Evolution of Genetic Codes

Thesis by

Charles A. Ofria

In Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy

California Institute of Technology

Pasadena, California

1999

(Submitted May 13, 1999)

Abstract

In this thesis, I use analytical and computational techniques to study the development of codes in evolutionary systems. We only know of one instance of such a genetic code in the natural world: our own DNA. However, the results from my work are expected to be universally true for all evolving systems. I use mathematical models and conduct experiments with avida, a software-based research platform for the study of evolution in "digital organisms." This allows me to collect statistically powerful data over evolutionary timescales infeasible in a biological system.

In the avida system, Darwinian evolution is implemented on populations of self-replicating computer programs. A typical experiment is seeded with a single ancestor program capable only of reproduction. This ancestor gives rise to an entire population of programs, which adapt to interact with a complex environment, while developing entirely new computational capabilities. I study the process of evolution in this system, taking exact measurements on the underlying genetic codes, and performing tests that would be prohibitively difficult in biological systems.

I have focused on the following areas in studying the evolution of genetic codes:

Information Theory: I treat the process of reproduction as a noisy channel in which codes are transmitted from the parent's genome to the child. Unlike most channels, however, evolution actively selects for codes received with a higher information content, even if this increased information was introduced via noise. A genetic code consists of information about the environment surrounding the organism. As a population adapts, this information increases, and can be approximated through measuring the reduction of per-nucleotide entropy - in effect sites freeze in place as they code for useful functionality. In the avida system, we know the sequence of all genomes in the population, and new computational genes can be identified as they are formed.

The Evolution of Genetic Organization: Organisms incapable of error cor-

rection (such as viruses) develop strong code compaction techniques to minimize their target area for mutations, the most prominent of which is overlapping genes. Higher organisms, however, are capable of reducing their mutational load and will explicitly spread out their code, cleanly segregating their genes. I investigate the pressures behind overlapping or segregation of genes, and demonstrate that overlaps have a side effect of drastically reducing the probability of *neutral* mutations within a gene, and hence hindering continued adaptation. Further, in a changing environment, overlapping genes have a significantly reduced ability to adapt independently. I compare overlapping and singly expressed sections of code in avida, and show a significant (two-fold) difference in the average per-site variation. I also demonstrate the evolutionary pressure for organisms to segregate their genes in a fluctuating environment to improve their adaptive abilities.

Evolving Computer Programs: I explore evolution in *digital* genetic codes, and isolate some of those features of a programming language that promote continuous adaptation. In the biological world evolution gives rise to complex organisms robust to changing situations in their environment. This increase in complexity and "functionality" of the organisms typically generates more stable systems. On the other hand, as computer programs gain complexity, they only become more fragile. If two programs interact in a way not explicitly designed, the results are neither predictable nor reliable. In fact, computer programs often fail even when put to the use for which they were explicitly intended. Computational organisms, however, have a level of robustness more akin to their biological counterparts, not only performing computations, but often doing so in a manner beyond the efficiency that a human programmer could produce.

Finally, all of this work is tied together, and future directions for its continuation are explored.

Acknowledgements

Thank you to Chris Adami, my advisor and mentor, for his time, his criticism, his encouragement, and his remarkable insights as he guided me in developing an effective research methodology. I also wish to thank Alan Barr; he has provided me with space and resources for my investigations and taken an active role in my development as a researcher.

I'd like to express my gratefulness to the members of my thesis committee for their guidance and helpful discussions: Chris Adami, Alan Barr, Yaser Abu-Mostafa, Jehoshua Bruck, and John Allman. They have each provided me with strong input on my research direction.

A multitude of people have profoundly influenced my research. Travis Collier has been involved at some level in every one of my projects, much to their betterment. Titus Brown co-designed and implemented the original version of the avida system, contributing his remarkable skills and innovations. Richard Lenski has been a fountain of ideas and information, and has opened my eyes to new potential for my work in understanding biological systems.

On a personal note, I want to express my deep gratitude to Charles and Alice Ofria, my parents, for their guidance and cultivation of my love of science. My heartfelt appreciation also goes to Amy Forth for her help, especially during the late nights when deadlines were fast approaching.

Many other people have had a significant impact on my work. My thanks go out to Dennis Adler, Johan Chu, Martijn Faasen, Mike Haggerty, Grace Hsu, Rob Schwartz and Sen Song for fruitful discussions or collaborations. My thanks also go to members of the Graphics Group who have provided a wonderful working environment and solid feedback: Matt Avalos, Cindy Ball, Alan Barr, Maret Bower, David Breen, Dan Fain, Dave Felt, Kurt Fleischer, Louise Foucher, David Laidlaw, Mark Montague, Preston Pfarner, and Erik Winfree.

My fond appreciation to Professor Peter Henderson of SUNY Stony Brook who aided me in starting on my research path as an undergraduate, and to Professors Steven Skeina and Alan Tucker who tapped my creativity and encouraged me to make frequent use of it. Finally, a sincere thank you to Professor Gerry Brown who provided me with the encouragement and support to come to Caltech.

Microsoft and the NSF supported my work through grants. My stipend was paid for by an NSF training grant. Access to a Beowulf system was provided by the Center for Advanced Computing Research at the California Institute of Technology. The URECA program at SUNY Stony Brook provided the funds that allowed me to begin my explorations into Artificial Life. All opinions, findings, conclusions, or recommendations expressed in this document are those of the author, and do not necessarily reflect the views of the sponsoring agencies.

Preface

Portions of this thesis rely on work performed in collaboration, most notably with my advisor, Chris Adami.

Chapter 2 contains material from [54]. Chapter 3 is based on two papers being readied for publication ([10] and [52]), while Chapter 4 and the latter half of Chapter 5 (evolution of genome organization and differentiation) are based on [51] and [53]. The studies on the evolution and robustness of computer languages in the first half of Chapter 5 are described in a paper in preparation [9]. The co-authors and collaborators of the above articles have agreed that their work is included in this thesis, and are gratefully acknowledged.

The postscript of the final version of this thesis is available from my web page at `http://www.krl.caltech.edu/čharles/thesis/`. Many pages require color; those are placed online, separate from the body of the thesis. For any questions or comments, please contact me at `charles@krl.caltech.edu`.

Contents

List of Figures

List of Tables

Chapter 1 Introduction

1.1 Overview

Evolution has traditionally been a formidable subject to study due to its gradual pace in the natural world. One successful experimental method uses microscopic organisms with generational times as short as an hour, but even this approach has difficulties; it remains impossible to perform measurements with high precision, and the time-scales to see significant adaptation are still on the order of weeks, at best[1].

Recently, however, the exponential increase in computational power has allowed us to explore methods of studying these problems in a digital medium—through the use of populations of self-replicating computer programs. These "digital organisms" are limited in speed only by the computers used, with generations in a typical trial taking a few seconds. These systems allow us to study the dynamics of evolution in a medium where we have full control over the environment and progress of the population.

Perhaps more importantly, artificial evolving systems give us the ability to explore evolutionary principles that are no longer at work in natural systems. Life on Earth is the product of approximately four billion years of evolution, with the vast majority of beginning and intermediate states lost to us forever. The exact details of how we evolved to become what we are may be impossible to know for sure, but what we can do is better understand the evolutionary pressures exerted on life, and from that reconstruct sections of the path our evolution is likely to have taken.

Here I look at fundamental issues to living systems as we know them focusing on the evolution of life's underlying program: the genetic code.

[1]Populations of *E.coli* introduced into new environments begin adaptation immediately, with significant results apparent in a few weeks [38]. Observable evolution in most organisms occurs on time scales of at least years [39].

With a computer reconstruction of a living system, we can gather data with perfect accuracy that would be very difficult or impossible to collect in its purely biological counterpart. Traditionally in a complex system, individual pieces are studied independently, in isolation before a full analysis of the collective behavior is attempted. However, this approach does not work as well in a biological medium—normally, a disassembled living organism is no longer alive. Likewise, a disassembled ecosystem results in each component being unstable and rapidly dying out. Therefore, we must look at life as a whole, but this historically has prevented us from getting much of the detail we need. These difficulties can be overcome by using a computational system. The observation of such a system does *not* bare any effect back on the organisms within it.

In this thesis, I concentrate on an information-theoretic perspective to understand the process of evolution of genetic codes. I analyze the process of reproduction as a noisy channel wherein information is transfered from parent to child. This allows us to further analyze the coding scheme in order to determine a measure of information content of the codes and, in turn, an intuitive metric for the complexity of the resulting organisms.

This framework allows us to answer specific questions about the structure of our own genetic codes. One such question that I tackle here is "Why are genes segregated in our DNA?" In all higher forms of life on Earth, genes have a distinct promoter sequence where their expression begins, a stop codon where it ends, and their entire sequence occupies a *unique* portion of the genome. On the other hand, it is quite common for viruses and some bacteria to have *overlapping* genes where a particular section of the genome provides code for more than one product. I look at both the information-theoretic reasons that such overlapping expression patterns are advantageous (they allow for shorter genomes, and therefore a smaller target for mutations and less code to be copied), and then explain why genetic segregation does occur in higher organisms to the point where only a handful of cases of overlapping genes have ever been found.

Finally, I apply this study of the emergence of complexity in evolution back to

Computer Science. Today, it is already common for computer programs to contain millions of lines of source code, interacting with other programs to form something akin to an ecosystem. These programs are expected to be able to interact independently; while in a biological system this is gracefully handled, this level of computational complexity is already proving difficult for humans to deal with. Programs of great length are not fully testable, and interactions with other software that are not explicitly planned for are unpredictable. I consider the elements of computer languages conducive to evolution and I study the impact of these design choices. I then look at the benefits of such evolvable languages in developing highly optimized and robust code. In the end, I consider other techniques of natural evolution and I make initial attempts at harnessing them for computer science.

In the remainder of this chapter, I provide a background for computational models of life, and further discuss the framework behind it. In Chapter 2, I talk about the details of avida, the primary software that I have developed and used for this research.

I apply information theory to these models of living systems in Chapter 3, with explicit results from avida, and the insights they provide into evolution. In Chapter 4, I discuss the organization of genetic codes, using principles from information theory to understand the ramifications of structures used, and I link this into actual biological experiments. Next, I tie in the applications to Computer Science in Chapter 5. Finally, in Chapter 6, I pull all of these ideas together discussing contributions and open directions where future work can continue. I close this thesis with a glossary, placing the definitions of all of the common terms and variables in one location.

1.2 Computational Models of Life

The most common application of evolutionary principles to computer science is the "Genetic Algorithm." A variety of variant strategies do exist, such as "Genetic Programming" or "Evolutionary Computation," but they all boil down to a similar recipe; that is:

1. Create a population of random solutions.

2. Evaluate all of the solutions, assigning each a "fitness" that represents its quality.

3. Select a subset of solutions with the maximal fitness, and remove all of the rest.

4. Refill the population with variations (or re-combinations) of the selected solutions.

5. Repeat from step 2 until a desirable solution is found.

This works well for some categories of problems, but lacks certain essential elements of natural evolution.

The models (both mathematical and instantiated in software) that I use in this thesis are *auto-adaptive* genetic systems [1] differing in key areas from traditional genetic algorithms, to encompass more of the features of natural evolution. Solutions in these systems take on the role that biological organisms do in nature, thus I shall use the term "digital organism" to describe them. Indeed, in most of these systems the digital organisms are self-replicating computer programs written in a computationally universal language, which theoretically allows them to adapt to calculate any computable function and can replicate through the execution of their own code.

A prominent feature of auto-adaptive systems is that there is no explicit selection of organisms that places them in the next generation. These organisms only have the ability to self-replicate, relying on their *offspring* to preserve their genetic information. Thus, in addition to the traditional fitness that determines an organism's rate of replication, an implicitly selected criterion is for the organism to optimally transmit its information into the genome of its children. As in the natural world, this pressure does not require offspring to be exact duplicates of their parent as long as the critical information is transmitted faithfully. In fact, a greater tolerance to variation can actually improve the ability for a species to adapt. The details of this genome-transmission process are explored in Chapter 3.

This slight shift in selection pressure from GAs to auto-adaptive systems has major ramifications on the system. The replication process in digital organisms will adapt

to be maximally robust to mutations. That is, the programs become less rigid, and more variation is introduced into the population among *high fitness* organisms. In a traditional genetic algorithm, the quality of the solution is the only selection pressure, and quite often this leads to the genetic encodings becoming trapped at fragile local maxima causing evolution to grind to a halt.

The avida system takes the "realism" of these algorithms several steps further. Avida comprises a population of self-replicating computer programs that have a computationally universal genetic basis, and a collection of tools used to study them. These digital organisms extend their code length, copy their genome into this extra space, and then divide off a child replacing the oldest organism in the local neighborhood. As the maximum size of the population is fixed, the birth of any organism equates to the death of another, and on the average each organism will place only one offspring into the next generation. In practice, a significant portion of the population is fatally mutated such that many programs that remain viable will produce a second copy of their genome. However, as this copying process is subject to mutations, those genomes less likely to break when mutated are *implicitly* selected for, and no organism is guaranteed accurate transmission into the next generation no matter its "fitness." In a limited (but very real) sense, the digital organisms can be considered to be alive.

1.3 Background

1.3.1 The Coreworld System

Work on constructing self-replicating computer programs that exhibit real evolution was first popularized in 1990 when Steen Rasmussen released work performed on the coreworld simulator [55]. This program was based off of the computer game Corewars in which competitors write computer programs in a simplified assembly language called "Redcode" [20], modeled on a subset of the Intel-i860 instruction set. In Corewars, these programs are placed into the memory space of a virtual computer, alternating execution, and the last program to remain active "wins."

Rasmussen introduced a single program into the virtual core capable only of copying its own genome into other portions of memory. The replication began immediately, and the memory space filled with identical programs. Next, mutations were applied to this copy process to determine if the digital organisms could be evolved or otherwise had any ability to adapt. Unfortunately, however, this portion of the experiment failed; not only did no evolution occur, but those programs that had been capable of self-replication were copied over with fragments of fatally mutated code, and the system as a whole collapsed into a "non-living" state. The dynamics of this system still turned out to be intriguing, displaying the partial replication of fragments of code, and repeated occurrences of simple patterns.

1.3.2 Tierra

In 1991, Thomas Ray at the University of Delaware decided to design a program of his own, with significant, biologically-inspired modifications. The result is the tierra system [56], the first software to successfully encompass evolution with computer programs. The changes introduced into the system were:

- A simple assembly language was used, containing 32 instructions that had *no* arguments. Instead, these instructions act on only the current state of the CPU.
- A template-based addressing scheme allows portions of the code to be identified. Two special no-operation (`nop`) instructions were introduced, used to form patterns (*templates* or *labels*) in the program. All instructions that needed to find other portions of code (e.g., the `jump` instruction) perform a pattern matching on sequences of these nops.
- The organisms were *write-protected.* Programs could not arbitrarily write over the genomes of each other; they could only be removed as a whole when memory had filled up.

These three differences not only allowed evolution to occur, but caused it to develop to very interesting outcomes.

The first tierra experiment was initialized with an ancestor program that was 80 lines of code. It filled up the allocated memory with copies of itself, many of which had mutations that typically caused loss of functionality. Yet, enough of these mutations were neutral and did not affect the organism's ability to replicate—and a few were even beneficial. In this initial experiment, the only selective pressure on the population was for increasing the rate of reproduction—in this case the minimization of the number of instructions per program (i.e., the gestation time).

As the experiment progressed, Ray witnessed a shrinking in the average length of the programs. Shorter programs had less to copy and could therefore do so in less time. Indeed, the size continued to shrink down to 60 lines, which seemed to be the minimum size. But this was not the end of evolution: after a long stasis, a 45 line program appeared and propagated itself through the population; such a condensed program seemed too small to be able to exist, and certainly to develop so suddenly. Oddly enough, however, it was not able to dominate the population—the length 60 programs persisted as well.

Upon further investigation, Ray determined that this new program acted as a parasite on the longer one. As a parasite, it was able to calculate its own beginning location, end, and allocate its own space for a child, but when it came time to copy its code to form the child's genome, it would jump its execution into a working host.

In time, the hosts developed immunity to the parasites. They altered the templates within themselves that the parasites were using to jump their execution to, and forced the parasites into extinction. Periodically, new parasites would arise, the hosts would adapt, and again they would die out. This changed later in the experiment when a host evolved that, rather than adapting to avoid the parasites, had moved the location of the template being targeted. Now, when the parasites tried to execute the copy procedure in these hosts, they would instead be forced to re-calculate size and location information—this time of the host organism. In effect, this forced the parasite to actually make copies of the host's own genome. These organisms were dubbed "hyper-parasites."

This interplay continued for some time, and developed many other types of organisms as part of a primitive, digital ecosystem.

In 1992, about a year after the release of this initial study of the tierra system, Chris Adami at the California Institute of Technology began experiments of his own with tierra, both to study the underlying process of evolution, and to perform "animal husbandry" (directed evolution) on these digital organisms to have them evolve specific computational abilities. Three main changes were applied to the system toward this goal:

- The amount of execution time (time-slice) given to an organism was set to be proportional to the length of the genome of that organism. This reduced the selective pressure to shrink, allowing some "junk code" to exist in the genome, from which these computations would arise. Unexecuted code was discouraged by multiplying alloted CPU time by the leanness (fraction of code executed) of the organism.

- The population size was limited to a power of 2, to allow masking for calculating integer numbers modulus population size (as was often necessary). This led to a three-fold speed increase.

- The input and output (`get` and `put`) instructions were modified to monitor those numbers being manipulated. If an output was a desired computation of inputs, a record would be made of this.

- When an organism divides, the time-slice of both it and its offspring was recalculated with a bonus applied for each computation completed. Thus, if all computations are performed each gestation cycle, then this lineage of organisms will always have their execution rates increased to reflect their computational capabilities.

The actual code to perform a task is never considered; only the inputs and output are used. This allows an organism to develop the desired computation by any method that works, not applying any preconceived limitations that the researcher might have.

Four tasks were rewarded in this system. When an organism would get any number at all, when it put out any number, when it actually connected the two to "echo" an input to an output, and finally when two inputs were taken, added together, and the sum was returned. In a short time, the population evolved to learn all of these tasks.

Initially, Adami studied the dynamics of learning events in these systems [1], analyzed self-organized critical effects [2], and began to formulate a framework to approach the evolution of living systems from a statistical mechanics standpoint [3], but found it to be difficult to make necessary modifications to the tierra system, and to gain the required accuracy on the data being collected. To that end, he decided to begin a project of his own.

1.3.3 The Avida Platform

In the summer of 1993, C. Titus Brown and I joined with Chris Adami to develop the avida platform.

The primary differences in avida from tierra are:

- A simulated parallel execution of all digital organisms. Each CPU has an associated value called *merit* that determines the exact rate at which it should be processed relative to all other CPUs in the population, thus removing the effects caused by organisms receiving large blocks of time as their time slice, and giving an arbitrary resolution of detail to this measure.

- A localized environment for the organisms. Each program exists on a point of a lattice, and can only affect its immediate neighbors.

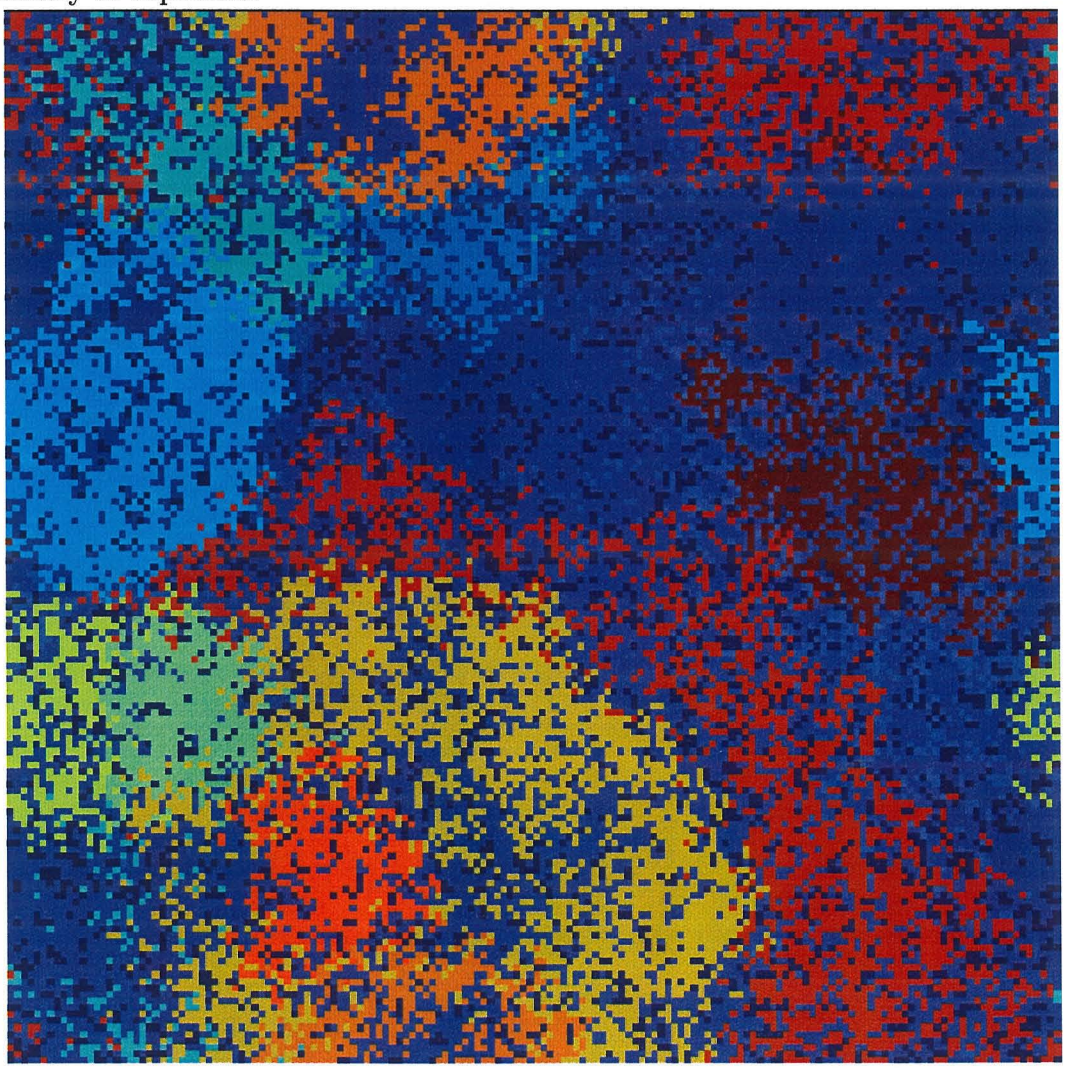
- High precision measurements that record features of the population exactly.

- A flexible interface, allowing for the easy modification of tasks, instruction sets, and other components involved in the population dynamics.

Figure 1.1 displays 25,600 avida genomes in their "artificial Petri dish." Avida is

detailed in Chapter 2. See the avida technical manual [54] for information on the use of the software.

Figure 1.1: The lattice from a typical avida experiment. Colors represent specific genotypes. Light blue displays those genotypes that were not abundant enough to warrant a unique color and dark blue signifies organisms that have not demonstrated an ability to replicate.



1.4 Artificial Chemistry

The metaphor underlying most of the work to date on adapting computer programs is that of an *artificial chemistry* [4]. In living organisms, chemical reactions provide the energy source that operates sensors, switches, signal transducers, and actuators

Such ordered, causal, functions may quite generally be viewed as *computation.* In turn, these chemicals are coded for in the DNA of the organism: the "program" of the computer. In the software systems that explore self-replicating computer code, from Rasmussen's coreworld through Ray's tierra to our avida, this analogy is taken seriously: the execution of computer code, based on low-level assembly-like instruction sets, is viewed as an analog of chemical reactions.

The idea behind these systems is to explore the complexity that the chemistry of self-replication has introduced, without actually simulating chemical reactions. Rather, the chemistry of the molecules coded for in the DNA is replaced by the execution of strings of instructions. Chemical reactions in biological organisms are evolved to make use of resources available in their environment to provide additional energy to aid in propagation of genetic information. In avida, CPU time takes the role of this energy, and computational "reactions" are evolved to acquire more. Replication and execution of instructions costs CPU time, thus consuming energy.

The accomplishment of a computation on numbers (found in the programs' environment) may be viewed as the catalysis of an exothermic reaction, benefiting the organism that carries the "gene" for this computation by speeding up its metabolism. As in real chemistry, it is problematic to specify which sequence of instructions is beneficial; rather, we construct the environment by rewarding *effects.* Any reaction that produces the desired outcome is positively reinforced with a CPU-speed increase.

The experiments that I report on here were performed with an instruction set that is computationally universal (ensuring that the system is not limited in the types of computations it can give rise to). The instruction set of biochemistry (the twenty amino acids) also appears to be computationally universal based on the observation that certain higher organisms display the capability to calculate (albeit to a finite extent) arbitrary functions.

Chapter 2 The Experimental Testbed

This chapter is a detailed description of the avida platform and related tools used for the research in this thesis. It is structured such that readers can skip it and still gain from the rest of the work presented, but it is strongly recommended to understand the intricacies of the system and design decisions that may be non-intuitive.

2.1 The Avida Platform

The computer program avida is an auto-adaptive genetic system designed as a platform for research in biological and computational evolution. The avida system consists of a population of self-reproducing strings with a Turing-complete genetic basis subjected to a variety of mutations. The population adapts in a biological manner, both to maximize its replication rate, and to beneficially interact with its environment. By studying this system, we can examine evolutionary adaptation and general traits of living systems (such as self-organization), and apply these concepts to evolving computer code.

Overview

The avida system creates an artificial environment inside of a computer. The system implements a toroidal lattice of virtual processors that execute a simplified assembly language; programs are stored as sequential strings of instructions in the individual memory of each processor. Every program string (typically termed *genome*) is associated with a processor, collectively referred to as a *digital organism.* Active lattice sites are *alive*, and inactive (or empty) lattice sites are *dead.* The maximum population size of these organisms is given by the dimensions of the lattice, $N \times M$. For purposes of research into evolving systems, the assembly language used must support self-reproduction; the default assembly language instructions, as well as several

specialized instructions for specific projects, are described in Section 2.4.

The virtual environment is initialized with a human-designed program that self-replicates. This program and its descendents are subjected to random mutations that change instructions within their memory, resulting in negative, neutral, and positive variations. Mutations are qualified in a strictly Darwinian sense: any mutation that results in an increased ability to reproduce in the given environment is considered positive. While it is clear that the vast majority of mutations will be negative—typically causing the organism to fail to reproduce entirely—or else neutral, those few that are *favorable* will result in organisms that reproduce more effectively and thus thrive. Mutations in avida are described in Section 2.5.

Over time, organisms that are better suited to the environment evolve from the initial (*ancestor*) program. All that remains is the specification of an environment such that specific tasks not directly useful to self-reproduction are selected for. This is achieved by altering the *time slice*, or amount of time apportioned to each processor, and is described in Section 2.2.

While avida is clearly a genetic algorithm (GA) variation (to which all evolutionary systems with a genetic coding can be reduced), the presence of a computationally (Turing) complete genetic basis differentiates it from traditional genetic algorithms. In addition, selection in avida more closely resembles natural selection than most GA mechanisms; this is a result of the implicit (and dynamic) co-evolutionary fitness landscape automatically created by the reproductive requirement. This co-evolutionary pressure classifies avida as an *auto-adaptive* system, as opposed to standard genetic algorithm (or *adaptive*) systems, in which the organisms have no interaction with each other. Overall, avida is an evolutionary system that is easy to study quantitatively yet maintains the hallmark complexity of living systems, as described in Chapter 3.

For all of the experiments presented here, co-evolution has been minimized, both by providing only a single niche, and by disallowing any direct interactions between organisms (such as parasitism). The reason for this is so that we can first understand the simplest of living systems before we attempt to unfold the emergent properties when multiple niches interact.

2.2 Time Slicing in an Artificial Chemistry

Time slicing is the method by which the organisms in an avida population are allocated CPU time to execute their code. Some of these organisms have faster CPUs than others, so the time slicing algorithm must make sure they are executed at the appropriate (relative) rates. The CPUs can be run either synchronously or asynchronously, closely approximating actual parallel machines. The closeness of this approximation depends largely on the *granularity*, or simplicity, of the instruction set. If a single instruction has a large effect on the surrounding environment, it will be correspondingly harder to approximate parallel execution using an asynchronous update technique. Each organism has a *merit* that determines the speed of its CPU; avida will adjust this merit as specified tasks are performed.

2.2.1 Time Slicing

The mechanism by which portions (or *slices*) of CPU time are distributed to the individual organisms significantly influences the global behavior of the population; here I examine these effects.

Two considerations go into the allocation of CPU time to the digital organisms in avida: the external bonus structure and the underlying system of CPU-time distribution that describe the low-level aspects of the virtual chemistry we are constructing. To define the time slicer, we must first decide how much time an organism should be "worth" *by default* (i.e., before the outcome of the string's execution is considered).

A simple choice would be to give all strings a constant time slice regardless of its features (most notably, its length). This is the primary mechanism used in the tierra system. With such a choice, each organism will attempt to minimize the length of its genome by shedding superfluous instructions, since gestation time is roughly linear in the length of the genome in such cases. The advantage gained by shrinking the code can be so dramatic, however, that programs often shed sections of code that trigger moderate bonuses. Such a method provides for efficient optimization, but discourages the evolution of complex code by magnifying the barrier to neighboring local optima

in the fitness landscape. As far as the structure of the fitness landscape for the strings is concerned, such a slicer increases the local slopes and thus accelerates convergence to a local energy optimum while reducing ergodicity.

Another possibility is to distribute CPU time in a manner proportional to the length of the code. This is the *size-neutral* regime also available in tierra. The resulting fitness landscape is intuitively smoother; strings that behave in the same way but differ in length of code are degenerate as far as their replication rate is concerned and far-lying regions in genotype-space can be accessed easily. Clearly this mechanism is more conducive to the evolution of complexity. However, it has a certain disadvantage from a practical perspective, as the instruction set provides the possibility to *jump over* sections of code, leaving them unexecuted. The organisms soon discover that they can "earn" CPU time by developing code that is neither executed, nor accurately copied into their offspring. This does not exist in real chemistry, as even DNA that is not expressed still participates in chemical interactions.

I developed a mechanism (similar to the one used by Adami in tierra) that counts only those instructions that both are copied into the genome and are executed by the organism's CPU, when evaluating an organism's effective length. Under these conditions, *lean* programs are favored over those that carry sections of unexecuted, uncopied code.

The slicers discussed here can be distinguished by a simple formula that describes the mechanism. Specifically, the time doled out (allocated) to each organism *a priori* is proportional to that organism's *merit*, where merit is determined by the effective genome length times any bonuses given to the organism through its interaction with the environment. This multiplication (as compared to the method of *addition* of bonuses used in previous incarnations of avida and in tierra) serves to ensure that there is no size bias in evolution. In the case where bonuses are additive, they soon overshadow the size component of merit thus returning to the constant time-slice paradigm.

2.2.2 Carving a Landscape

Since the time slicer defines the landscape (and thus the "physics" and artificial chemistry) associated with self-replication, we can superimpose any landscape we deem interesting. This is done by specifying bonus CPU time associated with the *phenotype* of the string. By rewarding actions rather than a particular sequence of commands within a genotype, we introduce the possibility for open-ended evolution. As the set of possible strings that have the same phenotype is effectively infinite (assuming no bounds are put on the length of strings), it is impossible to construct a string with maximum fitness given a sufficiently complex environment. The complexity of the landscape (here identified roughly as proportional to the number of distinct local optima) increases exponentially with the number of bonuses specified, as they can in principle be triggered simultaneously and in any order (often integrated so precisely that the same section of code will make progress on multiple tasks at once). For example, consider the landscape constructed such that the adapted population reflects a phenotype capable of adding integer numbers.

As a first step, any form of input or output is rewarded. Next, a connection between inputs and outputs is rewarded: the capability to *echo* numbers from the input, directly to the output. Finally, if the organism writes into the output a number that is the sum of two previously read inputs, the string is rewarded with another bonus. Each of these rewards can be triggered multiple times each gestation period (typically to a maximum of three). By default, the bonuses multiply the organism's merit by a specified factor the first time they are triggered, while only multiplying them by a smaller fixed factor (1.05) thereafter. This is both to encourage diversity in ability, and because it is typically easier to perform a task multiple times than it was to learn it initially.

How such a bonus structure *carves* a landscape in the space of all fitness improvements becomes obvious (or at least intuitive) if we analyze the population shortly after it has adapted to the *echo* bonus. At that point, strings that are mutations of the wild-type write all sorts of numbers, obtained via random manipulations, into the

output. Among those we find sums, differences and multiples of the input numbers. The gene for addition is simply filtered out by rewarding that particular task out of all those currently being performed (no matter how poorly), and then optimized through time. Any other task can be filtered in the same manner. Quite literally, rewarding addition creates a "fitness valley" that only those organisms with the appropriate gene can occupy. Since organisms in the lower regions of the landscape obtain more offspring than those higher up, they soon dominate the population and drive strings missing the gene into extinction. Once this is achieved, the adapted population spreads in diversity via the effects of mutation, to explore new regions of the landscape where more crevices may be found. Such a sequence of adaptations results in a fitness curve resembling a staircase, which can be a true fractal for environments of high complexity. Additionally, these learning events give rise to the propagation of information, and to Fisher waves [17] whose diffusion coefficients can be calculated exactly given the competitive advantage the new genotype received from its beneficial mutations.

Addition is an example of a particularly simple task for these organisms to perform. The default environment in avida selects for 80 tasks in all. Two of these are the trivial rewards for input and output. The remaining 78 are for the computation of bitwise logical operations, using only the nand instruction they are supplied with. The inputs they receive are 32-bit integers; for an output to trigger a bonus, all 32 bits must consistently represent a single logical function. There are 2 tasks rewarded that require a single input (ECHO and NOT), 8 two-input tasks (AND, OR, NAND, NOR, ¬A AND B, ¬A OR B, XOR, and EQUALS) and 68 requiring three inputs.

Although nand is the only operation the organisms have in their instruction set, it is sufficient for constructing any logical function. Additionally, as evolution develops the simpler (typically one- and two-input) tasks first, they are made use of as partial results in more complex formulas.

The successful computation of any of these logical operations can be viewed as beneficial metabolic chemical reactions that speed-up the virtual CPU accordingly; more complex tasks result in larger speed-ups. If the task is performed efficiently, the

corresponding speed increase is more than the time expended to perform the task, and the net effect is an increase in the replication rate of the organism.

2.2.3 Fitness

The fitness (w) of an organism in such simple digital systems is given by the expected number of offspring it can generate in its environment. This is closely approximated by the merit $\mathcal{M}$ earned by the organism divided by the time it takes to generate an offspring (the gestation time, t_g):

$$w \approx \frac{\mathcal{M}}{t_g}. \tag{2.1}$$

This fitness measurement can be directly compared to the fitness of any other organism to determine their relative reproduction speeds. If one organism has twice the fitness of another, then it will be able to have twice the number of offspring per unit time.

In practice, there is quite a bit more to measuring the fitness of an organism than simply its expected number of offspring. The main additional factor that we must take into account is the ability for that organism to faithfully transmit its genetic information into the next generation. This analytical study of selective pressures in evolution is explored further as the topic of Section 3.6. For the moment, the above formula will be used as a good approximation of the true fitness of an organism.

Fitness improvements come in two forms: the maximization of CPU speed by task completion, and the minimization of gestation time. As all tasks must be computed each gestation cycle to maintain a reward, this gestation time minimization includes the *optimization* of task completion in addition to speed-ups in the main replication process.

2.2.4 Time Slicing Algorithms

Time slicing algorithms handle the details of population execution; they ensure that organisms are executed with simulated parallelism to minimize any advantage dependent on execution order that might occur.

This system also provides a method of time-keeping independent of lattice size: the *update*. An update is defined as the time it takes an average CPU to execute 30 instructions. The entire population is executed during each update, with the time slicer partitioning CPU executions such that the average cycles per organism remains constant. Additionally, many of the algorithms provide a natural interface to distribute the avida system across multiple processors.

There are four main time-slicing algorithms implemented in avida. They are

- **Constant**: All CPUs receive the same execution time, independent of merit. This encourages shrinking, and removes all incentive to learn environmental tasks (unless merit is taken into account in the maximum age limits of offspring—see Section 2.3). CPU time is doled out evenly such that each organism executes a single instruction before any execute their second.

- **Block**: CPUs are allocated a block of execution time such that the size of each time-slice block is proportional to the organism's merit. The CPUs are executed in sequence for their entire block, as is the case in tierra. This method is available only for compatibility of experiments between systems.

- **Probabilistic**: Instructions are executed in a semi-random fashion, such that the probability of a single organism having an instruction executed is proportional to that organism's merit. Thus, on average, each program obtains CPU time that is proportional to its merit. This method has the most realistic "feel" to it, but the random component slows adaptation, and the constant use of the random number generator makes avida as a whole run slower.

- **Integrated**: This is the default slicing method in avida, used in all experiments unless otherwise stated. It has each CPU execute a single instruction at a time

in a deterministic fashion such that the relative speeds of the individual CPUs are proportional to the merit of the corresponding organism. Effectively, this comes as close to a perfectly synchronous parallel execution as mathematically possible.

2.3 Reproduction

The process of replication dominates the dynamics of the avida system. Here, I present an overview of the method by which programs reproduce, and I discuss their implementation.

Reproduction in avida is typically performed in four distinct phases:

1. Allocation of new memory at the end of the program's code (elongation).
2. Copying of the parent genome into the new memory, instruction by instruction.
3. Division of the program into parent and child programs.
4. Placement of the child program into the lattice.

The first three steps are implemented in the instruction set (and are thus the responsibility of the individual organism), while the fourth process is automatically handled by the environment when a successful division occurs.

In a correctly replicating program (see the example program in Section 2.4), the size of the allocated memory is the program's length (doubling the total memory from its original size), with division occurring at its midpoint at the end of the copying process. In principle, there is no reason that a program could not use a different method (such as tripling its size, and making two copies of itself, or creating a self-extracting smaller program); however, the instruction set (and the handwritten ancestor) bias the products of evolution toward the first method.

Placement of offspring is done in a *localized* manner; the child-program can only be placed within the immediate neighborhood of the parent's location (the eight nearest grid positions on a 2-D lattice).

The process of placement is entirely a function of the environment; as soon as a successful division occurs, the offspring is automatically placed in a manner determined during the configuration of the experiment. If there are any empty sites available, they will always have priority. If none of the eight immediate cells connected to the parent are vacant, the following mechanisms are in place for removing one of the surrounding organisms (or the parent itself):

- **Choose Randomly:** an organism is chosen at random from the parent and its eight neighbors. This method is poor for evolution because approximately half of the organisms will be replaced before producing their first offspring, and hence will not have been properly evaluated [6].

- **Choose Eldest:** this is the default method in avida. The oldest organism in the neighborhood around the parent (including the parent itself) is removed.

- **Choose max [Age/Merit]**: this placement method favors organisms with a higher merit, and is an additional way to encourage specific tasks to be learned. Combining this birth method with the constant time-slicing scheme causes organisms to be selected for by increased life span (as opposed to enhanced CPU speed).

- **Choose Empty:** this is a limited birth method that is useful only when organism *death*[1] is activated. In this mode, organisms are prevented from killing each other, and new offspring are placed only into empty locations.

2.4 The Virtual Computer

The virtual computers implemented in avida each consist of a central processing unit (CPU) operating on a memory space of commands from a specialized *instruction set.* These components define the low-level behavior of each program; the CPU and the instruction set together form the *hardware* of a universal computer.

[1] Organisms can be assigned a maximum number of executions, which, when reached, will cause them to be removed from the population.

When a genome is loaded into the memory (as the *software*) of a CPU, the initial state of the universal computer is set. The hardware, combined with the interaction with other CPUs, then governs the set of transitions between CPU states.

2.4.1 The CPU Structure

The CPU consists of the following set of components, as shown in Fig. 2.4.1.

- A *memory* that contains the assembly source code to be executed. Each location in memory contains a single instruction, and a set of flags to denote if the instruction has ever been executed, copied, mutated, etc. This memory is associated with an *instruction pointer* (IP) that indicates the next position to be executed.
- Three *registers* that contain arbitrary 32-bit values and are operated upon by most instructions.
- Two *stacks* that are used for storage. These are of variable (though finite) size. The default limit on stack size is 10 numbers.
- An *input buffer* and an *output buffer* used by the organisms to receive information, and return the processed results to the environment.
- A *facing* that determines which of the CPU's neighbors it is currently pointing toward. Those instructions that require interaction with other organisms use this facing to determine which neighbor to affect.

The memory of each CPU is *circular*, as is the genome in most bacteria and viruses. As a consequence, the instruction pointer never leaves the local organism unless it is forced to by explicit command (see the `jump-p` command below). This has a consequence of making parasitism more difficult to develop, but does add a higher level of biological realism to the process.

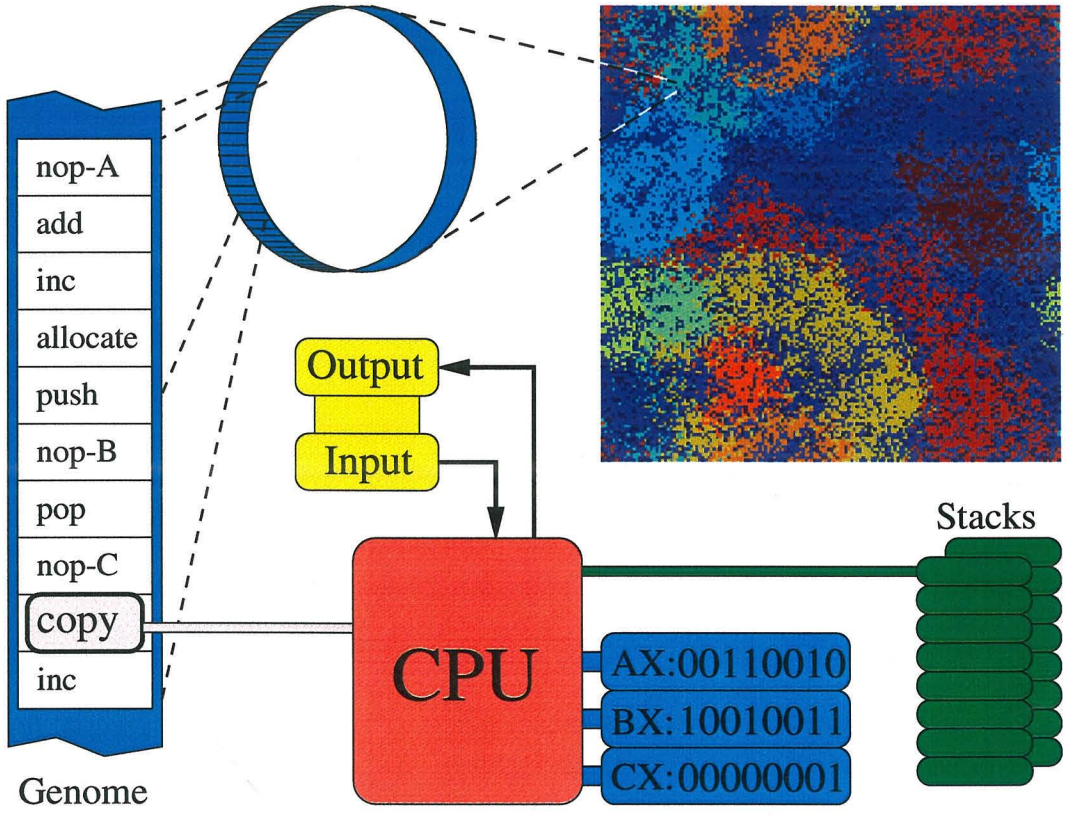


Figure 2.1: Structure of the virtual CPU in avida. The CPU operates on three registers (cyan), two stacks (green), and an instruction pointer (pink). Input and output from and to the environment is achieved via dedicated I/O buffers (yellow).

2.4.2 The Instruction Set Implementation

The instruction set in avida is loaded on startup from a configuration file allowing selection of different instructions without recompilation. These sets were designed with three goals in mind:

- To be as complete as possible; both in a Turing complete sense, and, more practically, to ensure that simple operations require only a few instructions to perform.

- For each instruction to be as robust and versatile as possible in all situations. Instructions should take an appropriate action any time they are executed, and gracefully handle those error conditions that do remain.

- To have minimal redundancy between instructions

One concept that differentiates this assembly language from its real-world counterparts is in the unconventional uses of nops (no-operation commands). They have no effect on the CPU when executed, but may modify the behavior of an instruction that *precedes* them. This occurs in two ways; most of the time it will change the register affected by a command. For example, an `inc` command followed by the instruction `nop-A` would increment the contents of the `AX` register, while an `inc` command followed by a `nop-B` would increment `BX`.

Below, whenever a register name is surrounded by ?'s in an instruction description, it is the default register to be used. If a `nop` follows the command, the register it represents will replace this default.

The second way nops can be used is as labels (reference points, or templates) for a search or a jump as in tierra. If `nop-A` follows a jump-forward command, it scans forward for the first complementary label (`nop-B`) and moves the instruction pointer there. Labels may be composed of more than a single `nop` instruction.

The label system used in avida allows for an arbitrary number of different nops. By default, we have three `nop` instructions, `nop-A`'s complement is `nop-B`, `nop-B`'s is `nop-C`, and `nop-C`'s is `nop-A`.

Below is a description of all of the 28 default instructions implemented in avida, followed by a limited collection of instructions used in specific projects. Hundreds of instructions are implemented from which the researcher may choose to assemble an appropriate set for their project, but here I only present those that will be used in work related to this thesis.

No-operations

There are three nops in the default instruction set:

1-3 `nop-A`, `nop-B`, and `nop-C`

Flow control operations

4 `if-n-equ` : If the ?BX? register does not equal its complement register, execute the next instruction, otherwise skip it. (Thus a `nop-A` following this command causes `AX` and `BX` to be compared; `nop-B`—the default—compares `BX` and `CX`, and finally, a `nop-C` compares `CX` and `AX`).

5 `if-less` : Execute the next instruction only if the ?BX? register is less than its complement register, otherwise skip it.

6 `if-bit-1` : Execute the next instruction only if the last bit of ?BX? is one.

7-8 `jump-b` and `jump-f` : If a label follows, search for its complement in the backwards/forwards direction; if a match is found, jump to it. If there is no label, jump by `BX` instructions in the proper direction. If there is a label, but its complement is not found, the jump will fail.

9 `call` : Put the location of the next instruction on the active stack, and jump forward to the complement of the label which follows. If there is no label, jump `BX` instructions.

10 `return` : Pop the top value from the active stack and jump to that index in the organism's memory.

Single argument math operations

11-12 `shift-r` and `shift-l` : Rotate the bits of the ?BX? register in the appropriate (right or left) direction.

13-14 `inc` and `dec` : Increment or decrement ?BX?.

15-16 `push` and `pop` : Push ?BX? onto the active stack or pop the stack into ?BX?.

17 `swap-stk` : Toggle the active stack to be used by instructions.

Double argument math operations

18 `add` : Set ?BX? to the sum of the BX and the CX registers : ?BX? = BX+ CX.

19 `sub` : ?BX? = BX - CX.

20 `nand` : ?BX?= BX NAND CX (in a bitwise fashion).

21 `swap` : Swap the contents of ?BX? with its complement register.

"Biological" operations

22 `allocate` : Allocate ?BX? instructions at the end of the memory for this CPU and return the start location of this memory into AX. Only one allocate may occur between successful divides; any additional ones will automatically fail. Not more than twice or less than half of the current memory size can successfully be allocated.

23 `divide` : Split the memory in this CPU at ?AX?, placing the instructions beyond the dividing point into a new site. There are a number of conditions under which a divide will fail. Those are:

(a) If either the parent or the offspring would have less than 10 instructions.

(b) If the organism has not completed a successful allocation of memory.

(c) If less than 50 percent of the parent was executed.

(d) If less than 50 percent of the offspring's memory was copied into.

(e) If the offspring would be less than half or more than double the parent's size.

24 `copy` : Copy a command from the memory location pointed to by the BX register to the memory location pointed to by AX + BX, i.e., copy the instruction at location BX into a location *offset* by AX. If a location is out of range of the memory, then it will be cycled back into range.

I/O and sensory operations

25 `get` : Place the next value from the input buffer into `?CX?`.

26 `put` : Place `?BX?` into the output buffer and set the register used to `0`. Avida will analyze this output (with respect to the values the organism has input) to determine if a merit bonus is warranted.

27-28 `search-f` and `search-b` : Search in the appropriate direction for the complement label and return its distance from the current IP position. The returned value is placed in the `BX` register, and the size of the label that followed is put in `CX`. If a complement label is not found, a distance of 0 is returned.

Additional instructions

These instructions are all used to test specific aspects of evolution, and are not in the default avida instruction set.

- `nop-X` : A pure no-operation instruction. It will do nothing when executed, and will not modify the execution of other instructions in any way. This instruction is used to test the functionality of specific portions of code (by replacing them with a `nop-X` and testing the change that has occurred) and to test the ability of organisms to withstand insertion mutations (other inserted instructions may cause side-effects that obscure the results).

- `read` : Copy a command from memory at `BX` into the `CX` register.

- `write` : Copy a command from the `CX` register into the memory location at `AX + BX`.

- `if-n-cpy` : Only execute the next line if the contents of memory locations `BX` and `AX + BX` are *identical*; otherwise skip it. This command has an error rate equal to the copy mutation rate. (It can be used to do error checking).

- `order` : Place `BX` and `CX` in the proper order, i.e., such that `CX > BX`.

- `jump-p` : Jump into the memory of another CPU, decided by the current facing. Jump to the position in the faced organism at an instruction after the first occurrence of the complement label being searched for. If no complement label can be found, this instruction fails. If no label is initially provided to the instruction, the IP (instruction pointer) will move to line `BX` in the faced CPU's memory. An organism's IP may only move to an immediate neighbor and no further (local interactions only).

- `rotate-l` and `rotate-r` : Rotate the current facing of the CPU in the appropriate direction.

2.4.3 An Example Program

Table 2.4.3 describes one of the simpler organisms distributed with avida. Due to its efficiency at self-replication, this organism is not well suited as an ancestor for adaptation experiments: it has a low degree of neutrality and will often become stuck in local optima.

This simple program contains two label pairs (α, $\bar{\alpha}$) and (β, $\bar{\beta}$), one for the purpose of calculating the genome length, and the other for the implementation of a copy loop. Execution commences as the `search-f` followed by label α searches forward in the genome for its complement $\bar{\alpha}$, and returns its distance (from the end of the first label to the end of the second) into the `BX` register, as well as the size of the label itself into `CX`. The program then adds `CX` to the `BX` register, to account for the length of the label itself, and finally increments `BX` to account for the single (`search-f`) instruction before the first label. The program now has its own length in `BX`. When the instruction on line 5 (`allocate`) is called, the memory is doubled in length and the absolute address of the new chunk of memory is put in `AX`. Now, `AX` contains the offset of the newly allocated section, and `BX` contains the length of the genome (which, if an organism is copying itself properly, will always be the same).

Lines 6 through 11 move the length of the genome (via the stack) into the `CX` register, and sets `BX` to zero.

Table 2.1: An example avida genome.

00	search-f	find distance to the end label
01	nop-A	label α
02	nop-A	
03	add	account for the end label's size
04	inc	account for the initial `search-f`
05	allocate	allocate space for offspring
06	push	move length from BX onto the stack
07	nop-B	
08	pop	move length off of the stack into CX
09	nop-C	
10	pop	since the stack is empty, pop 0 into BX
11	nop-B	label $\bar{\beta}$ (Copy Loop start)
12	nop-C	
13	copy	copy the current line...
14	inc	move onto the next line
15	if-n-equ	if we aren't done copying...
16	jump-b	...jump back to the loop's beginning
17	nop-A	label β
18	nop-B	
19	divide	done copying; separate the offspring
20	nop-B	label $\bar{\alpha}$
21	nop-B	

The copy loop follows. It starts by copying from line `BX` and uses `BX + AX` as the destination. Initially, `BX` is 0 and `AX` is the size (22). This means the first time through the copy loop, line 0 is copied to line 22 (the first line in the newly allocated memory). Then line 14 is executed and `BX` is incremented. Finally, `if-n-equ` tests to see if whether `BX` and `CX` are different, and jumps back to the beginning of the loop if this is the case. The loop will continue (copying a new line each time) until `BX` equals `CX` and hence all the lines have been copied. Finally, the `divide` instruction is reached.

The `divide` instruction partitions the memory at the offset specified in the `AX` register, creating distinct parent and child genomes. The offspring is placed in a

neighboring location (the mechanism of which is described in Section 2.3). At the end of the program, the instruction pointer is automatically looped back to the beginning.

Table 2.4.3 contains a trace of the execution of the program, with the values of the registers and stack at each moment in time.

Table 2.2: The trace of a sample genome execution.

line	**instruction**	**AX**	**BX**	**CX**	**stack**	**comments**
00	search-f	0	19	2		
03	add	0	21	2		
04	inc	0	22	2		We have genome length!
05	allocate	22	22	2		Allocate space
06	push	22	22	2	22	push `BX` onto the stack
08	pop	22	22	22		pop `CX`
10	pop	22	0	22		pop `BX`
12	nop-C	22	0	22		No-Operation
13	copy	22	0	22		Copy line 0...
14	inc	22	1	22		
15	if-n-equ	22	1	22		
16	jump-b	22	1	22		... to start of loop
13	copy	22	1	22		Copy line 1...
14	inc	22	2	22		
15	if-n-equ	22	2	22		
16	jump-b	22	2	22		...to start of loop
13	copy	22	2	22		Copy line 2...
	...					...
13	copy	22	21	22		Copy line 21..
14	inc	22	22	22		
15	if-n-equ	22	22	22		
17	nop-A	22	22	22		
18	nop-B	22	22	22		
19	divide	22	22	22		Divided!
20	nop-B	22	22	22		
21	nop-B	22	22	22		

For this program, the *gestation time* (the number of instructions required to reproduce) varies between 98 and 100 instructions. The first time through requires 98 instructions: the portion before the copy loop consists of 8 executed instructions; the copy loop contains another 4 instructions that are each executed 22 times, except for the last time the copy loop is executed, when the last `jump-b` is skipped over; from

there it is another 3 instructions until the `divide` is issued. So, $8+(4\times22-1)+3=98$. However, the second time through, an additional 2 instructions are executed because of the α label after the `divide` instruction, and a gestation time of 100, rather than 98, is averaged into the genotype record.

2.5 Mutations

Avida has a range of both explicit and implicit mutations. Five forms of explicit mutations have been implemented.

There are three intuitive methods by which mutations can be triggered in avida. The first is the cosmic-ray or *point* mutation, which is generated by an external random process independent of the action of the organism—they hit randomly chosen loci in the population at Poisson-randomly distributed times (thus, the time between point mutations is exponentially distributed, with the average time being the inverse of the mutation rate). Next there are *copy* mutations, which can occur whenever a program tries to copy a line. There is a small probability (fixed by the user) that a copy command will result in a random instruction instead of the instruction intended. Finally there are divide mutations, which occur during the birth of an offspring with a fixed probability. During a divide, a probability can also be set for a single *insertion* or *deletion* to occur at a random position in the code.

In all experiments presented in this thesis, the primary type of these explicit mutations are copy mutations. They are used most in avida experiments as they are the most prevalent in natural biological systems. Additionally, we maintain a low level of divide insertions and deletions so that genome length is not fixed. By default, the copy mutation rate in all experiments has been set to 0.0075/copy, and both divide insertions and deletions are set at 0.05/divide.

Naturally, the rates for all these mutations must be below a certain threshold to avoid killing the population, while a rate that is too low reduces variation, slowing evolution to a crawl ([4], p.265-296).

It is also important to note that each form of mutations has rates that are ex-

pressed in their own units. Copy mutations are specified *per instruction copied*, point mutations are *per-site per-update*, and divide mutations are *per-offspring*. What this means is that for the former two, longer genomes stand a higher probability of being mutated, inherently placing a selective pressure on the population to decrease genome length. On the other hand, organisms typically survive a single divide mutation better if they are longer, and therefore apply a slight pressure for genome growth.

Implicit mutations in avida involve mistakes (due to incomplete or faulty algorithms) committed in the act of self-copying, usually instigated by code that was corrupted by explicit mutations. There are a wide assortment of these, many of which have not been categorized due to the ability of life to always find new and surprising methods of operation.

A common form of implicit mutations is the duplication of code within a genome; sometimes the flow of execution will be distorted such that a section of the genome will be replicated multiple times within a single offspring. Interestingly, this has an analog in the biochemical code where repeated DNA sequences are abundant. A second (similar) form occurs when an organism only partially copies itself over a dead code fragment that previously occupied the CPU. In this manner, the two genomes (old and new) are effectively merged into a single one—a form of recombination that is disabled by default in avida. Other implicit mechanisms are possible. As a result of these, the *effective* copy-fidelity of a program can be significantly different from the one calculated with mutation rates alone.

2.6 Research with Avida

The avida system was designed to maximize flexibility, speed, and accuracy in order to be a versatile research platform. In addition to the characteristics discussed above (mutation rates, time-slicing mechanism, tasks, and instruction sets), the following features exist:

Clones: At any time during an experiment, the entire population can be stored on disk for later continuation, or to place into a variety of new environments in order

to study evolutionary dynamics from a well-equilibrated starting point.

Connectivity: The layout of the environment that the digital organisms evolve in is central to the course and dynamics of evolution. In avida, this is by default a lattice (equivalent to biological experiments using a Petri dish), but can have any other connectivity including a fully connected (well-stirred) environment.

Events: A full range of events exist in avida that can be set to occur at any time in an experiment. These events include changing most primary characteristics of an experiment (tasks, mutation rates, etc.), collection of specific data (i.e., recording the landscape of the dominant genotype, or analyzing the per-site entropy across the population), or one-time events, such as killing a fraction of the population in an "apocalyptic" event.

Gene Surgery: The researcher has complete control over the state of genomes in avida, and can either set an event or manually (via the user interface) modify the genetic code of any organism in the population.

Injection: New organisms can be introduced to an active population at any time to study the effects of invasion of specific organisms. One common experiment involves injecting parasites into a population to study the resulting effects (with parasite mode turned on).

Phylogenetic Trees: The complete phylogeny of an experiment can be recorded along with the genotypes and species of every organism that has ever existed. This allows a researcher to reconstruct a history of the genealogical relationships and understand the impact of evolutionary transitions on the structure of the phylogenetic trees.

User Interface: Avida has a detailed user interface that allows a researcher to observe the metabolic state of the average and the dominant organism, or even modify the experiment as it progresses. The lattice can be viewed to determine the geographical distribution of many features, histograms of genotype and species abundance are available, statistics can be observed, and configurations can be modified. Researchers can even monitor the CPU state of a specific organism as they step through the execution of its genome (one instruction at a time) to understand how it functions.

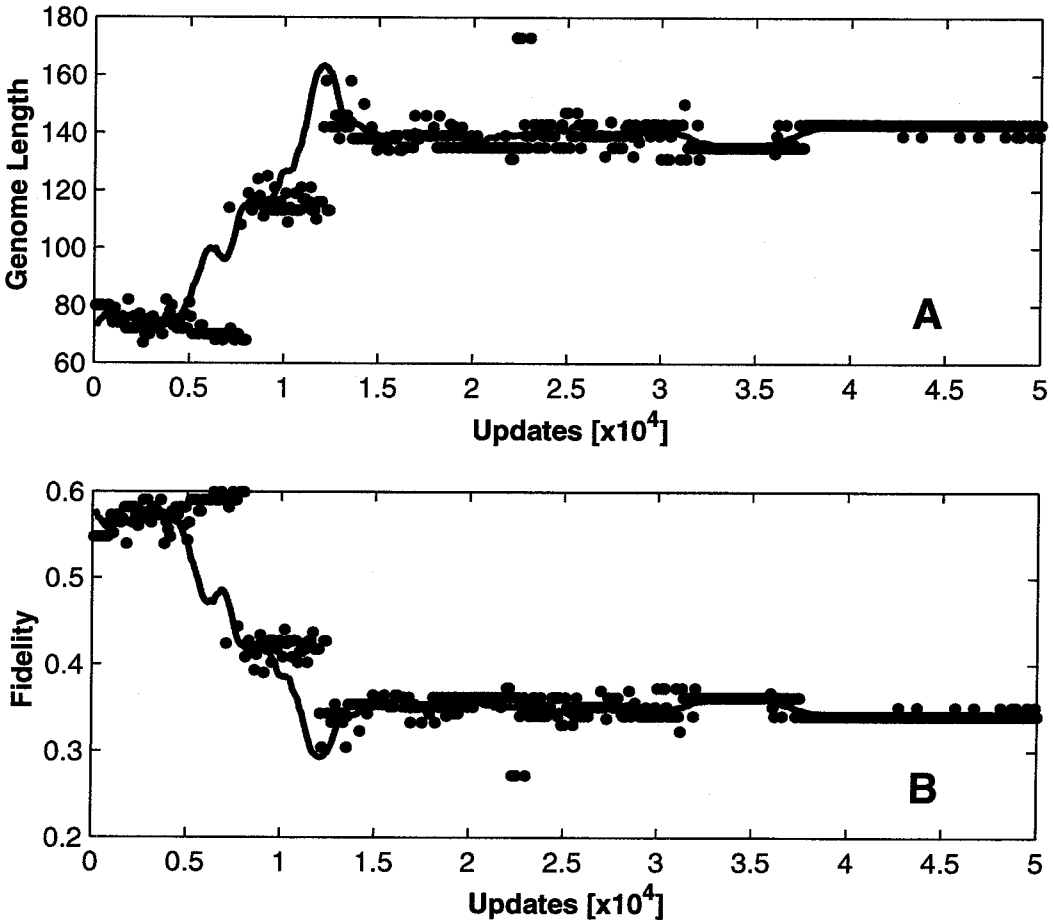
The main configuration files to set some of these conditions are described in Appendix A.

2.7 Basic Analysis Metrics

In this section, I discuss the dynamics of a typical avida experiment. There are a number of basic metrics that are used to study the course of evolution.

Genome Length (ℓ) is an obvious statistic to consider. Figure 2.2(A) displays the progression of length for both the average and dominant genotypes. As evolution incorporates new computations into the genomes, more information storage is required. As such, we witness regular increases in genome length through time.

Figure 2.2: Statistics from a typical avida experiment. Measurements for (A) Genome Length and (B) Fidelity are displayed for both the dominant (dots) and average (solid line) genotype, throughout the course of evolution over 50,000 updates.



Fidelity (F) is the probability for an organism to produce an offspring perfectly identical to itself, (i.e., the probability that the offspring is unaffected by mutations during the replication process). For a copy-mutation probability R_c per line copied,

$$F = (1 - R_c)^{\ell}. \tag{2.2}$$

If other explicit mutations are active, the fidelity can be calculated as the product of probabilities for each mutation *not* occurring. In an adapting population, other factors can affect the fidelity and lead to low-fidelity organisms even while the theoretical fidelity is high. One such example would be a sub-optimal replication algorithm. Conversely, the development of error-correction techniques could increase the actual fidelity beyond the value predicted in the above formula.

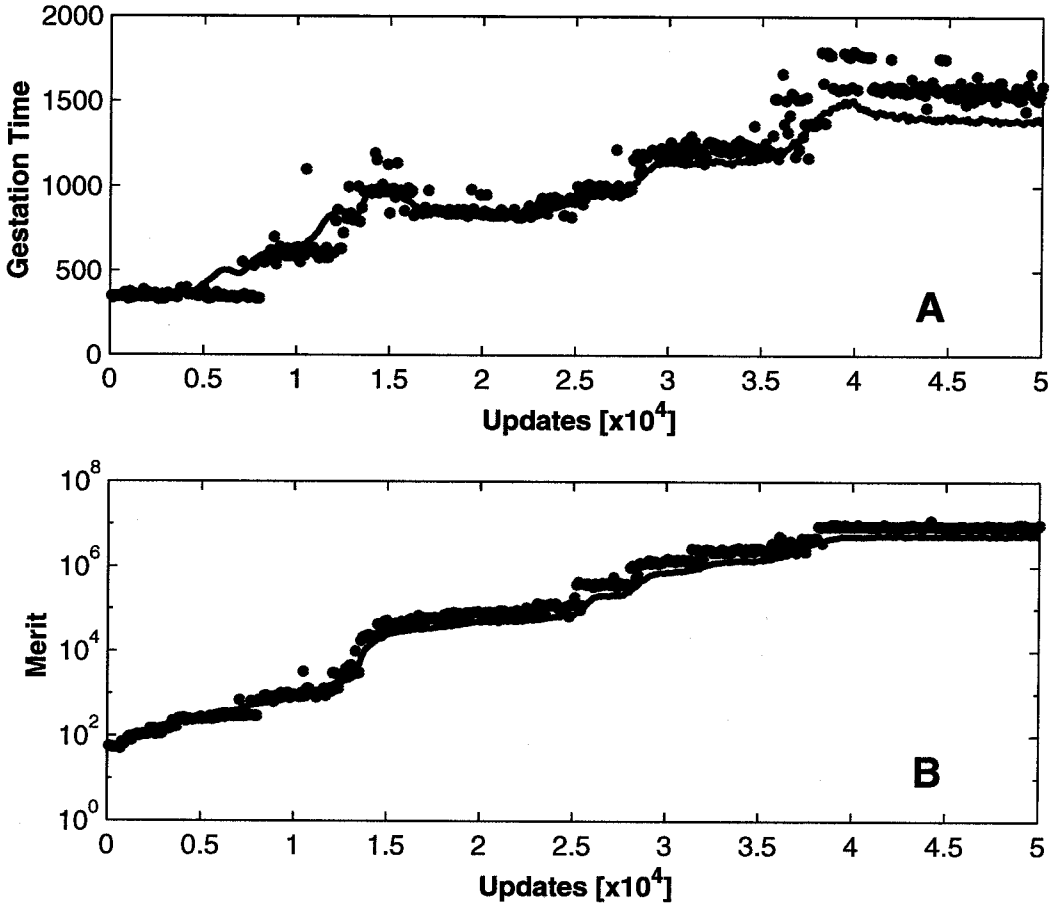
Mutation probabilities are fixed for all experiments examined here, making fidelity a function of genome length only, and thus varying accordingly in Figure 2.2(B). Longer genomes present a larger target for mutations, and are therefore more difficult to replicate faithfully.

Gestation Time (t_g) is the number of instructions an organism must execute to produce a single offspring. This counts both the execution time spent performing all computations, in addition to the replication of the actual genome. As the gestation time tends to be dominated by replication (typically taking four instructions executed to copy a single line), this statistic usually remains proportional to genome length, as demonstrated in Figure 2.3(A). Any gestation time fluctuation that is not correlated to a length change implies an algorithmic re-organization involving a shift in the number of executions of pre-existing code. This effect can be seen at updates 16,000 and 28,000 in this sample experiment.

Merit ($\mathcal{M}$) represents the relative rate at which the individual CPUs execute instructions. As bonuses increase the merit multiplicatively, merit is displayed on a logarithmic scale in Figure 2.3(B).

Note that gestation time grows over the course of evolution, despite the negative impact on replication speed. Any such gestation time increase must be compensated

Figure 2.3: Statistics from a typical avida experiment. Measurements for (A) Gestation Time and (B) Merit are displayed for both the dominant (dots) and average (line) genotype, throughout the course of evolution over 50,000 updates.

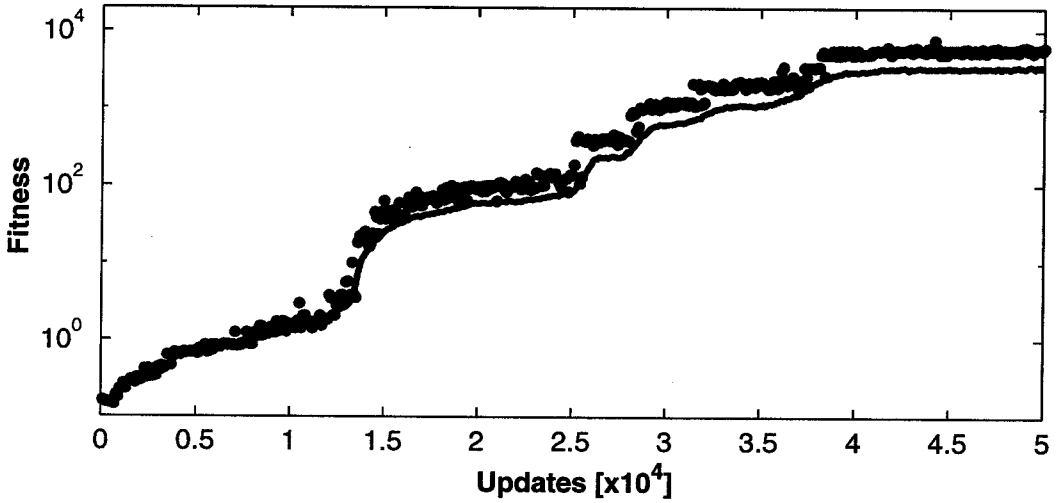


by at least a commensurate increase in merit. In practice, gestation time grows linearly while merit grows exponentially, causing the latter to dominate the adaptation dynamics.

Fitness (w), as previously described in Part 2.2.3, is the replication rate of a genome, approximately given by $w = \mathcal{M}/t_g$. Figure 2.4 shows a typical temporal history for fitness. We see both periods of gradual fitness growth (witness updates 16,000 through 25,000) as well as sharp punctuations (for example, update 25,000).

Each jump in fitness represents a major learning event, typically involving a significant change in genome structure. Often, a period follows in which these changes are fine-tuned, as an optimization of tasks is performed; each generation only producing incremental improvements. While individual trials will vary in the specific tasks that

Figure 2.4: Statistics from a typical avida experiment. Measurements for Fitness are displayed for both the dominant (dots) and average (line) genotype, throughout the course of evolution over 50,000 updates.



are learned, they all exhibit a similar pattern of fitness increases.

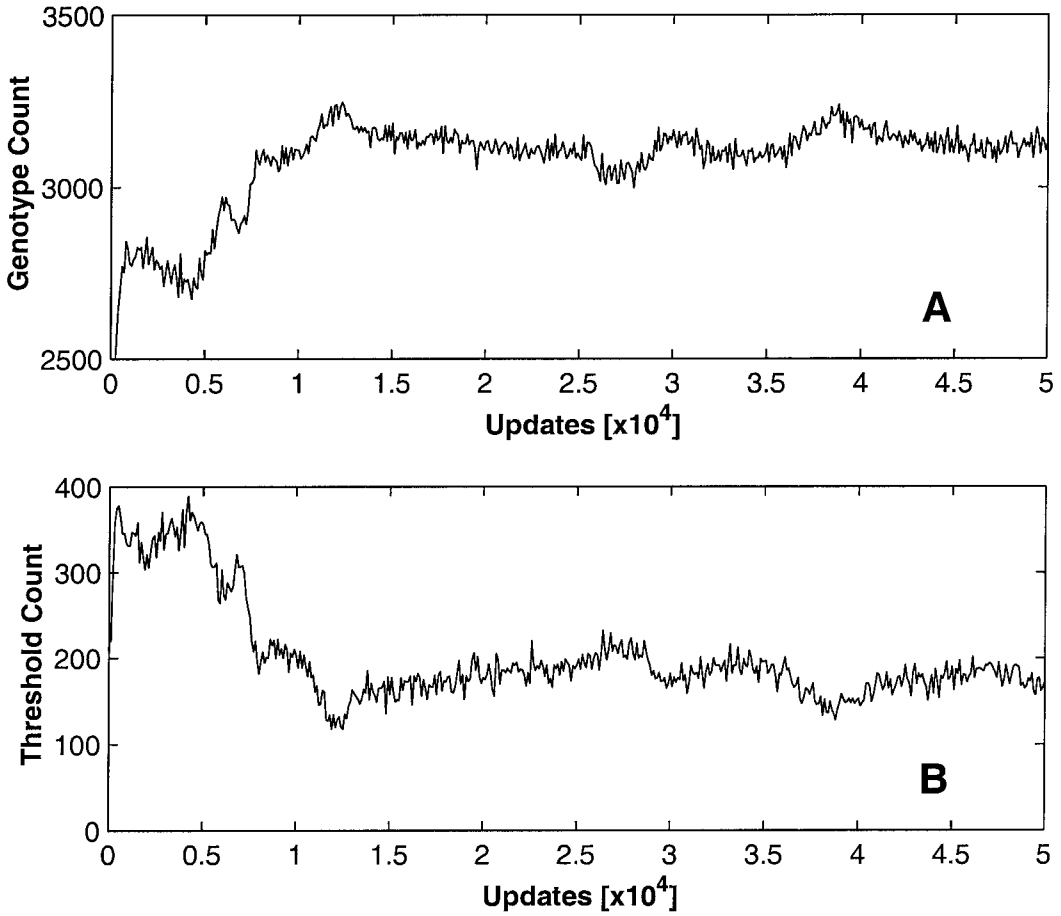
Other significant features of an avida experiment include the count and diversity of genotypes present in the population.

Genotype count (N_g) is the number of unique encodings found in the population, displayed in Figure 2.5(A). As genome lengths increase, so too does the variety of sequences found in the population. In this sample experiment, the high genotype count is mostly due to a dominant genotype (wild type) with a long genome, which is thereby affected by more mutations. Note the correlation between genotype count and genotype length.

Threshold count (N_t) is an approximation (lower bound) of the number of *living* genotypes in a population. A genotype is labeled as threshold if ever its abundance reaches three or more, as it is unlikely to observe that many copies of an organism if it cannot properly survive in this environment. Figure 2.5(B) shows the threshold count for our sample experiment. Note that as the total number of genotypes in the population increases, fewer of them ever reach threshold.

Genotype Entropy (H) is a measurement of the variation in the organisms in a population. Technically, entropy is a measure used to determine the *disorder*

Figure 2.5: Statistics from a typical avida experiment. Measurements for (A) Genotype Count and (B) Threshold Count are displayed throughout the course of evolution over 50,000 updates.



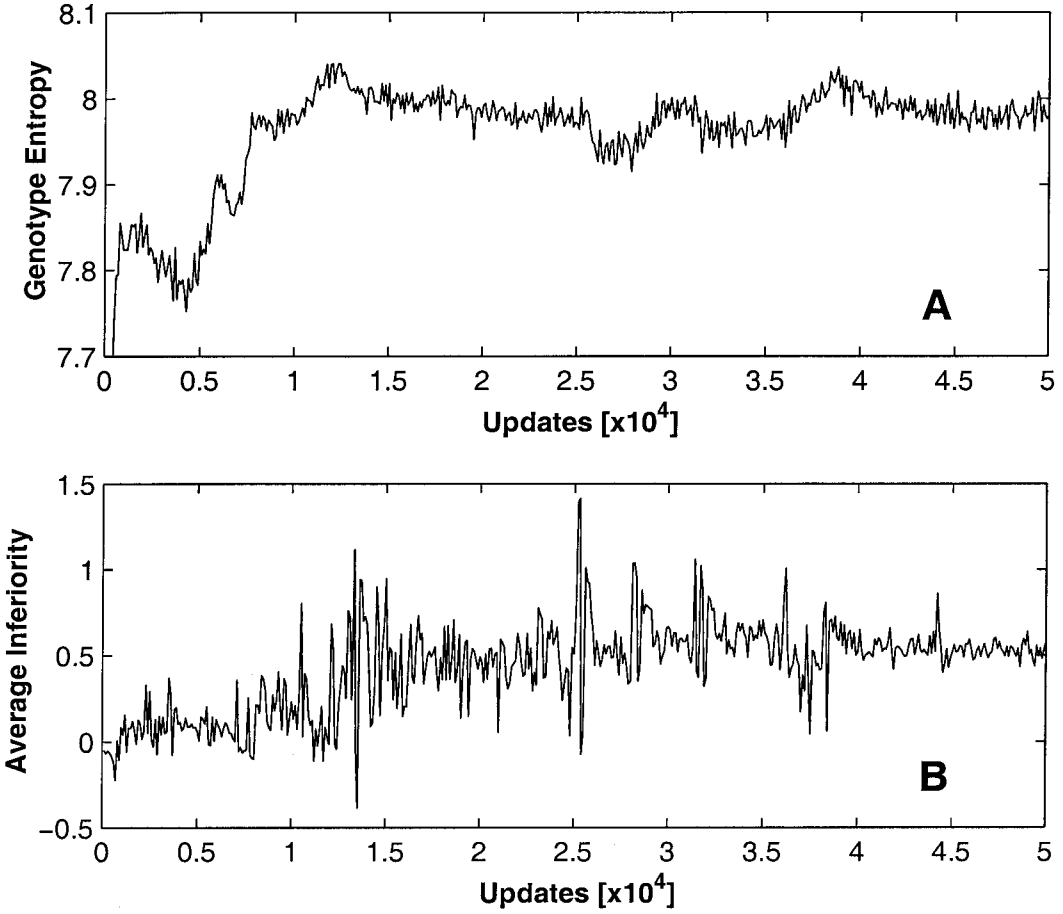
in a system, according to Shannon Information Theory [63]. For this measure, the probability of occurrence of a single genotype i in the population, p_i, is approximated by n_i/N:

$$H = -\sum_i \frac{n_i}{N} \log \frac{n_i}{N} \tag{2.3}$$

where n_i is the current abundance of genotype i and N is the total number of organisms in the population. This statistic is shown in Figure 2.6(A).

Average Inferiority ($\langle \mathcal{I} \rangle$) is a measure that determines how much *worse* the average organism is than the genotype that is currently dominating the population [3].

Figure 2.6: Statistics from a typical avida experiment. Measurements for (A) Genotype Entropy and (B) Average Inferiority are displayed throughout the course of evolution over 50,000 updates.



Each individual organism's inferiority ($\mathcal{I}_i$) is measured as

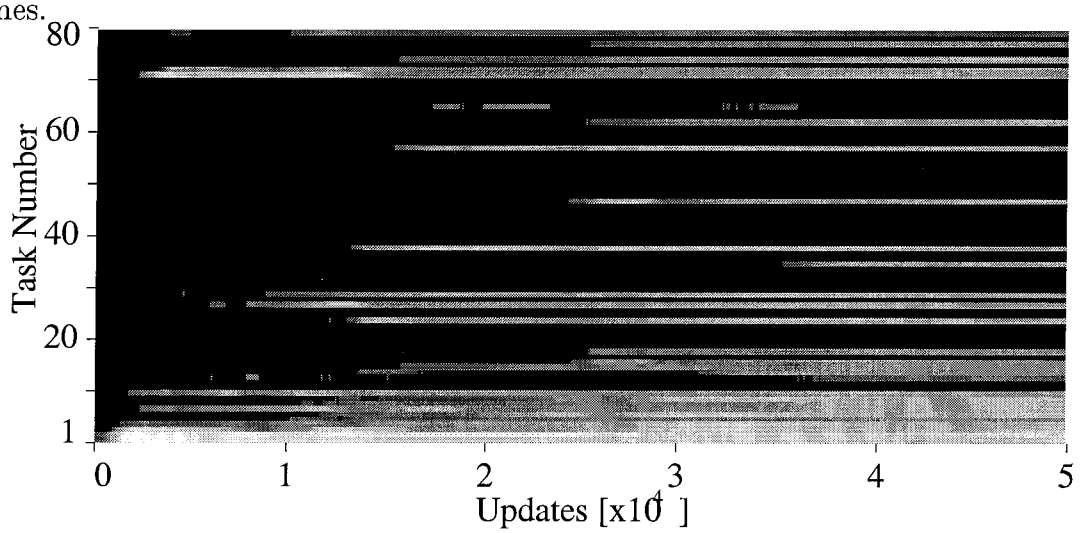
$$\mathcal{I}_i = \log(w_{\text{dom}}) - \log(w_i) \ . \tag{2.4}$$

Note in Figure 2.6(B) that when a new genotype begins to dominate the population, this is reflected by a downward then upward spike in average inferiority that gradually settles down as this new genotype becomes pervasive. This metric is often referred to as *energy* due to its correlation with the statistical mechanics measurement.

80 Tasks are available in the default avida environment for the organisms to incorporate into their genome. Each experiment will follow its own course as different tasks are acquired; typically between 20 and 30 are performed by each organism after

50,000 updates. In this sample experiment (whose tasks are displayed in Figure 2.7) the organisms have learned 27 of the available tasks before the experiment was halted.

Figure 2.7: Statistics from a typical avida experiment. Performance of the 80 awarded tasks is displayed throughout the course of evolution over 50,000 updates. Each horizontal line represents a single task: black indicates tasks never performed, dark gray are performed once by most organisms, and brighter shades are performed multiple times.



2.8 Analysis Tools

The experimental observables detailed in the previous section provide a solid basis for understanding the course of evolution in a population. However, many measurements are not as easily extracted simply by monitoring the actions of the organisms. To this end, I have assembled a collection of tools to further analyze avida experiments.

2.8.1 Test CPUs

All of the analysis tools in avida require the ability to study the behavior of an organism in isolation. I have constructed *test* CPUs that load a genome, process it to determine if it is viable and what tasks it can perform, and record its metabolical statistics involved without ever affecting the population from whence it came. Of course any organism that interacts with other organisms in its environment (such as

a parasite) cannot be studied in this way; most of this work is done excluding these possibilities.

Determining if an organism is viable is not a simple process. An organism is viable if it is "colony-forming"; that is, if it is able to produce a stable colony when introduced into a new environment. In the simplest form, this is an organism that can produce at least two identical copies of itself through reproduction.

One immediate use for these isolated CPUs is to collect statistics on any organism that we are extracting from the population (see Appendix B for examples of extracted organisms). More sophisticated uses include studying products of gene-surgery that we do not want to place into the population itself. Sections 2.8.2 and 2.8.3 exploit test CPUs in this manner.

2.8.2 Species

Up until this point, organisms in avida have been classified and studied on the basis of their *genotype*. That is, only if two organisms have identical genomes are their functionality considered together, in the form of genotypic statistics. However, from a biological perspective, we are interested in grouping organisms that have functionally and organizationally equivalent (though not identical) genomes, such that if they were to be recombined they would produce an "offspring" with identical functionality.

In Biology, it is quite difficult to classify asexual organisms in terms of species as sexual recombination rarely (if ever) occurs. However, we are not bound by such restrictions in digital organisms—if we want to know if two organisms can produce a successful offspring by the crossover of their genomes, we can construct and test such a hybrid in an isolated test CPU. In fact, we can test *all* possible recombinations between the two genomes. We line them up next to each other for the best possible match, and then move from one point to the next trying both possible recombinations for that crossover point.

In effect, this technique isolates those genotypes that differ only in non-functional regions of their code. If such non-coding portions are broken up and recombined, there

should be no change in the overall expression. On the contrary, if meaningful code is divided in this process, the resultant organism will most likely lose computational or reproductive capabilities.

In the end, we have a species definition that is satisfying from both a computational and biological perspective: only codes that are functionally identical are classified together, and a method akin to sexual recombination is used to perform this test.

Species Count (N_s) is portrayed in Figure 2.8(A). This approximates the number of different functional methods present in the population. So that non-living organisms do not each get counted as a new species, only genotypes that reach threshold have their species tested. All new genotypes are, by default, assigned the species of their parent until threshold is attained.

Species Entropy (H_s) provides a measurement of the diversity of species within a population. Figure 2.8(B) demonstrates a drop in the number of distinct functional techniques after each jump in fitness (Figure 2.4) due to the rise of a new species with a competitive advantage that drives all others to extinction, and then diversifies with time.

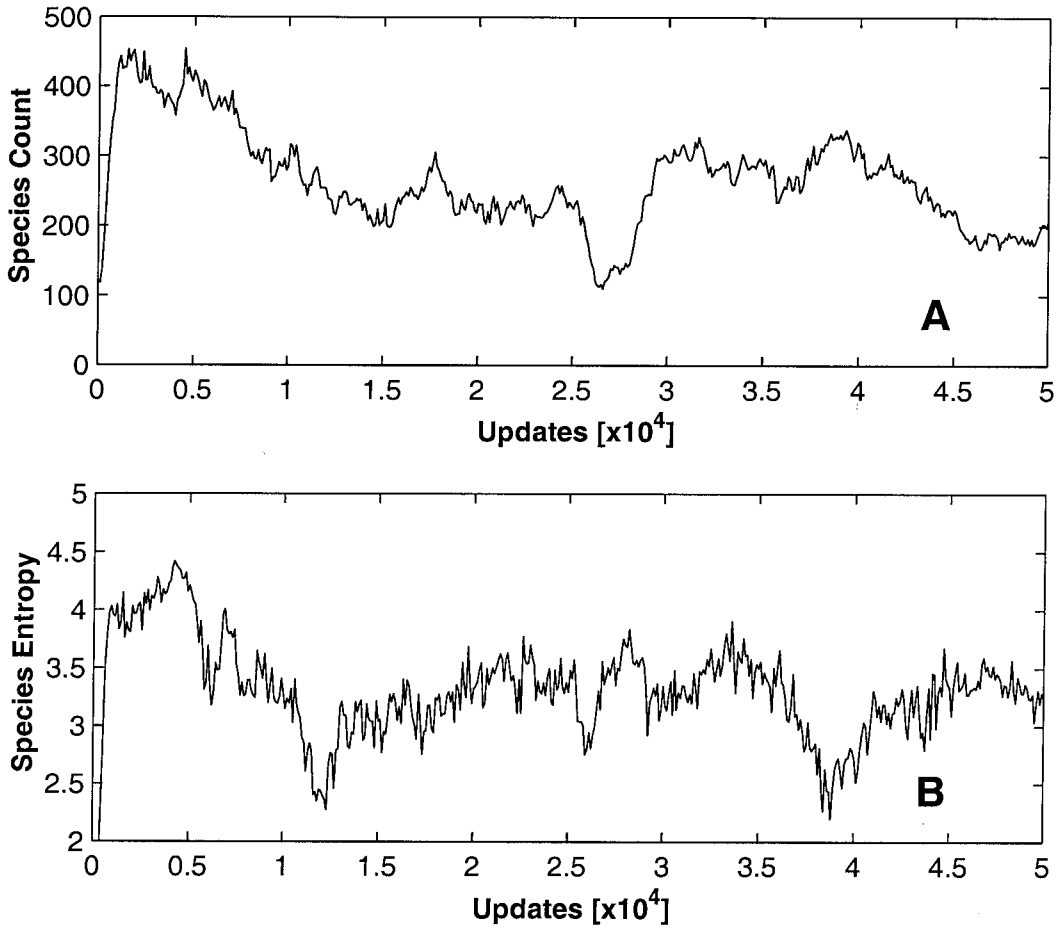
2.8.3 Local Landscape Analysis

In avida, we have the ability to experimentally determine the structure of the local genetic landscape around a genotype. In fact, we can look at all possible neighboring genotypes that are a single mutation away from the wild type, and study each of these neighbors in a test CPU in order to classify them as having a positive, negative, or neutral effect on fitness.

Given that we have D instructions in our instruction set, and we are considering a genome of length ℓ, the number of single-step mutations that can be performed on this genome (i.e., the size of the local fitness landscape) is

$$N_{\text{muts}} = \ell(D-1)\ . \tag{2.5}$$

Figure 2.8: Statistics from a typical avida experiment. Measurements for (A) Species Count and (B) Species Entropy are displayed throughout the course of evolution over 50,000 updates.



For example, the default instruction set in avida has $D = 28$, so if we consider a genome of $\ell = 80$, we would need to test $(28 - 1) \times 80 = 2,160$ neighboring genomes.

Immediately, we can obtain four numbers to describe this local landscape. These are the fraction of *fatal* mutations (those that cause the resulting organism to lose its ability to self-replicate), of *negative* mutations (those that impair replication rate), of *neutral* mutations (those that leave replication rate unaffected), and of *positive* mutations (those that actually improve replication rate). The fraction of neutral mutations in an organism, or its *neutrality* (ν), is a fundamental measure that we shall use to understand the evolution of that organism. Note that these categorizations of mutations only take into account an organism's speed of replication in isolation, not

any effects of fidelity or the susceptibility to subsequent mutations. Nonetheless, this is a good indicator of the local landscape.

Figure 2.9(A) displays the proportions of each of the mutation types over the course of the sample experiment. Note that the original (ancestral) organism is dominated by fatal mutations, but both detrimental and neutral mutations become more prevalent as the organisms evolve to become more robust to the noisy environment they must survive in. All human-written programs show a similar fragility, which is subsequently lost through evolution. The commensurate decline in neutrality (seen to initiate near update 25,000) is caused by learning events not associated with size changes. Under such circumstances, new functionality must be placed in available code, thereby reducing the neutrality at those sites. If any of the sites containing this new information are now disturbed, they deminish the bonus earned by the organisms, and thus are classified as detrimental mutations (as indicated by the corresponding rise in the detrimental fraction at this time).

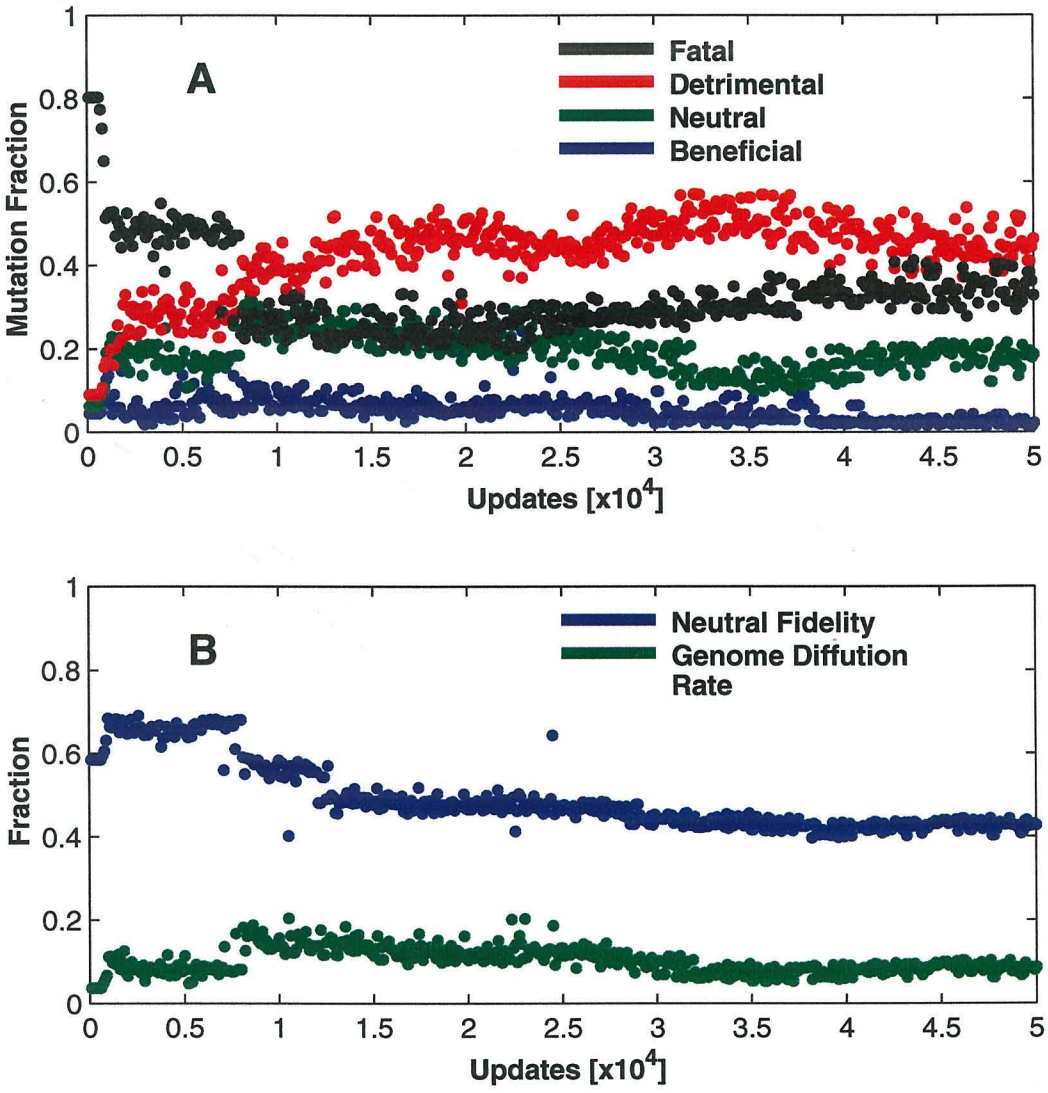
Fitness and *fidelity*, combined with *neutrality* are key in determining the ability of an organism to thrive in an avida environment. They correspond respectively to an organism's ability to create offspring, for those offspring to have a minimum mutational load, and for them to survive those mutations which they do bear. Apart from this, however, there is another aspect which is necessary for a phylogenetic branch to be successful, and that is its ability to further adapt to its environment. To characterize this, we define two more genomic attributes, both presented in Figure 2.9(B):

Neutral Fidelity (F_ν) is a measure that can be calculated once an organism's neutrality is known. It is the probability that an organism will give birth to an identical *or equivalent* offspring. Taking $f_c = R_c(1-\nu)$ to be the probability for an instruction to receive a mutation that is not neutral to the organism, we obtain the neutral fidelity as:

$$F_{\text{neut}} = (1 - f_c)^\ell \,. \tag{2.6}$$

Genomic Diffusion Rate (D_g) is the probability for an offspring to have a

Figure 2.9: Statistics from a typical avida experiment. Part (A) breaks down all of the possible mutations on the dominant genotype into the categories *fatal*, *detrimental*, *neutral*, and *beneficial*. Part (B) shows Neutral Fidelity and Genomic Diffusion Rate. Both graphs are displayed throughout the course of evolution over 50,000 updates.



genome *different* from its parent, but to be otherwise equivalent (i.e., neutral). This is obtained by subtracting the genome's fidelity from its neutral fidelity

$$D_g = F_{\text{neut}} - F \ . \tag{2.7}$$

This is a particularly important indicator as it is the rate at which new, viable genotypes are being created, which in turn is the pace at which genetic space is being explored, and therefore positively correlated with the rate of adaptation.

Chapter 3 An Analytic Approach to Evolution

Darwinian evolution is a simple yet powerful process that requires only a population of reproducing organisms in which each offspring has the potential for a heritable variation from its parent. This principle governs evolution in the natural world, and has gracefully produced organisms of vast complexity. Still, whether complexity is in fact increasing in evolution has become a contentious issue. Gould [29], for example, argues that any recognizable trend can be explained by the "drunkard's walk" model, where "progress" is due simply to a fixed boundary condition. McShea [44] investigates the evolution of certain types of structural and functional complexity, and finds some evidence of a trend but nothing conclusive. In fact, he concludes that "Something may be increasing. But is it complexity?" Bennett [12], on the other hand, explicitly defines complexity as "that which increases when self-organizing systems organize themselves" resolving the issue by fiat. Of course, in order to address this issue, complexity needs to have a clear definition.

In this chapter, I circumvent the issue of structural and functional complexity by examining *genomic complexity* instead, and assume that organismic complexity is incorporated into this. Of course, how the complexity of the organism is reflected in the complexity of its genome (or vice versa) is a contentious issue in itself, owing not only to the aforementioned ambiguous definition of complexity but also to the obvious difficulty of matching genes and function. Also, the gradual pace of evolution renders an observation of changes in complexity imperceptible on non-geological time scales.

Here, we are concerned with the evolution of complexity in biological genomes and the factors that influence such evolution. Several developments allow us to bring a new perspective to this old problem. On the one hand, genomic complexity can be defined

in a consistent information-theoretic manner [8] that appears to encompass intuitive notions of complexity used in the analysis of genomic structure and organization [15]. On the other hand, we can observe evolution in an artificial medium such as avida, allowing us to observe the growth of complexity explicitly, and also to distinguish distinct evolutionary pressures acting on the genome and analyze them within a mathematical framework.

If an organism's complexity is a reflection of the (mathematical) complexity of its genome (as we assume here), understanding evolutionary pressures on the genome is of prime importance in evolutionary theory. Equating genomic complexity with the full length of a genome in base pairs gives rise to a conundrum (known as the C-value paradox) because large variations in genomic complexity (in particular in eukaryotes) seem to bear little relation to the differences in organismic complexity [16]. The C-value paradox is partly resolved by recognizing that not all of DNA is functional; that is, there is a *neutral* fraction that can vary from species to species. If we were able to monitor the non-neutral fraction, it is likely that a significant increase in this fraction would be observed throughout, at least, the early course of evolution. For the later period, in particular the later Phanerozoic Era of natural history, it is unlikely that the growth in complexity of genomes is due solely to innovations in which genes with novel functions arise *de novo* with a commensurate increase in sequence length. Indeed, most of the enzyme activity classes in mammals, for example, are already present in prokaryotes [21]. Rather, gene duplication events leading to repetitive DNA and subsequent functional divergence [14] as well as the evolution of gene regulation patterns acting on those genes appears to be a more likely scenario for this stage. However, if we believe that the ancestral molecule of prokaryotic life was a simple RNA replicase, evolution toward complex genomes must have involved evolutionary transitions in which first simple, then more complex genes (coding for diverse proteins and enzymes) evolved out of randomness, and in which the growth of sequence length paralleled the growth of information coded into those genomes. This period, simpler than the subsequent era in which gene regulation is of primary importance, can be explored in detail in avida and compared to a

theory couched in the simple and intuitive language of information theory. Even after ontogenic gene regulation becomes the primary source of evolutionary innovation, the complexity growth dynamics described here still play a role, albeit less prominently.

In the next section, I present complexity and its evolution from the perspective of information theory and discuss how to measure complexity in evolving populations. I apply this measure to avida in Section 3.2 and study the course of its evolution in Section 3.3. Then I examine an evolutionary transition in detail in Section 3.4, demonstrating how complexity is increased by the action of a natural "Maxwell Demon." Finally, in Sections 3.5 and 3.6 I analyze evolutionary pressures on the genetic codes and show how they interact to promote complexity growth.

3.1 Information Theory and Complexity

The use of information theory to understand evolution and the information content of genetic sequences is not a new undertaking. Unfortunately, many of the earlier attempts (e.g., that of Schrödinger [62], Gatlin [28], and Wiley and Brooks [68]) confuse the picture more than they clarify it, often clouded by misguided notions of the concept of information[1].

A key aspect of information theory is that information cannot exist in a vacuum; that is, information is *physical* [36]. This means that information must have an instantiation (be it ink on paper, bits in a computer's memory, or even the neurons in a brain). Furthermore this information must be *about something.* Lines on a piece of paper, for example, are not inherently information until it is discovered that they correspond to something, such as (in the case of a map) the organization of local streets and buildings. Consequently, any arrangement of symbols might be viewed as *potential information* (also known as *entropy* in information theory), but acquires the status of information only when considering its correspondence, or correlation, to other physical objects.

In biological systems, the instantiation of information is DNA, but what is this

[1] In particular, Brillouin's book [13] has done nothing but confuse a generation of researchers.

information about? In part, it is the blueprint of an organism and as such information about its own structure. More specifically, it is a blueprint on how to build an organism that can best survive in its native environment, and pass on that information to its progeny[2]. Thus, those parts of the genome that do correspond to something (the non-neutral fraction, that is) are fixed only because they provide a benefit to the organism for its survival in the environment it lives in. Deutsch [19] summarized this as "Genes embody knowledge about their niches." This environment, of course, is extremely complex itself. It ranges from the ribosomes the genetic sequences are translated by, over other chemicals and the abundance of nutrients inside and outside the cell, to the environment of the organism proper (i.e., the oxygen abundance in the air as well as ambient temperatures, to name but a few). Thus, an organism's DNA is not only a book about the organism, but is also a book about the environment it lives in, including the species it co-evolves with.

It is well-known, of course, that not all the symbols in an organism's DNA correspond to something. This is (no doubt misleadingly) referred to as "junk-DNA", and usually equated to the portion of code that is unexpressed or untranslated (i.e., introns excised from the mRNA). More modern views concede that unexpressed and untranslated regions in the genome can have a multitude of uses, such as satellite DNA near the centromere, or the poly-C polymerase intron excised from *Tetrahymena* rRNA. In the absence of a complete map of the function of each and every base pair in the genome, how can we then decide which stretch of code is "about something" (and thus contributes to the complexity of the code) or else is entropy (i.e., random code)?

A true test for whether a sequence is information uses the success (fitness) of its bearer in its current environment. Note that this implies, naturally, that a sequence's information content is *conditional* on the environment it is to be interpreted within [8]. For example, *Mycoplasma mycoides* (which causes pneumonia-like respiratory illnesses), has a complexity of somewhat less than 10^6 base pairs in our nasal

[2]This is essentially Dawkins' view of selfish genes which "use" their environment (including the organism itself), for their own replication [18].

passages, but close to zero complexity most everywhere else; effectively it cannot interact with any other environment so its code would count for little more than a random nucleotide sequence outside this favored habitat.

A genetic locus that codes for information essential to an organism's survival will be *fixed* in an adapting population because mutations of that locus result in the organism's reduced ability to promulgate the tainted genome, whereas inconsequential (neutral) sites will be randomized by the constant mutational load. It would thus be sufficient to examine an *ensemble* of sequences large enough to obtain statistically significant substitution probabilities to separate information from entropy in genetic codes.

In order for new functionality to develop within a genetic code, some mechanism for change must exist. The "breeding grounds" for new functionality are usually neutral (variable) or redundant sections of the genome, which can be mutated without harm to the organism. According to the previous arguments, they do not contribute to the complexity of the organism (they encode no information about their environment[3]). However, they allow many potential (randomly generated) gene sequences to be explored. Should ever one of these random sequences code for a protein that improves survivability, then the organism that carries it will increase its relative abundance within the population, and the once-random sites will fixate. Thus, entropy is transmuted into information. Neutral sections of the code thus turn out to be critical for evolution to proceed, as has been pointed out by Maynard Smith [43]. Furthermore, neutrality may actually be selected for under certain circumstances, which I will discuss in Section 3.6.

In Shannon's information theory [63], the metric *entropy* (H) represents the expected number of bits required to specify a state, given a distribution of probabilities (i.e., it measures the amount of potential information that could be recorded in that

[3]A duplicated gene does not represent information if it can be replaced with a random sequence without affecting the fitness of the organism.

system). For a site i that can take on four nucleotides with probabilities

$$\{p_C(i), p_G(i), p_A(i), p_T(i)\} \ , \tag{3.1}$$

the entropy is

$$H_i = -\sum_{j}^{C,G,A,T} p_j(i) \log p_j(i) \ . \tag{3.2}$$

The maximal entropy per-site (if we agree to take our logarithms to base D—the size of the alphabet—in this case 4) is 1, which happens only if the probabilities are all equal at $1/D$. If the entropy is measured in bits (take all logarithms to base 2) the maximal entropy per site is $\log_2 4 = 2$ bits. Naturally, this also corresponds to the maximal amount of information that can be stored in a site: $\log D$. If a site is perfectly conserved across an equilibrated ensemble, then we assign the probability $p = 1$ to one of the symbols and zero to all others, rendering $H_i = 0$ for that site according to Eq. (3.2). The amount of information per site is thus [61]

$$I(i) = H_{\max} - H_i \ . \tag{3.3}$$

In the following, we shall approximate the complexity of an organism's sequence by applying Eq. (3.3) to each site and summing over the sites. Thus, for an organism of ℓ base pairs, the complexity is

$$C \approx \ell - \sum_i H(i) \ . \tag{3.4}$$

This value can only be an approximation to the true informational (or *physical*) complexity of an organism's genome because, in reality, sites are not independent and the probability to find a certain base at one position may be conditional on the probability to find another base at another position. Such *correlations* between sites are called *epistatic* and they can render the entropy *per sequence* significantly different from the sum of the per-site entropies. The entropy per sequence, taking into account

all epistatic correlations between sites, is defined as

$$H = -\sum_i p(i|E) \log p(i|E) \tag{3.5}$$

and involves an average over the logarithm of the conditional probabilities $p(i|E)$ to find genotype i *given* the current environment E. In every finite population, estimating $p(i|E)$ using the actual frequencies of the genotypes in the population (if those could be obtained) results in corrections to (3.5) larger than the quantity itself [11], rendering the estimate useless. Another avenue for estimating the entropy per molecule is the creation of clones, mutated at several positions [25, 40] to measure epistatic effects. The latter approach is feasible in single niche systems of digital organisms, as an extension of the local landscape analysis described in Section 2.8.3.

3.2 Complexity in Avida

Despite the apparent simplicity of the single-niche environment and the limited interactions between digital organisms, we can observe rich dynamics in avida experiments. As the populations I use are small (3600 organisms in all experiments reported here), we can assume that an equilibrium population will be dominated by organisms that all code for similar functionality and are of equivalent fitness (except for those organisms that have been adversely affected by mutation). In this world, an organism with different functionality can only obtain a significant abundance if it has a competitive advantage (increased Malthusian parameter) thanks to a beneficial mutation. As the system returns to equilibrium after the innovation, this new species will gradually exert dominance over the population, bringing the previously dominating species to extinction. In artificial evolution, perfect knowledge of the state of each organism's genetic code during the population's evolution gives us the ability to present a detailed view of the process and to accurately test our hypotheses. This dynamics of innovation and extinction has been monitored in detail and appears to mirror the dynamics of *E. coli* in single-niche long-term evolution experiments [24].

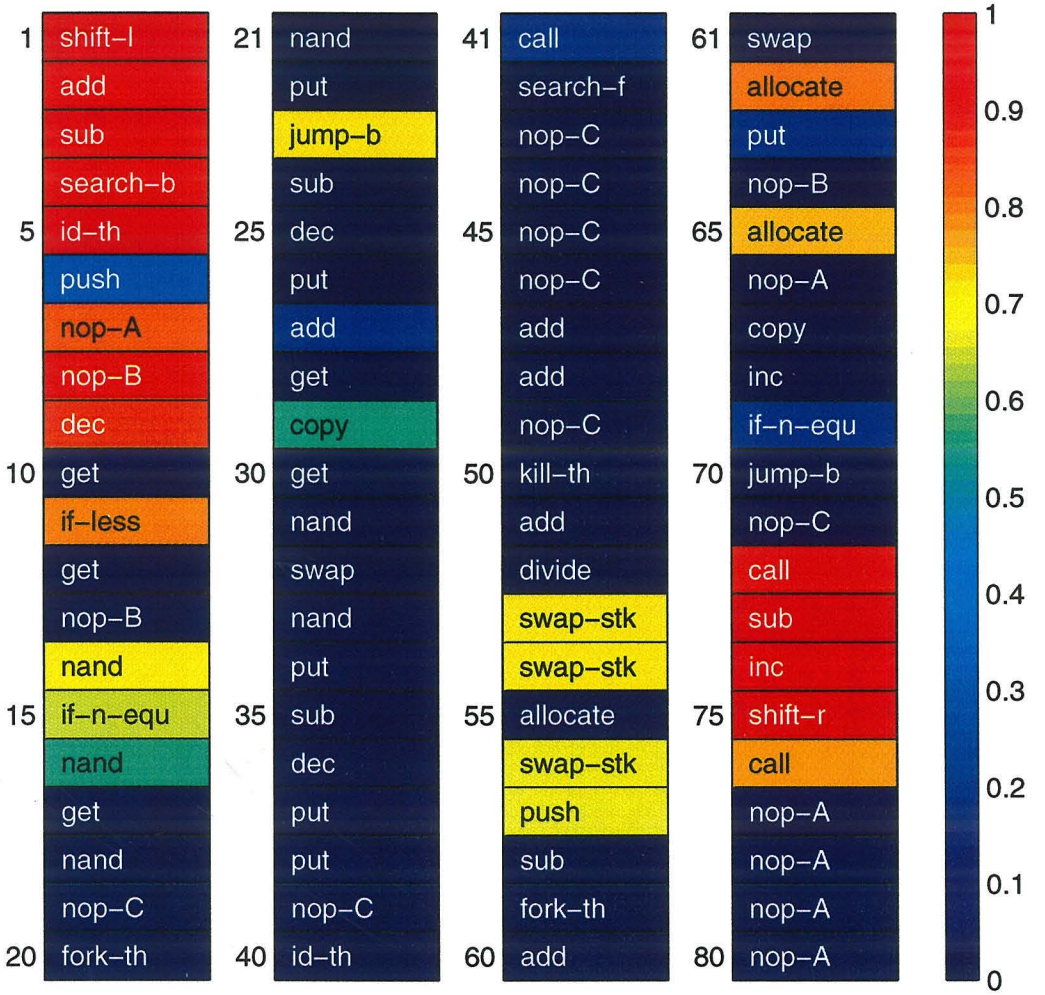


Figure 3.1: A typical avida organism, extracted 1,434 generations into an evolutionary experiment. Each site in the code is color-coded according to the entropy of that site, as determined by studying the effects of *all* single-point mutations in test-CPUs. Red sites are highly variable whereas black sites are perfectly conserved.

The complexity of an adapted digital organism according to Eq. (3.4) can be obtained by measuring substitution frequencies at each instruction across the population. This is easiest if genome size is constrained to be constant as is done in the experiments reported below, even though this can be relaxed by implementing a suitable alignment procedure, or testing the effects of mutations (to determine if they are neutral) in test CPUs. In order to correctly assess the information content of the ensemble of sequences, we need to obtain the substitution probabilities p_i at each position, going into the calculation of the per site entropy, Eq. (3.2). If we use

the distribution of instructions at each site that are present on the population, care must be taken to wait sufficiently long after an innovation, in order to give those sites within a new species that are variable a chance to diverge. This is because shortly after an innovation, previously 100% variable sites will appear fixed by "hitchhiking" on the successful genotype (see below).

For a population in which the sequence length was held constant at 80, collecting the p_i (for $i = 1, \cdots, 28$; $D = 28$) at equilibrium allows us to calculate the per-site entropy for any locus (k) in the organism as

$$H_k = -\sum_{i=1}^{28} p_i \log_{28} p_i \ , \tag{3.6}$$

an example of which is shown in Figure 3.1. This map of the genome shows that the organism is well-adapted (after 1,434 generations) and sports primarily conserved sites. Most of its variability (and ensuing neutrality) appears to be in the first ten lines, with smaller blocks elsewhere.

3.3 Progression of Complexity

Tracking the entropy of each site in the genome allows us to document the growth of complexity in an evolutionary event. Figure 3.2 displays the genome of an organism extracted 34 generations after the one displayed in Figure 3.1. This program is an offspring of an organism that underwent a mutation that allowed it to perform several new computational tasks, resulting in a fitness increase by a factor of 5.1. Comparing the two entropy maps, we can immediately identify the sections of the genome that code for those new tasks—the entropy at those sites has been drastically reduced. The organism shown in Figure 3.2 is unlikely to be a direct descendant of the one in Figure 3.1, as they differ widely in the neutral sections of code where most of the new gene appear to have arisen. Note, however, that all instructions that were frozen before this transition remain unchanged in the new organism, while most volatile sites differ.

Figure 3.2: An avida organism extracted 34 generations later than the one depicted in Fig. 3.1. A learning event has occurred, freezing most of the beginning of the genome and several other loci.

We can extend this analysis by continually surveying the entropies of each site during the course of an experiment. Figure 3.3 does this for the experiment just discussed. A number of features are apparent in this figure. First, the trend toward a "cooling" of the genome (i.e., to more conserved sites) is obvious. Second, major evolutionary transitions can be identified by vertical darkened "bands", which arise because the genome instigating the transition replicates faster than its competitors thus driving them into extinction. As a consequence, even random sites that are "hitchhiking" on the successful gene are momentarily fixed. This is documented clearly by plotting the sum of per-site entropies for the population (as an approximation for the entropy of

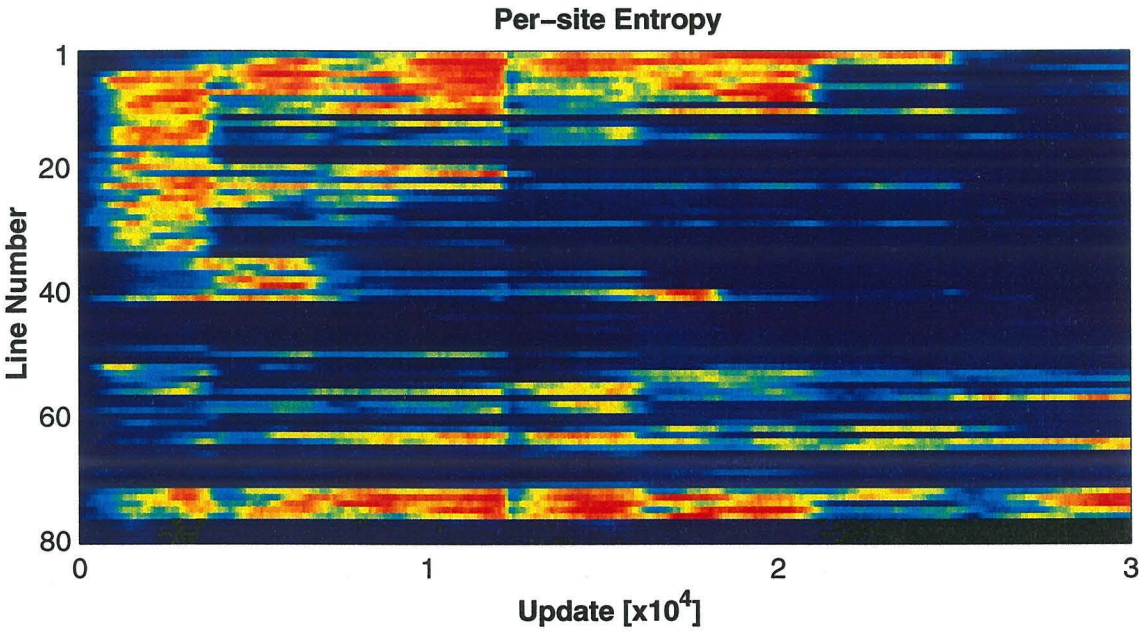


Figure 3.3: Progression of per-site entropy for all 80 sites throughout an avida experiment. The entropies are calculated at 60 points evenly spaced throughout the course of evolution.

the genome)

$$H \approx \sum_{k=1}^{\ell} H_k \tag{3.7}$$

across the transition in Figure 3.4(A). By comparing this to the fitness shown in Figure 3.4(B), we can clearly see a similar sharp drop in entropy followed by a slower rise for each learning event that the population undergoes. Quite often the population does not even reach an equilibrium state again before the next transition occurs.

While this entropy is not a perfect approximation of the exact entropy per program (Eq. 3.5), it does accurately reflect the disorder in the population as a function of time. I show this complexity estimate (3.4) as a function of evolutionary time for this experiment in Figure 3.5. While the measured complexity decreases briefly after transitions, it always settles down at a higher level. This overshooting of stable complexity is a result of the overestimate of complexity during the transition due to hitchhiking. This effect is also seen at the begining of evolution, as the population

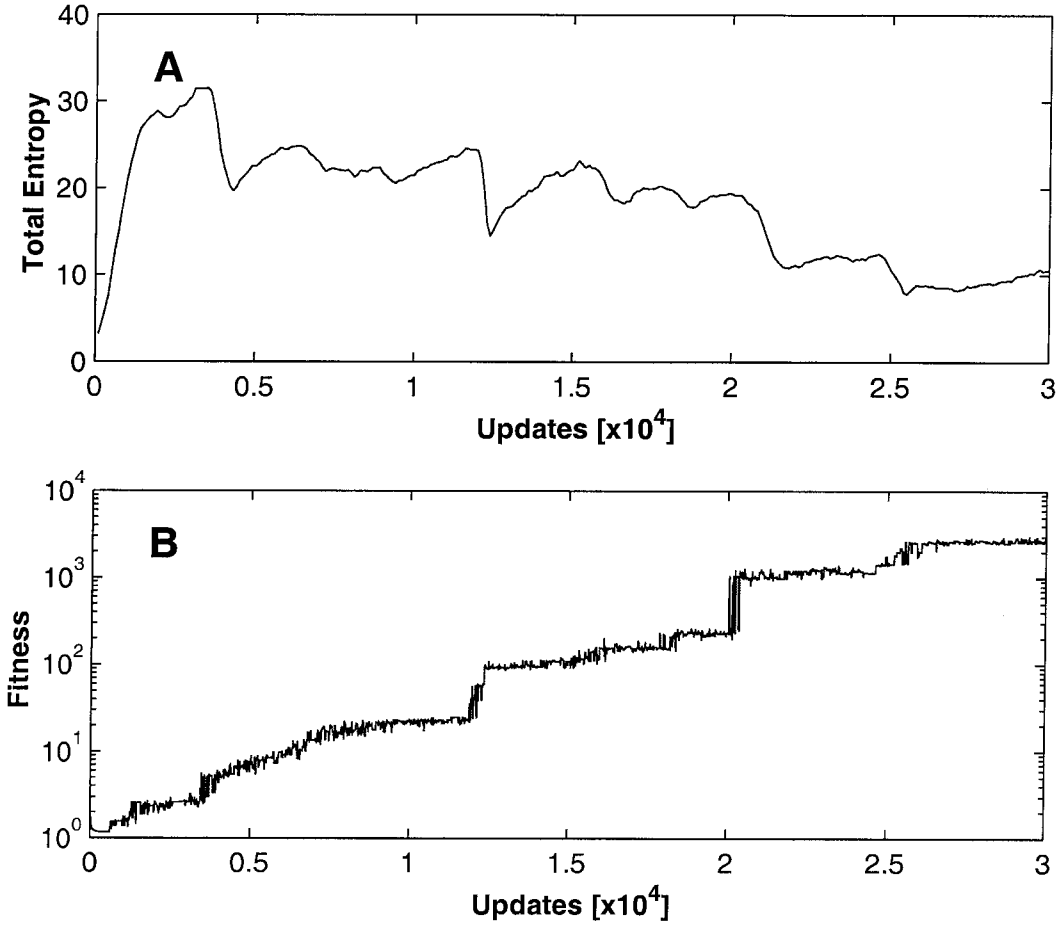


Figure 3.4: (A) Total entropy per program as a function of evolutionary time. (B) Fitness of the most abundant genotype as a function of time. Evolutionary transitions are identified with short periods in which the entropy drops sharply and fitness jumps.

is seeded with a single genome with no variation present. This typical evolutionary history documents that the complexity, measuring the amount of information coded in the sequence about its environment, steadily increases. The circumstances under which this is assured to happen is discussed in the following sections.

3.4 Maxwell's Demon and the Law of Increasing Complexity

Let us consider an evolutionary transition like the one leading from the genome in Figure 3.1 to that in Figure 3.2 in more detail. This transition is magnified in Figure 3.6,

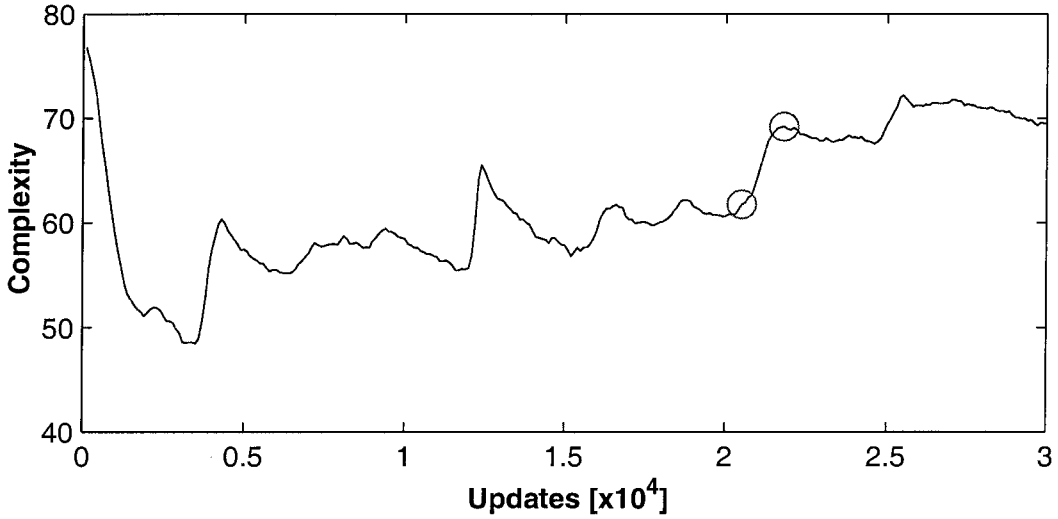


Figure 3.5: Complexity as a function of time. The organisms from Figures 3.1 and 3.2 are indicated by circles.

which shows that the entropy does not fully recover after its initial drop.

The difference between the equilibrium level before the transition and after is proportional to the information acquired in the transition, namely the number of sites that were frozen. This difference would be equal to the acquired information if the measured entropy (Eq. 3.7) were equal to the exact one given by Eq. (3.5). For this particular situation, in which the sequence length is fixed along with the environment, is it possible that the complexity decreases? The answer is that in a sufficiently large population this cannot happen[4] as a consequence of a simple application of the second law of thermodynamics. If we assume that a population is at equilibrium in a fixed environment, each locus has achieved its highest entropy given all the other sites. Thus, the entropy can only stay constant or decrease, implying that the complexity (being sequence length minus entropy) can only increase. How is a drop in entropy (such as the one evident in Figure 3.6) commensurate with the second law? That answer is simple also: the second law only holds for equilibrium systems. The transition we are witnessing in Figure 3.6 (and throughout the experiment) is decidedly *not* of the equilibrium type. In fact, such a transition can be described

[4]In smaller populations, there is a finite probability of all organisms being mutated simultaneously, referred to as Muller's ratchet [49].

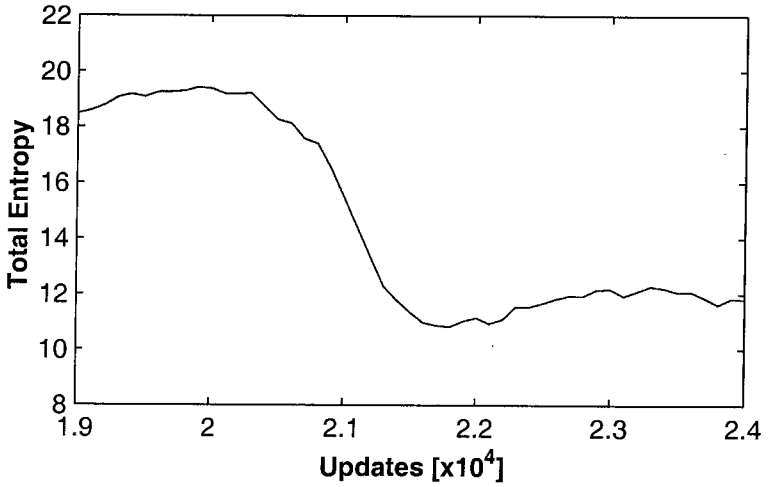


Figure 3.6: Entropy per program as a function of evolutionary time across a transition.

conveniently as a *measurement.*

In the classical measurement situation, the object to be measured is hooked up to a measurement device in such a manner that the state of the object is reflected in the state of the device. In other words, object and device are rigged so that they are *correlated*, such that one may give information about the other. Performing measurements on a system clearly reduces our uncertainty about the system, or in other words, reduces the entropy by increasing the amount of information we have about that system. This process cannot be described within equilibrium statistical physics either [7], but is perfectly well-described within information theory. It has been speculated for a long time that making judicious measurements would allow a violation of the second law (see, e.g., [37]). A "beast" that conducts such experiments was described first by Maxwell [42] and was later termed the "Maxwell Demon." The Demon was thought to operate a door separating two compartments filled with an inert gas at equal pressure. By measuring the molecules' speed and opening the door for the fast molecules to enter one half but closing it for the slow ones, the demon could produce a pressure difference between the halves and accordingly a difference in entropy, apparently violating thermodynamic's second law (see Fig. 3.7). Something quite analogous happens in evolution.

In effect, evolution can be thought of as a series of random measurements on the

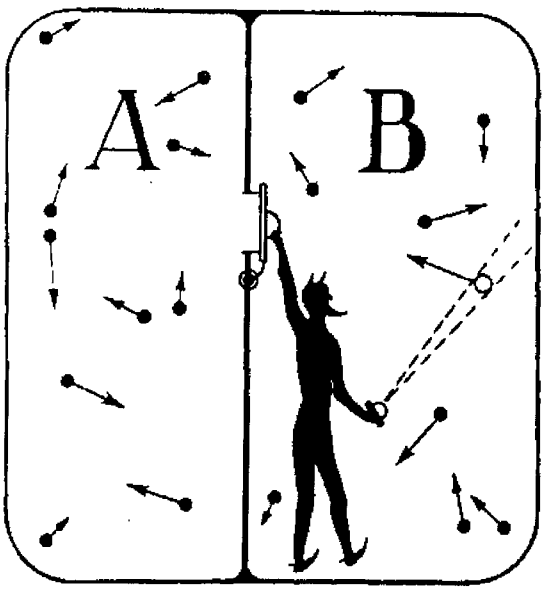


Figure 3.7: Maxwell's Demon at work (from [41]).

environment. Darwinian selection is a filter, allowing only informative measurements (those increasing the ability for an organism to survive) to be preserved.

A mutation that increases fitness leads to the fixation of some sites (or, at least to a net increase in the number of fixed sites) because the increased replication rate of the mutated genome causes (due to heredity) a non-equilibrium event ("avalanche") that installs the mutation as the new wild-type. Because the new genotype represents information about its environment and that information has increased, the transition in fact represents a measurement—performed by the population on its environment. Information cannot be lost in such an event because a mutation corrupting the information is purged from the population due to the corrupted genome's inferior fitness[5]. A mutation that corrupts the information cannot increase the fitness, because if it did then the population was not at equilibrium in the first place. As a consequence, only mutations that reduce the entropy are kept while mutations that increase it are purged. This is the classical behavior of the Maxwell Demon.

In an unchanging environment, an increase in the entropy can only occur if there is a rise in the average length of the organisms. In such an event, the number of bits

[5]This holds strictly for asexual populations only.

needed to describe an individual genome increases, while the amount of information held in those genomes remains fixed. The extra length comes in the form of entropy. Note, however, that these size-change events are critical to continued evolution as they provide a new space to record environmental information within the genome, and thus allows complexity to march ever forward.

3.5 Selective Pressures on Genome Size and Neutrality

In the preceding sections, I argued that a genomic complexity can be defined rigorously in the context of information theory. According to this simple and intuitive measure, a genome's complexity corresponds to the amount of information *about* its environment coded in the sequence. Thus, complexity is *context dependent* rather than absolute, and increases during every evolutionary transition as an organism obtains more information that improves it ability to survive in its environment. If the context, (i.e., the environment), does not change, complexity is forced to increase by a process akin to the operation of a Maxwell Demon. Thus, information can only enter a (statistically equilibrated) population, and never be lost. Naturally, should the environment itself be altered (including through the co-evolution of other organisms) or the organisms enter a *new* environment, the complexity of the sequences it harbors can decrease as portions of their genome information become invalid (transforming them into entropy). This does not, by itself, violate the law of increasing genomic complexity, which only declares that the complexity *given* a particular environment must always increase. Change the environment, and the complexity can change dramatically (a fact most clearly demonstrated by human interference with the ecosystem).

In the remainder of this chapter, I focus on the selective pressures of molecular evolution and how they contribute to the evolution of complexity. From the point of view of information theory, it is convenient to view Darwinian evolution of popula-

tions of code much like an *information transmission channel*, subject to a number of constraints. The information transmitted is the genome of a particular species, from one generation to the next, and it is subject to noise due to an imperfect copy process. Information theory is concerned with analyzing the properties of such channels, how much information can be transmitted and how the *capacity* of such a channel—the maximum amount of information that can be transmitted given a certain error rate—can be achieved. Selection acts to maximize the number of times the genetic information is transmitted to offspring, per lifetime. This maximization involves a number of factors, and illustrates the pressures evolution exerts on the code itself.

One way of decreasing transmission time is by *shortening* the message as long as such a length decrease does not affect the rate at which symbols are copied (i.e., code compression). This type of evolutionary pressure is analogous to r-selection in Evolutionary Biology, and is observed most frequently when a species invades a new niche. Its effects on the genome usually involve the removal of genes that are unimportant in the new environment thereby compressing the length of the *necessary* message, and increasing the rate of replication. This pressure has also been seen at work in the classical experiments of Spiegelman [45] involving RNA replicating in vitro.

Another way of increasing the channel's capacity consists of increasing an individual's expected lifetime and therefore raising the total number of copies of the genome that it has time to propagate. Typically, this occurs when an organism is mutated such that it takes better advantage of the environment it lives in, improving its own ability to make use of resources or avoid dangers. This type of adaptation is referred to as K-selection. Usually, both r- and K-selection act on the genome simultaneously while local conditions decide which dominates.

Improvements in the efficiency of the replication process brought about by r-selection are fundamentally limited by the amount of information that must be maintained within the genetic code. Once a species is established in a niche, most continuing evolution occurs via K-selection, which has no such limitation, but does have a cost involved. Specialization to a particular niche requires more information about

that environment to be stored within the genome, resulting in a longer code to transmit and, typically, a longer time frame to express that code. Thus, the complexity of organisms is restricted when the additional length required for new genes raises the gestation time and mutational load beyond that for which the improved life span will compensate. As a result, many niches are still occupied by primitive organisms such as bacteria, while other more intricate environments foster high levels of complexity. An example of an extreme case of r-selection appear to be the *Mycoplasmas*, whose genetic code has shrunk to less than 1×10^6 base pairs due to the nutritionally rich environment in which they thrive. Because of the simplicity of their environment, K-selection plays only a minimal role.

Theoretical arguments and experiments that I report on below have identified a third variable that affects information transmission and the rate at which complexity grows in adapting populations: the *neutrality* of a genome. These *neutral* mutations often occur on a non-critical nucleotide (synonymous substitutions), or else in code that is not currently expressed. That most evolutionary change is neutral with respect to the phenotype is of course well known at least since Kimura's seminal work on the subject [34]. What I suggest here is that there is a pressure, call it ν-selection, that acts upon genomes to *increase* the probability that a mutation is neutral. While such a possible pressure has been mentioned before [48], I demonstrate its importance in detail both theoretically and experimentally.

3.6 *Fitness*: The selective pressures of evolution

In Section 2.2.3, I introduced the term *fitness* (w) as a measurement of the ability for an organism to survive in a given environment. This fitness can be calculated approximately by the time apportioned to the organism (its merit, $\mathcal{M}$) divided by the amount of time it needs to produce an offspring (its gestation time, t_g); that is,

$$w \approx \frac{\mathcal{M}}{t_g}. \tag{3.8}$$

This formula clearly reflects the relative forces of both r- and K-selection. K-selection is the maximization of merit (and thereby lifespan, in computational units) achieved by gaining bonuses from tasks performed, while r-selection is the minimization of gestation time. These two selection pressures are very often contradictory. For every beneficial addition made to an organism's genetic code, there is also the cost of copying that code and expressing it.

Interestingly, there is an additional factor to fitness not represented by either r or K, not normally mentioned in the literature about selection pressure. As the new code has a chance of error, such a mutation can cause harm to the organism as a whole by its presence alone. Clearly then, a better approximation of the fitness of an organism must take into account the probability of the genetic information being correctly transmitted to the offspring. *Fidelity* (as defined in Section 2.7) is the probability for an organism to transmit its genetic code perfectly, but we are more interested here in the *neutral fidelity*: the probability of an organism transmitting correctly the *information* in its genome (introduced in Section 2.8.3) while disregarding mutations in the neutral portions of the genome. This neutral fidelity can be thought of as the fidelity of the quasi-species centered around the wild-type. Taking into account this probability for an offspring to be functionally identical to its parent, we multiply the original fitness (offspring per unit time) by neutral fidelity, to arrive at

$$w_{\text{eff}} = \frac{\mathcal{M}}{t_g} F_\nu. \tag{3.9}$$

Again, this effective fitness w_{eff} is only a good predictor of replication success in a fixed, single-niche environment. Any change to the environment or niche structure will affect the information content of the organisms, possibly resulting in a drop in fitness. I have investigated this fitness metric in 1600 experimental trials, each lasting 50,000 updates, for a total of over 12 million generations (or 15,000 CPU hours) and in every case the above measure of fitness only increases (other than fine-grain fluctuations) over the course of evolution. This seems to imply that it is the combination (3.9) that is selected for, rather than Eq. (3.8).

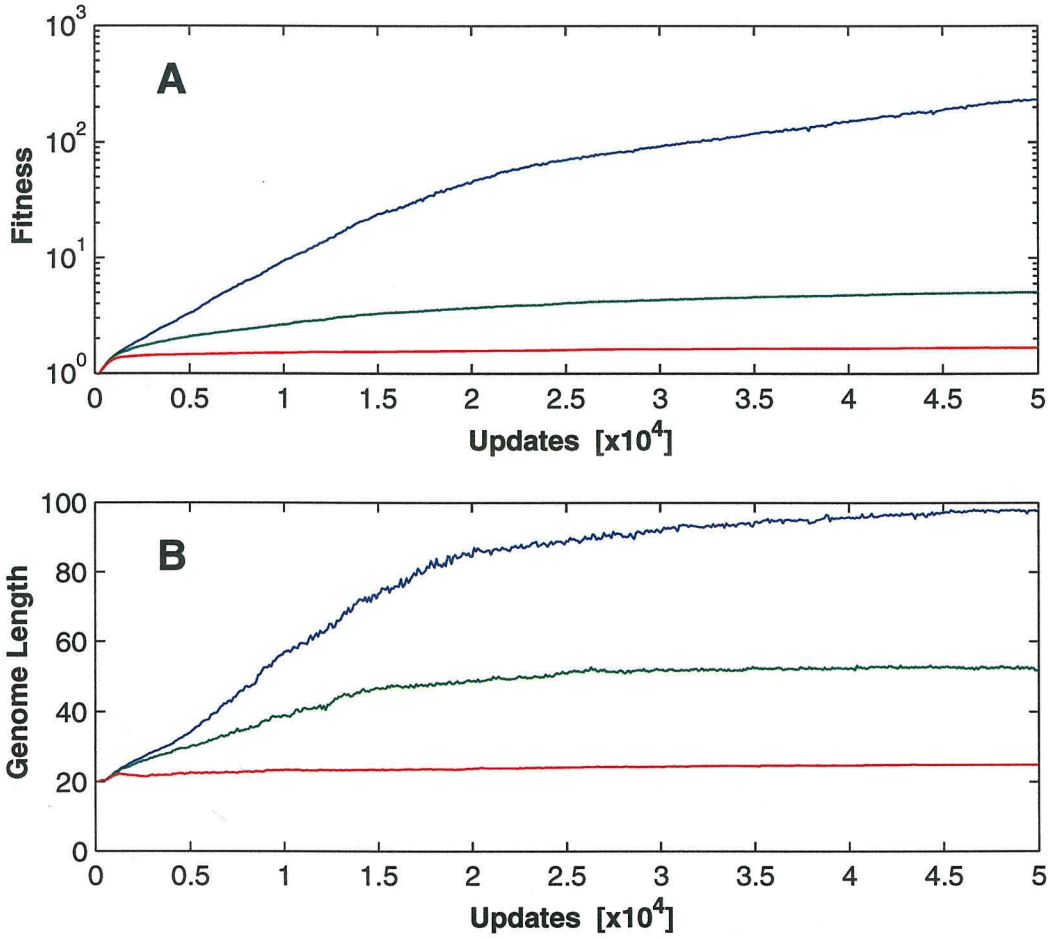


Figure 3.8: Average fitness (A) and average genome length (B) displayed for three experiments at distinct levels of complexity. Set I (red, bottom) is from the simplest environment (no tasks), set II (green, middle) has 12 tasks available, and set III (blue, top) has all 80 tasks present.

Here, I present individual experiments to demonstrate the relative effects of each pressure. Any learning event in evolution requires a tradeoff in its selection; for example, in order for merit to be improved through the completion of a new task, a portion of the lifetime must be spent to perform this task (increasing gestation time). Also, the sites that code for this task will be frozen, and therefore the organism, for a while, will have a lower neutrality. However, as long as the increase in merit outweighs combined effect of these losses, the new genotype will still be selected for.

K-selection is examined in Figure 3.8. Three sets of experiments were performed with initial conditions differing in the complexity of their environments. In Set I, no

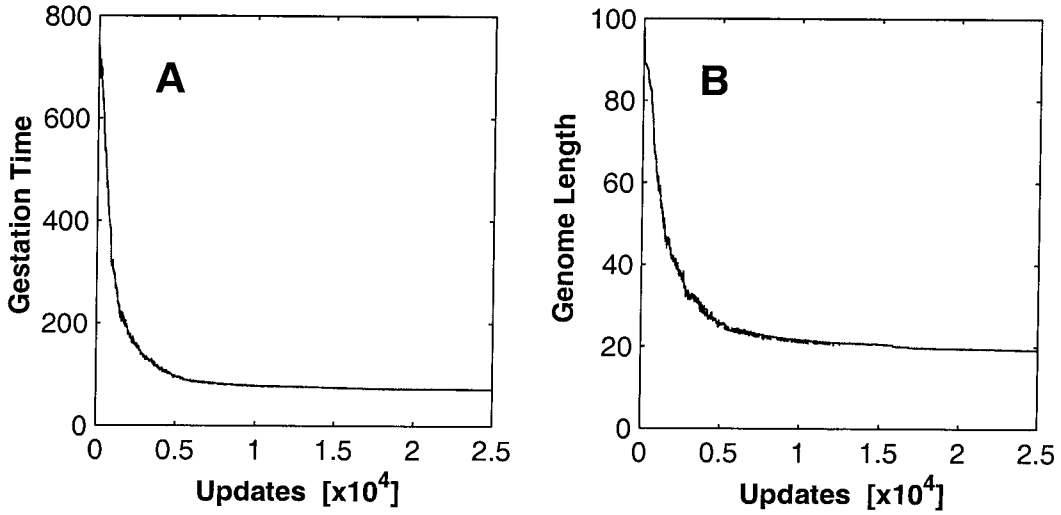


Figure 3.9: Average gestation time (A) and average genome length (B) displayed for set IV. Set IV is the continuation of set III, with all merit contributions deactivated.

tasks are rewarded at all, in Set II, 12 different tasks are rewarded, and in Set III, all 80 logic operations provide a bonus leading to the most complex landscape. The average fitness for the 100 trials of each experiment is displayed in Figure 3.8(A), clearly showing the correlation between environmental complexity (i.e., the potential for K-selection) and the resulting fitness. Figure 3.8(B), in turn, shows the average length of the genomes, indicating that those organisms performing more functions are also significantly longer. In a sense, the results of this experiment are quite trivial: populations rewarded with increased CPU time for performing simple tasks will adapt to perform those tasks. In this way, they maximize their own life expectancy and that of their offspring, clearly indicating the existence of K-selection.

To perform a corresponding test for r-selection, I removed the effects of K-selection by examining a set of trials (set IV) that are seeded with the final populations of set III (all 80 tasks available, above), but in which merit no longer determines the CPU apportioned to an organism. In other words, organisms that adapted to the complex environment of set III were subsequently exposed to the simple environment of set I. Consulting Figure 3.9(A), we see that, indeed, gestation time falls sharply as sections of the genomes that have become meaningless in the simple environment

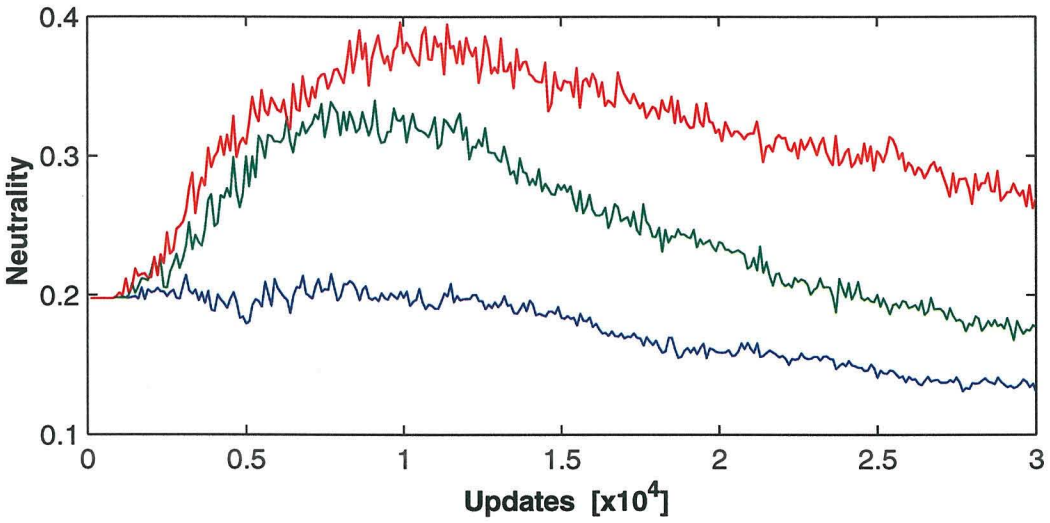


Figure 3.10: Average neutrality is displayed for sets V (blue), VI (green), and VII (red). All three sets are at a fixed length of 80 instructions and have the full range of tasks available. Their mutation rates are 0.5%, 1.0%, and 1.5% respectively. .

are stripped away so that they are no longer copied. In Figure 3.9(B), we witness the corresponding loss in genome length. Such experiments can be compared directly to the experiments by Spiegelman [45] who extracted viral DNA and replicated it *in vitro* and saw a seven-fold decrease in sequence length.

My final tests focus on the effects of ν-selection. This is harder to isolate than the others as it is a weaker form of selection, and acts mostly as a "cost" factor against additional complexity. We examine three new sets (V, VI, and VII), all of which have the full range of tasks rewarded (equalizing the pressure from K-selection between them) and are forced to remain at a genome length of 80 instructions (preventing size changes, and hence minimizing r-selection). The trials in set V were conducted with a copy mutation rate of 0.5% chance of error per copy, Set VI were conducted with a rate of 1.0%, and Set VII were conducted with a rate of 1.5%. As we would expect, those experiments with a higher mutation rate seem to be influenced more by ν-selection (Figure 3.10) as neutrality to mutations is more important when there are more mutations occurring. Note that all neutralities gradually drift downward in the latter half of the experiment. This occurs as their fixed-length genome starts filling up with information, which implies that they are no longer able to incorporate new

tasks without reducing some of the neutrality they have acquired.

Chapter 4 The Evolution of Genetic Organization

In this chapter, I examine the evolution of expression patterns and the organization of genetic information in populations of self-replicating digital organisms. Seeding the experiments with a linearly expressed ancestor, we witness the development of complex, parallel secondary expression patterns. Using principles from information theory, I demonstrate an evolutionary pressure toward overlapping expressions causing variation (and hence further evolution) to sharply drop. I compare the overlapping sections of dominant genomes to those portions that are singly expressed and observe a significant difference in the entropy of their encoding. Finally, I introduce a new model devised specifically for the study of gene organization, and demonstrate the segregation of genes in changing environments.

4.1 Overlapping Genes

Here we look at a fundamental issue to life as we know it; the organization of the genetic code and the differentiation in its expression. DNA is structured into many distinct genes that can be concurrently active, transcribed and expressed in an asynchronous, (i.e., differentiated) manner. Extant living systems have evolved to a state where multiple genes influence each other, typically without sharing genetic material. In all higher life forms, it appears that each gene has its own unique position on the genome, while the transcription products often interact with unique positions "downstream." Those organisms that do exhibit more primitive, *overlapping* expression patterns are mostly virii and bacteriophages [50]. This suggests that genomes with purely localized, non-overlapping genes evolved later on [33].

Upon initial inspection, the reason for a spatially separated (segregated) layout

is not obvious. A modular design may be quite common in artificial coding schemes such as computer programs, but, in fact, only reflects a designer's quest to create human-understandable structures. Evolution has no such incentive, and will always exert pressure toward the most immediate solution given the current circumstances. A more compressed coding scheme, perhaps with overlapping genes, would allow a sufficiently shorter code that would minimize the mutational load and hence be able to preserve its information with a higher degree of accuracy. Moreover, such overlapping regions might be used for gene regulation. Why this is not more common becomes clearer when we observe those examples from nature where these overlapping reading frames do exist, such as DNA phages [50] and eukaryotic viruses [60]. Even in these organisms only some sections of code overlap, but examination of those sections reveals that they contain little variation—almost all of the nucleotides are effectively frozen in their current state from one generation to the next [46, 47]. This occurs because for any mutation to be *neutral* in such a section of genetic code, it must not affect either of the genes that it is within. Further, the neutral mutations in DNA usually occur in the third nucleotide of a codon, as substitutions in that position are often synonymous. When overlapping genes have offset (out-of-phase) reading frames, however, the position of the third nucleotide in one gene maps to the first or second in the other, leaving no such redundancy.

Here, I study the development of genome organization and differentiation in the avida system, extended to allow for the expression of a *second* gene (here, the execution of a second thread via an additional instruction pointer) to occur in parallel. I processed the evolution of 600 populations from a seed program to complex information-processing sequences for 50,000 updates (an average of over 9000 generations). The 600 trials were divided into four sets that differ in initial and environmental conditions. All populations with a genetic basis allowing for the development of multiple *threads* learn to use them almost immediately (see below), but the methods by which this happens are quite distinct and varied. In the next section, I outline the experimental setup used in this study and discuss measures of differentiation. In Section 4.3 I present results obtained with the multiple-expression digital chemistry and compare

them to controls in which no secondary expression was allowed. In Section 4.4 I study the evolution of differentiation for different experimental boundary conditions, while Section 4.5 explores in more detail the organization and development of genes with an example. The final evidence for genetic segregation is presented in Section 4.6 where I compare gene organization between static and changing environments in a specially designed model. I close in Section 4.7 with a discussion of the evidence and conclusions, and issue caveats about applying the lessons learned directly to biochemistry. Later, in Section 5.2 I return to multi-threaded organisms in avida to explore this topic from a more computational perspective.

4.2 Experimental Details

In order to study the evolution of code expression, I extended the avida instruction set to allow for more than one instruction pointer to execute a program's code. Within the biochemical metaphor, the *simultaneous* execution of code is viewed as the concurrent expression of two genes, i.e., the chemical action of two proteins. The first new instruction allows a program to initiate a new expression: `fork-th`. Its execution creates a new instruction pointer ("forking off a thread") that immediately executes the next instruction, while the original thread skips it. Thus, `fork-th` is the rough equivalent of a promoter sequence in biochemistry. This secondary expression is initially rather trivial and leads to redundancy: if the second thread is not sufficiently altered by the instruction following the `fork-th`, it simply executes the same code as the first thread in lock-step. Of course, we are interested in how the organisms use this redundancy as a starting point to *diversify* the expression.

The second new instruction *inhibits* an expression: `kill-th` removes the instruction pointer that executed it, while a third addition, `id-th`, *identifies* which instruction pointer is currently executing the code by placing an identifying number in the BX register. The three commands together are expected to be useful in the *regulation* of expression. In principle, more than two instruction pointers can be generated by repeated issuing of the `fork-th` command, but here we restrict ourselves to a

maximum of two threads so as not to complicate the analysis. In nature, of course, complex genomes express hundreds of proteins simultaneously.

As all experiments begin with a self-replicating program which does not use any of the multiple expression commands, the first question might be whether or not multiple expression will develop at all. In fact, it does almost instantly, as secondary expression (typically in the trivial lock-step mode mentioned earlier) is immediately beneficial, because the second thread effectively doubles the amount of computational time the organisms receive. From here on, differentiation evolves, i.e., the two instruction pointers begin to adapt independently, to express more and more different code. Ultimately one might expect that each pointer executes an entirely different section of code, achieving local separation of genes and fully parallelized execution. The mode and manner in which this separation occurs is the subject of this investigation.

For this study, I obtained several hundred independent experimental trials and controls, testing different experimental conditions.

Set I consists of 200 trials, each initialized with a short (length 20) ancestor that did not have any threading ability. Computational tasks and the use of multiple threads were learned in all cases.

Set II consists of 100 trials similar to set I, but all trials were seeded with a longer (length 80) ancestor.

Set III consists of 100 trials initialized with the length 80 ancestor, but in which programs could not change in size.

Set IV is our control. It consists of 200 trials initialized with the short ancestor. Threading is *not* available in the instruction set, and therefore cannot evolve.

For each of these trials a record is kept of a variety of statistics, including the dominant genotype at each time step, from which we can track the progression of evolution of the population, in particular by studying the details of its expression patterns.

Basic Analysis Metrics

To track the differentiation of the threads, we need to develop a means to monitor the divergence between the two instruction pointers roaming the genome. This is a major advantage of digital chemistries: some of the data collected is impossible to accurately obtain in biochemical systems, and even less practical to analyze.

The following measures and indicators keep track of function-differentiation. In biochemistry, the differentiation of expression can be varied, and includes overlapping reading frames (in-phase and out-of phase), overlapping operons and promoter sequences, and gene regulation. There are no reading frames in our digital chemistry, but it is possible for a sequence of instructions to give rise to a different computation depending on the state of the thread that is executing it—in particular if one gene contains another (as is common in overlapping biochemical genes [67]). Also, thread-identification may lead one thread to execute instructions that are skipped by the other thread, and threads may interact to turn each other on and off: a case of digital gene regulation. All such differentiation however has to evolve from the trivial secondary expression discussed earlier, and we consequently need to monitor the divergence of thread-execution with suitable measures.

Thread Distance is the metric I use to determine the spatial divergence of the two instruction pointers. This measurement is the average *distance* (in units of instructions) between the execution positions of the individual threads. If this value becomes high relative to the length of the genome, it is an indication that the threads are segregated, executing different portions of the genome at any one time, whereas if it is low, they likely move in lock-step (or sightly offset) with nearly identical executions. Note, however, that if two instruction pointers execute the code offset by a fixed number of instructions, but otherwise identically, the thread distance is an inflated measure of function differentiation, as both threads do behave identically.

Code Differentiation distinguishes execution patterns with differing *behavior*. A count is kept of how often each thread executes each portion of the genome; code differentiation is the fraction of sites in the genome for which these counts differ

between threads. Thus, the ordering of execution (time-delay) is irrelevant for this metric; only whether the code ends up getting executed a different number of times by one thread vs. the other is important.

4.3 Single Expression vs. Multiple Expression

Let us first examine adaptability as measured by the average increase in fitness for both multiple and single expression chemistries. In Figure 4.1(A), the fitness is averaged for the 200 trials[1] that were seeded with small ($\ell = 20$) seed sequences and no size constraint (sets I and IV). While the average increases relatively smoothly in time, each individual fitness history is marked by periods of stasis interrupted by sharp jumps, giving rise to a "staircase" picture (refer back to Figure 2.4) reminiscent of the adaptation of *E. coli* [24]. During adaptation, the sequence length increases commensurately with the acquired information, as shown in Figure 4.1(B).

Clearly, the trials in which multiple expression is possible adapt more *slowly* than the single-expression controls, a behavior that may appear at first glance to be paradoxical as the only difference in the underlying coding of the multiple expression trials is an *increased* functionality. However, as I noted in Section 2.8.3, the neutral fidelity of an organism directly determines the fraction of its offspring that are viable. As this value is inversely correlated to the length of the genome, there is a pressure for the genomes to evolve toward shorter length. Normally, this pressure is counteracted by the adaptive forces which require the organism to store more information in its genome, requiring increased length. Overlapping expression patterns allows this adaptation to occur while minimizing the length requirement.

The pitfalls of compacting so much information into the same portion of the genome are illustrated in Figure 4.2 where I plot the average genomic diffusion rate D_g for both chemistries. It is evident in this graph that initially both sets of experiments

[1]Each trial was seeded with a single ancestor, which quickly multiplies to fill the maximum number of programs in the population, set to 3,600 for these trials. The populations were subjected to copy mutations at a rate of 7.5×10^{-3} per instruction copied, and a rate of 0.05 single insertion or deletion mutations per gestation period.

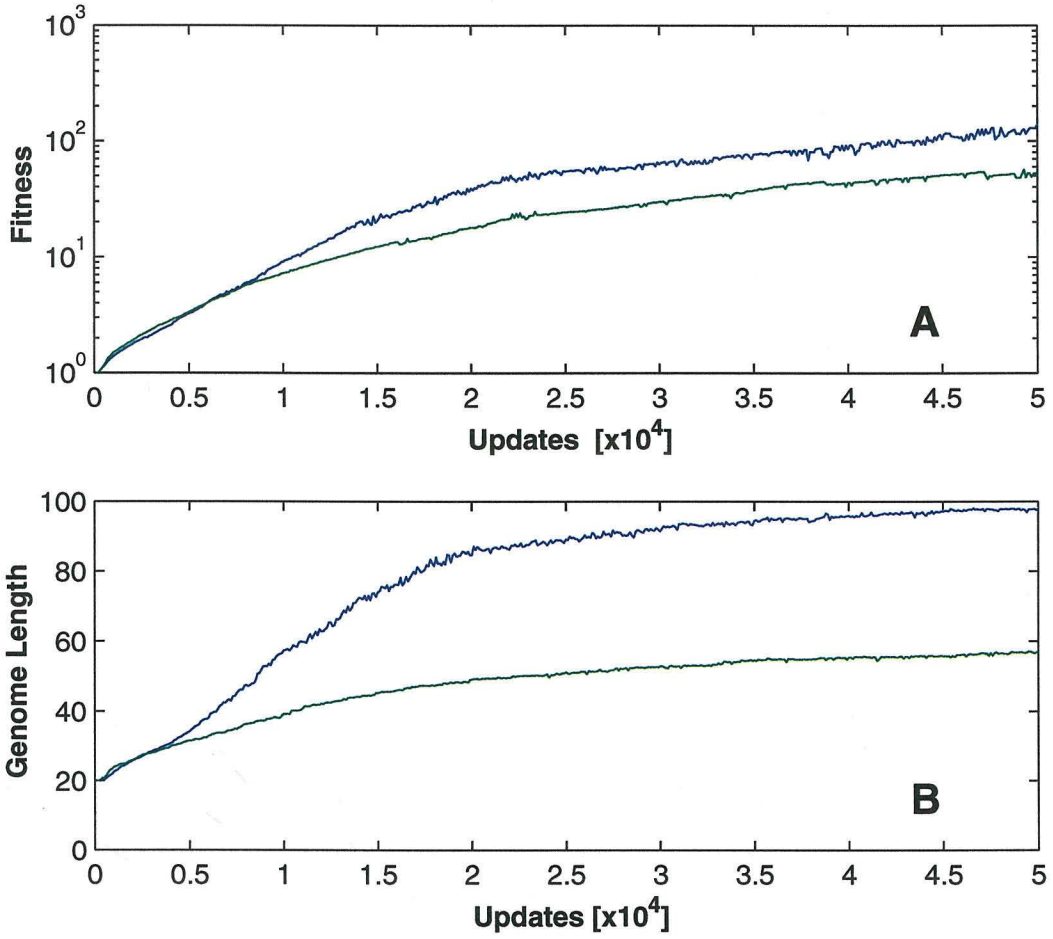


Figure 4.1: (A) Average fitness as a function of time (in updates) for 200 populations evolved from $\ell = 20$ ancestors and (B) their average sequence length, for the single expression chemistry controls (blue line) and the multiple expression chemistry (green line).

explore genetic space at a comparable rate, but at approximately 5000 updates (on average) the diffusion rates diverge markedly, followed by a corresponding divergence in the fitness of the organisms (that a higher diffusion rate leads directly to higher fitness in an information-rich environment is shown in [9]). Investigating the course of evolution further, we can see that it is precisely at this point that the differentiated, yet overlapping, use of multiple threads is typically established.

To further implicate overlapping expression in reduced adaptation for the populations, let us consider (as was done in Ref. [46] for the bacteriophage $\Phi X174$) the substitution rate of instructions for overlapping versus non-overlapping genes. The

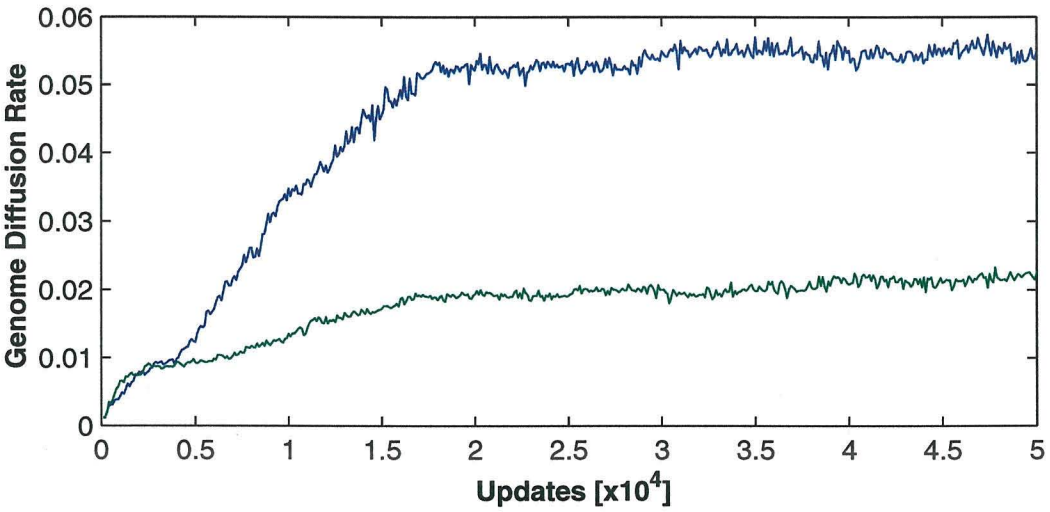


Figure 4.2: Average genomic diffusion rate as a function of time (in updates) for 200 populations evolved from $\ell = 20$ ancestors, for the single expression chemistry controls (blue line) and the multiple expression chemistry (green line).

substitution rate in avida is equal to the neutrality (at equilibrium). We find the *substitution suppression* (the neutrality in multiply expressed code divided by the neutrality in singly expressed code) to be between 0.53 and 0.57 for the three sets of trials (Table 4.3). In other words neutrality is suppressed by 0.43 to 0.47. This is in the same range as the suppression ratio of between 0.4 and 0.5 observed in bacteriophages [46]. When the instruction pointers do adapt independently and the threads differentiate, neutrality in overlapping regions is compromised. The instructions within these sections of overlapping code are comparatively "frozen" into their state.

Table 4.1: Average neutrality of the final dominant genotype: multiply-expressed code (column 1), singly expressed code (column 2), and their ratio (column 3), for 200 populations grown from $\ell = 20$ ancestors (variable length) [set I], 100 populations grown from $\ell = 80$ ancestors (variable length) [set II], and 100 populations grown from $\ell = 80$ ancestors (constant length) [set III].

Set	ν_{mult}	ν_{single}	ratio
I	0.109	0.202	0.539
II	0.197	0.346	0.569
III	0.082	0.145	0.566

4.4 Evolution of Differentiation

Let us now track the evolution of differentiation in more detail. First, I address the *de novo* evolution of multiple expression, i.e., the development of multi-threading from linear execution. In initial experiments with the tierra system, usage of multiple threads would not evolve spontaneously, but hand-written programs that had secondary expressions would evolve toward an increased level of multiple expression [64]. More recently, experiments were carried out within a network version of the tierra architecture, which showed that a program that used different instruction pointers to execute different genes would not lose this ability [59]. The failure of multiple expression to evolve spontaneously in that system can be tracked back to problems with tierra's digital chemistry and the lack of an information-rich environment [53].

Within avida, the ability to use more than a single thread begins to develop within the first 5000 updates and is common after about 10,000 updates, depending on the experimental boundary conditions. Figure 4.3A shows the (averaged) percentage of a program's lifetime in which more than one thread is active, for the populations of set I (blue line), set II (green line) and set III (red line). It is apparent that multiple expression develops more readily in smaller genomes, due to the fact that the logistics are less daunting.

In panels B and C of Figure 4.3 I show two indicators of differentiation (defined earlier), the thread distance and the code differentiation, respectively. The thread distance appears to be sensitive to the experimental starting conditions, as set II and set III show a value over twice that of set I. This is due to the small size of the ancestor used in set I: as that ancestor develops threading quickly, it loses adaptability earlier and lags both in average fitness and average sequence length. In fact, the averages for set I are dragged down by a significant percentage of the trials that become stuck in an evolutionary dead-end shortly after developing threading. Set II and III, which were seeded with an ancestor of length $\ell = 80$, did not suffer from this. Figure 4.3(C) shows the *code differentiation*, i.e., the fraction of code that is executed differently by the two threads. This fraction is less dependent on experi-

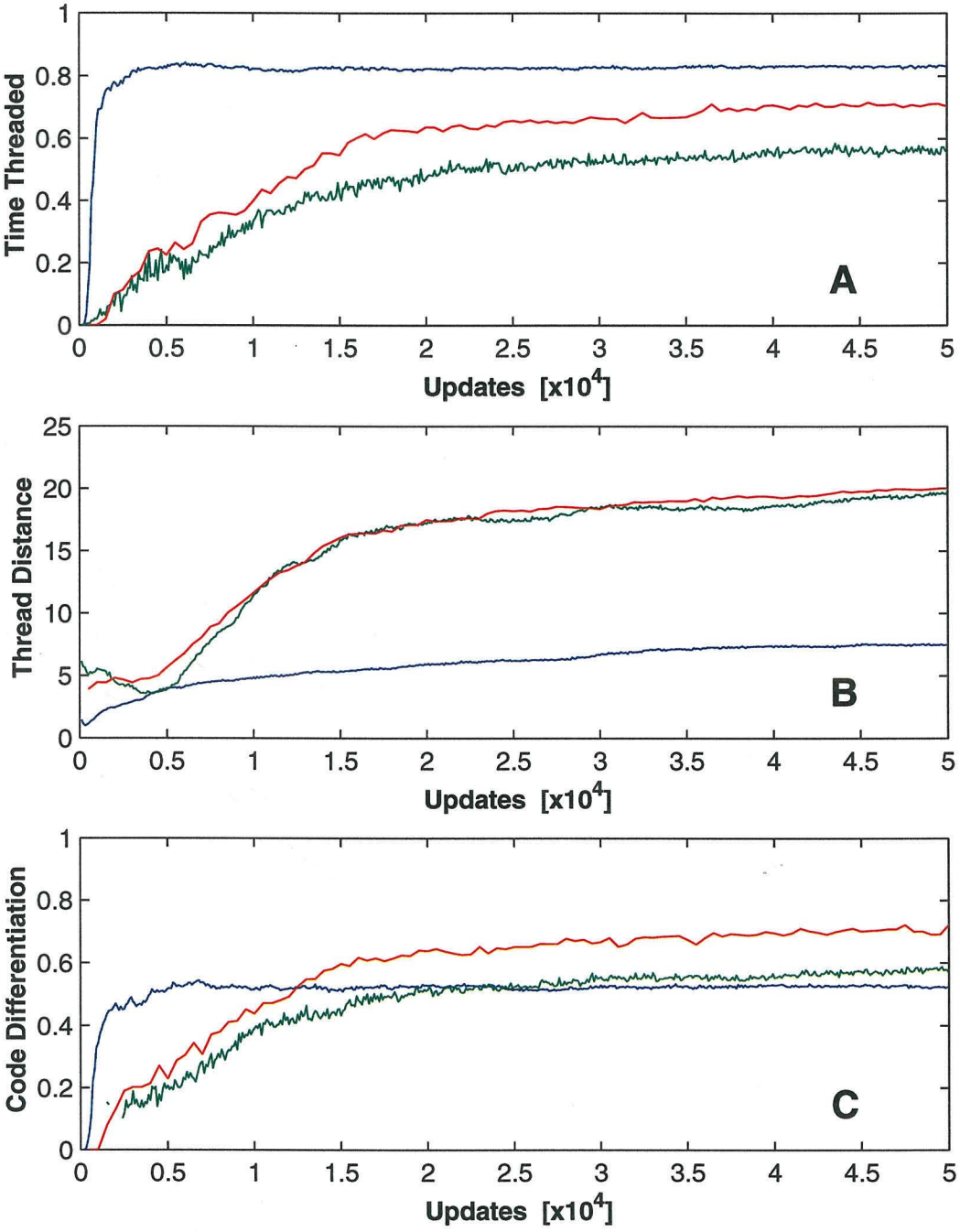


Figure 4.3: Differentiation measures. (A) Average fraction of lifetime spent with secondary expression, as a function of time (in updates), (B) average thread distance, (C) average code differentiation. Set I (blue line), set II (green line), and set III (red line).

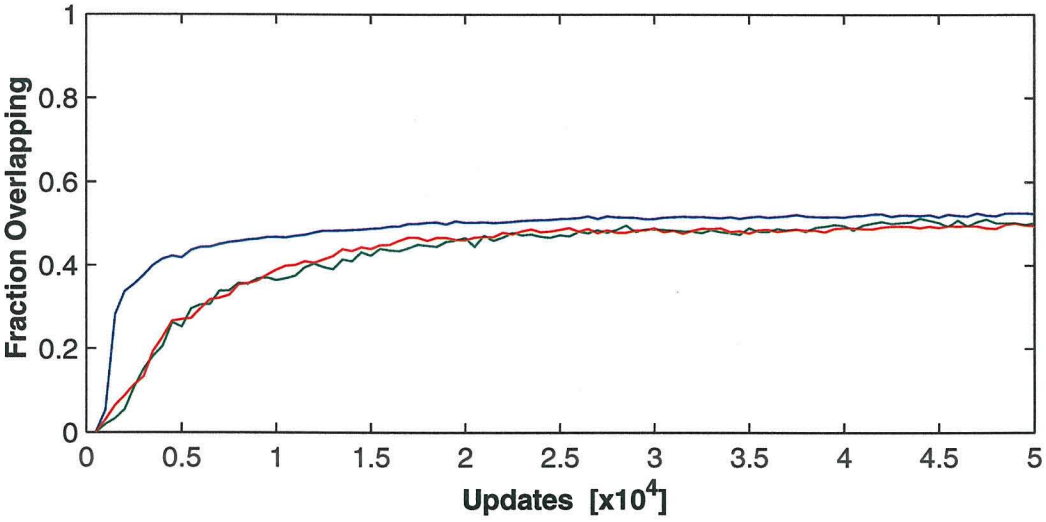


Figure 4.4: Average fraction of doubly expressed code for the three experimental sets. Blue (top) line: set I, green line: set II, red line: set III.

mental conditions, and the genomes appear to develop toward 0.5. Note, however, that this measure cannot accurately reflect functional differentiation, which is more subtle than threads executing particular instructions a different number of times. For example, two threads that execute a stretch of code in an identical manner but that start execution at different points "upstream" may calculate very different functions, and thus have quite different behaviors. This difference will thus be underestimated. While the preceding graphs seem to indicate that differentiation stops about halfway through the trials, this is actually not so, as the more microscopic analysis of the following section reveals. Rather, the use of these differentiated sections of code is optimized and integrated with one another to achieve greater functionality. Finally, Figure 4.4 shows the evolution of the fraction of code that is executed by multiple threads.

As anticipated, this fraction rises swiftly at first, but then levels off, as it is not advantageous to multiply express all genes (see below). However, we might expect that the fraction would start to decline at some point, when the organisms develop the ability to localize genes and use independent instruction pointers for each of them. This trend is not apparent in Figure 4.4, presumably because there is no cost associated with the development of overlapping expression. In the natural world,

such an environmental over-fitting would cause a negative impact upon the organism anytime a fluctuation in the environment occurred (due to the reduced neutrality and adaptability). Likewise, viruses develop in a more consistent environment than most higher organisms; they become very specialized to make use of specific hosts and the environments within them. In a virus, specific genes remain segregated (presumably those that need to be adaptable for the evolutionary arms race with its host), but the rest of the sequence often has two or three genes overlapped throughout. For example, in $\Phi X174$, only 4 of its 11 genes are segregated. Finally, both avida organisms and viruses have no error protection or correction mechanisms, thus the effects of mutation provide a much stronger incentive to decrease code length by whatever means necessary.

4.5 Evolution of Genetic Locality

To understand how evolution is acting upon programs harboring multiple threads, let us look at exactly what is being expressed. We can loosely characterize all organisms by tracking three categories of genes. They are "self-analysis" (*slf*), "replication" (*rpl*) , and "computation" (*cmp*). To follow the progression of these genes through time, I examine a sample experiment from the environment in which size-altering mutations are strictly forbidden (a trial from set III). This limitation is enforced to facilitate alignment, allowing us to cleanly study the functionality of the organism and the location of its genes. Similar analyses have been done with all sets, showing comparable behavior.

In Figure 4.5(A) we can follow the per-site entropies for each locus as a function of time, as described in Section 3.3. Positions are labeled 1 to 80 on the vertical axis, while time proceeds horizontally. A color coding has been employed to denote the variability of each locus, where red denotes more variable ("hot") positions and blue, fixed ("cold") positions. Figure 4.5(B) shows which portion of the code is expressed by which pointer, by two pointers simultaneously, or not at all.

The first gene *slf* uses pattern matching on `nop` instructions to find the limits

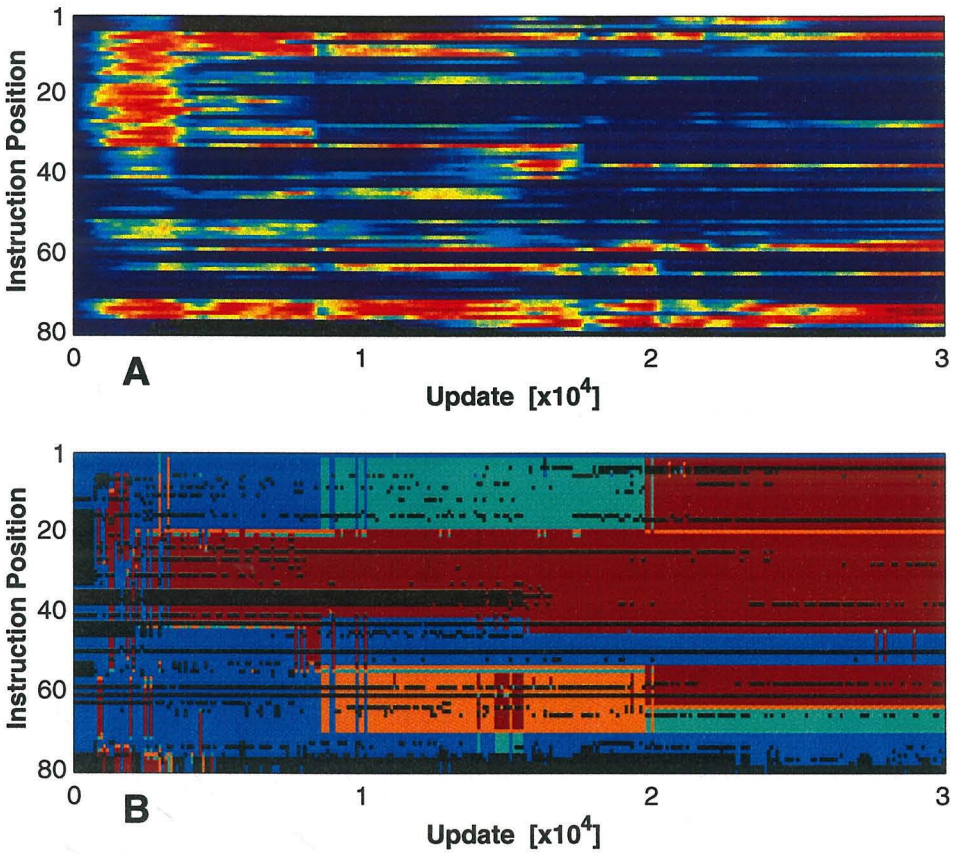


Figure 4.5: (A) Per-site entropy for each locus as a function of time for a standard (set III) trial. Random (variable) positions with near-unit per-site entropy are red, while "fixed" instructions with per-site entropy near zero are dark blue or black. (B) Thread identification within a genome. Black indicates instructions that are never directly executed, blue denotes instructions executed by a single thread when no other thread is active, while sections that are executed by a single thread while another thread is executing elsewhere are colored in green and orange. Finally, sections with overlapping expressions are red.

of its genome and from that calculate its length. This value is used for *elongation* (via the command `alloc`), which adds empty memory to the genome and prepares it for the expression of the replication gene. There are two interesting points to note about the evolution of *slf*. First, there are many methods by which the organism can determine its own genomic length, so this gene tends to vary widely. Most of the time the organism retains pattern matching techniques, but matches different portions of the code. However, often an organism shifts to purely numerical methods, performing mathematical operations upon itself that yield the genome length "by accident," and

thus making the organism brittle to size changing mutations (in those experiments where size change is allowed). The other evolutionary characteristic of this gene is that there is no benefit in expressing it multiple times as it can be applied only once during the gestation cycle. Looking at Figure 4.5, the *slf* gene initially spans from lines 44 to 61 plus the first four lines and last four lines of the genome, which are boundary markers fashioned from `nop` instructions. The first major modification to the *slf* gene occurs near update 3000. The pattern used to mark the limits of the genome is a series of four `nop-A` instructions. As a newly allocated genome has all of its sites initialized to `nop-A`, the genome is re-organized such that these lines are no longer copied. This reduces the possibility of variation in these sections of code to zero. This is apparent in Figure 4.5(A) as the positions of these limit patterns become fully black indicating vanishing entropy.

The *slf* gene is continuously undergoing minor changes as it becomes more optimized to require fewer lines of code to perform its function. Near update 13,000 it shifts dramatically and is replaced by one in which size is calculated using only the final boundary markers. The distance from the gene to the final marker is determined, and then manipulated numerically to obtain the number that is the size of the organism. Looking at the first four lines of Figure 4.5(A) near this update, we can see that they are slowly phased out and increase in entropy as they are no longer as critical to the organism's survival. Finally, the size of the pattern marking the end boundary of the organism is shortened until it becomes only a single line. By the end of the evolution shown, the *slf* gene only occupies lines 48 through 56. Note that all of these lines are only expressed a single time.

The next gene under consideration is the actual replication gene *rpl*. This sequence of instructions uses the blank memory allocated in the self-analysis phase and essentially consists of a "copy-loop" that moves line by line through the genome, copying each instruction into the newly available space. When this process is finished, it severs the newly created copy of itself which is then placed in an adjacent lattice site. These dynamics spawn off a new organism that, if the copy process was free of mutations, is identical to the parent. In Figure 4.5, the organism being tracked has its replication

gene on lines 65 to 71 until update 24,000 at which time this gene is reshuffled and takes up an additional line, thus becoming more efficient by "unrolling" its copy-loop. What this means is that it is now able to copy *two* lines each time through the loop. From the dark blue color of these lines, it is obvious that they have low entropy, and are therefore highly conserved. The copy-loop is a fragile portion of code, critical to the self-replication of the organism, yet we do see some evolution occurring here when multiple threads are in use. Often the secondary thread will simply "fall through" the copy-loop (not actually looping to copy the genome) and move on to the next gene, while the other thread performs the replication. However, sometimes the two threads will evolve together to use the copy loop in different ways, with each thread copying *part* of the genome. In Figure 4.5, most of the *rpl* gene is executed by only one thread. The *rpl* gene is followed by junk code that, while executed sporadically, does not affect the fitness in any way (as evidenced by the light shading in Figure 4.5A for these lines).

The most interesting of the genes is the computation gene *cmp*. The ancestor does not possess this gene at all, so it evolves spontaneously during the adaptive process. The *cmp* gene(s) evolve uniquely in each trial, enabling the organisms to perform differing sets of tasks. There are, however, certain themes that we see used repeatedly whereby the same section of code is used by both threads, but their initial values (i.e., the processing performed thus far on the inputs) differ. The unique "pre-processing" step causes this section of code to perform radically different tasks, allowing the multiple use of the same code to be beneficial. Portions of this algorithm that might have some neutrality for a single thread of execution will now be conserved due to the added constraints imposed by a secondary execution. The size of *cmp* grows during adaptation as a number of computations are performed, and the gene is almost always expressed by both threads as this is always advantageous for an individual organism. In Figure 4.5, the *cmp* gene stretches from line 1 to line 42 (at update 30,000), while it is considerably smaller earlier. Furthermore, the genome manages to execute the entire gene with both threads (the transition from single expression of part of *cmp* to double expression is visible around update 20,000). This gene ends up being expressed

many times (as the instruction pointers return to this section many times during the organism's lifetime). All in all, 17 different logical operations are being performed by this gene.

By the end of the evolution tracked in Figure 4.5, most of the genes appear to occupy localized positions on the genome. The *cmp* gene (red sections in Figure 4.5B) is revisited many times by both threads with differing initial conditions for the registers, allowing the genome to maximize the computational output. In the meantime, those sections have become fixed (their variability is strongly reduced) as witnessed by their dark blue shading in Figure 4.5A.

4.6 Genetic Segregation

Our analysis of organisms that exhibit overlapping expression patterns has led us to the theory that this is a form of overfitting, and only occurs in those environments that are relatively static. To test this theory in its simplest form, I have constructed the Auto-Adaptive Genetic Organization System (aagos) to simulate the evolution of genetic organization, removing many of the complexities involved in avida and focusing only on the evolution of gene positions.

4.6.1 The Aagos Model

Aagos is loosely based on Kauffman's $N-k$ model [31, 32]. Each organism is composed of a bitstring (of length ℓ) comprising n_g specific genes, each of length ℓ_g.

Figure 4.6 displays an example organism. Its genome (along the left most edge of the figure) is $\ell = 16$ bits long. It harbors $n_g = 4$ genes (labeled A through D) each of which corresponds to $\ell_g = 8$ bits of the genome. The blue bar in the figure marks the range of bits that determine the state of each gene. For example, gene A is the first 8 bits of the sequence—10110001.

The fitness contribution of each individual gene is uniquely determined by the bit-sequence associated with it. Each gene has a *fitness chart* associated with it; the sequence is indexed into this chart to determine the corresponding fitness. The

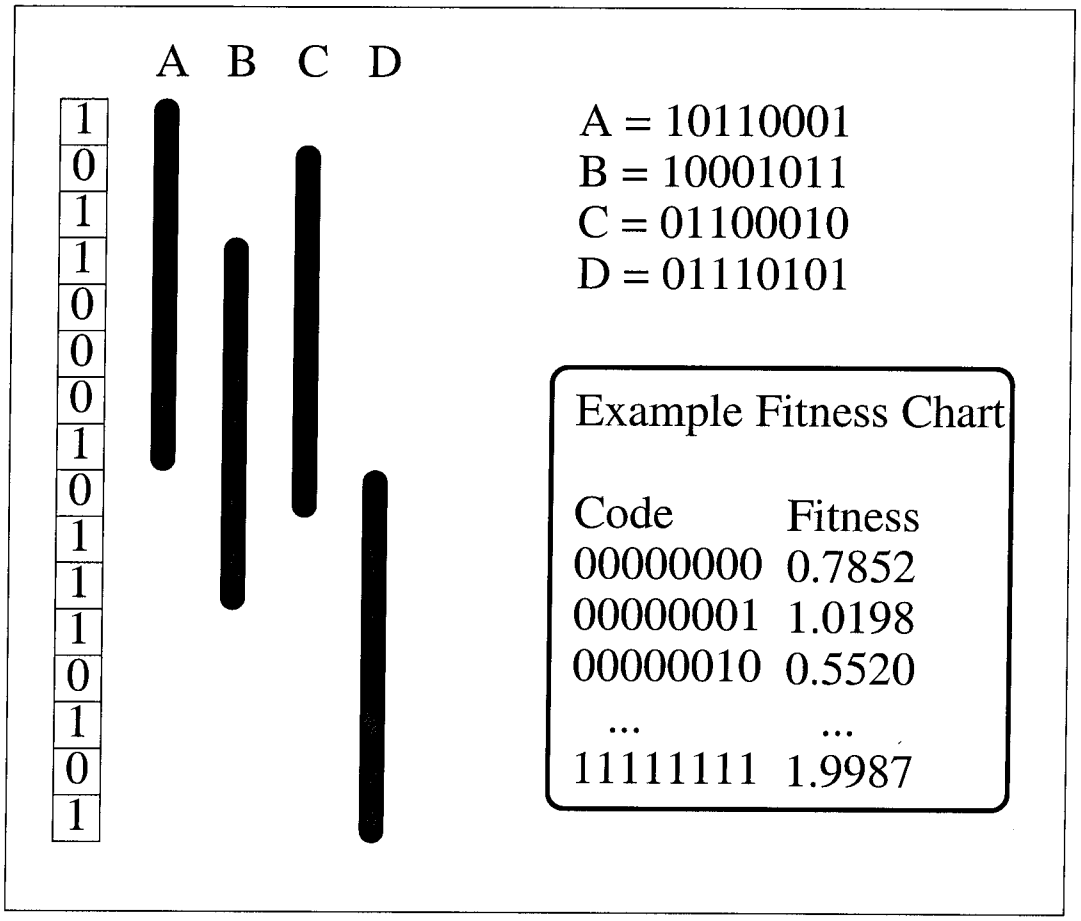


Figure 4.6: An example aagos organism.

entries in this chart are generated by the formula $f_i = 2^r$ where r is a random value evenly distributed between -1 and 1. Therefore, each entry ranges between 0.5 and 2.0 with an equal probability of being above or below one. The individual entries in this fitness chart are uncorrelated such that any change to a gene's sequence will alter its fitness contribution. The fitness of an organism is the product of its individual gene fitnesses.

In the example given, if genes A through D had the respective fitnesses of 1.2, 0.9, 1.6, and 1.1, then the overall fitness of the organism would be $1.2 \times 0.9 \times 1.6 \times 1.1 = 1.9008$.

A mutation in the first bit of the genome would cause gene A to change its contribution. If this flip happened to improve A, it could be selected for. On the other hand, a mutation in the fourth bit would alter the contributions of all three

genes A, B, and C, which are overlapping in this section.

Aagos is an auto-adaptive system: the fitness of each organism determines that organism's replication rate, but never guarantees the perfect transmission of its genome into the next generation. Thus, if one organism has twice the fitness of another, it will produce an offspring in half the time. Like avida, aagos has a fixed population size, so when a new organism is born, the eldest is removed. This encourages high-fidelity information transfer into the next generation: organisms with a low fidelity will have fewer viable offspring.

Several types of mutations exist in this system. The first are *copy* mutations. Every time an organism produces an offspring, each line has a fixed probability to be mutated. This provides a pressure for organisms to keep a short genome and minimize the target area for mutations. Next, there are *insertion* and *deletion* mutations that occur with a fixed probability per genome. When an instruction is inserted or removed, all genes in subsequent positions in the genome are shifted such that they remain associated with the same bit sequence. These two mutations allow the genomes to adjust their length as selection dictates.

Finally, there are *gene-position* mutations that will shift the location of any particular gene. The position of each gene is inherited by the offspring, and the evolution of these positions (the genetic organization) is what we are most interested in.

4.6.2 Experiments with Aagos

The initial experiments with aagos test the default organization of genes given a static environment. There are conflicting pressures—copy mutations encourage genomes to be as short as possible, but this requires overlapping genes. When genes overlap, mutations at any shared site affect *all* genes at that site. For the genes to evolve independently (each to their own maximal fitness), they must have as little overlap as possible.

Figure 4.7 displays the average number of sites used by genomes (sequence length) from two different experiments, each averaged over 50 trials. Organisms in both

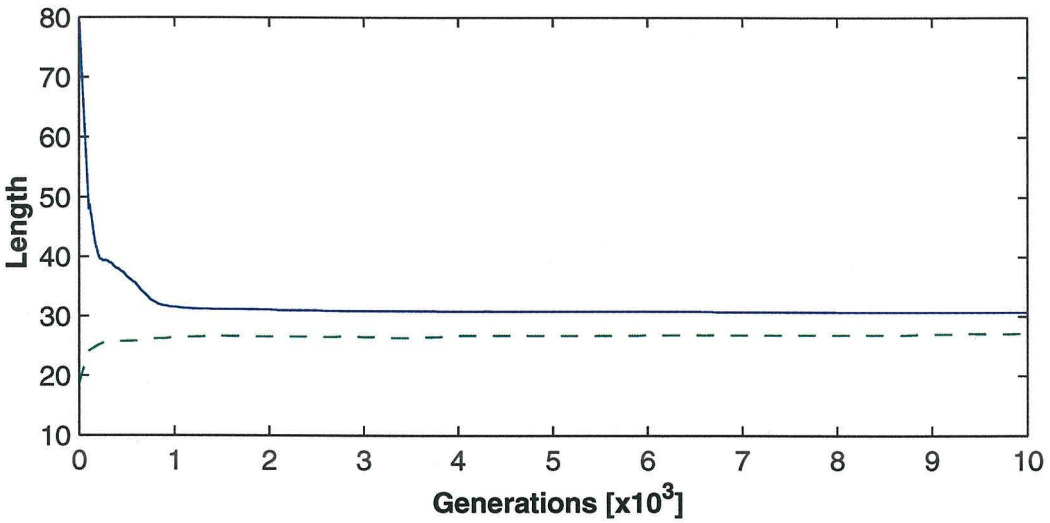


Figure 4.7: Genome length from 2 starting conditions (averaged over 50 trials each) in a static aagos experiment. Set one (solid line) began with length 80 genomes, while set two (dashed line) had length 20 ancestors.

of these experiments consisted of 10 genes, each 8 bits long. The copy mutation probability was 0.01 per line, insertion and deletion probabilities were both 0.02 per organism, and the probability of a gene shifting position was 0.005 per gene. The experiments differed in the starting conditions only: the first set began with an 80 bit ancestral genome while the second had a 20 bit ancestral genome.

These aagos experiments evolved overlapping genes, as expected from equivalent studies in both avida [51] and viruses [46]. The maximal genome length (assuming all genes to be fully segregated) would be 80 bits, while if they were perfectly overlapping, it would only be 8 bits (their individual length). In this case, both experiments converged to similar values; their average lengths evolved to 27 and 31 bits, respectively.

Next, I test the hypothesis that a changing environment will prevent overfitting. Intuitively, if two genes are overlapping in a fit state and the environment is altered to make one of them less fit, the suboptimal gene will be under pressure to change. However, any change will also cause the overlapping gene to alter its fitness contribution as well, likely causing it to lose its optimized state. Only through gene segregation can a genome adapt quickly to a turbulent environment.

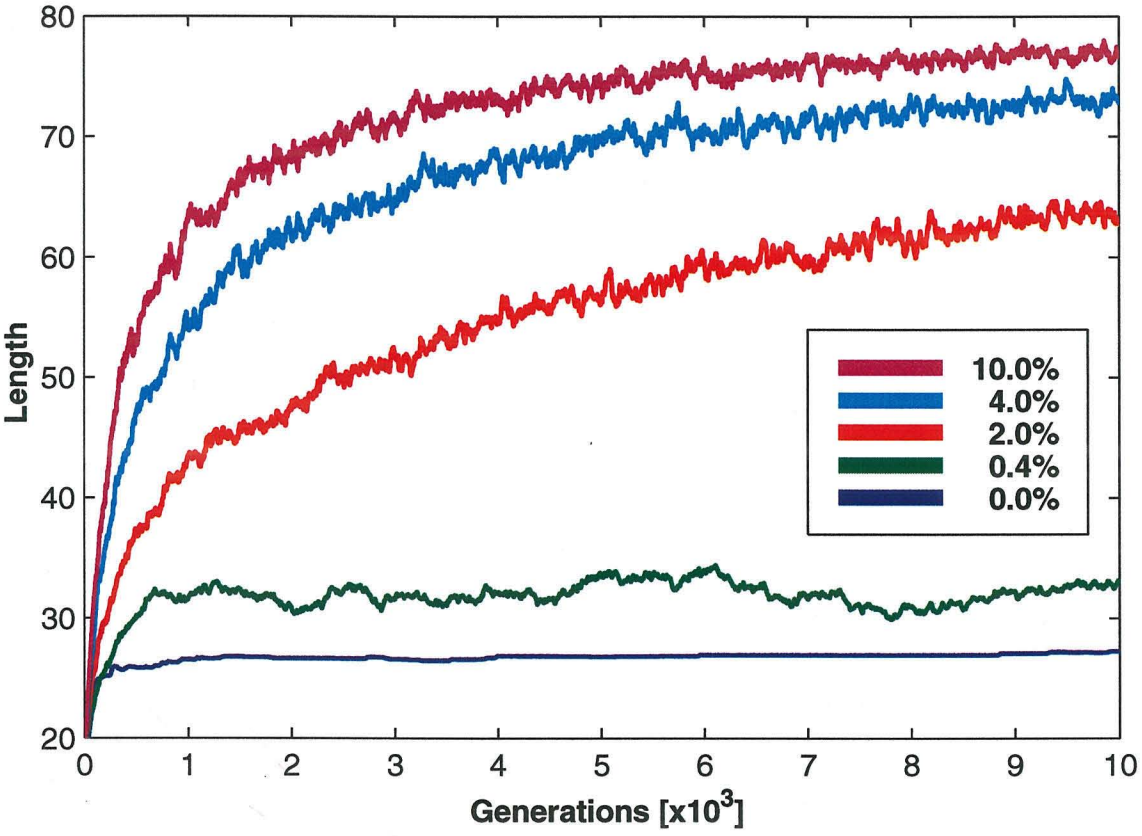


Figure 4.8: Genome length in 5 different changing environments in aagos (each averaged over 50 trials).

Fluctuating environments are easy to implement in aagos by setting up the fitness charts such that at each generation there is a fixed probability that a fitness value will be randomized. The higher that probability, the faster the environment will be changing.

In Figure 4.8, I show the progression of average sequence length with time for environments with a fluctuation rate of 0, 0.004, 0.02, 0.04, and 0.1, all initialized with a length 20 ancestor. As expected, as the rate of environmental fluctuation is increased, so too is the segregation of the genes in the genome (as seen from the increased sequence length).

4.7 Discussion and Conclusions

The path taken by evolution from simple organisms with few genes toward the expression of multiple genes via overlapping and interacting gene products in complex organisms is difficult to retrace in biochemistry. Artificial Life may help to understand some key principles in the development of gene regulation and the organization of the genetic code. We have examined the emergence and differentiation of code expression *in parallel* within a digital chemistry, and found some of the same constraints affecting multiply expressed code as those observed in the overlapping genes of simple biochemical organisms. For example, multiply expressed code is more fragile with respect to mutations than code that is transcribed by only one instruction pointer, and as a result evolves more slowly. During evolution, two constraints are notable: the pressure to reduce sequence length in order to lessen the mutational load, and the pressure to *increase* sequence length so as to be able to store more information. Simple organisms can give in to both pressures by using overlapping genes, gaining in the short term but mortgaging the future: the reduced evolvability condemns such organisms to a slower pace of adaptation, and exposes them to the risk of extinction in periods of changing environmental conditions.

This effect is clearly visible in the evolution of digital organisms, as is a trend toward multiple expression of as much of the code as possible. This latter feature is *not* universal, but rather due to the fact that organisms in avida exist in a static environment, and multiple expression is cheap (i.e., no additional resources are used to express more code). In a more realistic chemistry, this would not be the case: fluctuations in the environment would select against those organisms that had a reduced ability to adapt, placing an implicit cost to properly balancing the use of a secondary expression. Also, in such an environment where multiple expression is not free, we can expect more complex gene regulation to evolve as genes would be turned on only when needed.

Still, under certain extreme conditions I believe that multiple overlapping genes are a standard path that any chemistry would follow. Such organisms can be rescued

either by the development of error-correction algorithms, or an external change in the error rate. In either case, a drastic reduction of the mutational load would enable the sequence length to grow and the overlapping genes to segregate (for example by gene-duplication). Such a drastic event might thus give rise to an explosion of diversity followed by innovation.

Chapter 5 Evolution of Computer Languages

If current trends in the growth of computer hardware and software continue, we can speculate that in the not so distant future we shall be faced with a crisis in the development of complex operating systems and the programs designed to run under them. With computer codes beginning to exceed many millions of lines, intended to interact with other software written independently, such systems are becoming effectively untestable, and their behavior unpredictable. New paradigms of code generation, testing, and assembly may have to borrow principles from nature, by interpreting living organisms as complex machines that are constructed from—and operating on—"software" (the genome) several billions of lines long, assembled from various sources and operating in a remarkably robust and fault-tolerant manner.

Early experiments in *Genetic Programming* [35] and *Evolutionary Programming* [27] focused on the evolution of tree-like structures in which each "atom" already has a functionality related to the problem to be solved. Also, *Genetic Algorithms* [30] can be viewed as a tool to evolve specialized problem-solving code. In both instances, the brittleness of the coding—the tendency of evolved code to easily break under mutations—seems to go hand-in-hand with the specialization of the atomic instructions used, and therefore as the price to pay to ensure fast adaptation.

5.1 Evolvability and Robustness in Computer Languages

In this section, I show that evolvability and robustness can be achieved simultaneously, within the biologically-inspired, auto-adaptive paradigm discussed in this thesis.

Unlike in Genetic Programming and Genetic Algorithms, in systems of *self*-replicating computer code (auto-adaptive systems) robustness is selected for, as there is a premium for high-fidelity information transmission into the next generation. Rather than settling for a single instance of such an "artificial chemistry," I explicitly test elements that influence robustness and adaptability and take the first steps toward their systematic exploration. In particular, I study the role that evolution plays in generating populations of programs that react to changing and noisy environments in a predictable manner, while still maintaining evolvability.

5.1.1 Exploring Artificial Chemistries

The original experiments in self-replicating codes performed by Rasmussen [55] seemed to rule out the evolution of programs because the programs turned out to be extremely fragile. Self-replicating digital organisms written in the *Redcode* language could not survive even miniscule amounts of stochasticity in the replication process, leading to "dying" populations. Thus, *Redcode* represents a computationally effective chemistry that, however, does not survive mutations (i.e., is extremely brittle).

An important step was taken by Ray [56] who recognized that the brittleness of *Redcode* is primarily due to the *argumented* instruction set: independent mutations in the instruction and its arguments are unlikely to lead to a meaningful combination. In experiments with a version of *Redcode* designed to run on the avida system I determined that, in fact, over 99% of all non-trivial[1] mutations are deleterious in this architecture, even though information *can* be preserved in large populations if programs are protected during the replicative process. Evolvability in this language is severely limited.

Write-protection *and* an argument-free instruction set led to the first successful evolutionary experiments with assembly-like code in Ray's tierra world. Rather then using arguments for direct addressing, Ray's instruction-set relied on patterns of instructions whose execution has no effect (no-operation, or "nop", instructions) for

[1] A trivial mutation is defined as one that affects only non-executed portions of code.

relative addressing. Such instructions play a role analogous to untranslated binding sites in biochemical code (e.g., promoter sequences). Self-replicating programs survive well in the tierra world and can adapt to user-specified fitness landscapes and grow in complexity [1].

The artificial chemistry of the avida world differs from the coreworld and tierra systems: In avida, each program has a natural protection as it occupies a unique location on a two-dimensional lattice that other programs cannot directly access[2]. Consequently, interactions between programs are local, and organisms cannot corrupt the genome of other organisms by overwriting, as happens in Coreworld. This feature contributes significantly to the robustness of programs in this world, as it ensures that a program's fidelity does not depend on its neighbors.

Which aspects of the instruction set's design are directly responsible for evolvability and robustness is of fundamental importance [57] if dedicated, evolvable instruction sets are to be designed in the future. Here, I systematically test (within the avida system) the influence of design choices on robustness and evolvability by constructing several chemistries:

Set I is the *standard* instruction set described in Section 2.4, with 28 instructions and minimal redundancy.

Set II tests the importance of nop instructions matching to a complementary pattern, as does Set I. This *direct-matching* set is identical to the standard except that those instructions requiring nop patterns match directly to themselves (i.e., `nop-A` matches `nop-A`, not `nop-B`).

Set III tests the need for including nop instructions at all. This *no-nop* set lacks all three nop instructions entirely. The instructions `jump-f`, `jump-b` and `call` all require a value in the BX register (as opposed to a template) that set the distance to be jumped. Additionally, the `search-f` and `search-b` instructions are removed. Also, instead of `push` and `pop`, the register-specific `push-AX`, `push-BX`, `push-CX`, `pop-AX`, `pop-BX`, and `pop-CX` are used, for a total of 27 instructions in the set.

[2]The instruction pointer cannot execute or overwrite code of neighboring organisms except in special chemistries which explicitly allow for this. Thus, the phenomenon of *parasitism* [56] is excluded in the present experiments.

Set IV is nearly identical to the standard set, with the addition of the `memsize` instruction. In all of the other instruction sets, the organisms must calculate their own genome length before they can allocate memory to copy their offspring into. Often, this size-calculation mechanism is fragile, forcing the organisms to become stuck at a fixed size (which brings further evolution to a standstill). The *memsize* set overcomes this limitation with a single instruction that will return the genome length.

Set V tests the relevance of instruction set size to the dynamics of evolution. This *long* set comprises 84 unique instructions, with no additional functionality beyond the standard set. The new instructions are all variants of the normal instructions. For example, the standard set contains the conditional instructions `if-less`, `if-n-equ`, and `if-bit-1`. From these, any numerical comparison can be constructed. However, the long set contains the additional conditionals `if-equ`, `if-grt`, `if->=`, `if-<=`, `if-equ-0`, `if-not-0`, `if-grt-0`, `if-les-0`, `if->=-0`, `if-<=-0`, `if-A!=B`, `if-B!=C`, and `if-A!=C`.

5.1.2 Neutrality

One of the most important characteristics of a fitness landscape is its *neutrality* [43]. Intuitively, the neutrality of a landscape is a measure of its *connectedness*: the number of different ways a fitness improvement can be reached via a neutral walk (i.e., without ever taking a step down). Either extreme of neutrality brings evolution to a halt: zero neutrality implies that any mutation affects the fitness—leading to fragile programs (a rugged fitness landscape in which evolution "freezes" into a local optimum). Maximal neutrality, on the other hand, implies that no selection can occur. Landscapes ideal for evolution should therefore sport regions with neutrality somewhere in between, where neutral mutations can take a population out of local optima and avoid dead ends [22].

The neutrality of a landscape is a *local* characteristic that can change drastically in major evolutionary transitions. The landscape encountered by the programs in avida is not *self-averaging*, so no particular evolutionary path is representative of

the space of all possible paths. Thus, as in natural genetic landscapes, the result of a particular evolutionary experiment is contingent upon the ancestral program or genome; the process is *non-ergodic*. However, the area in genotypic space adjacent to the dominating wild-type is statistically occupied, giving rise to a "cloud" of programs that forms a *quasispecies*, a concept introduced to molecular evolution by Eigen [23]. By focusing only on the quasispecies, we ignore the vast volumes of genetic space that are—and always will be—unoccupied. Section 2.8.3 discussed the landscape analysis tool in avida that allows us to directly measure the neutrality of the dominant organisms. Genomes with a higher frequency of neutral mutations give rise to clusters of functionally equivilent organisms that diverge from their progenetor to form new strains.

For each experiment in each chemistry, I test the local landscape around any program that is ever most abundant (dominant) in the population. The fraction of mutations are recorded that fall into each of the categories of fatal, deleterious, neutral, or beneficial. In practice, all deleterious mutations prove fatal for the mutant as it is out-competed. Most beneficial mutations turn out to be fatal for the *rest* of the population, as the edge in replicative ability spells doom for inferior genotypes in this single-niche environment. Yet, the percentage of advantageous mutations is so small that no statistical significance can be attributed to it. Consequently, we can classify almost all mutations as either neutral or effectively fatal. One of the central questions addressed here is how this degree of neutrality is affected by changes in the artificial chemistry.

5.1.3 Results

Due to the contingent nature of evolution, experiments with identical conditions but different random number seed can—and do—lead to wildly different outcomes. Here, we have the opportunity to repeat such trials many times (*replicas*) to gain statistical significance, and extract global characteristics that set one chemistry apart from another. I focus on two such measures: the percentage of neutral mutations to

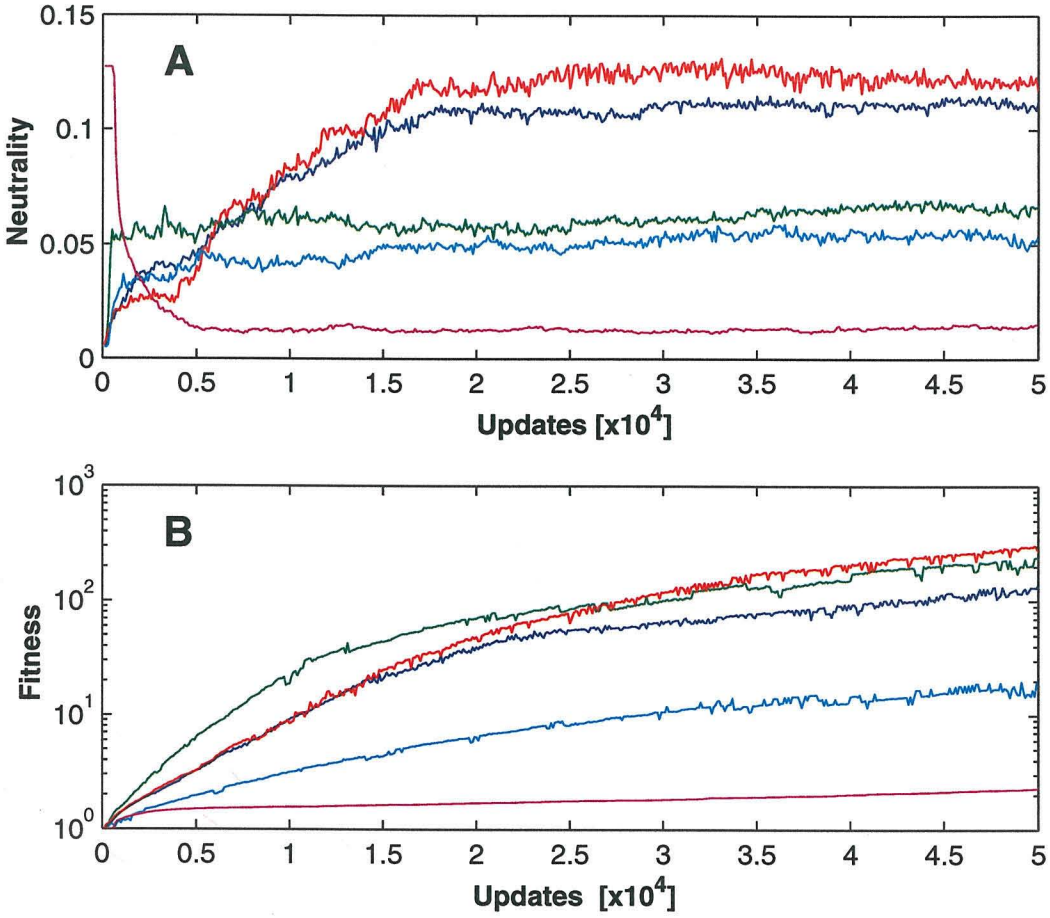


Figure 5.1: (A) Neutrality averaged over 100 trials for each of the five chemistries, as a function of time (in updates). Set I (Standard, blue curve), Set II (Direct-Matching, red), Set III (No-nop, magenta), Set IV (Mem-size, green), and Set V (Long, cyan). (B) Average fitness (relative replication rate with respect to ancestor) across trials for each chemistry. Color code as in A.

measure robustness, and the average fitness (relative replication rate with respect to the ancestor) as a function of time to measure evolvability.

For each of 100 replicas in each of the five chemistries[3], the neutrality history is obtained by extracting the dominant genotype every 100 updates and recording its neutrality. This time series is then averaged across the replicas of a particular set to produce Figure 5.1(A). We notice first that there are significant differences between chemistries, and that the neutrality changes (in general, increases) during the course

[3]Each population of 60×60 programs was allowed to adapt for 50,000 updates subject to a mutation rate of 7.5×10^{-3} errors per instruction copied, as well as a 5% probability for a single insert or delete mutation per gestation cycle.

of an experiment. This is primarily due to the increase in sequence length which accompanies adaptation, but the neutrality levels out while the fitness, in most cases, continues to increase (see Figure 5.1B). This will be discussed later on.

The neutrality of a genotype is essentially determined by the fraction of hot loci, which we can think of as the sequence length minus the amount of information stored in the sequence, in units of instructions. Thus, learning events *decrease* the neutrality if the sequence length stays constant, while size-increases without commensurate acquisition of information *increase* neutrality [4]. The mechanism by which the sequence length changes thus is crucial for both neutrality and evolvability.

In three of the chemistries (sets I, II, and V), size changes are possible only as long as the program calculates its size by finding its end (marked off by a pattern of `nop` instructions), which is the type of algorithm used by the ancestral program. Any insertions or deletions that occur in such an organism will be measured properly, and thus accounted for in the next memory allocation. If, however, size is calculated by some other means, such as a mathematical operation that produces a result equal to the length, then this calculation is not likely to change (and certainly not appropriately) when mutations have led to an altered length.

The chemistry in which size changes are easiest is set IV, as it contains an instruction (`mem-size`) that directly returns the sequence length without self-inspection. If `mem-size` is used for size calculation (as is developed in 96 of the 100 trials) insert or delete mutations become more neutral and size changes occur as needed. The opposite extreme is set III, in which no `nop` instructions are available rendering relative addressing impossible. For these experiments the standard ancestor must be replaced by one in which the genome length is explicitly coded into the sequence. Consequently, changes in size are rare as they involve significant code rearrangements.

Instruction set II differs from the standard set only in that the search for a pattern of `nop`s seeks the pattern itself rather than its complement, as described earlier. This seemingly innocuous change leads to important differences in the evolutionary history. In set I, two pairs of patterns are required for the correct functioning of the digital organism: a pattern and its complement to mark the end of the genome, and a

second pair framing the copy loop of the organism. With complementary template matching, we have witnessed that only 18% of the replicas retain such a flexible size-calculation structure. In most others it is replaced by one in which the organism's search for its end returns the location of the copy loop instead, then finds its own length (accidentally) by manipulating this number. While this is an effective way to calculate program size, it is also quite brittle: size changes can occur only if several instructions are changed in a commensurate manner. As a consequence, the standard set develops difficulties in adjusting the program's length the moment this algorithm is locked in. On the contrary, in set II the original algorithm is maintained more frequently (in 29% of the replicas), because the direct-matching of templates tends to avoid misdirected searches.

The differences in the way size changes occur is reflected in both evolvability and robustness. First, the neutrality is highest for chemistries that lead to the development of junk code, i.e., loci which do not code for information (Figure 5.1A). These are chemistries I and II where size changes occur frequently, but the direct-matching chemistry (set II) holds the edge after 20,000 updates when the trials in set I begin to lock in a non-robust algorithm for size calculation leading to problems in the acquisition of more information[4]. Set IV changes size most easily, and uses this ability to eliminate junk code. Consequently, its neutrality is lower except for the early stages of evolution (Figure 5.1A). This set beats all other chemistries as far as evolvability is concerned, mainly because the early neutrality gives it a head-start in the acquisition of information.

Set V, which consists of 84 instructions, lags in both neutrality and fitness. This is attributed to the smaller rate of advantageous substitutions (due to the larger set of instructions to choose from), while the versatility of the instructions seems to result in smaller sizes and less junk code. The chemistry that is deprived of the possibility of relative addressing offered by templates of `nop` instructions (set III) is extremely inflexible: size changes are infrequent leading to poor neutrality and adaptation. In

[4]This difference in chemistries is only apparent in the average. Because of contingency, the observables are not normally distributed, and the standard deviation can be as large as the average itself.

that respect, it is more akin to the *Redcode* chemistry mentioned earlier. Also, the neutrality of the ancestral genome is larger than the type selected by evolution, no doubt due to a clumsy design by humans.

In conclusion, I have examined different artificial chemistries with respect to their robustness and evolvability, and found that while the differences among them are attributable mainly to the manner in which genome-size changes occur, in almost all cases evolution guides the population toward a region in genotypic space with significant neutrality. While fitness continues to increase during the evolutionary process, the neutrality (on average) stays constant, suggesting that the adaptive process has led the population to a "comfortable" level that avoids evolutionary dead ends.

5.2 The Evolution of Parallel Processing

5.2.1 Introduction

In this section, I address a critical difference between the standard avida experiments and the natural world, previously touched on in Chapter 4. In nature, many chemical reactions and genome expressions occur simultaneously, with a system of gene regulation guiding their interactions. However, in most of the work done with digital organisms only one instruction is executed at a time. This implies that no two sections of the program can actively interact, as opposed to natural systems where interaction is the rule. Due to this, an obvious extension is to examine the dynamics of adaptation in artificial systems that have the capacity for more than one thread of execution (i.e., an independent CPU with its own instruction pointer, operating on the same genome). I study the emergence, adaptation, and differentiation of parallel code execution ("multi-threading") from a more computational perspective than was done in Chapter 4, and examine its impact on the evolutionary dynamics.

Work in this direction began in 1994 with Thearling and Ray using the program tierra [56]. These experiments were initialized with an ancestor that creates two threads each copying half of its genome, thereby doubling its replication rate.

Evolution produces more threads up to the maximum allowed (16), each copying a subsection of the genome [64]. In subsequent papers [65, 59] this research extended to organisms whose threads are not performing identical operations. This is done in an enhanced version of the tierra system called **Network Tierra** [58], in which multiple "islands" of digital organisms are processed on real-world machines across the Internet. In these later experiments, the organisms exist in a more complex environment in which they have the option of seeking other islands on which to place their offspring. The ancestor used for these experiments reproduces while searching for better islands using independent threads. Thread differentiation persists only when island-jumping is actively beneficial; that is, when a meaningful element of complexity is present in the environment.

In the experiments reported here, I survey the initial emergence of multiple threads and study their subsequent divergence in function. Further, I investigate the hypothesis that environmental complexity plays a key role in the pressure for the thread execution patterns to differentiate. Finally, I discuss how research on the evolution of multiple threads in digital organisms can be applied to more practical issues in both Computer Science and Biology.

5.2.2 Experimental Details

We examine the development of multi-threading in populations exposed to different environments at distinct levels of complexity, comparing them to each other and to controls that lack the capacity for multiple threads. For this purpose, I measure thread development and subsequent divergence in the research platform avida, modified to allow the generation of multiple threads, as described in Chapter 4.

I performed experiments on three environments of differing complexity, with both the extended instruction set that allows multiple expression patterns and the original (standard) instruction set as a control. Each setup was repeated in two hundred trials to gain statistical significance. Individual trials often differ extensively in the course of their evolution, generating radically different genetic structures, or getting caught

in evolutionary dead-ends.

These experiments were performed on populations of 3,600 digital organisms for 50,000 updates, equating to approximately 9,000 generations per trial utilizing about twenty hours of execution on a Pentium Pro 200. Mutations are set at a probability of 7.5×10^{-3} for each instruction copied, and a 0.05 probability for an instruction to be inserted or removed in the genome of a new offspring.

The first environment (I) is the least complex, with no explicit environmental factors to affect the evolution of the organisms; that is, the optimization of replication rate is the only adaptive pressure on the population. In the next environment (II), the organisms have available collections of numbers that they may retrieve and twelve basic (one- and two-input) logic operations for which rewards are given out. The final environment (III) studied is the most complex, with 80 logic operations selected for. For each trial, dominant genomes are analyzed to produce a time series of thread use and differentiation.

Differentiation Metrics

The following measures and indicators keep track of functional differentiation. I keep this initial analysis manageable by setting a maximum of two threads available to run simultaneously. Under this limitation, we can study the basic mechanism of differentiation, while preventing the analysis from becoming overwhelming. The relaxation of this constraint does lead to the development of more than two threads with characteristically similar interactions. The first two measures here are repeated from Chapter 4.

Thread Distance is the metric I use to determine the spatial divergence of the two instruction pointers. This measurement is the average *distance* (in units of instructions) between the execution positions of the individual threads. If this value becomes high relative to the length of the genome, it is an indication that the threads are segregated, executing different portions of the genome at any one time, whereas if it is low, they likely move in lock-step (or sightly offset) with nearly identical executions. Note, however, that if two instruction pointers execute the code offset by

a fixed number of instructions, but otherwise identically, the thread distance is an inflated measure of differentiation, as both threads do behave identically.

Code Differentiation distinguishes execution patterns with differing *behavior*. A count is kept of how often each thread executes each portion of the genome. The code differentiation is the fraction of instructions in the genome for which these counts differ between threads. Thus, the ordering of execution (time-delay) is irrelevant for this metric; only whether the code ends up getting executed differently by one thread vs. the other is important.

Execution Differentiation is a more rigorous measure of functional differentiation. It uses the same counters, taking into consideration the *difference* in the number of times the threads execute each instruction. Thus, if one thread executes a line 5 times and the other executes it 4 times, it would not contribute as much toward differentiation as an instruction executed all 9 times by one thread, and not at all by the other. This metric totals these differences in execution counts at each line and then divides the sum by the total number of multi-threaded executions. Thus, if the threads are perfectly synchronized, there is zero execution differentiation, and if only one thread exclusively executes each line, this metric is maximized at one. An execution differentiation of 0.5 indicates that half of the executions were not matched in each thread.

5.2.3 Evolution of Multi-Threaded Organisms

For my initial investigations, I focus on the 200 trials in environment III (the most complex), with the extended instruction set, allowing for multi-threading.

Emergence of Multiple Execution Patterns

Describing a universal course of evolution in any medium is generally not feasible due to the numerous random and contingent factors that play key roles. However, there are a number of trends that appear in a significant portion of experiments, which will be discussed further.

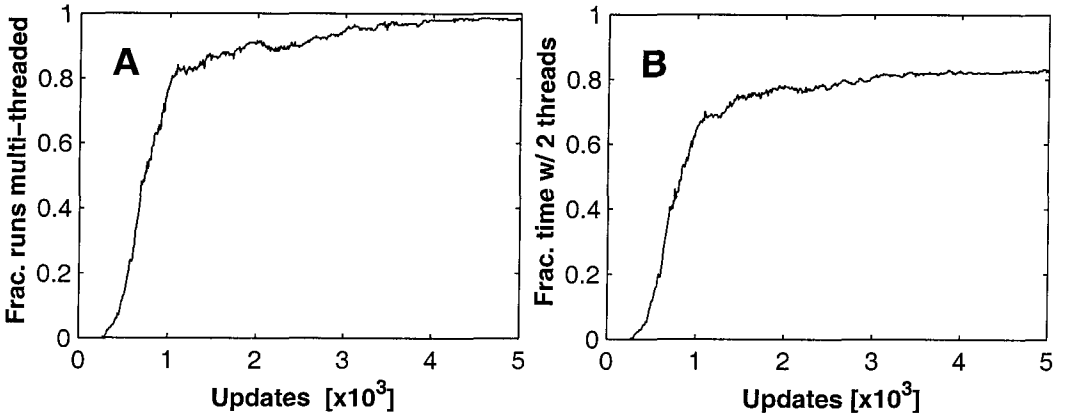


Figure 5.2: The time progression of organisms learning to use multiple threads averaged over 200 trials. (A) The fraction of trials which thread at all, and (B) the average fraction of time organisms spend using both threads at once. The data displayed here is for the first 5000 updates of 50,000 update experiments in environment III.

Let us first consider the emergence of organisms from a purely linear execution to the use of multiple threads. In Figure 5.2(A), we see that most populations do, in fact, develop a secondary thread near the beginning of their evolution. Secondary threads come into use as soon as this extra execution grants any benefit to the organisms. The most common way this occurs is by having a `fork-th` and a `kill-th` appear around a section of code, which the threads thereby move through in lock-step, performing rewarded computations twice. While multiple completions of a task provide only a minor speed bonus, this is often sufficient to warrant a double execution.

Once multiple execution has set in, its use will be optimized with time. Smaller blocks of duplicated code will be expanded, and larger sections will be used more productively, sometimes even shrinking to improve efficiency. Once multiple threads are in use, differentiation follows.

Execution Patterns in Multi-threaded Organisms

A critical question is "What effect does the introduction of a secondary thread have on the process of evolution?" The primary measure to denote a genome's level of adaptation to an environment is its *fitness*. The progression of fitness with time (av-

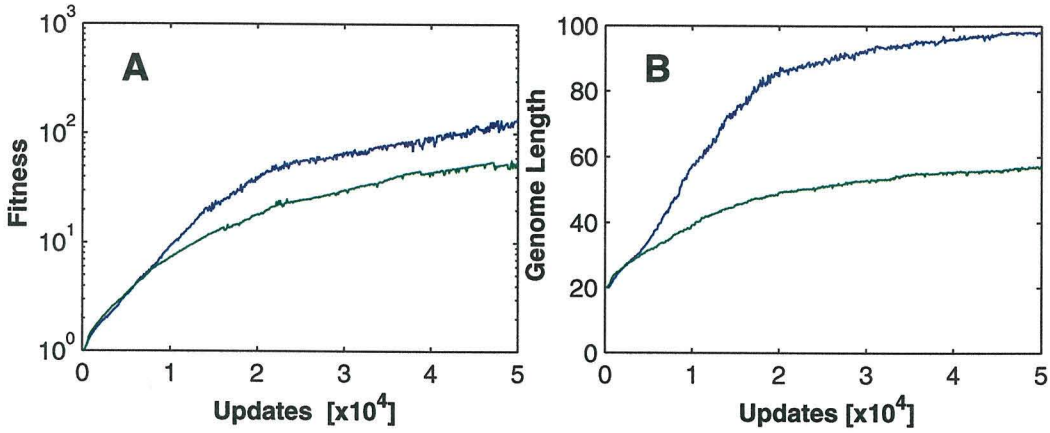


Figure 5.3: (A) Average fitness as a function of time (in updates) for the 200 environment III trials, and (B) average sequence length for the linear execution experiments (blue) and the multiple execution experiments (green).

eraged over all trials for each experiment) is shown in Figure 5.3(A)[5]. Most increases to replication rate occur as a multiplicative factor, requiring fitness to be displayed on a logarithmic scale.

Contrary to expectations, the availability of additional threads leads to reduced adaptation. However, the average length of the genomes (Figure 5.3B) reveals that the code for these marginally less fit organisms is stored using 40% fewer instructions, indicating a denser encoding. Indeed, the very fact that multi-threading develops spontaneously in these populations implies that it is beneficial. How then can a beneficial development be detrimental to an organism's fitness? Inspection of resulting genomes has allowed us to determine that this code compression is accomplished by overlapping execution patterns that differ in their final product. See Chapter 4 for more details.

Figure 5.4(A) displays an example genome. The initial thread of execution (the inner ring) begins in the D "gene" and proceeds clockwise around the diagram. The execution of D divides the organism when it has a fully developed copy of itself ready. This is not the case for this first execution, so the gene fails with no effect to the organism. Execution immediately progresses into gene C_0 where computational

[5]Note that the fitness curves of *individual* trials display a sharp punctuation in learning events, appearing as an uneven staircase.

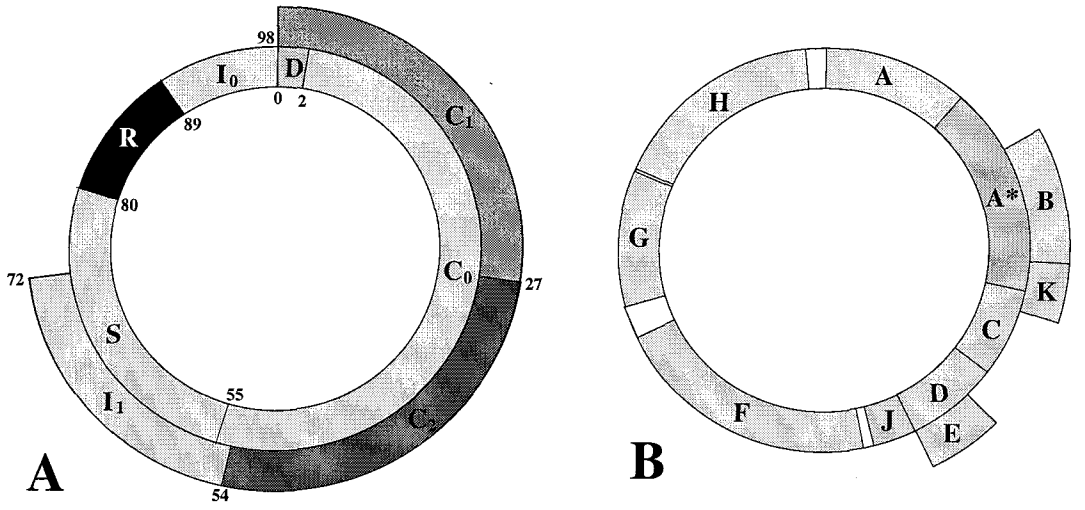


Figure 5.4: (A) Execution patterns for an evolved avida genome. The inner ring displays instructions executed by the initial thread, and the outer ring by the secondary thread. Darker colors indicate more frequent execution. The initial thread primarily executes gene R, which performs the copy process, while the other thread centers on genes C_1 and C_2 for task computation. (B) Genome structure of the phage ΦX174. The promoter sequence for gene $A*$ is entirely within the gene A, causing the genes to express the same series of amino-acids from the portion overlapped. Genes B, E, and K are also entirely contained within others, but with an offset reading frame, such that different amino acids are produced (i.e., the expression is different).

tasks are performed, increasing the CPU speed. Near the center of C_0, a `fork-th` instruction is executed initiating secondary execution (of the same code) at line 27, giving rise to gene C_2. The primary thread continues on to line 55, the S gene, where genome size is calculated and the appropriate memory space for its offspring is allocated. Next, the primary instruction pointer runs into gene R, the copy loop, where the actual replication occurs. It is executed once for each of the 99 instructions in the genome (hence its dark color in the figure). When this process is complete, it moves on through gene I_0 shuffling numbers around, and re-enters gene D for a final division.

During this time, the secondary thread executes gene C_2 computing a few basic logical operations. C_2 ends with a `jump-f` (jump forward) instruction that initially fails. Passing through gene I_1, numbers are shuffled within the thread and the jump at its last line (72) diverts the execution back to the beginning of the organism. From this point on, its execution loops through C_1 and C_2 for a total of 10 times, using the

Table 5.1: Example genetic encoding in avida.

00: pop	17: if-bit-1	34: push	51: pop	68: nop-C	85: inc
01: divide	18: push	35: swap	52: jump-f	69: add	86: if-n-equ
02: get	19: id-th	36: nand	53: nop-C	70: if-bit-1	87: jump-b
03: nop-A	20: jump-f	37: put	54: push	71: get	88: nop-B
04: nand	21: get	38: nop-C	55: search-b	72: jump-f	89: get
05: nand	22: shift-r	39: get	56: nop-C	73: fork-th	90: dec
06: nand	23: add	40: nand	57: nop-C	74: put	91: nop-C
07: nop-C	24: put	41: get	58: add	75: allocate	92: nand
08: push	25: fork-th	42: swap	59: if-less	76: push	93: push
09: nop-A	26: push	43: nand	60: swap	77: search-f	94: nop-B
10: nop-A	27: nand	44: put	61: dec	78: pop	95: nand
11: nop-A	28: if-n-equ	45: nop-B	62: nop-C	79: nop-C	96: inc
12: nop-B	29: nand	46: divide	63: kill-th	80: swap-stk	97: pop
13: put	30: put	47: nand	64: fork-th	81: swap-stk	98: pop
14: add	31: get	48: shift-r	65: inc	82: if-n-equ	
15: push	32: nop-B	49: nand	66: add	83: copy	
16: put	33: nand	50: put	67: add	84: copy	

results of each pass as inputs to the next, computing different tasks each time. The full genome for this organism is presented in Table 5.1. Note that for this organism, the secondary thread is never involved in replication.

Similar overlapping patterns appear in natural organisms, particularly viruses. Figure 5.4(B) exhibits a gene map of the phage ΦX174 containing portions of genetic code that are expressed multiple times, each resulting in a distinct protein [67]. Studies of evolution in the overlapping genes of ΦX174 and other viruses have isolated the primary characteristic hampering evolution. Multiple encodings in the same portion of a genome necessitate that mutations be neutral (or beneficial) in their net effect over *all* expressions or they will be selected against. Fewer neutral mutations result in a smaller variation and in turn a slower adaptation rate. It has been shown that in both viruses [46] and Avida organisms [51] (see Chapter 4), overlapping expressions have between 50 and 60% of the variation of the non-overlapping areas in the same genome. Therefore, fewer mutated offspring survive, causing genotype space to be explored at a slower pace.

In higher organisms, multiple genes do develop that overlap in a portion of their

encoding, but are believed to be evolved out through gene duplication and specialization, leading to improved efficiency [33]. Unfortunately, viruses and avida organisms are both subject to high mutation rates with no error correction abilities. This causes a strong pressure to compress the genome, thereby minimizing the target for mutations. As this is an immediate advantage, it is typically seized, although it leads to a decrease in the adaptive abilities of the population in the long term.

Environmental Influence on Differentiation

Now that we have witnessed the development of multiple threads of execution in avida, let us examine the impact of environmental complexity on this process. Populations in all environments learn to use their secondary thread quite rapidly, but show a marked difference in their ability to diverge the threads into distinct functions. In Fig 5.5(A), average Thread Distance is displayed for all trials in each environment showing a positive correlation between the divergence of threads and the complexity of the environment they are evolving in.

More complex environments naturally require more information to be stored within the organism, and hence promote longer genomes [4], possibly biasing this measure. To account for this, I consider this average thread distance normalized to the length of the organisms, displayed in Fig 5.5(B). When threads fully differentiate, they often execute neighboring sections of code, regardless of the length of the genome they are in. Indeed, longer genomes now need their threads to be further spatially differentiated to obtain an equivalent fractional thread distance. Thus, the fact that more complex environments still have a marginally higher distance is quite significant.

Interestingly, Code Differentiation (Fig 5.6A) does not firmly distinguish the environments, averaging at about 0.5. In fact, the distribution (between 0 and 1) of code differentiation turns out to be nearly uniform. This indicates that the portion of the genomes that are involved with the differentiated threads are similarly distributed between complexity levels. Execution Differentiation (the measure of the fraction of executions that occurred differently between threads, shown in Fig 5.6B), however, once again positively correlates environments with thread divergence. The

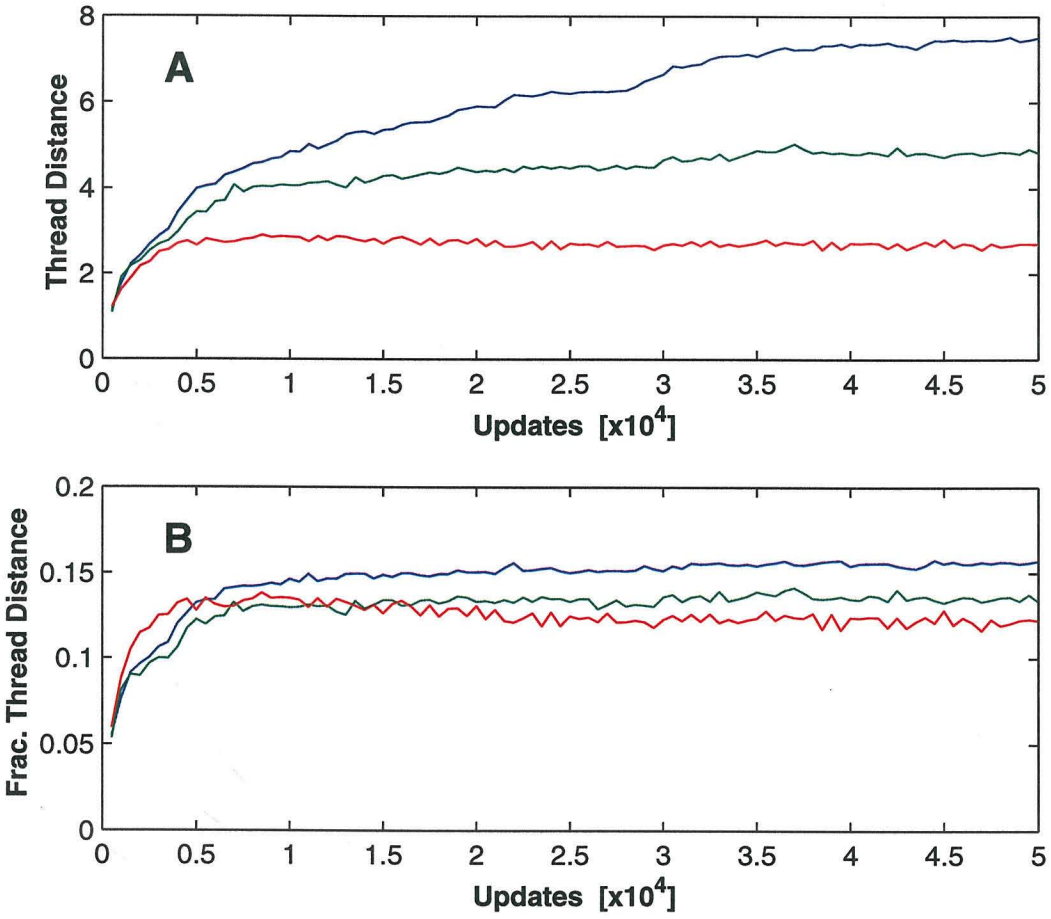


Figure 5.5: Differentiation measures averaged over all trials for each experiment. Average values of (A) Thread Distance and (B) Fractional Thread Distance are displayed for experiments in environment I (red), environment II (green), and environment III (blue).

degree of differentiation between the execution patterns is more pronounced in the more complex environments.

5.2.4 Summary

We have witnessed the development and differentiation of multi-threading in digital organisms, and exhibited the role of environmental complexity in promoting this differentiation. Although this is an inherently complex process, the ability to examine almost any detail and dynamic within the framework of avida provides insight into what I believe are fundamental properties of biological and computational systems.

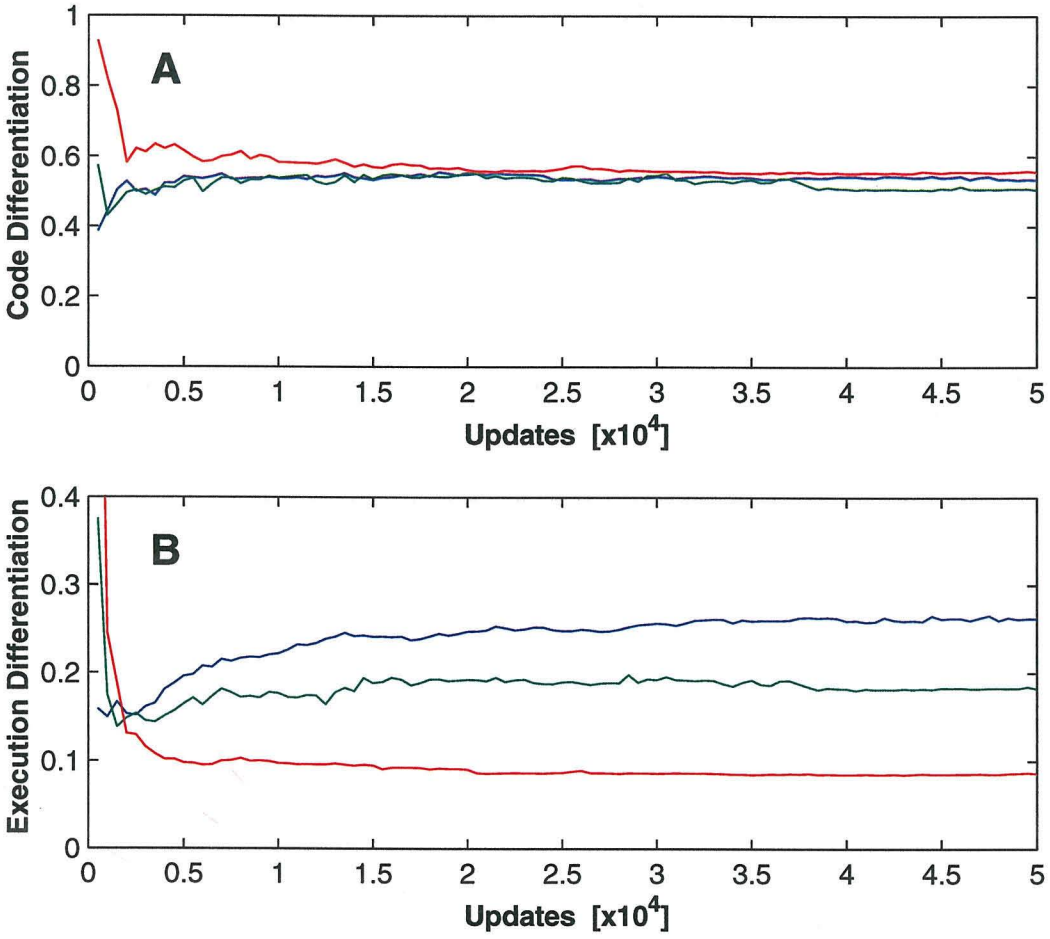


Figure 5.6: Differentiation measures averaged over all trials for each experiment. Average values of (A) Code Differentiation and (B) Expression Differentiation are displayed for experiments in environment I (red), environment II (green), and environment III (blue).

The patterns of expression (lock-step, overlapping, and spatial differentiation) are selected by balancing the "physiological" costs of execution and differentiation against the implicit effects of mutational load. Clearly, multiple threads executing single regions of the genome provides for additional use of that region. This is analogous to the use of both the overlapping genes as seen in many viruses, as well as the more common use of single gene products within various pathways. The benefit is in the form of additional functionality and a reduction in the mutational load required for that functionality. Within the context of this thinking, the correlation between environmental complexity and the usage of multiple threads makes a great

deal of sense: multiple threads are advantageous only if they can provide additional functionality.

However, there is a cost side in this equation. Even though in avida there currently is no *explicit* cost for adding threads, when a gene or gene product is used in multiple pathways, variations are severely reduced as the changes to each affected gene must result in a net benefit to the organism. This cost was demonstrated: We observed a negative correlation between rates of adaptation and use of multiple threads. Furthermore, the ability to analyze the entropy of each site in the genome quantifies the loss in variability predicted by this hypothesis.

Implications of this work with potentially far reaching consequences for Computer Science involve the study of *how* the individual threads interact and what techniques the organisms implement to obtain mutually robust operations. The internal interactions within computer systems lack the remarkable stability of biological systems to a noisy, and often changing environment.

In a computer, a substantially altered environment will cause a program to cease to function. Likewise, if two pieces of software interact in a way not explicitly designed, the results are neither predictable nor reliable. An even more acute problem seems to be that in contemporary computer systems we are reaching a level of code complexity that renders systems untestable, and fragile even under normal use.

Conversely, as biological organisms have evolved to higher levels of complexity, they deal more gracefully with change or unexpected situations. Life as we know it would never have reached such vast multi-cellularity if every time a single component failed or otherwise acted unexpectedly, the whole organism shut down. Although failure or pathological behavior of a single cell within an organism can cause traumatic results (such as cancer), this is an exception. The typical outcome is for the body to remove the offending cell and have its function carried out by others. This is accomplished by inter-cell communication and gene regulation.

Clearly, we are still taking the first steps in developing systems of computer programs that interact on similarly robust levels. Here we have performed experiments on a simple evolutionary system as a step toward deciphering these biological prin-

ciples as applied to digital life. Systems at levels of integration anywhere near that of biological life are still a long way off, but more concrete concepts such as applying principles from gene regulation to develop self-scheduling parallel computers may be much closer.

Chapter 6 Future Work

In this thesis, I have used the avida software to systematically study specific aspects of the relationship between a population's evolutionary dynamics and the underlying genetic encoding employed by the organisms. Throughout, I consider genetic codes in the form of computer languages, and attempt to exploit the analogy in both directions. For one, evolving systems can be studied in a computational medium, allowing us to observe dynamics and verify predictions at a level of detail that would otherwise be infeasible to obtain. In the other direction, biological evolution has been able to produce and manage vast levels of complexity that are currently impossible to deal with in software systems; those techniques that are employed in the software systems are not nearly so elegant. If we can better understand the robustness of, and complexity growth in evolved systems, this may provide us with insights into handling similar aspects of man-made software.

The next steps in my research involve strengthening the comparison between the evolution of biological genetic codes and the evolution of computer programs. I have begun a collaboration with Richard Lenski at Michigan State University, to use avida to reproduce several experiments performed by him on *E. coli*. In the first of these, we have compared the epistatic effects from his biological work [25] to our digital organisms [40]. In the next steps of this collaboration, we are considering the relative roles of chance, history, and adaptation on the products of evolving systems [66], and the nature of punctuated versus gradual evolution [24].

I am also continuing my studies of complexity in evolving systems. The equations for calculating complexity presented here neglect the possibility of correlations occurring in codes, which can significantly skew the accuracy of the complexity estimates. The landscape tool in avida has been expanded to sample millions of genomes at an arbitrary mutational distance away. If the expected fitness from this test is not multiplicative (i.e., if a single mutation reduces fitness by a factor of α, a multiplicative

model would predict n mutations to reduce it to α^n), then we know such epistatic effects are at work, and can be estimated in a simple model.

All of the other projects presented here are equally open for continued study. There is significantly more to the topic of *genetic organization* than the overlapping or segregation of genes. The expression of specific amino acids in DNA seems to maximize the probability that the mutation of a single nucleotide will not affect the protein that it is involved in expressing. Likewise, there are many structures (promoter sequences, transcription levels, catalytic sections of introns, etc.) that we do not account for in the current instantiation of avida.

The topic of *evolvability and robustness of computer languages* is quite extensive; in this thesis I have only considered a few of the key issues that have been assumed, but not tested, to be critical. On the topic of label matching alone, we have shown the importance of labels, but we are far from isolating the optimal type of labels to be used.

In several portions of this thesis, I explore the effects of allowing the digital organisms to have multiple threads of execution. In the limited model that I present, only two threads are possible, and while they can communicate, they have no ability to directly promote or suppress one another. I plan to expand this research into the study of *digital gene regulation*, with the goal of understanding how to develop more complex software with interacting parts.

Looking forward, extensions I have planned for this work should be able to enhance both the fields of Computer Science and Biology. A project I am initiating on the computational side is the study of a genetic algorithm variation that I call an *auto-adaptive algorithm* (A^3). Such a system would have solutions replicating themselves proportional to their fitness. Unlike a standard GA, however, no genome can ever guarantee having its information propagated to the next generation except through its offspring. This will introduce ν-selection into the system, forcing the codes to maintain a neutrality to mutations, and in turn making it more difficult for the population to become stuck in an evolutionary dead-end.

Finally, I am adding a resource model to avida with the goal of constructing

primitive *ecosystems*. We have been very successful in using avida to understand the progression of evolution in a single niche; I am now expanding this work to cover higher-level biological structures. The simplest eco-system model would include two resources, each of which requires a different type of task in order to obtain the merit bonus associated with it. When one resource is used, it is converted to the other encouraging two niches to exist at once, each fueling the other. Such experiments should allow us to witness the evolution of co-operation and small co-evolving ecosystems.

Appendix A Configuration Files

Avida is designed to be highly flexible such that the researcher can setup a configuration to perform a wide varieity of experiments.

A.1 The genesis File

The genesis file is the primary configuration file for avida. Most environmental initial conditions can be adjusted here.

The format is: <variable name> <value>

The genesis file is divided into several sections, each of which contains a selection of variables (described briefly within the actual file).

```
VERSION_ID 1.4                # Do not change this value!

### Architecture Variables ###
MAX_UPDATES 50000     # Maximum updates to run simulation (-1 = no limit)
MAX_GENERATIONS -1    # Maximum generations to run simulation (-1 = no limit)
WORLD-X 60            # Width of the world in Avida mode.
WORLD-Y 60            # Height of the world in Avida mode.
MAX_CPU_THREADS 2     # Number of Threads CPU's can spawn
RANDOM_SEED 1         # 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.28.base    # File containing instruction set.
TASK_SET task_set            # File containing task set.
EVENT_FILE event_list        # File containing list of events during run.
START_CREATURE creature.base # Organism to seed the population.

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

### Reproduction ###
BIRTH_METHOD 1      # 0 = Choose Random Organism (poor for evolution!)
                    # 1 = Choose Oldest Organism
                    # 2 = Choose largest Age/Merit
                    # 3 = Choose only empty cells.
                    # 4 = Choose Random from full population (Mass Action)
                    # 5 = Choose Eldest from full population (Tierra)
DEATH_METHOD 0      # 0 = Never kill organisms.
```

```
                    # 1 = Kill when inst executed = AGE_LIMIT
                    # 2 = Kill when inst executed = length * AGE_LIMIT
AGE_LIMIT 5000      # Modifies DEATH_METHOD
ALLOC_METHOD 0      # 0 = Allocated space is set to default instruction.
                    # 1 = Set to section of other organisms (Necrophilia)
                    # 2 = Allocated space is set to random instruction.

### Divide Restrictions ####
CHILD_SIZE_RANGE 2      # Maximal differential between child and parent sizes.
MIN_COPIED_LINES 0.5    # Code fraction which must be copied before divide.
MIN_EXE_LINES    0.5    # Code fraction which must be executed before divide.

### Mutations ###
POINT_MUT_PROB  0.0     # Mutation rate (per-location per update) (x10^-6)
COPY_MUT_PROB   0.0075  # Mutation rate (per copy).
INS_MUT_PROB    0.0     # Insertion rate (per site, applied on divide).
DEL_MUT_PROB    0.0     # Deletion rate (per site, applied on divide).
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 organisms get default...
                      # 1 = BLOCK: Block slice scaled to merit.
                      # 2 = PROBABILISTIC: Run _prob_ proportional to merit.
                      # 3 = INTEGRATED: Perfectly integrated deterministic.
SIZE_MERIT_METHOD 4   # 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

### Task Bonus Modifiers ###
MAX_NUM_TASKS_REWARDED -1     # -1 = Unlimited

### Genotype Info ###
THRESHOLD 3           # Number of organisms in a genotype needed for it
                      #   to be considered viable.
GENOTYPE_PRINT 0      # 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 2   # 1 = full, 2 = limited search (parent only)
SPECIES_THRESHOLD 2   # max number of failures organisms to be same species

### Data and Log Files ###
SAVE_AVERAGE_DATA   10   # Print these files every x updates.  Enter 0 for
SAVE_DOMINANT_DATA  10   # those that should never be printed
SAVE_COUNT_DATA     10
```

```
SAVE_TOTALS_DATA    0
SAVE_TASKS_DATA     10
SAVE_STATS_DATA     10
SAVE_TIME_DATA      10

SAVE_GENOTYPE_STATUS  0  # Print these files every x updates.  Enter 0 for
SAVE_DIVERSITY_STATUS 0  # those that should never be printed

LOG_CREATURES    0        # 0/1 (off/on) toggle to print file.
LOG_GENOTYPES    0        # 0 = off, 1 = print ALL, 2 = print threshold ONLY.
LOG_THRESHOLD    0        # 0/1 (off/on) toggle to print file.
LOG_SPECIES      0        # 0/1 (off/on) toggle to print file.
LOG_BREED_COUNT  0        # 0/1 (off/on) toggle to print file.
LOG_PHYLOGENY    0        # 0/1 (off/on) toggle to print file.
LOG_GENEOLOGY    0        # 0 = off, 1 = all, 2 = parents only.
LOG_LANDSCAPE    0        # 0/1 (off/on) toggle to print file.
LOG_MUTATIONS    0        # 0/1 (off/on) toggle to print file.

### Debug ###
DEBUG_LEVEL 2           # 0 = ERRORS ONLY, 1 = WARNINGS, 2 = COMMENTS

### END ###
```

A.2 The event_list File

The `event_list` file configures the events to occur during the course of an avida experiment, by listing the trigger of the event, the name of the event, and any relevant arguments required. The format is:

```
<trigger type> <trigger> <event name> [<args>...]
```

Currently there are two trigger types available: updates (u) and generations (g). The trigger itself takes the form of `start:step:stop`; the first number indicates when the event should initially be triggered, the second number is how often it should be triggered, and the final number is when it should no longer be triggered. Thus, if an event were

```
u 1000:50:2000 save_clone
```

then starting at update 1000, a clone of the population would be saved every 50 updates until update 2000. If only two number are specified, the even will continue

periodically until the experiment terminates. If a single value is given, the event happens only once.

If an event requires any arguments to specify how it should behave, those come at the end.

Here is an example `event_list` file:

```
g 100:100 calc_landscape
u 5000:5000 save_clone

u 9000:10:10000 mod_copy_mut 0.0001
u 20000:10:21000 mod_copy_mut -0.0001

u 30000 inject creature.parasite

u 40000 apocalypse 0.75

u 50000 exit
```

The file starts off by setting two events that occur continuously; every one hundered generations a local landscape analysis (see Section 2.8.3) is performed on the dominant organism in the population, and every 5000 updates a clone is made of the population.

Specific events begin at update 9000 when, for the next 1000 updates (until 10,000), the copy mutation rate is slowly increased. This "high temperature" environment persists for 10,000 more update when it begins to cool at update 20,000 returning to its original level by update 21,000. At update 30,000 a parasite is injected into the population, and then at 40,000 an apocalypse event occurs killing off 75% of the population.

Finally, at update 50,000 the experiment is ended.

A.3 The inst_set File

The instruction set file determines which instructions are to be included in the computational chemestry. It consists of a list of possible commands for the virtual assembly language, each with a 1 or 0 next to them, which determines if they should be included. Any instruction not listed here will automatically be assumed to have a 0 next to it (so, theoretically only the included instructions need to be in the list). Avida is distributed with a collection of pre-configured instruction set files for the specific types of languages.

The default 28-instruction set follows. Here, it is divided over two columns to conserve space, but progresses linearly in the actual file.

```
nop-A       1
nop-B       1
nop-C       1
nop-X       0
if-equ-0    0
if-not-0    0
if-n-equ    1
if-equ      0
if-grt-0    0
if-grt      0
if->=-0     0
if->=       0
if-les-0    0
if-less     1
if-<=-0     0
if-<=       0
if-A!=B     0
if-B!=C     0
if-A!=C     0
if-bit-1    1
jump-f      1
jump-b      1
jump-p      0
jump-slf    0
call        1
return      1
pop         1
push        1
swap-stk    1
flip-stk    0
swap        1
swap-AB     0
swap-BC     0
swap-AC     0
```

```
set-num     0
inc         1
dec         1
zero        0
neg         0
square      0
sqrt        0
not         0
add         1
sub         1
mult        0
div         0
mod         0
nand        1
nor         0
and         0
order       0
xor         0
copy        1
read        0
write       0
stk-read    0
stk-writ    0
compare     0
if-n-cpy    0
allocate    1
divide      1
c-alloc     0
c-divide    0
inject      0
get         1
stk-get     0
stk-load    0
put         1
```

```
copy-reg    0
set_A=B     0
set_A=C     0
set_B=A     0
set_B=C     0
set_C=A     0
set_C=B     0
reset       0
pop-A       0
pop-B       0
pop-C       0
push-A      0
push-B      0
push-C      0
shift-r     1
shift-l     1
bit-1       0
abs-jp      0
search-f    1
search-b    1
mem-size    0
rotate-l    0
rotate-r    0
set-cmut    0
mod-cmut    0
fork-th     0
kill-th     0
id-th       0
repro       0
dm-jp-f     0
dm-jp-b     0
dm-sch-f    0
dm-sch-b    0
re-jp-f     0
re-jp-b     0
```

A.4 The task_set File

The task_set file lists the possible bonuses that an avida organism can receive, and sets the type and level of that bonus.

The file is in the format

```
<category>  <task name>  <bonus type> <bonus value> [<type> <value> ...]
```

In the file below, each task entry is followed by a comment describing that task. The type of each bonus determines how it affects merit; it can either be additive ($\mathcal{M}_{\text{new}} = \mathcal{M}_{\text{old}} + \text{bonus}$), multiplicative ($\mathcal{M}_{\text{new}} = \mathcal{M}_{\text{old}} \times \text{bonus}$), or exponential ($\mathcal{M}_{\text{new}} = \mathcal{M}_{\text{old}}^{\text{bonus}}$).

If more than one bonus is given for a task, the first bonus is rewarded the first time that task is completed, the second bonus is awarded the second time, and so on. If a task is performed more often than there are bonuses listed for, the final bonus that is listed is applied thereafter. Note that all of the task descriptions below end in a "+ 0" mandiating that additional completions of the task garner no bonus at all.

The naming convention for the logic task is a number-letter pair. The number is how many inputs the task makes use of, and the letter is a unique code.

```
# Cat.  Task    Bonuses...                # Meaning

misc    get     * 1.05  * 1.05  * 1.05  + 0
misc    put     * 1.05  * 1.05  * 1.05  + 0
misc    echo    + 0
misc    add     + 0

logic   0a      + 0                      # Bitstring of all Zeros
logic   0b      + 0                      # Bitstring of all Ones

logic   1a      * 1.05  * 1.05  + 0      # Echo
logic   1b      * 1.1   * 1.05  + 0      # Not

logic   2a      * 1.2   * 1.1   + 0      # A and B
logic   2b      * 1.25  * 1.1   + 0      # A or B
logic   2c      * 1.2   * 1.1   + 0      # A or ~B
logic   2d      * 1.25  * 1.1   + 0      # A and ~B
logic   2e      * 1.3   * 1.1   + 0      # A nor B
logic   2f      * 1.15  * 1.1   + 0      # A nand B
logic   2g      * 1.5   * 1.1   + 0      # A xor B
logic   2h      * 1.5   * 1.1   + 0      # A equ B

logic   3a      * 1.5   * 1.1   + 0      # A and B and C
logic   3b      * 1.5   * 1.1   + 0      # A and B and ~C
logic   3c      * 1.5   * 1.1   + 0      # A and ~B and ~C
logic   3d      * 1.5   * 1.1   + 0      # ~A and ~B and ~C
logic   3e      * 1.5   * 1.1   + 0      # A and (B xor C)
logic   3f      * 1.5   * 1.1   + 0      # A & (B | C)
logic   3g      * 1.5   * 1.1   + 0      # A + B + C = 2
logic   3h      * 1.5   * 1.1   + 0      # A + B + C >= 2
logic   3i      * 1.5   * 1.1   + 0      # A & ~(B xor C)
logic   3j      * 1.5   * 1.1   + 0      # A xor (B & C)
logic   3k      * 1.5   * 1.1   + 0      # A | (B & C)
logic   3l      * 1.5   * 1.1   + 0      # A & (B | ~C)
logic   3m      * 1.5   * 1.1   + 0      # (A & ~B) | (~A & B & C)
logic   3n      * 1.5   * 1.1   + 0      # (A & ~B) | (B & C)
logic   3o      * 1.5   * 1.1   + 0      # A & (B nand C)
logic   3p      * 1.5   * 1.1   + 0      # A xor (B & C)
logic   3q      * 1.5   * 1.1   + 0      # A | (B & C)
logic   3r      * 1.5   * 1.1   + 0      # (A xor B) & ~C
logic   3s      * 1.5   * 1.1   + 0      # ~A & ( B xor C) ) | (A & B & C)
logic   3t      * 1.5   * 1.1   + 0      # (A & ~B) | (~A & B & ~C)
logic   3u      * 1.5   * 1.1   + 0      # A & (B | ~C) | (~A & ~B & C)
logic   3v      * 1.5   * 1.1   + 0      # (A xor B) | (A & C)
logic   3w      * 1.5   * 1.1   + 0      # (A nor B) nor C
logic   3x      * 1.5   * 1.1   + 0      # (~A & (B | C)) | (B & C)
logic   3y      * 1.5   * 1.1   + 0      # (~A & B) | (~A & C) | (B & ~C)
logic   3z      * 1.5   * 1.1   + 0      # A | (B & ~C)
logic   3aa     * 1.5   * 1.1   + 0      # (A & ~B) | (A & ~C) | (~A & B)
logic   3ab     * 1.5   * 1.1   + 0      # A + B + C = 1
logic   3ac     * 1.5   * 1.1   + 0      # A xor B xor C
logic   3ad     * 1.5   * 1.1   + 0      # (A & ~C) | (B & ~C) | (~A & ~B & C)
logic   3ae     * 1.5   * 1.1   + 0      # [very complex]
```

```
logic   3af     * 1.5   * 1.1   + 0     # [very complex]
logic   3ag     * 1.5   * 1.1   + 0     # A | (B xor C)
logic   3ah     * 1.5   * 1.1   + 0     # ~( (A & B & C) | (~A & ~B & ~C) )
logic   3ai     * 1.5   * 1.1   + 0     # A or B or C
logic   3aj     * 1.5   * 1.1   + 0     # (A & B & C) | (~A & ~B & ~C)
logic   3ak     * 1.5   * 1.1   + 0     # A nor (B xor C)
logic   3al     * 1.5   * 1.1   + 0     # [very complex]
logic   3am     * 1.5   * 1.1   + 0     # [very complex]
logic   3an     * 1.5   * 1.1   + 0     # (C & (A | B)) | (~A & ~B & ~C)
logic   3ao     * 1.5   * 1.1   + 0     # A xor ~(B xor C)
logic   3ap     * 1.5   * 1.1   + 0     # A + B + C != 1
logic   3aq     * 1.5   * 1.1   + 0     # (~A & ~B) | (A & B & C)
logic   3ar     * 1.5   * 1.1   + 0     # (A xor B) nor (A & C)
logic   3as     * 1.5   * 1.1   + 0     # ~A & (B | ~C)
logic   3at     * 1.5   * 1.1   + 0     # (A & ~B) | (A & C) | (~B & ~C)
logic   3au     * 1.5   * 1.1   + 0     # [very complex]
logic   3av     * 1.5   * 1.1   + 0     # [very complex]
logic   3aw     * 1.5   * 1.1   + 0     # [very complex]
logic   3ax     * 1.5   * 1.1   + 0     # A | (B nor C)
logic   3ay     * 1.5   * 1.1   + 0     # (~A & (~B | C)) | (A & (B xor C))
logic   3az     * 1.5   * 1.1   + 0     # A | ~(B xor C)
logic   3ba     * 1.5   * 1.1   + 0     # A nor (B & C)
logic   3bb     * 1.5   * 1.1   + 0     # A equ (B & C)
logic   3bc     * 1.5   * 1.1   + 0     # (A & ~B) nor (B & C)
logic   3bd     * 1.5   * 1.1   + 0     # (A & ~B) nor (~A & B & C)
logic   3be     * 1.5   * 1.1   + 0     # A nor (B & C)
logic   3bf     * 1.5   * 1.1   + 0     # A equ (B & C)
logic   3bg     * 1.5   * 1.1   + 0     # A nand B nand C
logic   3bh     * 1.5   * 1.1   + 0     # ~A | (B & ~C)
logic   3bi     * 1.5   * 1.1   + 0     # A nand ~(B xor C)
logic   3bj     * 1.5   * 1.1   + 0     # ~A | B | C
logic   3bk     * 1.5   * 1.1   + 0     # A + B + C <= 1
logic   3bl     * 1.5   * 1.1   + 0     # A + B + C != 2
logic   3bm     * 1.5   * 1.1   + 0     # A nand (B | C)
logic   3bn     * 1.5   * 1.1   + 0     # A nand (B xor C)
logic   3bo     * 1.5   * 1.1   + 0     # ~A | ~B | C
logic   3bp     * 1.5   * 1.1   + 0     # ~A | ~B | ~C
```

Appendix B Extracted Organisms

Organisms can be extracted from avida for analysis by a researcher, or for use in future experiments. This appendix displays several organisms, both those that are used as ancestor organisms, and those that were the product of evolution.

Two main ancestors have been used in the experiments prestented here: one possesing short, 20-line genome, and the other with a longer 80-line genome. Each of these were placed into avida and immediatly extracted; all comments are automatically generated. By default, genotypes are named with a 3 digit code (indicating their length) followed by a unique 5 letter id. The length-20 genome follows:

```
# Filename........: 020-aaaaa
# Update Output...: 0
# Is Viable.......: 1

# Generation: 0         Divide-1     Divide-2      Average
# Merit...........:           17           19           18
# Gestation Time..:           88           90           89
# Fitness.........:     0.193182     0.211111     0.202146
# Errors..........:            0            0
# Code Size.......:           20           20
# Copied Size.....:           20           20           20
# Executed Size...:           17           19           18
# Offspring.......:         SELF         SELF

# get:       0      logic:3i:  0      logic:3ac: 0      logic:3aw: 0
# put:       0      logic:3j:  0      logic:3ad: 0      logic:3ax: 0
# logic:1a:  0      logic:3k:  0      logic:3ae: 0      logic:3ay: 0
# logic:1b:  0      logic:3l:  0      logic:3af: 0      logic:3az: 0
# logic:2a:  0      logic:3m:  0      logic:3ag: 0      logic:3ba: 0
# logic:2b:  0      logic:3n:  0      logic:3ah: 0      logic:3bb: 0
# logic:2c:  0      logic:3o:  0      logic:3ai: 0      logic:3bc: 0
# logic:2d:  0      logic:3p:  0      logic:3aj: 0      logic:3bd: 0
# logic:2e:  0      logic:3q:  0      logic:3ak: 0      logic:3be: 0
# logic:2f:  0      logic:3r:  0      logic:3al: 0      logic:3bf: 0
# logic:2g:  0      logic:3s:  0      logic:3am: 0      logic:3bg: 0
# logic:2h:  0      logic:3t:  0      logic:3an: 0      logic:3bh: 0
# logic:3a:  0      logic:3u:  0      logic:3ao: 0      logic:3bi: 0
# logic:3b:  0      logic:3v:  0      logic:3ap: 0      logic:3bj: 0
# logic:3c:  0      logic:3w:  0      logic:3aq: 0      logic:3bk: 0
# logic:3d:  0      logic:3x:  0      logic:3ar: 0      logic:3bl: 0
# logic:3e:  0      logic:3y:  0      logic:3as: 0      logic:3bm: 0
# logic:3f:  0      logic:3z:  0      logic:3at: 0      logic:3bn: 0
```

```
# logic:3g:  0      logic:3aa: 0      logic:3au: 0      logic:3bo: 0
# logic:3h:  0      logic:3ab: 0      logic:3av: 0      logic:3bp: 0

search-f
nop-A
nop-A
add
inc
allocate
push
nop-B
pop
nop-C
sub
nop-B
copy
inc
if-n-equ
jump-b
nop-A
divide
nop-B
nop-B
```

Next, here is the length 80 ancestor; its genome has been manually commented.

```
# Filename........: base.out
# Update Output...: 0
# Is Viable.......: 1

# Generation: 0         Divide-1     Divide-2      Average
# Merit...........:           47           57           52
# Gestation Time..:          353          344        348.5
# Fitness.........:     0.133144     0.165698     0.149421
# Errors..........:            0            0
# Code Size.......:           80           80
# Copied Size.....:           80           80           80
# Executed Size...:           47           57           52
# Offspring.......:         SELF         SELF

# get:       0      logic:3i:  0      logic:3ac: 0      logic:3aw: 0
# put:       0      logic:3j:  0      logic:3ad: 0      logic:3ax: 0
# logic:1a:  0      logic:3k:  0      logic:3ae: 0      logic:3ay: 0
# logic:1b:  0      logic:3l:  0      logic:3af: 0      logic:3az: 0
# logic:2a:  0      logic:3m:  0      logic:3ag: 0      logic:3ba: 0
# logic:2b:  0      logic:3n:  0      logic:3ah: 0      logic:3bb: 0
# logic:2c:  0      logic:3o:  0      logic:3ai: 0      logic:3bc: 0
# logic:2d:  0      logic:3p:  0      logic:3aj: 0      logic:3bd: 0
# logic:2e:  0      logic:3q:  0      logic:3ak: 0      logic:3be: 0
# logic:2f:  0      logic:3r:  0      logic:3al: 0      logic:3bf: 0
# logic:2g:  0      logic:3s:  0      logic:3am: 0      logic:3bg: 0
# logic:2h:  0      logic:3t:  0      logic:3an: 0      logic:3bh: 0
# logic:3a:  0      logic:3u:  0      logic:3ao: 0      logic:3bi: 0
```

```
# logic:3b:  0      logic:3v:  0      logic:3ap: 0      logic:3bj: 0
# logic:3c:  0      logic:3w:  0      logic:3aq: 0      logic:3bk: 0
# logic:3d:  0      logic:3x:  0      logic:3ar: 0      logic:3bl: 0
# logic:3e:  0      logic:3y:  0      logic:3as: 0      logic:3bm: 0
# logic:3f:  0      logic:3z:  0      logic:3at: 0      logic:3bn: 0
# logic:3g:  0      logic:3aa: 0      logic:3au: 0      logic:3bo: 0
# logic:3h:  0      logic:3ab: 0      logic:3av: 0      logic:3bp: 0

nop-A          # Start Label
nop-A
nop-A
nop-A
call           # Call to size-calclulation module.
nop-A
nop-B
nop-A
nop-A
swap-stk       # Save the size on the second stack.
push
nop-B
swap-stk
call           # Call the copy module.
nop-A
nop-B
nop-B
nop-A
swap-stk       # Restore the size from the second stack.
pop
nop-B
swap-stk
jump-b         # Restart the copy section of the genome
nop-C
nop-A
nop-C
nop-C
divide         # Section divisor (never executed)
divide
nop-B          # Beginning of size-calculation module.
nop-C
nop-B
nop-B
search-b       # Search for distance to beginning of code.
nop-C
nop-C
nop-C
nop-C
add
push
nop-B
search-f       # Search for distance to end of code.
nop-C
nop-C
nop-C
```

```
nop-C
add
add             # Add forwards, backwards, and 4 extra for middle.
pop
nop-C
add
return          # Return size in BX (end size-calculation module)
nop-B           # Beginning of ''copy'' module
nop-C
nop-C
nop-B
allocate        # Allocate space
push            # Move size into BX and clear CX
nop-B
pop
nop-C
sub
nop-B           # Copy Loop
nop-A
nop-B
nop-C
copy
inc
if-n-equ
jump-b
nop-A
nop-C
nop-A
nop-B
divide          # Divide off offspring
return          # End of copy module.
nop-A           # End of genome label
nop-A
nop-A
nop-A
```

Finally, here are two organisms extracted from the sample experiment presented in Chapter 2. The first is the dominant organism at update 25,000. The genome has been divided over two columns to conserve space.

```
# Filename........: genebank/147-aaann
# Update Output...: 25000
# Is Viable.......: 1

# Generation: 0         Divide-1     Divide-2      Average
# Merit...........:       134553       103503       119028
# Gestation Time..:          952          908          930
# Fitness.........:      141.337       113.99      127.664
# Errors..........:           22           19
# Code Size.......:          136          136
```

```
# Copied Size.....:          147          147          147
# Executed Size...:          147          147          147
# Offspring.......:         SELF         SELF

# get        12     logic:3i   0     logic:3ac  0     logic:3aw  0
# put        43     logic:3j   0     logic:3ad  0     logic:3ax  0
# logic:1a    2     logic:3k   0     logic:3ae  0     logic:3ay  0
# logic:1b    1     logic:3l   2     logic:3af  0     logic:3az  0
# logic:2a    2     logic:3m   0     logic:3ag  0     logic:3ba  0
# logic:2b    2     logic:3n   0     logic:3ah  0     logic:3bb  0
# logic:2c    2     logic:3o   2     logic:3ai  0     logic:3bc  0
# logic:2d    2     logic:3p   0     logic:3aj  0     logic:3bd  0
# logic:2e    0     logic:3q   2     logic:3ak  0     logic:3be  0
# logic:2f    2     logic:3r   0     logic:3al  0     logic:3bf  0
# logic:2g    0     logic:3s   0     logic:3am  0     logic:3bg  2
# logic:2h    0     logic:3t   0     logic:3an  0     logic:3bh  2
# logic:3a    0     logic:3u   0     logic:3ao  0     logic:3bi  0
# logic:3b    0     logic:3v   0     logic:3ap  0     logic:3bj  2
# logic:3c    1     logic:3w   0     logic:3aq  0     logic:3bk  0
# logic:3d    0     logic:3x   0     logic:3ar  0     logic:3bl  0
# logic:3e    0     logic:3y   0     logic:3as  1     logic:3bm  0
# logic:3f    0     logic:3z   4     logic:3at  0     logic:3bn  0
# logic:3g    0     logic:3aa  0     logic:3au  0     logic:3bo  1
# logic:3h    0     logic:3ab  0     logic:3av  0     logic:3bp  0
```

```
nop-A
put
if-n-equ
if-less
if-less
call
nop-A
put
get
pop
if-bit-1
put
nop-B
nand
push
nand
get
swap
nand
nand
push
put
add
dec
nand
search-b
pop
swap
```

```
search-f
put
copy
nop-A
inc
nop-B
allocate
dec
inc
allocate
search-f
add
call
inc
copy
copy
shift-r
dec
allocate
if-n-equ
swap
sub
add
get
push
pop
nand
nand
```

```
dec
sub
put
add
put
sub
swap
inc
allocate
if-less
return
search-b
nop-C
inc
allocate
add
add
nop-C
add
copy
add
divide
add
allocate
push
swap
allocate
copy
nand
allocate
nop-A
put
push
nop-B
copy
inc
if-less
jump-b
nop-A
push
push
search-f
inc
push
copy
swap
put
shift-r
inc
allocate
call
inc
shift-r
nop-B
if-bit-1
if-less
call
nand
put
jump-b
nand
get
nop-B
nand
nop-B
push
if-n-equ
nand
nop-B
put
nop-C
put
copy
shift-l
jump-b
allocate
pop
swap
nand
nand
put
return
return
search-b
allocate
shift-r
search-f
nop-A
inc
copy
if-less
```

This last genome is the result after all 50,000 updates of the sample experiment.

```
# Filename........: genebank/143-abcyx
# Update Output...: 50000
# Is Viable.......: 1
```

```
# Generation: 0          Divide-1      Divide-2      Average
# Merit.............:  9.79906e+06  9.79906e+06  9.79906e+06
# Gestation Time....:         1625          1539          1582
# Fitness...........:      6030.19       6367.16       6198.67
# Errors............:            1             0
# Code Size.........:          129           129
# Copied Size.......:          143           143           143
# Executed Size.....:          143           143           143
# Offspring.........:         SELF          SELF

# get        48    logic:3i   0    logic:3ac  0    logic:3aw  0
# put       134    logic:3j   0    logic:3ad  0    logic:3ax  2
# logic:1a    6    logic:3k   0    logic:3ae  0    logic:3ay  0
# logic:1b    2    logic:3l   5    logic:3af  0    logic:3az  0
# logic:2a    2    logic:3m   0    logic:3ag  0    logic:3ba  0
# logic:2b    4    logic:3n   0    logic:3ah  0    logic:3bb  0
# logic:2c    6    logic:3o   2    logic:3ai  2    logic:3bc  0
# logic:2d    9    logic:3p   0    logic:3aj  0    logic:3bd  0
# logic:2e    2    logic:3q   4    logic:3ak  0    logic:3be  0
# logic:2f    2    logic:3r   0    logic:3al  0    logic:3bf  0
# logic:2g    0    logic:3s   0    logic:3am  0    logic:3bg  2
# logic:2h    0    logic:3t   0    logic:3an  0    logic:3bh  4
# logic:3a    1    logic:3u   0    logic:3ao  0    logic:3bi  0
# logic:3b    3    logic:3v   0    logic:3ap  0    logic:3bj  9
# logic:3c    7    logic:3w   2    logic:3aq  0    logic:3bk  0
# logic:3d    2    logic:3x   0    logic:3ar  0    logic:3bl  0
# logic:3e    0    logic:3y   0    logic:3as  3    logic:3bm  2
# logic:3f    2    logic:3z  10    logic:3at  0    logic:3bn  0
# logic:3g    0    logic:3aa  0    logic:3au  0    logic:3bo  2
# logic:3h    0    logic:3ab  0    logic:3av  0    logic:3bp  0
```

```
nop-C                          copy
push                           push
copy                           search-b
put                            dec
put                            nand
call                           jump-b
nop-A                          if-bit-1
sub                            get
get                            sub
get                            copy
swap                           nop-A
nop-B                          call
nop-B                          if-less
nand                           dec
nand                           jump-b
get                            put
swap                           push
nand                           jump-f
nand                           sub
push                           swap
if-less                        if-n-equ
```

```
add
inc
put
search-f
pop
swap
dec
nand
put
add
if-less
put
put
pop
if-bit-1
return
search-b
nop-C
inc
divide
add
add
add
nop-C
add
add
push
add
allocate
if-n-equ
swap
swap-stk
nop-B
nop-B
put
nop-B
nop-C
nand
put
nop-B
copy
inc
if-less
jump-b
nop-A
swap
sub
swap-stk
call
search-f
call
nop-A
get
push
pop
nand
nand
put
jump-b
if-less
if-less
call
inc
inc
nop-B
call
dec
copy
shift-l
get
nop-B
nand
copy
push
copy
nand
put
nop-C
put
search-f
inc
inc
shift-r
pop
add
swap
nand
nand
put
return
pop
swap
search-b
if-less
call
sub
nand
add
sub
get
nop-C
```

Summary of Variables

w_i	fitness of genotype i
w_{dom}	fitness of dominant genotype
$\langle w \rangle$	fitness of average genotype
A_i	birth rate of genotype i
D_g	genomic diffusion rate
F	fidelity
F_ν	neutral fidelity
H	genotype entropy
H_s	species entropy
$\mathcal{I}$	inferiority
$\mathcal{M}$	merit
N	number of organisms in population
N_g	number of genotypes in population
N_s	number of species in population.
N_t	number of threshold genotypes in population.
n_i	abundance of genotype i
t_a	allocated time
t_g	gestation time

Glossary

Abundance: The total number of sub-taxa within a taxon. For example, we will commonly look at the total abundance of organisms within a genotype, or the abundance of genotypes within a species.

Adaptive System: A system in which a population of solutions will evolve to optimize an *extrinsic* fitness function. Typically these solutions will have no direct interactions with each other; they will only be evaluated for their fitness, and those with the maximal fitness will be selected to survive and propagate.

Ancestor: The organism used to initialize the population in an avida experiment.

Auto-Adaptive System: A system of *self-replicating* agents ("solutions") in an environment with an *implicit* fitness function. An agent's ability to interact with both the environment and other agents determines how well it is able to reproduce. Only indirect control over that environment is used to direct the evolution of these agents.

Birth-Rate: The number of offspring per update an organism (or genotype) is expected to have. This can be approximated by

$$A = \frac{\text{fitness}}{\text{ave\_merit}} * \text{ave\_time\_slice} = \frac{w}{\langle \mathcal{M} \rangle} \langle t_a \rangle \tag{G.1}$$

This value depends on the average merit $\langle \mathcal{M} \rangle$, and hence on other organisms currently active in the environment. The inverse of this value is the expected number of updates it would take for the organism to have an offspring given its current competition.

Clone: An exact copy of an entire population. These are typically used to begin experiments at an adapted state.

Consensus Sequence: the genotype obtained by picking at each location the instruction that is the most frequent in the population. This measure can strictly only be defined for sequences of the same length. If the most abundant genotype has more than 50 percent of the population, this genotype automatically becomes the consensus sequence. After equilibration, the consensus sequence usually has zero or close to zero representatives in the population (approach to the error threshold).

Copied Length: The number of lines in a program that were copied into it from its parent. All lines that were *not* copied from the parent are typically random.

Copy Mutation: A stochastic event occurring when copying a single line of code from one point in memory space to another. Typically this event, affecting the `copy` instruction, results in the instruction being written to be different from the one that was read (while still being a legal instruction).

Cosmic-Ray Mutation: See Point Mutation.

CPU (Central Processing Unit): This is the machine that processes the genome of an organism. By default, it consists of a memory space, three registers, two stacks, a facing, I/O buffers, and an instruction pointer. The CPU will move through the instructions in memory, executing each and then advancing. Most instructions (unless defined otherwise) will deterministically alter the state of the CPU.

Digital Organism: A single program located at a lattice point in avida. A digital organism consists of a genome and a CPU executing that genome.

Effective Mutation Probability: The probability of a specific program (or its offspring) to be mutated during the replication process. See also: *Fidelity.*

Energy: A measure of the average *inferiority* in the population. See also: *Inferiority.*

Entropy: A measure used to determine the *disorder* in the population, according to Shannon Information Theory. In this measure, the probability of occurrence of

a single genotype i in the population, p_i, is approximated by n_i/N:

$$H = -\sum_i \frac{n_i}{N} \log \frac{n_i}{N} \tag{G.2}$$

where n_i is the current abundance of this genotype and N is the total number of strings in the population. See also: *Per-Site Entropy.*

Executed Length: The number of instructions in the genome of a program that are executed at least once during the course of its lifetime. A single `nop` used to modify the register an instruction interacts with *does* count as an executed instruction, but full labels only have their first `nop` counted (if these were counted in full, it could cause programs to develop long labels to maximize their executed length).

Fidelity: The probability for a string to correctly transmit its code to its offspring. The fidelity F is just 1-P, where P is the error probability. If only copy errors arise with probability R, the fidelity is

$$F = (1 - R)^{\ell} , \tag{G.3}$$

where ℓ is the length of the code. If insert and delete mutation occur with probability P_i and P_d respectively, the effective fidelity is

$$F = (1 - R)^{\ell}(1 - P_i)(1 - P_d) . \tag{G.4}$$

Fitness: A unit-less measure of the replication ability of a particular organism in a specified environment. By itself, fitness has little intrinsic meaning, but when compared to that of another organism it provides a comparison of their respective replication rates. Specifically, to calculate fitness, we take an organism's merit and divide it by its gestation time. ($w = \mathcal{M}/t_g$). Since merit increases exponentially with the number of tasks acquired, fitness is best described by the log of its actual value (see also *Inferiority*).

Genome: The assembly language program used to define an organism. The genome initializes the memory component of the CPU when an organism is executed.

Genome Length: The number of assembly language instructions that form the program of the organism. See also *Copied Length* and *Executed Length.*

Genotype: A taxonomic level recorded in avida that represents all organisms with identical genomes. Genotype is one of the standard tools used to study avida, as all organisms of a particular genotype behave similarly given a fixed environment.

Gestation Time: The number of instructions an organism must execute to produce a single offspring. This is typically proportional to the length of the genome.

Inferiority: A measure which determines how much *worse* a particular genotype is than the genotype that is currently dominating the system. If $w_i \geq 0$ represents the fitness of genotype i, its inferiority is

$$\mathcal{I}_i = w_{\text{best}} - w_i \; . \tag{G.5}$$

Instruction: A single command in the assembly language of the CPU. When executed, an instruction modifies some of the parts of the CPU in a deterministic fashion.

Instruction Set: The collection of instructions in the assembly language the genomes are written in. Whenever an instruction is mutated, the new instruction is chosen at random from the instruction set (with all instructions given an equal probability of being selected).

Label: (Also called a template) A sequence of nops (no-operation instructions) in the genome that are used to modify the instruction that precedes them. They are also used to reference another point in the code where the *complement* label is located.

Merit: A value indicating the CPU time a particular organism has earned, taking into account its length and the tasks that it has successfully completed.

Mutation: Any random change caused to the genome of an organism. See *Copy Mutation* and *Point Mutation.*

Organism: A living individual program. See *Digital Organism.*

Per-Site Entropy: The expected number of bits required to specify the instruction that lies at a particular site in an organism's genome, given that we know the distribution of genomes in the population it was extracted from. Sites are typically labeled "hot" if they very wildly across the population (maximal entropy), and "cold" if they remain conserved among all of the genomes. See also: *Entropy.*

Phenotype: A classification system that measures *what* an organism can do without ever checking *how* it is done. In other words, the phenotype reflects gestation time, tasks completed, bonus earned, and the like, but never takes into account the source code (the genotype).

Point Mutation: (Also called cosmic-ray mutations). This form of mutation is a random change from one instruction to another in the memory space of an organism. This can occur at any time and is not limited to whether the CPU is executing a particular task, or even executing at all.

Population: The collection of all active organisms on the lattice in an avida experiment.

Quasispecies: A cloud of organisms that are functionally equivalent, all being neutral mutations off of a single wild-type genome.

Replication Rate: The absolute speed at which an organism can self-replicate, i.e., the number of offspring per unit time. Typically, this is the same as *Fitness.*

Self-Replication: The process an organism goes through in making an exact copy of its genotype into an offspring organism.

Species: A taxonomic level above genotype. All organisms in a species are similar on a functional and structural level, but not necessarily in all instruction positions on their genome. Species can be used to study clouds around an archetype (*quasispecies*) in genome space.

Task: A feature imposed on the environment that can be triggered by certain actions of an organism that will cause the merit (and hence fitness) of that organism to increase.

Template: See Label; this a the biological term for a sequence of genetic code used in pattern matching.

Threshold Genotype: A genotype that has reached a minimum abundance (usually three). This is used to determine if the genotype is properly self-replicating (since it would be unlikely to observe this many copies of an organism that could not properly replicate itself). For this reason, some statistics are taken only on threshold genotypes.

Time-slice: The number of instructions executed in a particular CPU during a single update. By default (and in most avida configurations), this is proportional to the *merit* of that organism.

Time-slicer: The portion of code in avida that doles out time slices to CPUs, and is responsible for executing the proper number of instructions in those CPUs.

Unrolling the Loop: An evolutionary step populations will often take to lower gestation times. This process involves copying *two or more* instructions each time through the copy loop to minimize the effect of loop overhead.

Update: An artificial unit of time during which all organisms execute their time-slice, which is on average 30 instructions. All statistics are collected at the end of each update.

Bibliography

[1] Adami C, Learning and complexity in genetic auto-adaptive systems, *Physica* **D 80**: 154 (1995).

[2] Adami C, Self-organized criticality in living systems, *Phys. Lett. A* **203**: 29-32 (1995).

[3] Adami C, On Modeling Life, *Artificial Life* **1**: 429-438 (1995).

[4] Adami C, *Introduction to Artificial Life*, Telos Springer-Verlag, NY (1998).

[5] Adami C and Brown CT, in *Proc. of Artificial Life IV*, R. A. Brooks and P. Maes, Eds., MIT Press, Cambridge, MA (1994), p. 377.

[6] Adami C, Brown CT, and Haggerty M, Abundance Distributions in Artificial Life and Stochastic Models: "Age and Area" revisited, *Lect. Notes Artif. Intell.* **929**: 503 (1995).

[7] Adami C and Cerf NJ, Prolegomena to a non-equilibrium quantum statistical mechanics. *Chaos, Solitons, Fract.* **10**: 1637-1650 (1999).

[8] Adami C and Cerf NJ, Physical complexity of symbolic sequences. *Physica* **D**, *to appear.*

[9] Adami C, Collier TC, and Ofria C, Robustness and evolvability of computer languages, *in preparation.*

[10] Adami C, Ofria C, and Collier TC, Evolution of biological complexity: I. Physical complexity of genomes, *in preparation.*

[11] Basharin GP, On a statistical estimate of the entropy of a sequence of independent random variables, *Theory Probability Appl.* **4**: 333-336 (1959).

[12] Bennett CH, Universal computation and physical dynamics, *Physica* **D 86**: 268-273 (1995).

[13] Brillouin L, *Science and Information Theory*, Academic Press, NY (1962).

[14] Britten RJ and Davidson EH, Gene regulation for higher cells: A Theory, *Science* **165**: 349-357 (1969).

[15] Britten RJ and Davidson EH, Repetitive and non-repetitive DNA sequences and a speculation on the origins of evolutionary novelty. *Quart. Rev. Biol.* **46**: 111-138 (1971).

[16] Cavalier-Smith T, Eukaryotic gene numbers, non-coding DNA and genome size. Pp. 69-103 *in* Cavalier-Smith T, ed. *The evolution of genome size*, John Wiley, New York (1985).

[17] Chu J and Adami C, in *Proc. of Artificial Life V*, Langton CG and Shimohara T, Eds., MIT Press, Cambridge, MA (1997), p. 462.

[18] Dawkins R, *The Selfish Gene*, Oxford University Press (1976).

[19] Deutsch D, *The Fabric of Reality*, The Penguin Press, NY (1997), p. 179.

[20] Dewdney A, In the game called Core War hostile programs engage in a battle of bits. *Sci. Amer.* **250/5**: 14 (1984).

[21] Dixon M and Webb EC, *The Enzymes*. Academic Press, NY, 2nd. Ed. (1964).

[22] Eigen M, The physics of molecular evolution, in *Molecular Evolution of Life*, Baltscheffsky H, Jörnvall H, and Rigler R, eds., Cambridge Univ. Press, Cambridge, MA (1986), p. 13–26.

[23] Eigen M, *Naturwissenschaften* **58**, 465 (1971).

[24] Elena SF, Cooper VS and Lenski RE, Punctuated evolution caused by selection of rare beneficial mutations, *Science* **272**: 1802-1804 (1996).

[25] Elena SF and Lenski RL, Test of synergistic interactions among deleterious mutations in bacteria, *Nature* **390**: 395-398 (1997).

[26] Ereshefsky M (Ed.) *The Units of Evolution: Essays on the Nature of Species.* MIT Press, Cambridge, MA (1992).

[27] Fogel DB, *Evolutionary Computation: Toward a New Philosophy of Machine Intelligence*, IEEE Press, NY (1995).

[28] Gatlin LL, *Information Theory and the Living System*, Columbia University Press, New York (1972).

[29] Gould SJ, *Full House*, Harmony Books, NY (1996)

[30] Holland JH, *Adaptation in Natural and Artificial Systems*, Ann Arbor, University of Michigan Press (1975).

[31] Kauffman SA and Levin S, Towards a general theory of adaptive walks on rugged landscapes, *J. Theor. Biol.* **128**: 11 (1987).

[32] Kauffman SA and Johnsen S, Coevolution to the edge of chaos—Coupled fitness landscapes, poised states, and coevolutionary avalanches, *J. Theor. Biol.* **149**: 467 (1991).

[33] Keese P and Gibbs A, Origins of genes: "Big bang" or continuous creation? *Proc. Natl. Acad. Sci.* **89**: 9489-9493 (1992).

[34] Kimura M, *The neutral theory of molecular evolution*, Cambridge University Press, Cambridge (1983).

[35] Koza JR, *Genetic Programming*, MIT Press, Cambridge, MA (1992).

[36] Landauer R, Information is physical. *Physics Today* **44(5)**: 23-29 (1991).

[37] Leff HS and Rex AF, Eds, *Maxwell's Demon: Entropy, Information, Computing*, Princeton University Press, Princeton (1990).

[38] Lenski R, Evolution in experimental populations of bacteria, in *Population Genetics of Bacteria*, Society for General Microbiology, Symposium 52, Baumberg S, Young JPW, Saunders SR, and Wellington EMH, eds., Cambridge University Press, Cambridge (1995) 193-215.

[39] Lenski R, Rose MR, Simpson SC, and Tadler SC, Long-term experimental evolution in *Escherichia coli.* I. Adaptation and divergence during 2,000 generations, *American Naturalist*, **138**: 1315-1341 (1991).

[40] Lenski RL, Ofria C, Collier TC, and Adami C, Genomic Complexity, robustness, and genetic interactions in digital organisms, *Nature, in press.*

[41] Lerner AY, *Fundamentals of Cybernetics*, Plenum Pub. Corp., NY (1975).

[42] Maxwell JC, *Theory of Heat*, Longmans, London (1871).

[43] Maynard Smith J, Natural selection and the concept of a protein space. *Nature* **225**: 563 (1970).

[44] Mc Shea DW, Metazoan complexity and evolution: Is there a trend? *Evolution* **50**: 477-492. (1996).

[45] Mills DR, RL Peterson, and S Spiegelman, An extracellular Darwinian experiment with a self-duplicating nucleic acid molecule, *Proc. Nat. Acad. Sci. USA* **58**: 217 (1967).

[46] Miyata T and Yasunaga T, Evolution of overlapping genes, *Nature* **272**: 532 (1978).

[47] Mizokami M, Orito E, Ohba K, Lau JYN, and Gojobori T, Constrained evolution with respect to gene overlap of Hepatitis B virus, *J. Mol. Evol.* **44**(Suppl. 1): S83-S90 (1997).

[48] Modiano G, Battistuzzi G, and Motulsky AG, Nonrandom patterns of codon usage and of nucleotide substitutions in human α- and β-globin genes: An evolu-

tionary strategy reducing the rate of mutations with drastic effects? *Proc. Nat. Acad. Sci. USA* **78**: 1110-1114 (1981).

[49] Muller HJ, The relation of recombination to mutational advantage. *Mut. Res.* **1**: 2-9 (1964).

[50] Normark S, Bergström S, Edlund T, Grundström T, Jaurin B, Lindberg FP, and Olsson O, Overlapping genes, *Ann. Rev. Gen.* **17**: 499-525 (1983).

[51] Ofria C and Adami C, Evolution of genetic organization in digital organisms, *Proc. of DIMACS Workshop on Evolution as Computation*, Jan 11-12, Princeton University, Landweber L and Winfree E, eds., Springer-Verlag, NY, (1999).

[52] Ofria C, Adami C and Collier TC, Evolution of Biological Complexity: II. Selective Pressures on Genome Size and Neutrality, *in preparation*

[53] Ofria C, Adami C, Collier TC, and Hsu GK, Evolution of differentiated expression patterns in digital organisms, *Proc. of the Fifth European Conference on Artificial Life*, to appear (1999).

[54] Ofria C, Brown CT, and Adami C, *The Avida User's Manual*, in [4] 297-350 (1998).

[55] Rasmussen S, Knudsen C, Feldberg R, and Hindsholm M, *Physica* **D 42**: 111 (1990).

[56] Ray TS, An approach to the synthesis of life, in *Proc. of Artificial Life II*, Langton CG, Taylor C, Farmer JD, and Rasmussen S, Eds., Addison Wesley, Redwood City, CA (1992), p. 371.

[57] Ray TS, Evolution, complexity, entropy, and artificial reality, *Physica* **D 75**: 239 (1994).

[58] Ray TS, A proposal to create a network-wide biodiversity reserve for digital organisms, ATR Technical Report TR-H-133 (1995).

[59] Ray TS and Hart J, Evolution of differentiated multi-threaded digital organisms, in *Proc. of Artificial Life VI*, Adami C, Belew RK, Kitano H, and Taylor CE, eds. MIT Press, Cambridge, MA (1998), p. 295.

[60] Samuel CE, Polycistronic animal virus messenger RNAs, *Prog. Nucleic Acids Res. Mol. Biol.* **37**: 127-153 (1989).

[61] Schneider TD, Stormo GD, Gold L, and Ehrenfeucht A, Information content of binding sites on nucleotide sequences, *J. Mol. Biol.* **188**: 415-431 (1986).

[62] Schrödinger E, *What is Life?* Cambridge University Press, Cambridge (1945).

[63] Shannon CE and Weaver W, *The Mathematical Theory of Communication*, University of Illinois Press, Urbana (1949).

[64] Thearling K and Ray TS, Evolving multi-cellular artificial life, in *Proc. of Artificial Life IV*, Brooks RA and Maes P, Eds., MIT Press, Cambridge, MA (1994), p. 283.

[65] Thearling K and Ray TS, Evolving parallel computation, *Complex Systems* **10**: 229-237 (1997).

[66] Travisano M, Mongold JA, Bennett AF, and Lenski RE, Experimental tests of the roles of adaptation, chance, and history in evolution, *Science* **267**: 87-90 (1995).

[67] Watson JD, et al., *Molecular Biology of the Gene*, Fourth Ed., Benjamin Cummings, Menlo Park (1987).

[68] Wiley EO and Brooks DR, Victims of history—A nonequilibrium approach to evolution, *Syst. Zool.* **32**: 209-219 (1982).