs adjacent to a redundant instruction, but as we don't want to consider the exact structure of the genome we will use this simplified form.

$$F_e = (1 - R)^{L_E} - P_i - P_d \quad (8.5)$$

Now assume that we have two creatures from the same phenotype, but with different gene correlation. As they can perform the same functions the one with the higher gene correlation will have lower L_E than the other.

If we now look at the difference in effective fidelity ΔF_e between the two creatures with L_E and L_E' , where $L_E > L_E'$

$$\begin{aligned}\Delta F_e &= (1 - R)^{L_E} - P_i - P_d - ((1 - R)^{L_E'} - P_i - P_d) = [\text{Taylor expansion}] = \\ &= 1 - RL_E' - 1 + RL_E = R(L_E - L_E')\end{aligned}$$

This shows that the effective fidelity depends approximately linearly on the number of essential instructions. As the gene correlation increases when the number of essential instructions are reduced (within a phenotype class), a high gene correlation yields a high effective fidelity.

A high effective fidelity corresponds to that the creature protects the information that is has gained, by compressing it. Note that it's beneficial for a creature to have a high effective fidelity, which is not the case too with the standard fidelity, as the creature can't adapt to the environment if the standard fidelity is high [13]. Experimental evidence shows that the learning rate is maximised when $F = 1/e$ [13].

8.4 Development

The third and final mechanism that reinforces gene correlation is that of development. The functions that are rewarded in the avidian environment are basic boolean functions. The building block of all these functions is the `nand` instruction. This means that it's possible to extend a function by adding a `nand` and thus turning it into a higher function, which gives a higher bonus, section 5 shows this in more detail. Note that this can be done in a way that the original function still is kept. This way of constructing the genes requires less effort, as it uses existing functions as building blocks. The probability for a higher function to evolve from a lower is much higher than the probability for the higher function to be coded from scratch. It has even been shown in [10] that higher functions like EQU won't appear in the population if lower functions aren't rewarded. This implies that the use of lower functions as building blocks is the only way to construct more advanced functions, and consequently creatures that can perform higher tasks will have a gene correlation that is non-zero, if the lower tasks are kept.

8.5 Conclusion

The three mechanisms described above all contribute to a high gene correlation in the genome of Avida creatures. A highly correlated structure is actually easier to transfer to the offspring because it's less likely to be struck by a mutation, which will result in a lower fitness of the offspring. This, together with the fact that the fitness variability is very small in Avida, promotes correlated genes. The correlated structures are also much easier to develop for

creatures, as the boolean functions in Avida can be built on top of each other. A higher function is more probable to arise from a current gene than to be coded from scratch. Summing this up one can say that the correlated structures are easier to both develop and maintain in Avida.

9 Comparison to human code

9.1 Introduction

Computer programs can be constructed in many different ways. The classical way is that it is the product of a human programmer as a product of her/his creative thinking, but recently new ways to produce programs or algorithms have appeared. This new approach takes inspiration from nature and uses natural selection and variation in an evolutionary process to create code that will solve a particular problem. The same approach is used in the field of artificial life, but here the main objective is not to solve a particular problem, but rather to study the process in which a population evolves in a specific environment.

It is a well known fact that code produced in an evolutionary process is very hard to understand and often contains instructions that seems very illogical to a human programmer, it can be said to have an evolutionary coding style that is very far from the coding style of its human counterpart [4]. The idea of this section is to compare the human coding style with the coding style of programmes that are a product of an evolutionary process, but let us first begin by distinguishing these two different styles of coding.

9.2 The human coding style

The human coding style, which was the only known until two decades ago, is what we humans consider good programming. The code should be well structured, not contain any unnecessary instructions, different functions in the code should be separated, it should be easy for an outsider to understand the code and it should be easy to make changes to the code in the future.

These are only a few things that influence human coding but we think that the above features capture the essence of the human coding style.

9.3 The evolutionary coding style

The evolutionary coding style is harder to describe, as it depends on the environment in which the code has evolved (see section 6), but I think some generalisation can be made. The code often holds redundant code or even unexecuted instructions, the code often contains complex connections between different parts of the code and it often contains unexpected solutions. An example of this can be found on p. 17, where an `add` instruction is used to perform a boolean function.

9.4 A top to bottom perspective

If we look at a code produced in a creative process be it human or evolutionary, what this program tells us by just examining the code is the purpose of the code. In the evolutionary case it tells us how to survive as effectively as possible in the environment in which the code evolved. It also tells us what traits or computations that were beneficial in that environment. In the human case it rather tells us what specification the programmer was given before he started coding and what features the code should have, should be as fast as possible or with low complexity etc. But it also gives us an idea of the individual coding style of that programmer.

So the product of the creative process reveals facts about the process in which it was created, we can say that there is a connection between the specification given to the programmer, the demands put on his program and the environment that the evolutionary code was created in. We can say that the human written code is a result of the specification together with the programmer's notion of what constitutes good coding.

9.5 Human fitness

If a programmer is given a number of programs that have the same specification (i.e. they perform the same function), but are programmed by different persons he will most probably be able to distinguish them by their coding style, and if he was asked to rate the programs that too wouldn't be an impossible task. The possibility to rate programs suggests that that one can assign fitness to human written programs as well. This fitness function, which of course depends on who is rating the programs, is a combination of all the features that constitute good programming mentioned earlier together with how well the program manages the specification. A program that doesn't fully manage the specification, but is coded in a very good fashion could still get a very high rating. Although human written code is not evolved in the sense of biological or digital evolution with random mutations and survival of the fittest, it can be said to have been developed under certain selection pressure which somewhat corresponds to the environment in the evolutionary case.

9.6 Local vs. global properties

In the human case many of the features that constitute the fitness depend on local properties, i.e. properties that can be defined for a subset of the code. Examples of these properties are that the structure is too complex in some part of the program, the fact that some instructions maybe redundant and that functions are separated. This means that local properties of the code influence the fitness in the human case. This implies that although a program manages the specification very well it maybe disregarded for its local properties.

In the evolutionary case this local optimisation doesn't occur, it's only how well the creature manages the specification that determines its fitness, which is a global property. In Avida the fitness is computed using (9.1), which only depends on global properties.

This implies that there occurs no optimisation of local properties, but the fitness actually depends on the genes, which are a local structure. So what we have in the evolutionary case is

an optimisation of a global property that depends on local structure, on which there is no selection pressure.

$$\alpha = \frac{bM}{t_g}, \text{ where } b \text{ is the bonus acquired by performing tasks} \quad (9.1)$$

If one would re-write Avida so that it also based its fitness on local properties, like separated functions and easy-to-read code, then the result would probably be something that resembled human written code. But this modification would of course remove the biological relevance of the system.

9.7 A case study

In this section we will investigate two programs that have can perform the same function, but come from two different sources. The programs are written in the assembler-like language of Avida and they can both perform the following functions: ECHO, NOT, NAND, OR_N, OR, AND_N, NOR and they have the capability to reproduce themselves. One was written by me (080-aaaaa) and the other is the product of a run with Avida (042-accbt). As the programs are a bit hard to analyse just by looking at the code (which can be found in Appendix B), we will instead look at their Functional Genomic Array (FGA), which reveals information about where the different functions are coded and which instructions are redundant and at their graphic genome representation. We will also compute their coding style vector, a concept defined in section 6. Table 9.1 shows the stylistic vector for the two programs, fig. 9.1 and 9.2 show their FGA's and fig 9.3, 9.4 show a graphical representation of their genomes. From table 9.1 we see that the two codes differ quite much in their coding style, and we shall now discuss each stylistic measure in detail.

Type	Gene correlation	Redundancy	Introns	Gene reuse	Fragility
Avida	0.4476	0.1190	0	1.0244	0.4512
Human	0.0593	0.1750	0.0375	1.3051	0.5458

Table 9.1 The stylistic measures of one human written and one evolved program

Gene correlation – the human code has less correlation between functions (genes). The reason for this is because it is considered good programming to separate different functions in the code and don't let them overlap like in the avidian. The reason for the high gene correlation in the Avida creatures is investigated in detail in section 8.

Redundancy – The human code actually has a higher redundancy than the avidian. There reason for this is that this particular avidian has an extremely low redundancy. The average redundancy in size merit method 4, under which this creature evolved, is actually 0.5230. This implies that most avidians have a redundancy much higher than the human written code. In human code redundant instructions don't have any purpose, but in avidians the redundant instructions serve as development areas where new functions can be coded without affecting the current genes.

Introns – This is the measure where the two coding styles agree the most. The reason for this is that the size merit-method that the avidian evolved in punishes introns, and the reason why the human code doesn't contain introns is quite obvious.

Gene reuse – Both programs have a high gene reuse, which indicates that they both have a linear execution without loops.

Fragility - The fragility is lower for the avidian, because it was created in a process where it is beneficial to handle mutations, while the human code doesn't take this into account.

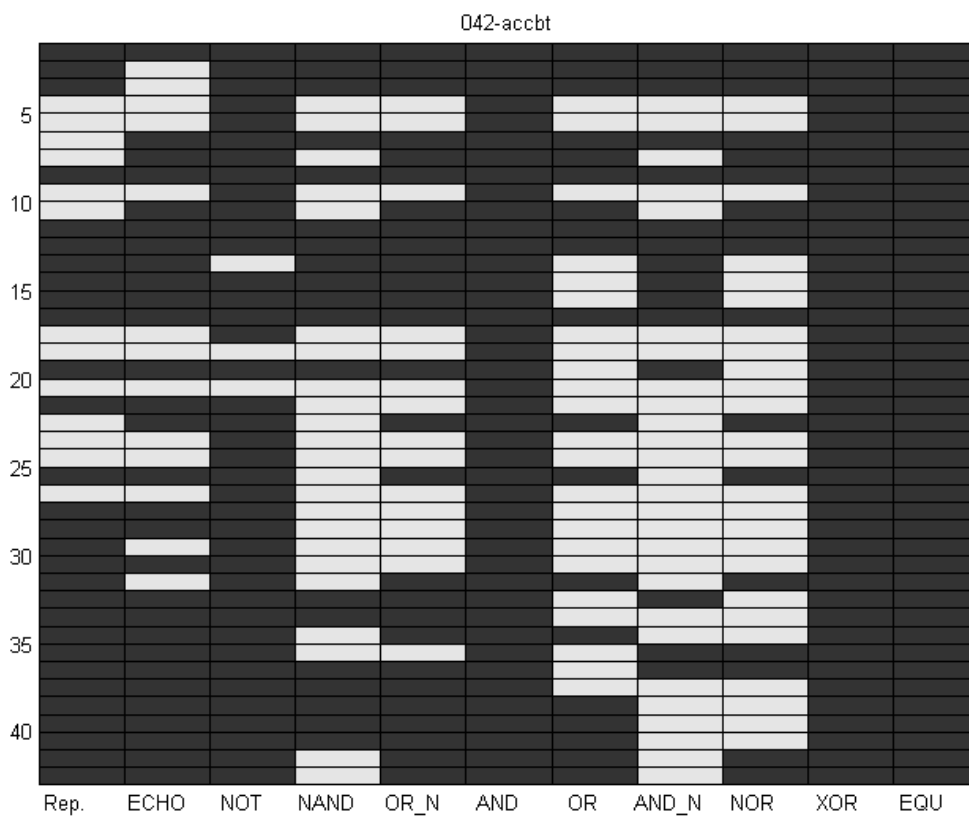


Figure 9.1. FGA for the Avida code

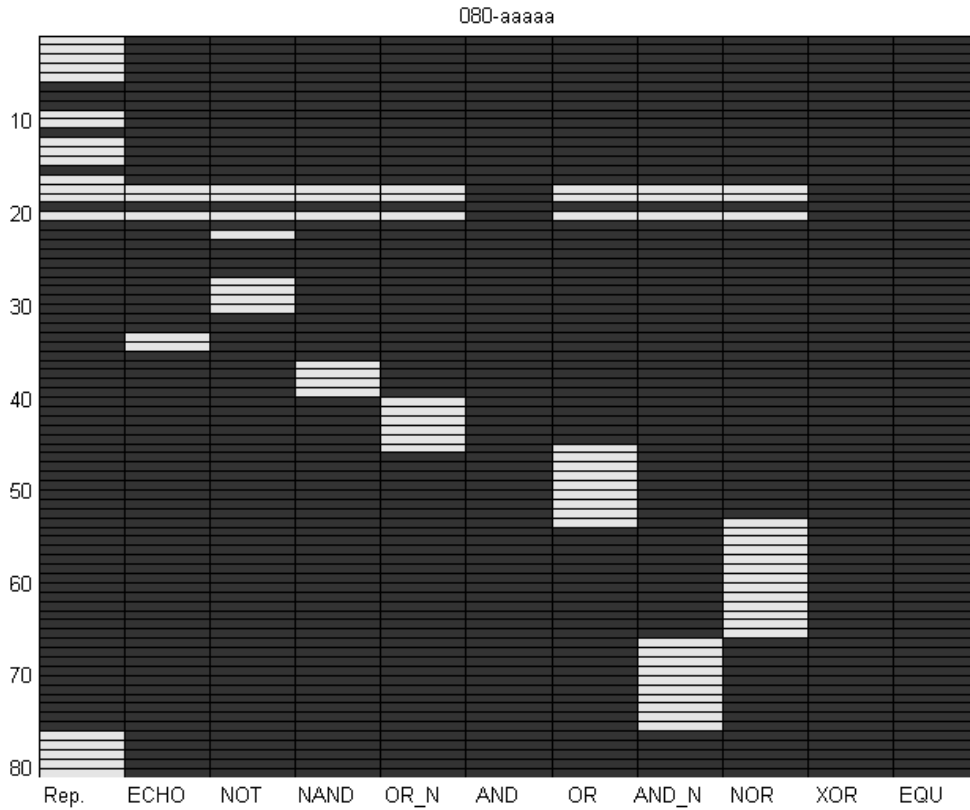


Figure 9.2. FGA for the human written code

If we now look at the FGA's the differences are striking. The functions in the human code are well separated, while they in the avidian are highly connected and overlapping, there are even several instructions that affect all output functions. Seeing this in an evolutionary perspective we can say that the lack of gene separation in the avidian is simply a result of the global fitness function in Avida, while in the human case, we have a fitness function which rewards separated functions (genes). The differences in the graphical genome representation are also striking, the genes in the avidian overlap heavily while they are well-separated in the human written code. The reason why it seems like the Avida code has introns is because it executes the child part of the genome before a divide occurs, which allows it to use the same code part for different outputs.

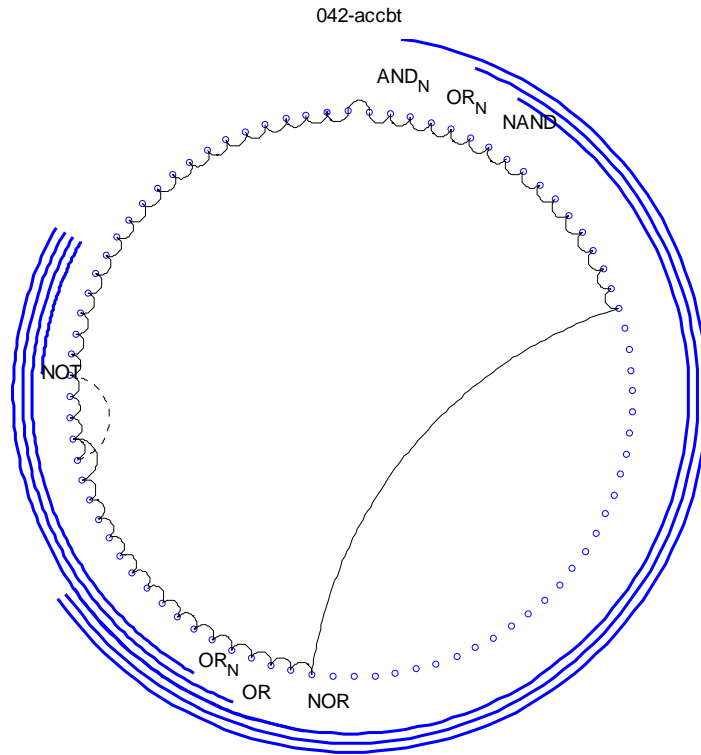


Figure 9.3 Graphic genome representation of the Avida code

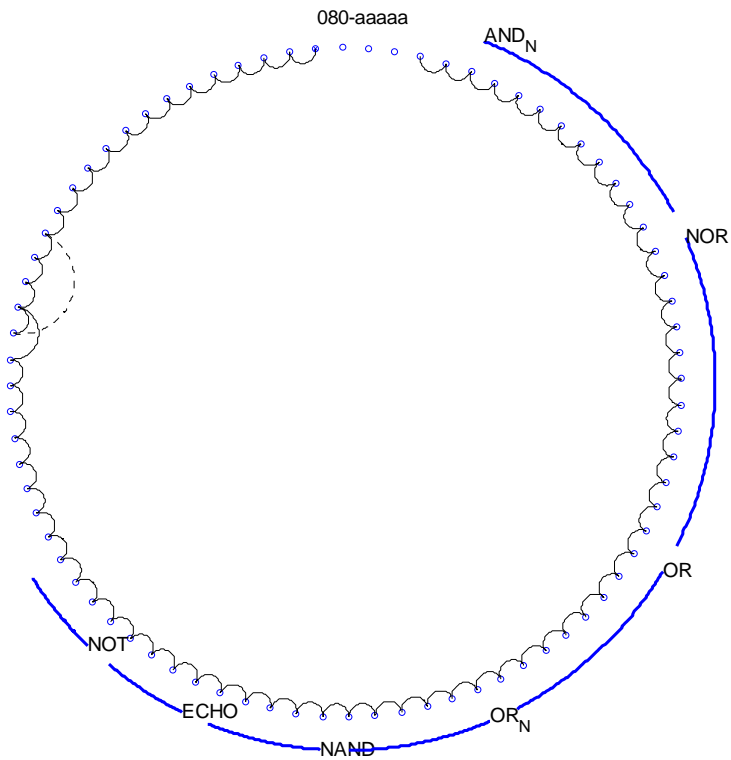


Figure 9.4. Graphic genome representation of the human written code

9.8 Conclusion

As seen in the case study the human and evolutionary coding style differ vastly. The underlying reason for this is the difference in the fitness functions that govern the processes. The avidian fitness function is only based on global properties, while the human fitness function takes both global and local properties into account. The existence of local selection pressure in the human case leads to features like, easy-to-read, well-structured code and non-existent redundancy. We believe that it is the difference between local and global selection pressure that give the vast differences between human and evolutionary coding.

Both coding styles are optimal in respect to the environment in which they were created. Human coding is well structured because it should be easy to understand and easy to change, while evolutionary code has a complex structure because it should withstand mutations and store and transmit information as effectively as possible. The evolutionary code was never meant to be inspected and understood, its only purpose is to survive in the given environment. When a human programmer examines a code that is produced in an evolutionary process, he examines it using his preferences of what is good programming. This means that he looks at local properties of the code, as it is these properties that constitute a good human coding style, but the evolutionary code was never under selection pressure that depended on local properties, as it only evolved under a global fitness pressure. This implies that the human programmer will find lots of illogicalities and bad structuring in the code.

In the case of Avida the lack of local selection pressure and optimal information storage/transmission creates a high correlation between different genes/functions. One can say that creatures from Avida are robust in the sense that they have a low probability of a deleterious mutation, but this doesn't compare very well with the human sense of robustness. From a human point of view one would rather lose a single function if an error occurred than, in the extreme case, all functions performed by the program.

10 The Blindfolded programmer

10.1 Introduction

As we have shown in the preceding chapter the human and evolutionary coding styles differ in several aspects, and that these differences arise because of the different selection pressure the codes are subject to. But the question of specification is also interesting when we compare human and evolutionary code.

A human programmer always has a specification of his program before he starts to code it, he always has a goal or purpose with the program. This naturally yields structure and sense to the code. The evolution on the other hand, which can be said to programme evolved programs, has no purpose and certainly no specification. The only thing evolution does is to produce programs that are as fit as possible in a given environment. In a sense one can say that the specification of the evolved programs is hidden in the environment and is revealed during the process of evolution, as a gene can never be known to increase fitness until it actually has been coded in the genome of a creature. So in the process of evolution the specification has to be discovered and implemented. This always happens simultaneously, a task can never be known to raise the fitness until it has been coded into the genome, but the implementation can then be optimised after the specification of this task has been discovered.

If we now turn the tables and give the human programmer the task to write a program without a specification, what would he do?

Consider the following thought experiment:

Imagine a programmer whose task it is to write an efficient program in an unknown environment (unknown specification). He doesn't know the programming-language with which the program is written, he is given an initial program, which has non-zero fitness, and in his possession he has a test CPU in which he can test a program. This test CPU only gives the fitness of the program nothing more. How would he proceed to construct a program with highest possible fitness?

The simplest approach would be a kind of trial and error method, where he modifies the program randomly and checks if the fitness is raised. This algorithm can be described as follows: Change/insert/delete a random instruction from the given program. If the fitness is raised keep the modified program and modify that, otherwise modify the original program again. This is obviously a kind of hill-climb method in the fitness landscape defined by the test CPU, and we call it "The Blindfolded Programmer" or BFP for short.

There are of course more advanced strategies than the above, where the programmer keeps track of where fitness lowering changes in the code occurred, but the above algorithm is very simple and that's why I shall investigate its properties in more detail.

10.2 Avida vs. the Blindfolded Programmer

The process of evolution actually resembles the hill-climbing strategy of the Blindfolded Programmer. The change/insertion/deletion of a random instruction corresponds to mutations and survival of the fittest in evolution is a kind of hill-climbing, where the population serves as memory. If a mutation is fitness lowering the creature will die, but the genotype from which the creature was spawned still exists and therefore serves as a kind of memory.

The main difference lies in the way this hill-climbing is done. In the BFP only one point in the fitness landscape is explored at a time, while in the evolutionary process it's rather a case of a distributed hill-climb. Each creature in the population is then represented by a point in the fitness landscape and offspring with mutations give rise to new points which, depending on their fitness survives or goes extinct.

Although the two processes share the hill-climbing aspect there are aspects in which they differ. In Avida the creatures replicate under mutations, which implies that creatures that can handle mutations (e.g. have a low fragility or compressed genes) are advantageous, further the population in Avida explores a larger section of the sequence space due to the distributed nature of the hill-climb, the locality in Avida allows for a high diversity in the population, as information spreads in a diffusive process, and the existence of implicit mutations allows for larger jumps in the sequence space.

None of these features are taken into account in the BFP. The question is if the similarities in the coding process yield similarities in the coding styles of the resulting programs, or if these two processes give rise to different coding styles.

The aim is to run the BFP in the Avida fitness landscape and then compare the coding style of creatures from the BFP to creatures that have evolved in Avida. The comparison will be done using the stylistic measures introduced in chapter 6.

10.3 Implementation of the Blindfolded Programmer

The Blindfolded Programmer was implemented in Java and has the following structure. The programs are represented by a class called `Individual`, which simply holds the genome and methods used for modifying the genome, for example inserting a random instruction. The individuals are executed in a virtual CPU, which is a copy of the one in Avida, and all the statistics of the execution are stored in a class called `Run`. From this class one can extract characteristics like the gestation time, tasks performed, number of executed instructions, introns etc., using the methods of this class.

The algorithm is seeded by an ancestor of choice, which is then modified by a random change of an instruction, insertion of an instruction, or deletion of an instruction. The ratio between these events is decided by the mutation probabilities like in Avida. As we want a change to occur every time step the original mutation probabilities are transform to new ones which sum up to one, so that a mutation occurs every time step. The probability that a child is copied incorrectly in Avida is $1-(1-R)^L$, where R is the copy mutation probability and L is the length of the creature. We therefore set the probability for a "copy mutation", which in the BFP corresponds to a random change of one instruction to $p_m = 1-(1-R)^L$. If we let $p_i = p_d$ be the probabilities for insert and delete mutations, the normalisation constraint can be written:

$$p_m + p_i + p_d = p_c + 2p_i = 1 \quad (10.1)$$

If we now assume that $p_m/p_i = k$, where k is a real valued positive constant, we can use the normalisation constraint (10.1) to find the new probabilities:

$$p_m = \frac{k}{k+2}$$

$$p_i = p_d = \frac{1}{k+2}$$

The two individuals, the original and the modified, are then executed in the CPU and their fitness's are compared. In order for an individual to be considered viable it must produce an offspring that also is viable. The reason for this is that I don't want to generate individuals that wouldn't reproduce correctly if they were placed in the Avida environment. If a creature isn't viable the fitness is set to zero.

If the fitness of the modified individual is higher it is kept and overwrites the original individual, and the procedure is repeated. If the fitness of the modified is lower then it is discarded and the process is repeated again. Figure 10.1 shows the development of the fitness in a typical run.

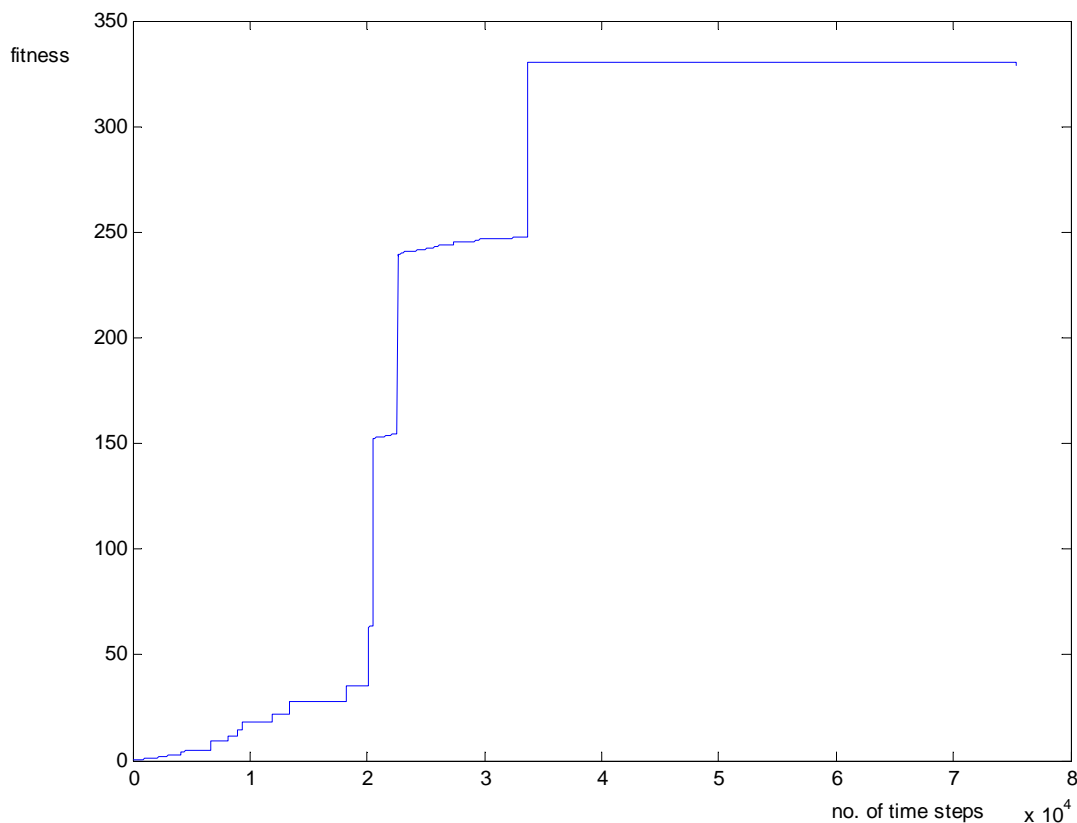


Figure 10.1 The fitness development as a function of the number of updates in a typical run of the "Blindfolded Programmer". The fractal scale-free behaviour of the curve can clearly be seen.

One can clearly see the fractal scale-free behaviour of the curve in fig 10.1 with fitness jumps of all length-scales, which is a result of the fractal fitness landscape in Avida [15]. The

resemblance to the development of the dominant fitness in Avida (fig. 10.2) is clear. Although the fitness development in Avida isn't as smooth, it still shows the fractal scale-free behaviour. The reason why the two curves resemble each other is simply because the same fitness landscape is used.

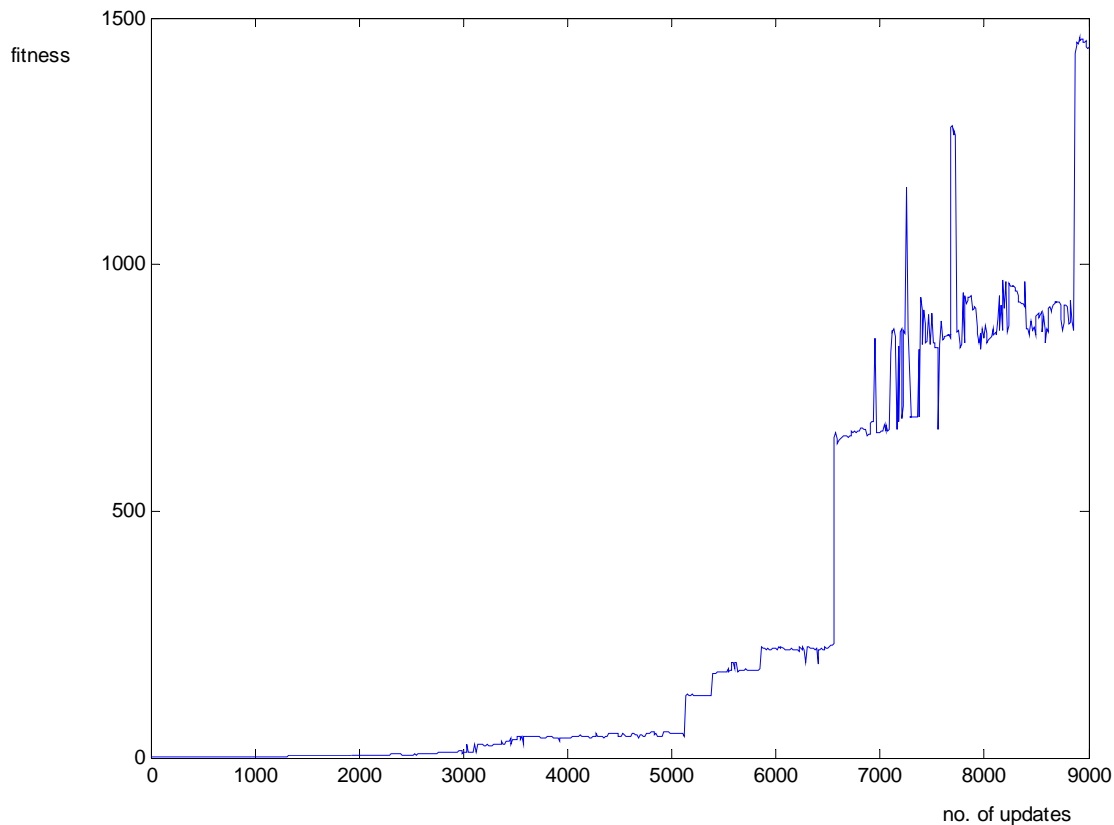


Figure 10.2 The development of the dominant fitness in a typical Avida run.

10.4 The coding style of the Blindfolded Programmer

In order to compare the coding style of the Blind Programmer to Avida we created a set of creatures using the blindfolded programmer algorithm with the CPU being set to size merit method 4, and one set of creatures from Avida with the same size merit method, the exact settings of the runs can be found in Appendix C. In order to compare creatures of approximately the same complexity (i.e. collect approximately the same bonus) we had to make sure that the number of time steps before a creature was extracted from the BFP corresponded somewhat to the 40 000 updates in Avida. This was simply done by trial and error and a value of 100 000 time steps gave in most cases a creature that could perform three or more boolean functions.

The two sets, containing approximately 50 creatures each, were then analysed using the stylistic measures introduced in the section 6. The result of the stylistic analysis can be seen in the table 10.1.

Process	Gene correlation	Redundancy	Introns	Gene reuse	Fragility
Avida	0.2505 (0.1038)	0.4279 (0.1531)	0.0225 (0.0220)	0.4063 (0.1425)	0.5289 (0.0598)
Blindfolded programmer	0.2702 (0.1457)	0.6486 (0.2037)	0.0334 (0.0346)	0.3860 (0.1224)	0.6507 (0.1124)

Table 10.1 The average and standard deviation for the profile measure of creatures evolved in the “Blindfolded programmer” and Avida, both under size merit method 4.

In order to compare if the coding styles were similar they were also plotted in a PCA-plot together with creatures from size merit method 0 and 1 generated in Avida.

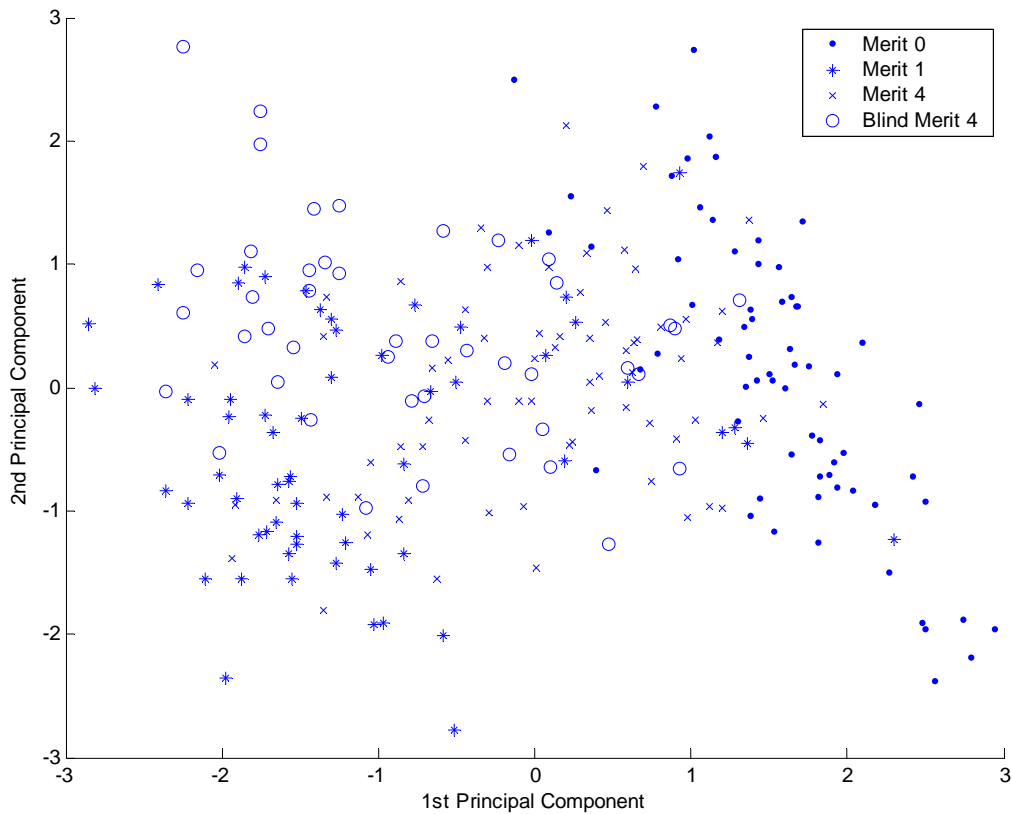


Figure 10.3 A PCA-plot of the three sets of Avida creatures from different size merit methods together with creatures from the “Blindfolded programmer”. The plot shows that the coding style of the creatures from the “blindfolded programmer” varies more than the coding style of creatures that have evolved in Avida.

10.5 Discussion

The stylistic measures presented in the table 10.1 show that the two coding styles are similar in some aspects but differ in others. The gene correlation, intron and gene reuse measures are similar for both processes, while the measures that differ significantly are the redundancy and fragility measures, which are both higher for creatures from the BFP.

One big difference between the coding styles doesn't actually show in the profile measure. The creatures from the BFP have an average length of 268 instructions, while the creatures from Avida only have an average of 79 instructions. This suggests that the creatures from the BFP are coded in a less efficient manner. This also shows if we look at the number of essential instructions in the creatures. The average number of essential instructions in the creatures from the BFP are $L_E = 65.1$ while the creatures from Avida have $L_E = 41.7$. As the creatures from Avida and the BFP collect approximately the same bonus we can conclude that the creatures from the BFP are coded in a less efficient fashion.

Figure 10.3 shows that the coding style of the creatures from the BFP is not as coherent as the coding styles from the creatures from Avida. The creatures from the BFP are spread out over all the three size merit methods, but lie mainly together with method 1 and 4.

10.6 Conclusion

The fact that the creatures don't need to replicate in the BFP, as opposed to Avida, seems to give rise to a coding style that differs in some aspect compared to the coding style of the Avida creatures. As the information stored in the genome doesn't have to be transferred to the next generation under mutations, there's no pressure on efficient coding of the information, which results in a less efficient coding and a higher fragility.

On the other hand there are measures that are similar for both Avida and the BFP. This indicates that some aspects of the coding style are related to the fitness function rather than to the dynamics of the process. If this is the case then the coding style could be broken down to one part that depends on the dynamics of the process and one part that depends on the fitness function. One way to test this would be to come up with other algorithms that generate creatures that are viable in the Avida environment. One could for example use different forms of genetic programming to accomplish this. Then one could compare the codes using the concept of stylistic measures to test which measures are constant in the different coding processes and which measures change.

11 Conclusion

In this thesis the evolutionary coding of Avida creatures has been studied. This has been done by creating a graphical representation of their genomes, representing the genes as logical circuit diagrams and by measuring stylistic properties of the coding in the genomes. Through this analysis we have shown that the genome of Avida creatures are characterised by strong correlations between different genes, substantial redundancy in the genome and by a reuse of code. The coding style has been shown to depend on the size merit method and copy mutation probability under which the population evolved. The analysis has also shown that the size merit methods and mutation probabilities affect different stylistic properties of the code, for example the size merit method influences the fraction of introns while the mutation probability influences the gene correlation.

The gene correlation was also studied in more detailed and the existence of correlation between genes has been explained with mutational effects, compression of information and the development of boolean functions in the genome.

When compared to human written programs, which are coded in a different fashion, the structure of evolutionary generated code seems complicated and illogical. We have argued that these differences arise from the selection pressures put on the codes. Human written code is characterised by local selection pressures such as easy-to-read code, separated functions and code that is easy to make changes to. The evolutionary code on the other hand is subject to a fitness function that only takes global properties into account. That together with the fact that the code replicates under mutations gives rise to a coding style that is optimal with respect to information storage and transmission in the environment where the code evolved. Finally we have shown that a simple hill-climbing algorithm, named the Blindfolded programmer, gives rise to a coding style that is similar in some aspects to the coding style of Avida creatures. This result may indicate that the fitness function and dynamics of the process each affect different stylistic properties of the genome.

Bibliography

- [1] John Sundman, http://www.salon.com/tech/feature/2003/10/21/genome/index_np.html
- [2] A.O. Schmitt and H. Herzel *Estimating the entropy of DNA sequences*, Journal of Theoretical Biology, 188: (3) 369-377, 1997
- [3] C. Adami, *Introduction to Artificial Life*, Springer-Verlag, 1998
- [4] T. Lundh, *In search of an evolutionary coding style*, pre-print
- [5] C. Adami, *Introduction to Artificial Life*, Springer-Verlag, 1998, p. 6
- [6] C.G Langton, *Artificial Life*. Proc. of an interdisciplinary workshop on the synthesis and simulation of living systems, Los Alamos, 1988
- [7] C.G Langton, *Studying Artificial Life with cellular automata*, Physica **D 22**, 120, 1986
- [8] S. Rasmussen, C. Knudsen, R. Feldberg and M. Hindsholm, *The Coreworld: Emergence and evolution of cooperative structures in a computational chemistry*, Physica **D 42**, 111, 1990
- [9] Digital Life Lab at Caltech, <http://dllib.caltech.edu/avida/>
- [10] R.E Lenski, C. Ofria, R.T Penncock, C. Adami, *The evolutionary origin of complex features*, Nature **423**, 139 - 144, 2003
- [11] R.E Lenski, C. Ofria, T.C Collier, C. Adami, *Genome complexity, robustness and gene interactions in digital organisms*, Nature **400**, 661 – 664, 1999
- [12] N. Baas *Emergence, Hierarchies, and Hyperstructures*, Artificial Life III, Ed. C. Langton, SFI Studies in the Sciences of Complexity, Proc. Vol. XVII, Addison-Wesley, 1994
- [13] C. Adami, *Introduction to Artificial Life*, Springer-Verlag, 1998, section 11.2
- [14] T. Ray, Tierra homepage, <http://www.his.atr.jp/~ray/tierra/>
- [15] C. Adami, *Introduction to Artificial Life*, Springer-Verlag, 1998, section 8.6

Appendix A

All runs in this experiment were performed with the below settings (log file settings excluded) in the genesis file, only size merit method or copy mutation probability was changed. Each run lasted 40 000 updates after which the dominant genotype was extracted and saved. If the dominant genotype didn't manage three boolean tasks it was discarded, as a certain degree of complexity was needed. The Java application and matlab files used for the stylistic analysis can be found at: <http://www.dd.chalmers.se/~f00phge/thesis/CodingStyle.zip>

```
# This file includes all the run-time defines...
VERSION_ID 1.3 # Do not change this value!

### Architecture Variables ###
MAX_UPDATES 100000 # Maximum updates to run simulation.
WORLD-X 40 # Width of the world in Avida mode.
WORLD-Y 40 # Height of the world in Avida mode.
MAX_CPU_THREADS 1 # Number of Threads CPU's can spawn
RANDOM_SEED 0 # Random number seed. (0 for based on time)

### Configuration Files ###
DEFAULT_DIR ../work/ # Directory in which config files can be found.
INST_SET inst_set.24.base # File containing instruction set.
TASK_SET task_set.txt # File containing task set.
EVENT_FILE event_list.txt # File containing list of events during
run.
START_CREATURE /genebank/creature.base # Creature to seed the soup.

### Viewer ###
VIEW_MODE 1 # 0=BLANK, 1=MAP, 2=STATS, 3=HIST, 4=OPTIONS, 5=ZOOM

### Reproduction ###
BIRTH_METHOD 1 # 0 = Choose Random Creature (poor for evolution!)
# 1 = Choose Oldest Creature
# 2 = Choose largest Age/Merit
# 3 = Choose only empty cells.
# 4 = Choose Random from entire soup (Mass Action)
# 5 = Choose Eldest from entire soup (Tierra)
DEATH_METHOD 0 # 0 = Never kill creatures.
# 1 = Kill when inst executed = AGE_LIMIT
# 2 = Kill when inst executed = length * AGE_LIMIT
# Modifies DEATH_METHOD
AGE_LIMIT 5000
ALLOC_METHOD 0 # 0 = Allocated space is set to default instruction.
# 1 = Set to section of other creatures (Necrophilia)

### Mutations ###
POINT_MUT_PROB 0.0 # Mutation rate (per-location per update) (x10^-6)
COPY_MUT_PROB 0.005 # Mutation rate (per copy).
DIVIDE_MUT_PROB 0.0 # Mutation rate (per divide).
DIVIDE_INS_PROB 0.05 # Insertion rate (per divide).
DIVIDE_DEL_PROB 0.05 # Deletion rate (per divide).

### Time Slicing ###
AVE_TIME_SLICE 30
SLICING_METHOD 3 # 0 = CONSTANT: all creatures get default...
# 1 = BLOCK: Block slice scaled to merit.
# 2 = PROBABILISTIC: Run prob proportional to merit.
# 3 = INTEGRATED: Perfectly integrated deterministic.
SIZE_MERIT_METHOD 1 # 0 = off (merit is independent of size)
# 1 = Merit proportional to copied size.
# 2 = Merit prop. to executed size.
# 3 = Merit prop. to full size.
# 4 = Merit prop. to min of executed or copied size.
# 5 = Merit prop. to sqrt of the minimum size.
TASK_MERIT_METHOD 1 # 0 = No task bonuses
# 1 = Bonus just equals the task bonus
```

```

### Genotype Info ###
THRESHOLD 5          # Number of creatures in a genotype needed for it
                    # to be considered viable.
GENOTYPE_PRINT 0     # 1 0/1 (off/on) Print out all threshold genotypes?
SPECIES_PRINT 0      # 0/1 (off/on) Print out all species?
GENOTYPE_PRINT_DOM 0 # Print out a genotype if it stays dominant for
                    # this many updates. (0 = off)
SPECIES_RECORDING 0  # 1 = full, 2 = limited search (parent only)
SPECIES_THRESHOLD 0  # max number of failures creatures to be same species

### END ###

```

For size merit method 1,4 the following task bonuses were used:

```

# Should be input tasks, then output tasks (otherwise, they will get reordered)
# Bonus types can be:
#      * -> merit *= bonus
#      + -> merit += bonus
#      ^ -> merit = pow(merit,bonus)
#
# Use "+ 0" for no bonus
#
# Task      type bonus type bonus ... # Meaning & Difficulty

# IO - Depends only on executing the instructions (no logic)
get         -          * 1.05      * 1.05      * 1.05
put         -          * 1.05      * 1.05      * 1.05

# Task (any last output is function of _last_ input)
echo        -          * 1.25      * 1.25      * 1.25      # A
not         -          * 1.5       * 1.25      * 1.25      # ~A

# tasks (any last output is function of _last_ two inputs)
ggp         -          * 1.25      * 1.25      * 1.25
nand        -          * 1.5       * 1.25      * 1.25      # ~(A and B) - 1 nand
or_n        -          * 2         * 1.25      * 1.25      # A or ~B - 2 nands
and         -          * 2         * 1.25      * 1.25      # A and B - 2 nands
or          -          * 3         * 1.25      * 1.25      # A or B - 3 nands
and_n       -          * 3         * 1.25      * 1.25      # A and ~B - 3 nands
nor         -          * 5         * 1.25      * 1.25      # ~(A or B) - 4 nands
xor         -          * 9         * 1.25      * 1.25      # A xor B - 5 nands
equ         -          * 9         * 1.25      * 1.25      # ~(A xor B) - 5 nands

```

For size merit method 0 the task bonuses were modified slightly in order to stop the population from going into a size minimising state, which happens easily with size merit method 0 as the creatures are punished if they increase their length. An increase in size is only beneficial if it is followed by a fitness increase. The task bonuses used for size merit method 0 are given on the next page.

```

# Should be input tasks, then output tasks (otherwise, they will get reordered)
# Bonus types can be:
#      * -> merit *= bonus
#      + -> merit += bonus
#      ^ -> merit = pow(merit,bonus)
#
# Use "+ 0" for no bonus
#
# Task          type bonus type bonus ... # Meaning & Difficulty

# IO - Depends only on executing the instructions (no logic)
get             -             * 1.25      * 1.05      * 1.05
put             -             * 1.25      * 1.05      * 1.05
# Task (any last output is function of _last_ input)
echo            -             * 1.5       * 1.25      * 1.25      # A
not             -             * 1.5       * 1.25      * 1.25      # ~A

# tasks (any last output is function of _last_ two inputs)
ggp             -             * 1.5       * 1.25      * 1.25
nand            -             * 2         * 1.25      * 1.25      # ~(A and B) - 1 nand
or_n            -             * 3         * 1.25      * 1.25      # A or ~B - 2 nands
and            -             * 3         * 1.25      * 1.25      # A and B - 2 nands
or             -             * 4         * 1.25      * 1.25      # A or B - 3 nands
and_n          -             * 4         * 1.25      * 1.25      # A and ~B - 3 nands
nor            -             * 6         * 1.25      * 1.25      # ~(A or B) - 4 nands
xor            -             * 10        * 1.25      * 1.25      # A xor B - 5 nands
equ            -             * 10        * 1.25      * 1.25      # ~(A xor B) - 5 nands

```

The ancestor, which was used in all runs was `creature.base` and has the following genotype:

```

search-f       # Calculates the size
nop-A          # Start label
nop-A
add
inc
allocate
push
nop-B
pop
nop-C
sub
nop-B          # Start copy Loop
copy
inc
if-n-equ
jump-b
nop-A          #End copy loop
divide         # Divides the child from the parent
nop-B          # End label
nop-B

```

Appendix B

The human written program with comments can be seen below.

```
# Filename.....: 080-aaaaa
# Update Output...: 272
# Is Viable.....: 1
# Repro Cycle Size: 0
# Depth to Viable.: 0

# Generation: 0          Divide-1      Divide-2      Average
# Merit.....:          51014          52339          51676.5
# Gestation Time..:          378           380           379
# Fitness.....:          134.958        137.734        136.346
# Errors.....:           0              0
# Code Size.....:           80             80
# Copied Size.....:           80             80           80
# Executed Size...:           77             79           78
# Offspring.....:          SELF          SELF

# get          12
# put          7
# echo         1
# not          1
# ggp         6
# nand        1
# or_n        1
# and         0
# or          1
# and_n       1
# nor         1
# xor         0
# equ         0

search-f #Calculates the length
nop-A   #Start label
nop-A
add
nop-C
inc
nop-C
zero
add
allocate #Allocates memory
nop-C
zero
push
nop-A
nop-B   #Beginning of copy loop
copy
inc
if-n-eq
jump-b
nop-A   #End of copy loop
nop-X
zero
zero
nop-A
zero
nop-C
get
dec
sub
put #NOT
get
zero
add
put #ECHO
get
get
nop-B
nand
put #NAND
```

```
get
get
nop-B
nand
nand
put #OR_N
get
dec
sub
get
nand
nop-C
nand
put #OR
get
dec
sub
get
nand
nop-C
nand
nop-C
zero
dec
sub #NOR
put
get
get
nop-B
nand
nand
nop-C
zero
dec
sub
put #AND_N
pop
nop-A
divide
nop-B #End label
nop-B
```

The Avida generated program with comments below, was generated under size merit method 4 and the same settings as in Appendix A. The task bonuses were modified so that only one computation of each task was rewarded:

```
# Filename.....: genebank/042-accbt
# Update Output...: 80000
# Is Viable.....: 1
# Repro Cycle Size: 0
# Depth to Viable.: 0

# Generation: 0          Divide-1      Divide-2      Average
# Merit.....:          17809          17809          17809
# Gestation Time..:          211          207            209
# Fitness.....:          84.4028         86.0338         85.2183
# Errors.....:           1              0
# Code Size.....:          42            42
# Copied Size.....:          42            42             42
# Executed Size...:          42            42             42
# Offspring.....:          SELF          SELF

# get      3
# put      8
# echo     1
# not      1
# ggp      2
# nand     1
# or_n     1
# and      0
# or       1
# and_n    1
# nor      1

if-n-equ
pop
put #ECHO
search-f #Get distance to label
nop-C #Start label
divide
shift-1 #Double the distance to get length
nop-B
push
allocate #Allocate memory
search-f
nop-C
nand
get
sub
nop-B
pop
nop-C
push
put #NOT
nop-B #Start copy loop
copy
inc
if-n-equ
jump-b
nop-A #End copy loop and End label
pop
nop-C
get
nop-B
push
nand
nand
nop-C
put #OR_N / NAND
add
put #OR / OR_N
nand
nand
put #NOR / AND_N
jump-f
nop-C
```

Appendix C

The Avida generated creatures used in this experiment are from the same set used in section 6, for detailed description see Appendix A. The “blindfolded programmer“-algorithm was seeded with `creature.base` which can be found in Appendix A. The mutation probabilities were set to the default in Avida, namely $p_c = 0.005$ and $p_i = p_d = 0.05$. After 100 000 time steps the current creature was extracted and if it could perform three or more boolean functions it was kept and analysed. The full source code of the application can be found at: <http://www.dd.chalmers.se/~f00phge/thesis/blindfolded.zip>