IMPLEMENTING VON NEUMANN'S ARCHITECTURE FOR MACHINE SELF REPRODUCTION WITHIN THE TIERRA ARTIFICIAL LIFE PLATFORM TO INVESTIGATE EVOLVABLE GENOTYPE-PHENOTYPE MAPPINGS

Declan Baugh, M.Eng. B.Sc.
School of Electronic Engineering, Dublin City University

Supervisor: Prof. Barry McMullin

A thesis submitted for the degree of Ph.D.

July 2015

Declaration

I hereby certify that this material, which I now submit for assessment on the programme of study leading to the award of Ph.D., is entirely my own work, and that I have exercised reasonable care to ensure that the work is original, and does not to the best of my knowledge breach any law of copyright, and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my work.

Signed: _____ (Candidate), ID No.: 53605574, Date: _____

Dedication

All of my work is dedicated to Trevor - who inspires, supports, and protects.

Acknowledgements

I would like to express my special appreciation and thanks to my supervisor Prof. Barry McMullin; you have been a tremendous mentor for me. I would like to thank you for directing and encouraging my research over the years. For all the long meetings delivering advice and guidance as well as the countless hours correcting and honing my academic writing skills. Your support over the past years has been invaluable and I sincerely appreciate the effort that you invested in me. I would also like to thank my colleague, Dr. Tomonori Hasegawa for his assistance and companionship over the years as we battled this Ph.D. together.

I would also like to express my sincere gratitude to Dr. Tim Taylor and Dr. Alistair Sutherland for assuming the task of reading my thesis and sitting as my external and internal examiners, and Dr. Darragh O'Brien for sitting as my examiner for my transfer report.

I wish to thank the Tierra developers, mainly Thomas S. Ray, without whom this thesis would have never been, the European Complexity Network (Complexity-NET) through the Irish Research Council for Science and Technology (IRCSET) under the collaborative project EvoSym for granting the funding for this Ph.D., and our partner labs, the Artificial Intelligence Laboratory - Vrije Universiteit Brussel, and the Bioinformatics Group - Universiteit Utrecht for their collaboration.

Finally I would like to say a special thanks to my family, friends and girlfriend Catherine. Words cannot express how grateful I am to my Mother, for all of the sacrifices that you've made on my behalf; I am forever grateful.

Contents

Abstract

John von Neumann first presented his theory of machine self reproduction in the late 1940's in which he described a machine capable of performing the logical steps necessary to accommodate self reproduction, and provided an explanation in principle for how arbitrarily complex machines can construct other (offspring) machines of equal or even greater complexity. In this thesis, a machine having the von Neumann architecture for self reproduction is designed to operate within the computational world of Tierra. This design implements a (mutable) genotype-phenotype mapping during reproduction, and acts as an exploratory model to observe the phenomena which may arise with such a system. A substitution mapping was chosen to carry out the genotype-phenotype mapping, and two specific implementations of a substitution mapping were investigated, via the use of a look-up table and a translation table. During implementation of the look-up table, preliminary experiments showed a degeneration to *self copiers* where a lineage of von Neumann style self reproducers degenerated into self copiers. Further experiments showed that a particular phenomenon emerges, where *pathological constructors* quickly develop, which can ultimately lead to total ecosystem collapse. If redundancy is introduced to the genotype-phenotype mapping, certain inheritable perturbations (mutations) prove to be non-reversible via a change to the genotype, which leads to a bias in the evolution of the genotype-phenotype mapping, consistently resulting in the loss of any target symbols from the mapping which are not vital for reproduction. It demonstrated how instances of Lamarkian inheritance may occur, which allowed these genetically "non-reversible" perturbations to be reversed, but only when accompanied by a very specific perturbation to the phenotype. The underlying dynamics of the chosen coding system was studied in order to better understand why these phenomena occur. When implementing a translation table, the space of possible mutations to the genotype-phenotype mapping was investigated and the same phenomena observed, where non vital symbols were lost from the mapping, and an instance of Lamarkian inheritance is necessary in order to introduce symbols to the mapping.

Chapter 1

Introduction

1.1 Chapter Overview

The story of life, from its origin on Earth approximately 3.5 billion years ago to the extremely complex living systems that we see today is a still under much investigation. The field of Artificial life aims to study the logic of living systems in artificial environments in order to gain a deeper understanding of the complex information processing that defines such systems.

To quote Christopher Langton "The aim of artificial life research is to study life-as-it-could-be, in order to advance our understanding of life-as-we-know-it" (Langton, 1989). However, it is difficult to have any preconceptions what form life might take in the digital medium.

In order to design artificial life systems which accurately emulate the properties of living systems, it is important that we first develop a thorough understanding of the fundamental properties which define systems which have proven to support evolution in the carbon based medium, and explore if they can provide us with the foundations to realize complex evolution in silico.

This chapter discusses the fundamental properties of life which must be understood before attempting to recreate artificial life, in silico. Project EvoSym is introduced, which was a collaborative project between the Dublin City University Artificial Life group, the Bioinformmatics group of Utrecht, and the Artificial Intelligence Laboratory at Vrije University Brussels, whose aim was to investigate different aspects of the evolution and emergence of biological symbol systems in different fields of artificial life. The contributions that this thesis made towards Project Evosym are listed.

1.2 What is Life?

1.2.1 Fundamental Properties of Life

Some of the earliest theories of life were materialist, holding that all that exists is matter, and that life appears to be a property of the organisation of matter, not a property inherent to the fundamental components which comprise lifeforms themselves. Basic components which carbon based life forms are comprised of, such as amino acids, nucleotides, proteins etc., are not categorised as living organisms, but when these basic components are organised in the correct manner, emergent properties develop, which are characterised as life. Emergent properties are global behavioural patterns which arise from complex systems as a result of fundamental, local interactions between the basic underlying components. The emergent properties of life are the result of systems of chemical components, where the complexity of their global behaviours are qualitatively distinct from the complexity of all their local behaviours. Living systems do not obey any simple principle of superposition, so it is difficult to examine these emergent properties by axiomatizing the system and studying the individual parts in isolation. The principal feature of an emergent system is that its properties are a result of the complex interaction between its lower level components, and these properties may cease to exist if the system is de-constructed in order to be examined.

Rather then studying the components in isolation, we can compare and contrast the most primitive examples of life as we know it, and classify the primary behaviours which are common across all forms of such carbon based life forms. The fundamental properties which a system must incorporate in order to be classified as a genuine form of life has been subject to much debate. There is currently no universally accepted definition of the properties which a system must possess in order to be considered living.

An early attempt to gather together several strands of scientific research and question "what is life", and "how can the events in space and time which take place within the spatial boundary of a living organism be accounted for by physics and chemistry?" was put forth by Erwin Schrödinger (1944). According to Benner (2010), a discussion in 1994 by The National Aeronautics and Space Administration (NASA), following a suggestion by astrophysicist Carl Sagan, proposed that life is a "self-sustaining chemical system capable of Darwinian evolution." (Joyce et al., 1994). However, the "NASA definition" only illustrates a definition of life-as-we-know-it, as it excludes any conceivable non-chemical system which may also fit the description of life. The field of artificial life investigates life-as-it-could-be, which includes the possibility of electro-chemical systems or digital systems which may support life.

When attempting to realize the limits of artificial life, which takes into account any conceivable form of life, not just the carbon based life as we know it, Claus Emmeche (1992) put forward the following list.

- Life is not medium-independent, but shows an interdependence of form and matter.

- Life may be realized in other media than the carbon-chain dominated as a result of a long, natural evolutionary process.
- Artificial life research may contribute to theoretical biology by: simulating developmental and evolutionary phenomena of life on Earth, simulating life as it could have evolved in non-earthly environments given some set of realistic boundary conditions, providing new concepts and models of emergent phenomena belonging to a general set of complex systems of which biological systems (under particular kinds of descriptions) may be a subset.
- Artificial life may inspire attempts to realize life artificially in other media by in vitro experiments. Such prospects include the experimental approach of molecular biology and protobiology research. However, this is not yet the centre of interest in the present artificial life research programme.

There is no however, no universally accepted, single property that can be used to perfectly characterise life. Any single property that might be assigned to life will either be too broad in that it also characterises many non-living systems, or too specific, so that there will exist biological systems, which are already categorised as living, but will not satisfy the single proposed property. Instead, a formal definition of life might constitute a list of specific criteria which describes all living systems, and excludes all systems which are not considered to be living. An attempt to define life was proposed by Farmer and A Belin which consisted of a set of eight essential, fundamental criteria which bear on the nature of life as we know it (Farmer & Belin, 1991). Their proposed list of properties that they associate with life is as shown in Figure 1.1.

1.2.2 Hierarchy of Living Properties

The aforementioned list of properties is referred to by many artificial life researchers as an accepted definition of life e.g, (Heudin, 1995), however, from the point of view of designing an artificial life system which characterises life-like behaviours, it may not be completely satisfactory. This set of criteria does not accurately categorise the essential, fundamental criteria which bear on the nature of life, as it does not highlight the relative level of complexity of each criterion, which is an important aspect to take into consideration when designing systems which attempt to simulate the properties of life. In order to accurately define a framework to support life, it is more informative to construct a level-based hierarchy which consists of multiple levels of organisations, having dynamics within and between the members depicted at each level of hierarchy. The members (properties) are organised into levels and each level is a list. A chief

1. *Life is a pattern in space-time*, rather than a specific material object. For example, most of our cells are replaced many times during our lifetime. It is the pattern and set of relationships that are important, rather than the specific identity of the atoms.

2. *Self reproduction*, if not in the organism itself, at least in some related organism. (e.g. mules are alive, but cannot reproduce.)

3. *Information storage of a self-representation.* For example, contemporary natural organisms store a description of themselves in DNA molecules, which is interpreted in the context of the protein/RNA machinery.

4. *A metabolism* which converts matter and energy from the environment into the pattern and activities of the organism. Note that some organisms, such as viruses, do not have a metabolism of their own, but make use of the metabolisms of other organisms.

5. *Functional interactions with the environment.* A living organism can respond to or anticipate changes in its environment. Organisms create and control their own local (internal) environment.

6. *Interdependence of parts.* The components of living systems depend on one another to preserve the identity of the organism. One manifestation of this is the ability to die. If we break a rock in two, we are left with two smaller rocks; if we break an organism in two, we often kill it.

7. *Stability under perturbations and insensitivity to small changes*, allowing the organism to preserve its form and continue to function in a noisy environment.

8. *The ability to evolve.* This is not a property of an individual organism, but rather of its lineage.

Figure 1.1: Proposed fundamental criteria which bear on the nature of life as we know it. (Farmer & Belin, 1991, verbatim quotation, p. 4)

Level 0. Life is a pattern in space-time & functional interactions with the environment.

Level 1. Stability under perturbations.

Level 2. Interdependence of parts.

Level 3. Information storage of a self-representation & a metabolism.

Level 4. Self reproduction.

Level 5. The ability to support inheritable variation.

Figure 1.2: A proposed hierarchy which displays the relative levels of complexity of each fundamental property of life.

aspect of this concept is the fact that the different members at each level have different functionalities that emerge from the interactions from entities of the same, and lower-levels, therefore, I propose a specific hierarchy which is displayed in Figure 1.2. This level based hierarchy was developed based on the relative level of complexity of each fundamental property, with the least complex fundamental property on the bottom level. The properties of each level must be a result of the emergent properties of the levels below, while also capable of existing independently of the properties of the levels above.

Although the number of properties described by Farmer and Belin was initially 8, this has been reduced here to 6 levels, where certain properties are considered as sharing the same level of hierarchy.

Life is a Pattern in Space Time & Functional Interactions with the Environment

These are not criteria specifically for life, but the fundamental criteria for the universe in which life resides.

In order for matter as we know it to exist, firstly it must reside within space and time, and secondly, there must be fundamental laws of this universe which determine the behaviour of matter within space and time. In our physical universe, these interactions are expressed by the laws of physics and chemistry. Within computational systems, before a system is even inoculated with a digital presence, the system must contain memory, which allows patterns to be stored in space time, and a set of rules which specify the functional interactions between the patterns.

Stability Under Perturbations

The fundamental components of life must possess some appropriate degree of stability in order for life to preserve its form in dynamic environments. They must consist of

stable components, robust enough that they are capable of holding form when external forces are acting upon them, such as interactions with other entities within the system. In biological systems, this might correspond to the emergence of macromolecules, very large stable molecules typically composed as a polymerization of smaller subunits. Biologically important macromolecules include nucleic acids, proteins, carbohydrates and lipids, which are relatively insoluble in water and similar solvents.

Interdependence of Parts

It has already been discussed that life appears to be a property of the organisation of matter, and not a property inherent to the fundamental components which comprise a life form itself. Therefore, living systems such as cells are an organisation of interdependent parts, such as macromolecules, and if one component is removed the system may fail. Individual components cannot constitute life on their own, but a life-form as a whole may fail to function if one or more of its fundamental components is removed.

Information Storage of a Self Representation & Metabolism

Information storage must exist at a less fundamental level than the interdependence of parts, as stored information must be interpreted by a separate component which decodes and constructs the described entity. Similarly, a metabolism could not exist without the interdependence of parts, as it must convert energy and matter in the environment into the individual arrangement and activities of the organism.

There are two main schools of thought in how the origin of life unfolded, *genes-first* or *metabolism-first*. The genes-first hypotheses postulate that the early emergence of nucleic acids (an information storage) gave rise to biochemical reactions (a metabolism).

The metabolism-first hypotheses postulates that metabolism arose which gave rise to nucleic acids. However, we are not specifically focusing here on the origin of life, but the criteria for life, as it exists today. As the metabolism's function is to build, repair and guide the activity of matter within an organism, it must in general receive its instructions to do so via some form of information storage. Similarly, a storage of information must be initially organised from inorganic matter by a metabolism. Therefore in order for life as we know it to continue existing, there must be a co-dependence between the metabolism and information storage.

Self reproduction

Self reproduction relies on a metabolism, which exploits energy and organic matter from the environment, and arranges this matter in a form that replicates the relevant structure and behaviour of the parent. It may be argued that for simple organisms, in order to self reproduce the metabolism can simply self inspect the parent's arrangement, and construct the offspring to the same specifications, without the use of information storage of a self-representation. However, self reproduction cannot exist without a

metabolism, which in turn, is co-dependent on an information storage, so self reproduction should arguably be positioned a level of hierarchy higher than both metabolism and information storage.

The Ability to Support Inheritable Variation

The ability to support heritable variation is the final fundamental behavioural criterion which is stated as being common to all forms of life as we know it. Self reproduction is an essential criterion for Darwinian evolution, so this would suggest that the ability to evolve should be depicted as the highest hierarchical level, as it is an emergent property of the complete system of lower-level interactions.

However, a single organism in isolation cannot be subject to evolution by natural selection. Natural selection requires several interacting lineages of different instances of life. With this in mind, Farmer and Belin's exact wording of the criterion "the ability to evolve" may lead to some confusion.

In order for evolution to occur, instances of life must exist already, and the interactions between these life forms are what drives evolution by natural selection. If primitive life did not possess the ability to evolve and adapt to its surroundings, then as the environment changed in such a way that the rate of reproduction could no longer keep pace with the mortality rate, the population would have fallen to extinction.

The Earth's biosphere is a highly non-quiescent environment, therefore, one may state that "the ability to evolve, is *a fundamental property* of life on Earth". However, Darwinian evolution itself could not have emerged had a primitive instance of life not already existed. Therefore, one may also state that "the ability to evolve is *an emergent property* of life on Earth". These two statements contradict each other and may lead to confusion, as the ability to evolve cannot be both a fundamental property, and an emergent property. Therefore, "the ability to evolve" seems too vague a statement to use. *The ability to support inheritable variation* is a much more suitable criterion to define a single, primordial expression of life and less open to interpretation. With this new criterion, we can say that incorporating the ability to support inheritable variations is the final fundamental behavioural criterion which is common to all forms of life as we know it. *The ability to evolve* is simply an emergent property, which arises when selection occurs within an ecosystem, but not a fundamental property of a single instance of life.

1.3 Project EvoSym: Emergence and Evolution of Biological Systems

1.3.1 Introduction

Of the suggested criteria which bear on the nature of life that were discussed, the use of information storage as a self representation is a specific criterion which will

be the focus of this thesis. In order for information to be stored within a biological system, specific mechanisms must be in place which can store a pattern of tokens or symbols and interpret these symbols as information. This information in turn can be used as instructions to construct further biological systems. Project EvoSym was a collaborative project set up to investigate the emergence and evolution of examples of such systems, in which the Artificial Life group in DCU participated.

1.3.2 Project Collaboration and Affiliation

The Rince Institute, based in the Faculty of Engineering and Computing at Dublin City University, is an Irish national research centre focussed on innovation in engineering technologies. The DCU Artificial Life Group, within Rince[1], collaborated with partners in the Bioinformatics group of Utrecht University (The Netherlands), and the Artificial Intelligence Laboratory at Vrije University Brussels (Belgium), to carry out the EvoSym project[2] between October 2010 and September 2013. The partners combined inter-disciplinary expertise in biological evolution, language evolution, computation and robotics to enable a collaboration where specific relevant systems could be studied simultaneously, and important general principles could be extracted.

This project was funded locally by the Irish Research Council[3], under the Complexity-Net[4] funding scheme (European Network for the Coordination of Complexity Research and Training).

1.3.3 Project EvoSym Synopsis

Out of the previously discussed list of criteria which bear on the nature of life as we know it, this project most closely focused on understanding a specific aspect, "information storage as a self representation". This criterion requires the emergence and evolution of symbolism, where a sequence of symbols are read and interpreted as information.

Symbols play an extensive role in biological organisation on the micro and macro level, such as genetic coding, RNA and protein modification, cell signalling, epigenetics, gene regulation etc. The aim of the EvoSym project as a whole was to investigate the conditions for the emergence and evolution of such biosemiotic systems, and investigate potential technological applications of this understanding in robotics and distributed agent-based software. In summary, the two core questions which were addressed in the initial project proposal, were as follows (McMullin, 2010):

- How do complex representational and communicative coding systems emerge, self-organise and evolve, from micro to macro level in the natural biosphere?

[1] http://www.rince.ie
[2] http://evosym.rince.ie
[3] http://research.ie
[4] http://www.complexitynet.eu

- How can this biological understanding be applied to the artificial evolution of complex coding systems in computational and/or robotic systems?

The investigation of these questions was structured into three distinct, but complementary and inter-related Work Packages (WP), with each partner leading one package. These work packages will be briefly summarised here.

WP1: Evolutionary Emergence of Codes and the Role of Coding in the Evolution of Complex Biotic Systems (Universiteit Utrecht) A fundamental characteristic, shared by of all living systems on Earth, is that the responsibility of the storage of genetic information is exclusively dedicated to DNA, while proteins are responsible for catalytic behaviour. However, the *RNA world hypothesis*, first proposed by Gilbert (1986), posits that at earlier stages of evolution, the roles of both information storage and catalytic function were undertaken by RNA alone.

Throughout the natural progression of life on Earth, life has undergone major transitions in evolution, including the transition from asexual clones to sexual populations, the transition from prokaryotic cells to eukaryotes, and the transition from an RNA world to a DNA-RNA-protein world (Maynard Smith & Szathmáry, 1995). It has been argued by De Beule (2011) that "The crucial factor that determines whether an ecosystem will make a transition will be the capacity of the organisms to communicate. Only after having established a shared convention of code will the ecosystem undergo a transition."

This suggests that transitions in evolution which result in an increased level of complexity were accompanied by an increased level of symbolism or communication. The transition from an RNA world to a DNA-RNA-protein world must have been facilitated by the emergence of an accompanied coding system. It is well documented that an RNA system suffers from a severe limitation with respect to the amount of information which can be preserved accurately under a Darwinian replication-mutation selection regime, given the high mutation rates during early evolution (Eigen, 1971; McMullin, 2010). WP1 investigated whether the emergence of a shared coding system could help alleviate the problem of this information threshold in a prebiotic RNA world (Takeuchi et al., 2011).

WP2: Evolutionary Emergence of Codes and the Role of Coding in The Evolution of Complex Mobile Software Agent Systems (Dublin City University) As with WP1, this work package dealt with the emergence and evolution of abstract coding systems, however, specifically within computational environments. The problem originates from John von Neumann's work from 1948 - 1952 (Von Neumann, 1948), where he demonstrated how a machine architecture (real or virtual) which incorporates a division of labour between information storage (genotype), and catalytic behaviour (phenotype) satisfies some of the conditions for open ended evolutionary growth of complexity. This division of labour must be facilitated by an abstract genotype-phenotype

mapping, analogous to the translation of DNA to complex organic forms in biological, genetic reproduction.

It has been argued that evolutionary platforms such as Tierra (Ray, 1991) have demonstrated the evolutionary growth of complexity of digital automata (Ray, 1994). However, reproduction is conventionally implemented purely by self inspection (self copiers), somewhat analogous to the template-replication mechanism of the RNA world. This work package focused on implementing conditions in which von Neumann style reproduction may emerge and evolve in such computational worlds. Furthermore, it investigated what distinctive evolutionary phenomena may emerge within the genotype space when an evolvable genotype-phenotype mapping is implemented.

WP3: Using Principles and Insights from Evolutionary Linguistics to Understand the Evolutionary Emergence of Codes and the Role of Coding in the Evolution of Complexity (Vrije Universiteit Brussels) The main goal of WP3, was to investigate the generic mechanism for the emergence and evolution of complex multi-levelled codes, such as natural language. The mechanisms which compositional and grammatical languages are built upon are not fully understood. This work package aimed to develop new tools and techniques for both analysing and synthesizing such multi-levelled linguistic systems (De Beule, 2011).

1.3.4 Work Package 2: Elaboration and Thesis Question

The aim of this thesis is to investigate a subset of the questions proposed by Work Package 2. Within the natural biosphere, it is clear that biological organisms possess the ability to produce offspring of equal or higher complexity than themselves through the process of sexual or asexual reproduction. When observing the evolutionary growth of complexity of biological organisms from primitive replicating molecules which presumably existed following the origin of life on Earth, to the highly complex organisms that exist today, it is clear that biological organisms not only possess the ability to produce offspring of greater complexity than themselves, but this procedure must have occurred repeatedly and consistently throughout the duration of life on Earth.

If one was to assume that biological organisms could be categorised as extremely complex, organic machines, which abide by the same fundamental laws of physics as any machine which we can engineer today, then this poses the question, *how can we artificially engineer machines which also possess the ability to produce offspring of equal or greater complexity than themselves?*

There appears to be an inconsistency between biologically engineered, and humanly engineered machines, as humanly engineered machines typically produce machines of less complexity then themselves, while occasionally biologically engineered machines can produce offspring of greater complexity than themselves through mutation and evolution.

In order to investigate this problem, von Neumann envisaged a machine architecture which not only satisfied the conditions to maintain complexity under reproduction, but also allowed for increasing complexity (Von Neumann & Burks, 1966, pp. 82-87)[. This machine architecture consisted of two main components, a passive and an active component. The passive component is responsible for the storage of genetic information by containing an encoded description of the active component, analogous to a genotype, or a DNA sequence in biological organisms. The active component is responsible for all functional activity of the machine, analogous to a phenotype in biological organisms[5].

The phenotype of von Neumann's machine was further divided into four interrelated sub-components which cooperate to facilitate self reproduction: a general constructor, a copier, a control unit, and an additional, relatively arbitrary "ancillary machinery". Firstly, the control unit activates the general constructor which can inspect and decode an arbitrary genotype in order to construct an offspring phenotype. Upon construction of the offspring phenotype, the control unit deactivates the general constructor, and activates the copier. The copier inspects and duplicates the genotype (analogous to template-directed DNA polymerase in biological organisms) and attaches the duplicate genotype to the offspring phenotype. The ancillary machinery represents all other arbitrary functional activities of the machine. If the genotype which is copied and decoded contains a self description of the machine itself, then this process will result in self reproduction and the offspring machine will be an exact replica of the parent, assuming no errors occurred in the offspring's construction. However, errors or perturbations may indeed occur during the process of copying the genotype from parent to offspring.

[5] Although von Neumann never actually referred to the passive and active components as a genotype and phenotype, they serve the same purpose as the genotype and phenotype in biological terminology. Hence, we will refer to them as such from here forth.

This will result in an offspring whose genotype does not describe its phenotype exactly, but describes a new arbitrary phenotype which may be of less, equal, or greater complexity than the parent.

Furthermore, as the genotype includes the description of the entire phenotype, the mechanism in which the general constructor decodes a genotype is also encoded within the genotype. Should an error occur during reproduction which alters the description of the mechanism in which the general constructor decodes a genotype, this would potentially result in a change in the genotype-phenotype mapping of any offspring constructed from this altered description, analogous to a change in the genetic code of biological organisms.

As the emergence and evolution of symbolism appears to play a crucial role in the evolutionary growth of complexity, von Neumann's architecture seems to be a possible mechanism in which this phenomenon can be modelled and studied. The aim of this thesis is to implement von Neumann's architecture for machine self reproduction within a specific, currently existing, artificial life platform (Tierra); observe how such a mutable coding system might evolve; and investigate what effects this may have on the subsequent evolutionary dynamics of the system.

1.3.5 Thesis Contribution

The emergence and evolution of biological symbol systems appears to coincide with increasing levels of complexity in many biological systems. This thesis focuses on investigating evolvable symbol systems in which the utilised coding system is subject to inheritable variation. Such a system is developed and implemented where a class of artificial reproducing automaton are programmed with an initial "language" where distinct symbols are interpreted as specific distinct instructions or information. The specific architecture of this class of automaton allows for heritable variation so that the interpretation of these symbols are subject to change through evolution. This results in a mutable language or coding system, analogous to an evolvable symbol system in biology. The implementation of such a system within an Artificial Life platform leads to specific findings and publications which are documented in this thesis.

Firstly, when implementing automata which differentiated between the active catalytic component and the passive information storage, a phenomenon was observed where so called *pathological constructors* emerged.

These pathological constructors are automata which exploit specific anomalies in the Tierra artificial life platform to construct multiple short, malfunctioning automata which quickly exploit all available resources and drive all other automata in the neighbourhood to extinction. These findings were presented in the following publications:

1. Baugh, Declan & McMullin, Barry (2012), *The Emergence of Pathological Constructors when Implementing the Von Neumann Architecture within Tierra.* In: Proceedings of the European Conference on Complex Systems 2012 (ECCS).

2. Baugh, Declan & McMullin, Barry. (2012), *The Emergence of Pathological Constructors when Implementing the Von Neumann Architecture for Self Reproduction in Tierra.* In: From Animals to Animats, 12th International Conference on the Simulation on Adaptive Behaviour (SAB).

3. Baugh, Declan & McMullin, Barry (2012), *The Emergence of Pathological Constructors when Implementing the Von Neumann Architecture within Tierra.* In: Proceedings of the Frontiers of Natural Computing Workshop, University of York (NCFrontiers).

Further investigation was carried out where space for redundancy was introduced to the symbol system mapping. This class of automaton had an alphabet of symbols which comprised the genotype, which were mapped onto an alphabet of symbols which comprised the phenotype. However, approximately 75% of the phenotypic alphabet was not required for reproduction and so these symbols were "unused". The mutable mapping system created a "mutational ratcheting effect", where non-reversible variation occurred, so when an automaton "lost" a symbol mapping which was imperative to its survival, it did not give rise to functional offspring and the variation was not carried out to future generations. If a symbol mapping which was not imperative to survival was lost, the lineage would progress. However, as this variation proved to be non-reversible, there was a bias towards the entire population losing every symbol mapping which mapped a symbol to an unused instruction. This bias was not due to Darwinian selection, but due to the underlying physical dynamics of the mapping system where specific mutations were not directly (genotypically) reversible. Eventually, only instructions necessary for reproduction were implemented in the symbol system. Furthermore, in order to reintroduce a lost mapping to the symbol system, an instance of Lamarkian inheritance must occur. Specifically, a phenotypic perturbation must occur which affects the automaton's ability to decode the genotype, resulting in a phenotypic change inherited by its offspring without any change to the genotype.

This work was been published and orally presented with following publications:

5. Baugh, Declan & McMullin, Barry (2013), *Evolution of G-P Mapping in a Von Neumann Self reproducer Within Tierra.* In: Advances in Artificial Life, European Conference of Artificial Life (ECAL).

6. Baugh, Declan & McMullin, Barry (2013), *Evolution of Genotype-Phenotype Mapping of Von Neumann style Self reproduction within the Platform of Tierra.* In: European Conference on Complex Systems (ECCS).

Further work included a new class of automaton that implemented a different mechanism to achieve a symbol mapping system. This mapping system was more complex and required vastly more computer resources in order to achieve self reproduction, therefore, it was infeasible to achieve a large data set of results through the conventional means with the available resources. For this set of experiments, the architecture was analysed and all possible variations were manually inspected and documented in order to determine the range of possible phenomena which may occur using this alternative mapping system. Although a different mapping system was implemented, the same mutational ratcheting effect caused by non-reversible variation was also apparent with these results, which were a direct result of implementing a mutable mapping system.

1.4 Thesis Overview

The aim of this thesis was to investigate the evolution of a genotype-phenotype mapping of von Neumann style self reproducers within the artificial life platform of Tierra.

Previous work has been carried out in the investigation of agents which implement a mutable genotype-phenotype mapping, however, much of this work appears to focus on symbol systems which are not designed to investigate reproduction, but for the investigation of social communication in robotics such as swarm theory (Lukas & Schmeck, 2009). Other research which deals with symbol systems responsible for reproduction, take a "top down" approach, which examine the emergent properties of genotype-phenotype mappings, and attempt to understand the underlying mechanics responsible for these emergent properties (Altenberg, 1995).

The aim of artificial life is to take a "bottom up" approach in which the fundamental logical formalism of a biological system is simulated in a computational model in order to observe and study the emergent properties which arise. This was accomplished via the implementation of the von Neumann architecture for machine self reproduction within the artificial life platform of Tierra.

Chapter 2 summarises a brief history of the developments and research in the field of Artificial Life, which brings us up to date with the current advancements in the field

which eventually lead to the evolutionary platform of Tierra with which the evolutionary experiments were performed for this thesis. Chapter 3 describes the von Neumann architecture for machine self reproduction and the logical formalism to how a self reproducer with a mutable genotype-phenotype mapping may be designed. The artificial life platform of Tierra, and how a von Neumann style ancestor may be implemented within Tierra is described in detail in Chapter 4 and Chapter 5 respectively. The experimental procedures and results are documented in Chapter 6. Within this chapter, two separate implementations of a genotype-phenotype mapping are experimented with, and both sets of experiments are analysed and compared. Finally Chapter 7 provides a summary of entire the thesis and highlights the major findings, and potential the future work which builds on the findings of this thesis.

Chapter 2

Developments in Artificial Life

2.1 Chapter Overview

The previous chapter argued that the emergence and evolution of symbol systems appears to play a profound role in the evolution of biological systems. This chapter briefly discusses the advances and developments in Artificial Life, from Darwin's theory of evolution by natural selection, to operating systems capable of evolving digital agents. These developments have supplied us with platforms, allowing us to test theories in silico, and provide proof-of-principle examples of evolving symbol systems in order to further study this phenomenon.

2.2 An Introduction to the Theory of Evolutionary Dynamics

When Charles Darwin published On The Origin Of Species (1859), he provided a beautifully simple, yet elegant mechanism to explain the origins and maintenance of the large diversity of species which exist on Earth. Gregor Mendel's extensive work with the selective breeding of pea plants between 1856 and 1863 demonstrated that the inheritance of certain traits follow particular patterns, which formed the foundation of the modern science of genetics. However, there existed a gap in our knowledge of evolutionary theory until Julian Huxley et al, published "Evolution: The Modern Synthesis" (1942). The modern evolutionary synthesis bridged the gap between Darwinian selection and Mendelian genetics, and reflects the current consensus among the scientific community. The modern synthesis states that all evolutionary phenomena can be explained in a way which is consistent with known genetic mechanisms. Natural selection is the fundamental mechanism which drives Darwinian evolution, where incremental improvements in structure or behaviour will lead to the prevalence of different genotypes. Darwinian evolution is typically a gradual process, where small incremental genetic changes regulated by a process of selection accumulates over time. In general, the progression of natural selection in the animal and plant kingdom is negligible on the time-scale of a human life and therefore is very difficult to examine as it may take thousands or even millions of years for a single species to diversify into a categorically different species.

Evolutionary change can occur more rapidly if selective breeding takes place, as is the case with artificial selection, to the point where phenotypic divergence can be seen on the time-scale of a human life. Since 1959 The Soviet Union/Russia have been performing long term selective breeding experiments which have domesticated lineages of wild silver foxes (Trut, 1999) by selecting foxes which display more docile personality traits for breeding. These descendents not only behaved differently, but their physical appearance had shown drastic changes. It was also shown in small islands in Florida that one species of lizard, *Anolis carolinensis*, evolved larger toe pads in order to help it climb to higher perches, following the introduction of another closely related

species, *Anolis safrei*, to the environment, which introduced competition between the two species (Stuart et al., 2014). However, although these experiments show evolutionary change on observable time scales, these time scales may span across decades, which require huge amounts of funding and dedication before results can actually be observed.

2.3 Biological Synthesis of Evolutionary Dynamics

Due to evolution's gradual pace within the Earth's biosphere, experiments in evolution have been generally limited. One method of studying the effects of evolution involves using microscopic organisms with generation times on the order of hours, as shown by Richard Lenski's *E. coli* ongoing long-term evolutionary experiment (Lenski, 2011), where twelve initially identical populations of asexual *Escherichia coli* bacteria were studied over 50,000 generations. However even this approach has difficulties, as it is difficult to perform measurements on such a small scale with high precision, and the time-scale to see significant adaptation may still span weeks (Adami et al., 2000).

However, the advent of computer hardware and software over recent decades has allowed us to design evolutionary systems in silico, which simulate at least some of the properties of living biological systems. The dynamics of Darwinian evolution over thousands or even millions of generations can be applied to these systems within the time frame of minutes, hours or days, finally allowing us to examine the evolutionary dynamics of such systems under fully controlled parameters and under practical time frames.

2.4 Computational Synthesis of Evolutionary Dynamics

2.4.1 Von Neumann's Kinematic Model

The first computational approach to the generation of life-like behaviour, was proposed as early as 1948 by mathematician John von Neumann (1948) in his attempt to understand the evolutionary growth of complexity. Although von Neumann's untimely death prevented him from completing his work on this topic, his colleague Arthur Burks proceeded to compile and complete von Neumann's work, *On The Theory of Self Reproducing Automata* (Von Neumann & Burks, 1966). Burks stated that von Neumann posed the following question, "What kind of logical organisation is sufficient for an automaton to reproduce itself?". In an attempt to answer this question, von Neumann first developed his *Kinematic Model* as an initial thought experiment. The Kinematic Model is represented by a machine which floats on the surface of a two dimensional pond along with instances of the fundamental physical subcomponents with which machines may be composed (Von Neumann & Burks, 1966, pp. 82-87).

The floating machine consists of several different, yet essential, interdependent as-

semblies, one of which is a *programmable constructor*. If the floating machine is supplied with the blueprints, or description, of an arbitrary target machine, the machine's programmable constructor has the computing capacity to analyse and decode the description, and the motor skills to select any parts required from its environment and build the described machine. The assembly which contains the description is referred to as *the tape* by von Neumann as it was inspired by the tape of Alan Turing's computational machine (Turing, 1936). The initial machine also contains a *tape copier*. Upon constructing an offspring machine, the tape copier will copy the tape, and attach a duplicate copy to the offspring, and the offspring machine is now "switched on". Should the contents of a tape contain an exact description of the initial machine, then this machine will proceed to create copies of itself, legitimately exhibiting machine self reproduction.

2.4.2 Cellular Automata

Von Neumann wished to devise a formal, logical system which modelled self reproduction, however, this required a large computational workload, and general purpose computers were not available at the time, circa 1948. A colleague of von Neumann while working at Los Alamos National Laboratory, Stanislaw Ulam, suggested using a mathematical abstraction, such as a lattice network in which he was using to study crystal growth. From this foundation, von Neumann worked on developing the logical formalism of what is now called cellular automata.

Von Neumann attempted to extract the logical form of his Kinematic Model, and implement it within a cellular automaton, however, there are two fundamental limitations on behaviours which we can expect when implementing logical behaviours within any computing machine (Langton, 1995).

The first limitation is that of computability. There are certain functions which are possible to describe perfectly, but which are impossible to compute. Alan Turing previously proved this limitation in 1936 with the classic example of the Halting Problem (Turing, 1936). The halting problem can be stated as follows: "Given a description of an arbitrary computer program, decide whether the program finishes running or continues to run forever". This is equivalent to the problem of deciding, given a program and an input, whether the program will eventually halt (or in fact execute any prescribed behaviour) when run with that input, or will run forever. There is no general mechanism which can be applied to any arbitrary machine, within the complete space of all possible machines, to determine whether or not this machine will halt. The machine must be run in the context of its environment to observe whether the machine halts, however, if the machine does not halt, one can only deduce that the machine did not halt up to the point that they stopped observing it.

To minimise the computational limitation of his self reproducer, von Neumann incorporated a Turing universal computation machine within the programmable con-

structor to ensure that his machine was computationally universal, and all possible logical computations were possible.

The second limitation of logical behaviours of any computing machine is one of computability in practice. In order to provide a machine with a formal specification of how to exhibit a certain behaviour, we must first establish a sequence of logical steps which the machine must obey in order to exhibit this behaviour. This limitation can be loosely thought of as the complement of the Halting Problem. There can be no general, formal mechanism which can be applied to the complete set of possible machine arrangements to deduce whether *any* arbitrary machine will exhibit a specific behaviour such as halting. Similarly, within the set of all possible arbitrary machine behaviours, there cannot be a general, formal mechanism which can be used to deduce the description of the underlying machine when supplied with *any* arbitrary behaviour. There may be certain behaviours where it is impossible to deduce the mechanical arrangement of the underlying machine.

This is a consequence of the levels of hierarchy that exists with complex systems. It may not be possible to determine the emergent properties of a complex system by simply examining its description, and similarly, it may not be possible to determine the description of a complex system system by examining its emergent properties. Thus, we have no conventional methods for deriving a machine which will display a specifically required behaviour, short of educated guessing, trial and error. In order to implement the phenotypic behaviour of self reproduction within a cellular automata, von Neumann had to design his system ab initio. Unfortunately, von Neumann's final model was not completed upon his death, however Burks (Von Neumann & Burks, 1966), completed the work and published a working design of von Neumann's machine architecture in the cellular automata formalism (See Figure 2.1).

The machine was implemented within a 29 state cellular automaton with a two dimensional, five cell neighbourhood, and accurately exhibited self reproduction. This was constructive proof that fundamental characteristics of life could be achieved by machines. In hindsight, we see that von Neumann also incorporated many of the other fundamental criteria of life which were described earlier, such as; life being a pattern in space-time, information storage of a self-representation, functional interactions with the environment and interdependence of parts. In fact, it wasn't until five years later, in 1953 when Watson and Crick co-discovered the structure of DNA, that it was realised that von Neumann's architecture showed amazing prescience, precisely describing how heritable information is passed from parent to offspring, consisting of both a transcription/translation process (performed by the general constructor) and a replication process (performed by the description copier). However, it should be noted that it was possible that von Neumann was aware of the idea proposed earlier by Schrödinger where genetic information must be contained within an "aperiodic crystal" in its configuration of interchangeable covalent bonds (Schrödinger, 1944).

Edgar Codd and James Thatcher attempted to improve von Neumann's automa-

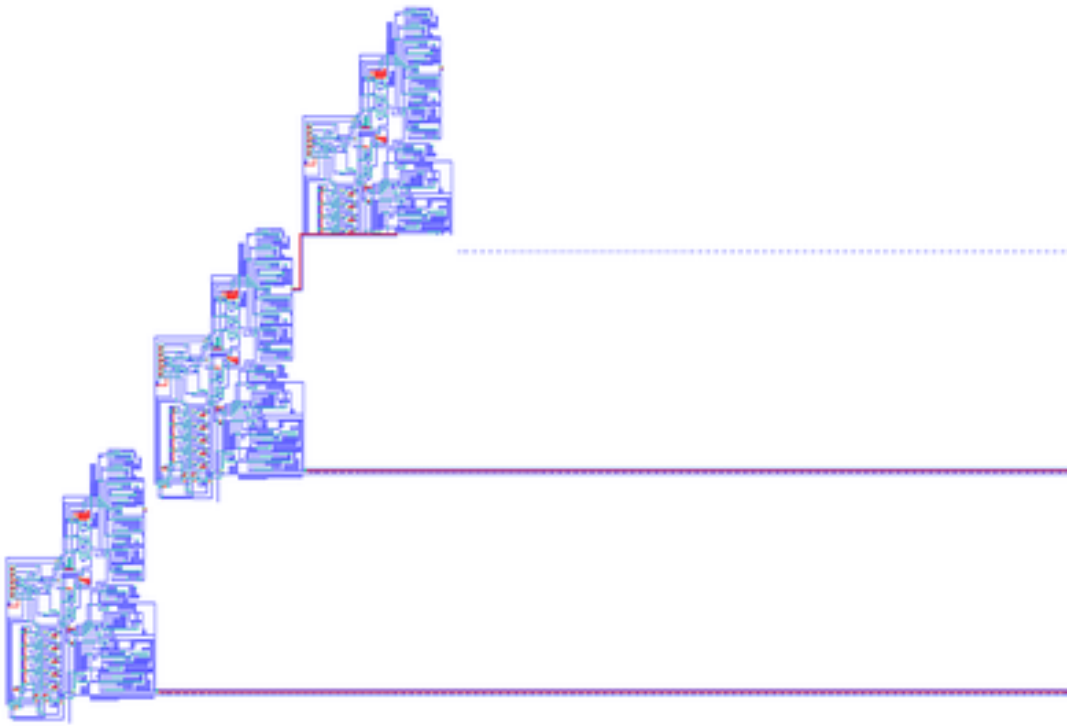

Figure 2.1: Implementation of von Neumann's Tessellation Automaton in cellular automata. Three generations of the Tessellation Automaton are shown, the second automaton has nearly finished construction of the third. The tapes can be seen protruding to the right.(Public domain image reproduced from the sq3 website; based on the (1995) implementation of von Neumann's self-reproducing machine and generated using the Golly software, located at http://golly.sourceforge.net.)

ton by adding simplifications such as reducing the number of cell states within the cellular automata and simplifying parts of the general constructor of the automaton itself, (Codd, 1968; Thatcher, 1964), however, it is arguable that none of the applied adjustments actually added greatly to the functionality of the model but just acted as incremental improvements (Pesavento, 1995). Again, with hindsight, this is not surprising, as in order to make the model significantly more "life-like" it would be expected that one would include a greater number of the eight essential criteria which bear on the nature of life. However, due to the extremely large computational workload required to simulate models, it would not be until the advent of affordable, programmable, general purpose computers that further major developments would be witnessed in this field.

2.4.3 Genetic Algorithms & Genetic Programming

Genetic algorithms (GAs) are computational models of evolution that are employed as a method of solving practical problems in many areas of science, but play a central role in models of evolution in artificial life. GAs as they are known today, were initially described by John Holland in the 1960s and were further developed and published with the release of his book "Adaptation in Natural and Artificial Systems", which presents the GA as an abstraction of biological evolution and displays the theoretical framework for adaptation to occur (Holland, 1975). Holland wished to "extract the logical form" of the natural process of evolution, using genetic algorithms. Holland's GA is a method for transforming a population of "chromosomes" (bit strings which represent functional or phenotypic characteristics of an abstract organism) to a new population, by using selection, paired with the application of genetic operators (crossover, mutation, deletions) to the chromosomes. A selection mechanism will choose which chromosomes in the population will be reproduced and decides how many offspring each chromosome will bring forth. Selection chooses which chromosomes will reproduce by measuring its "fitness". This fitness is a measure of how well the chromosome performs a specific task. Chromosomes with a higher fitness value will give rise to a greater number of offspring (fitness-proportionate selection). Genetic operators cause random perturbations to the bit string representations of the offspring chromosomes. The inclusion of genetic operators introduce inheritable variation to members of the population, and inspection of the fitness function selects members of the population with increased fitness to be reproduced. Members of the population with a lesser fitness are not reproduced in the next generation. The general form of a GA is as follows (Mitchell & Forrest, 1994):

1. Start with a randomly generated population of chromosomes (i.e., representations of candidate solutions to a problem)

2. Calculate the fitness of each chromosome in the population.

3. Apply selection and genetic operators to the population and generate a new population.

4. Revert to step 2 and begin the next generation.

This process is iterated over many generations until a population of highly fit chromosomes are produced which can perform the specified task at higher fitness than the initial chromosomes which inoculated the population of the first generation.

Although GAs are an optimisation tool which provide a proven method of evolving chromosomes to achieve higher performance levels, GAs do not generally display open ended evolution, as eventually the fitness of a population of chromosomes will plateau and no further increase in fitness will be observed. GAs are generally not used as a tool for displaying open ended evolution, but more as a search tool for searching a space of possible chromosomes for a representation which will represent the fittest possible solution.

We can see that GAs are missing many of the fundamental criteria which appear in living organisms, such as a metabolism and self reproduction (bit strings within memory are not manipulated or reproduced by the chromosomes themselves, but by the externally applied selection tool). As a result of this, it would not seem likely that evolution of GAs could provide the same level of increased complexity as observed in organic biology.

Genetic Programming (GP), first introduced by Nils Barricelli (1962), and Lawrence Fogel (1966), can be considered a particular case of GAs, where each individual is a computer program. The output of GAs are generally strings of data which represent a solution to a specific problem, however the GP is a machine learning technique where the output itself is a population of computer programs, optimised according to a fitness landscape determined by each program's ability to perform a given computational task. In particular, the question of "would it be possible to select symbioorganisms able to perform a specific task assigned to them, for example, the task may consist of deciding moves in a game being played against a human or against another symbioorganism" (Barricelli, 1957, p 120). However the term "Genetic Programming" was not coined until much later by John Koza (1990), where he describes how this paradigm provides a method of creating computer programs which were genetically bred using the Darwinian principle of survival of the fittest to evolve to perform complex tasks which include but are not limited to solving equations, pattern recognition, neural network design and game playing strategies, without the need of a human programmer guiding the entire design process.

The evolving agents in GPs are typically written in Lisp-like languages[1] which can be represented in memory as *tree structures*. An example of a simple mathematical function represented as tree structure is illustrated in Figure 2.2. The use of a tree structure based programming language allows perturbations to be made to the code, where mathematical operations and numbers can be changed within the code, but still

[1]Lisp is a family of computer programming languages which employs fully parenthesized Polish prefix notation.

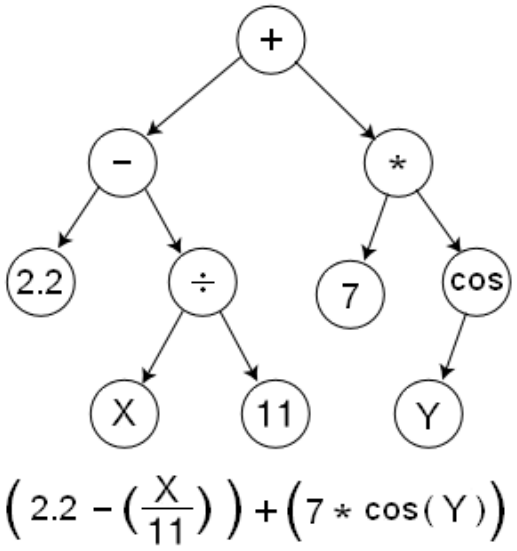

Figure 2.2: Example of a simple mathematical function represented as a tree structure, adapted from Wikipedia (2014).

render the resulting code syntactically correct. Perturbations such as crossovers can be applied where a node on the tree can be switched with a node from itself or one from another program in the population. Point perturbations can also occur where the information within an individual node can be perturbed to a random number or mathematical operation.

2.4.4 Computer Viruses and Core Worlds

The early 1970's saw the advent of early computer viruses, such as *The Creeper Virus* which was an experimental self replicating program developed by Bob Thomas at BBN Technologies in 1971 (Chen & Robert, 2004). Creeper used *ARPANET*[2] to infect computers operating on the TENEX operating system where it created a copy of itself and and displayed the message "I'm the creeper, catch me if you can!". However, after replication, Creeper would delete the previous copy of itself to prevent it from progressively consuming the hard disk space on the affected computer. Subsequently, the *Reaper* program was developed which was a computer program which used the computer network to spread itself, locate, and delete the Creeper virus.

The advent of computer viruses sparked the realisation that self replicating computer programs in the memory of computers could serve as a research platform for Artificial Life. An early attempt to simulate artificial life within a computer was described by A. K. Dewdney (1984) with the release of a programming game called Core Wars. Within this game, two or more programs called "Warriors" reside within a 1 dimensional circular memory (core world), and compete with each other for the control

[2]The Advanced Research Projects Agency Network (ARPANET) was one of the world's first operational packet switching network and the progenitor to the internet.

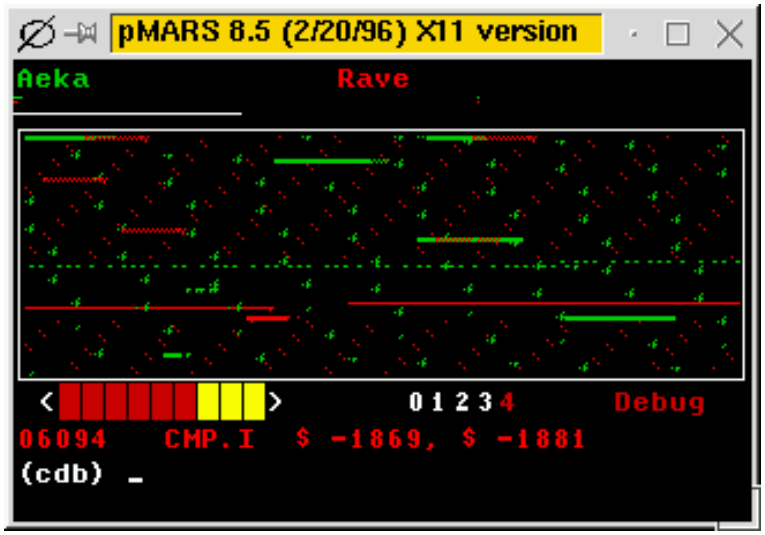

Figure 2.3: A Core Wars battle, simulated on the pMars simulator. The source code for pMars can be downloaded from the official site: http://www.koth.org/pmars/.

of the virtual computer (Memory Array Redcode Simulator "MARS"), Figure 2.3.

Programs in Core Wars are written in a specifically designed assembler language called Redcode. Redcode is loosely based on a complex instruction set computer (CISC). Each Redcode instruction occupies a single location within the circular memory. There are no registers and all data are stored and manipulated within memory. Redcode has 10 different instructions where each instruction can take two numbers as arguments and each argument is preceded by an addressing mode. There are three addressing modes, which specify whether the numbers which represent the arguments are interpreted as numerical data, as a pointer to a location in memory, or as a pointer to another pointer.

This game introduced the fundamental criterion of *a metabolism*, which converts matter (memory) and energy (control over the virtual computer) into the pattern and activities of the programs (warriors). Within this framework, warriors can propagate throughout memory, and compete for memory space and control over the virtual computer.

Building upon the framework of Core Wars, Steen Rasmussen designed a different program called "Coreworld", (Rasmussen, 1990). Random noise was added to the system incorporated within the `MOV` command, which copies instructions from one location in memory to another. This command was flawed with a certain probability so that it would copy an instruction to a random destination. The specific goal of this system was to construct an artificial computational chemistry that could support artificial replicating programmes in the hope that these programmes would display emergent computational behaviour.

Although Core Wars, and subsequently Coreworld introduced the concept of a metabolism to digital programs, its instability and sensitivity to small changes was

a major defect which prevented the demonstration of further life-like phenomenon. In a non-quiescent environment, it is difficult for warriors or programs to preserve their form and may eventually malfunction as their code may be over written by other programs.

2.4.5 Tierra

In 1991 ecologist Thomas S. Ray developed a new system which could be used to experimentally explore, in silico, the basic processes of evolutionary and ecological dynamics (Ray, 1991). This system, Tierra, is similar in many ways to the Core War system such as having a circular memory and machine language programs, however it has a number of major advancements. Upon birth of offspring programs (creatures), the memory block in which creatures are located are issued "write protection". This ensures that creatures do not overwrite each other's code, and hence implement the fundamental criterion of *stability under perturbations and insensitivity to small changes*, allowing the creature to preserve its form and continue to function in a noisy environment where neighbouring creatures cannot overwrite each others code. This is one of the fundamental criteria which characterise life as we know it, that Farmer and Belin proposed, which was not implemented within Core Wars.

The concept of random variation had been introduced in Coreworld. While data was being copied from one location memory to another, the destination memory address was subject to errors, allowing data to be copied to random locations rather than the intended destination. This concept of introducing random variation was expanded on in Tierra. Several *genetic operators* were introduced which allowed biologically inspired changes such as point perturbations, crossovers, insertions and deletions. This introduced noise allowed for a wider range of possible variation to emerge between creatures and their offspring.

In order for life in our universe to exist, we rely on the delicate balance of finely tuned physical laws. Within astrophysics and cosmology, the anthropic principle states that our observations of the physical universe must be compatible with the conscious life that observes it. However, there is no such anthropic principle within the digital medium, so there is no pre-designed set of rules which we must follow in order to create artificial life. As we have no formalisation for developing a set of fundamental laws within the digital universe that may eventually support digital life we must act as "God", and by deliberate design, attempt to create a set of laws which is capable of sustaining digital life.

Ray acknowledged this problem, and spoke about "conventional" and "unconventional" factors that affected evolution (Ray, 1999). Conventional factors are aspects which have been thoroughly analysed in the field of biology and are variable, e.g., population size, mutation rates etc. Unconventional factors are aspects which are not variable in biology, but can be varied in the design of artificial systems, such as the

physical laws of the universe.

The field of genetic algorithms had already provided a good understanding of conventional factors, however, the study in the understanding of the unconventional factors was relatively limited. Ray decided to use biology as a basis, where he would try to implement unconventional factors to resemble the laws which take place in real life. Ray noted that proteins are composed of a set of 20 amino acids, therefore he implemented a 5 bit assembler language instruction set, which allows a total of 32 distinct assembler language instructions with no arguments or addressing modes, which only include the minimum number of instructions necessary to facilitate self reproduction. Although Redcode includes a smaller instruction set of 10, these instructions can take 2 numbers as arguments which range from 1 to the total number of address locations in memory. These arguments were also preceded by one of 3 addressing modes. This significantly reduced space of possible instructions in Tierra compared to Redcode was intended to increase its mutational robustness and its *stability under perturbations*. Random perturbations applied to the reduced instruction set, may perturb it to only one of the other 31 instructions available. This allows for random perturbations to be applied to instructions, and have a higher probability that the result will itself be functional, in some degree at least.

With genetic algorithms, reproduction is incorporated with an externally applied fitness function. However, Tierran creatures perform the act of self reproduction autonomously, within the memory, and so their fitness is a direct measure of the ability of the creatures to actually reproduce. This removes the role of the external programmer in determining the fitness function directly and allows a more substantive model of natural selection.

Tierran creatures use a so-called template addressing mode rather than relative addressing which was used in Redcode. Templates are complementary sequences of two specifically reserved instructions (`nop0` and `nop1`). This idea was also borrowed from molecular biology, as biological molecular structures commonly interact by matching complementary shapes. The intention of introducing this feature was to improve the *functional interactions with the environment*, as it forces the Tierran creatures to behave more like their analogous carbon based representation where the locations in memory can be recognised without knowing their absolute address location.

However, evolution in Tierra generally still reaches a plateau where, after a period of rich interactions between creatures within the ecosystem we meet a stage of stasis (Ray, 1991). Simply implementing Farmer & Belin's essential criteria for life is not sufficient to observe, effective, on going, open ended evolution.

One may also focus on fine tuning the fundamental laws of the universe (unconventional factors), which in turn improve the cooperation between the different levels of the hierarchy of living properties, so as to improve evolvability. We understand that life must exist as an ordered pattern in space-time, and it must bear functional interactions with its environment. However, it is not clear as to what form these criteria must

be met in the digital universe, i.e., how the matter must be ordered, or what form of interactions must take place between it and its environment. In order to answer these questions, the only approach which we currently have is to study biological processes, and attempt to incorporate these processes within our artificial life platforms.

2.4.6 Avida

As a successor to Tierra, a different Artificial Life simulator, Avida, was developed by Chris Adami, Charles Ofria and C. Titus Brown at Michigan State University (Adami & C. Titus, 1994). As mentioned before, unconventional factors which affect evolvability are aspects that are not variable in biology, but can be varied in the design of the artificial system. When examining the proposed hierarchy which bears on the nature of life as we know it, we see that *life is a pattern in space-time* and *functional interactions with the environment* are at the lowest level of hierarchy. These two factors are essentially what Ray considers "unconventional factors". Ray focused on improving the functional interactions with the environment, by proposing new means by which the Tierran creatures will interact with each other which bear a closer resemblance to that of biological organisms. Adami however, seemed to focus on improving the patterns in space-time by which the Avidan creatures exist.

A potential weakness in Tierra is the one dimensional linear memory space. Although this doesn't prevent evolution from occurring, it is quite different from the complex three dimensional interactions which are observed in life on Earth. Two dimensional cellular automata already existed, however, they are less suited to study evolution as they are typically quite brittle. Any small change in an organism will generally result in a very low probability of it preserving its ability to self reproduce. There has been examples of more "robust" evolution in two dimensional cellular automata, as is the case with Evoloop (Sayama, 1999), which is essentially an improved version of Langton's Loop (Langton, 1984). Due to the high adaptability of the self-reproductive mechanism employed within Evoloop, when the construction arms of two self-reproducing automata collide, two parent loops simply abort the construction of their offspring and continue their self-reproducing activity by constructing an offspring elsewhere. However, the wall-clock time required to carry out a complex procedure such as self reproduction is still considerably slower in typical CA compared to that in a core world type system.

Adami introduced a two-dimensional toroidal memory system. Each Avidan organism resides within its own protected region of memory (cell) and cannot read or write information outside its own local memory space. Because of this, an Avidan may only affect other organisms in its direct neighbourhood rather than the entire memory system as is the case with Tierra.

The second major difference is that the virtual CPUs of different organisms can run at different rates, based upon each organism's ability to perform particular, externally

specified tasks, or logical computations. This new feature can be thought of as an amalgamation of properties of both Tierra and genetic programming. An organism is assessed by its ability to perform a specific task, much akin to how a fitness function is a measure of how well a chromosome performs a specific task. In the case of GAs, fitness-proportionate selection will select the chromosomes of higher fitness to be reproduced at a higher rate, and similarly, Avidans which can perform specific tasks, are rewarded with extra CPU time as a bonus, which in turn, can accelerate their rate of reproduction, relative to neighbouring Avidans.

This new feature was another unconventional factor, which affected the digital organism's functional interactions with the environment. This amendment was an attempt to allow digital organisms to increase in complexity under natural selection, by posing increasingly complex external computational tasks.

Evolution in an information poor landscape where there are no factors which drive natural selection rather than speed of reproduction usually leads to shrinking of the genome size[3]. This was first demonstrated *in vitro* in the 1960s when Sol Spiegelman demonstrated the Darwinian evolution of Qβ genomic RNA (Joyce, 2007). Spiegelman was aware that Qβ replicase would produce the occasional mutation during amplification[4]. He allowed the amplification of RNA molecules within a controlled solution by introducing Qβ replicase. After a specific period of time, a sample of the solution would be extracted, and used to inoculate a new generation. After a number of generations, he noticed that the RNA molecules would require less time for replication, and therefore would have a selective advantage. This was due to the molecules decreasing in size. In an informationally poor landscape, speed of replication was the only selective force in effect, so under evolution, all genomic information was lost, with the exception of "instinctual knowledge" which is fundamentally required for the molecules to replicate, and nothing else.

This result can also be observed in the previously developed digital medium of Tierra. The default ancestral creature in Tierra only contains instinctual knowledge, which is information embedded within the creature, without which it couldn't function correctly and reproduce. Under evolution, these genes become more efficient at performing the tasks that they can already perform, so the creatures shrink in size as efficiency rises. However, no new knowledge is ever added, so it can be argued that the complexity within Tierran creatures never actually rises.

In order to make a case for or against the evolution of complexity, complexity must be defined and measurable. There are many approaches one may take, for example, Kolmogorov complexity is a measure of the computational resources necessary to define

[3] In the experiments demonstrated in this thesis, this phenomenon is also seen to occur, however, the aim of this thesis is not to search for "increased complexity" which usually coincides with an increase in information/creature length, but is specifically looking at the possibilities of a heritable change in the genotype-phenotype mapping, which does not inherently require an increase in "complexity".

[4] Amplification within molecular biology refers to the process of creating multiple copies of a seed molecule via a polymerase chain reaction.

an object. However, the Kolmogorov complexity of a random bit string would actually be higher than that of an ordered bit string of the same length, because it would take more computer resources to exactly describe a random bit string than it would a more complex but ordered bit string of the same length. Shannon's information theory states that "the quantity entropy (H) represents the expected number of bits required to specify the state of a physical object given a distribution of probabilities" (Shannon, 1948; Adami et al., 2000). It seems that complexity of an organism may be defined as some measure of how much information can be stored within the organism. Physicist David Deutsch stated that "genes embody knowledge about their niches" (Deutsch, 1997), in other words, during evolution, genes store information about their environment, which in turn, increases the level of information stored within the gene, hence increasing complexity. Because of this, Adami stated that an organism's genes are not simply a "book" about the organism, but also a book about the environment in which the organism lives (Adami et al., 2000). With this in mind, if there is no information within an organism's environment which the organism can interact with, then there is no possibility of the organism increasing in complexity. In order for an organism to increase in complexity, it may have to reside within an informationally rich landscape. Adami developed a system where knowledge could be learned from interacting with an external environment, and this knowledge can become incorporated within the organism's genes.

While this development was successful at demonstrating how information of a creature's surroundings can become incorporated into its genes, this system still eventually suffers from the same fate as Tierra, as it does not display open ended evolution. Once the creatures have learned the basic knowledge that is available in its environment, evolution hits a period of stasis and plateaus.

2.4.7 Conclusion

There are many speculations as to why the mentioned artificial life systems inevitably reach a plateau in evolution. The division of labour between passive information and an active catalyst is a fundamental property of all living systems (Takeuchi et al., 2011). If we wish to observe open ended evolution within an artificial life platform, it seems reasonable that we should implement all the essential criteria which are common to real life, which are already proven to be capable of hosting open ended evolution without plateauing.

One essential criterion which is common to all organic life, *information storage of a self-representation*, is not implemented in Tierra or Avida, although it was present within von Neumann's Tessellation Automaton. In order to incorporate this criterion, von Neumann's design included a mechanism which could accurately decode data and interpret it as instructions to construct a physical automaton. As this decoding mechanism must be described within the self-representation, it too must also be subject to

inheritable variation just as the rest of the automaton. The next chapter describes the automaton proposed by von Neumann in detail, and describes how this architecture may be used to investigate how a mutable decoding system may evolve. If it is possible for automata to emerge which incorporate a mutated decoding mechanism, or *genotype-phenotype mapping*, than the plateau may be delayed and may also result in a wider range of automata which are possible through evolution.

Chapter 3

Von Neumann's Architecture

3.1 Chapter Overview

The previous chapter discussed several major developments in artificial life and discussed how they incorporated many of the fundamental criteria which bear on the nature of life as we know it. The most recent developments which were discussed, the Tierra and Avida platforms, and automata implemented within these systems failed to incorporate one fundamental criterion, *information storage of a self-representation.* In order to design an architecture for artificial life which incorporates an information storage of a self-representation, an automaton must be designed which not only includes an information storage, but has the ability to decode and interpret this information in order to build an offspring. A logical formalism for a self reproducing automaton capable of evolution, which performs this task, was articulated by John von Neumann. However, this formalism has not generally been investigated in the most recent artificial life platforms. This chapter summarises the problem which von Neumann set out to solve, and explains the details of the architecture which he devised as a solution.

3.2 Von Neumann's Problem: The Evolutionary Growth of Machine Complexity

As indicated in Chapter 2, John von Neumann investigated the problem involved with the evolutionary growth of complexity of arbitrary machines (Von Neumann & Burks, 1966; McMullin, 2000). Assuming that biological organisms are essentially extremely complex organic machines, then over time these machines must, in at least some cases, have constructed offspring of greater complexity than themselves in order to facilitate an evolutionary growth of complexity[1]. Von Neumann pondered how a machine might construct an offspring machine of greater complexity than itself. At face value, the parent machine must already contain a description of the offspring's design, hence being of equal or greater complexity than the offspring.

Julian Huxley had recently published his book, *Evolution: The Modern Synthesis* (Huxley, 1942), which attempted to bridge the gap between Darwinian evolution and Mendelian genetics, however, the exact architecture that machines must possess in order to accommodate the evolutionary growth of complexity was not rigorously defined. Furthermore, a satisfactory definition of biological complexity has still not been explicitly defined to this day. Von Neumann simply adopted an informal definition of complexity as "the ability to do very difficult and involved things" (Von Neumann & Burks, 1966, p. 78).

[1] Although it is acknowledged that evolution may also occur through symbiogenesis, where increasingly complex organisms can be created via the symbiosis of two simpler organisms, which explains the origin of eukaryotic cells from prokaryotes, here we are referring to the generic Darwinian process of evolution via inheritable variation among diverging organism lineages.

Von Neumann progressed to formulate a general and abstract machine architecture, where the set of potential machines included an infinite subset which were:

1. Arbitrarily complex.
2. Capable of self reproduction
3. Capable of undergoing spontaneous inheritable mutations.
4. The entire set of such machines are connected under mutation. (McMullin, 2000)

Populations of machines that exhibit these criteria may possess the ability to undergo an evolutionary growth of complexity, and with a single relatively simple seed machine there may exist many evolutionary lineages leading to machines of higher complexity.

This formalisation was the basis for his Schematic Kinematic Model. However, in order to prove that his model indeed demonstrated true machine evolution, he would have to extract its logical form, and exhibit a fully detailed design. To do so, he used cellular automata as a platform to design what will be referred to here as the von Neumann Architecture for Machine Self Reproduction.

3.3 Von Neumann's Solution: The Von Neumann Architecture For Machine Self Reproduction.

Von Neumann's architecture for machine self reproduction, presented in his theory of self reproducing automata (Von Neumann & Burks, 1966; Baugh & McMullin, 2012a), describes an abstract class of machine, M, which is decomposed into two primary components, a functional component P, and a passive component G, denoted by $M = (P+G)$ (McMullin, 2012). G represents a one-dimensional string of symbols which has no active/functional capability, but can be interpreted as information, similar to the *tape* of a Turing machine. The information within G is used to describe (or "encode") an arbitrary target machine X under some function, $\phi()$, such that $G = \phi(X)$.

P is further divided into four fundamental subcomponents, a *general constructive automaton* A, a *general copying automaton* B, a *control unit* C, and so called *ancillary machinery*, D. G will be referred to as the *genotype* while P will be referred to as the *phenotype*[2].

The general constructive automaton A can read the symbols within G, and interpret them as an encoded description of the arbitrary machine X. A therefore has the capability to apply an inverse "decoding" function, $\phi^{-1}()$, or $\psi()$, to G, and construct the described machine X. We denote this by saying $\psi(G) = \psi(\phi(X)) = \phi^{-1}(\phi(X)) = X$. In other words, when supplied with a genotype, the general constructive automaton

[2]Although von Neumann never actually used these specific terms to describe the components in question, they are analogous to the genotype and phenotype in modern biological terminology.

applies the decoding function $\psi()$, to G, in order to construct some arbitrary machine phenotype X.

The general copying automaton B, reads and duplicates the machine description G. A control unit C is required to govern the automaton $(A + B)$, directing its operation, activating A and B in order, and ensuring that the offspring creature is appropriately activated in a controlled fashion once its construction is complete.

The forth component, the ancillary machinery D, refers to all conceivable functionality that the machine may possess which does not interfere or hinder the reproductive operation of $(A + B + C)$.

When a machine $(A + B + C + D)$ is supplied with a description, $G = \phi(X)$, the control unit C first commands B to duplicate G. Upon duplication, C instructs A to decode G under its characteristic genotype-phenotype mapping function $\psi()$, and thus construct the described machine X. Finally C will attach the new instances of G and X, and sever them from the parent automaton $(A + B + C + D) + \phi(X)$, after which there exists the new entity, $X + G \triangleq X + \phi(X)$.

Now consider the case where $X \triangleq (A + B + C + D)$. This system, $(A + B + C + D) + \phi(A + B + C + D)$ will proceed to construct an offspring automaton and attach it to the description of itself, $(A + B + C + D) + \phi(A + B + C + D)$. The parent and offspring are identical, therefore achieving self reproduction. This machine architecture is shown schematically in Figure 3.1.

Next we consider the case where a random phenotypic perturbation occurs during the construction of P. If a perturbation affects the reproductive subcomponents A, B or C, than the automaton will most likely be rendered infertile. Next consider the possibility of the perturbation affecting D, so that a machine $M = (P + G)$ produces $M' = (P' + G)$, where $P' = (A + B + C + D')$. This machine $(A + B + C + D') + \phi(A + B + C + D)$ will proceed to decode and copy the unaltered G under the decoding function $\psi()$, to recreate the original machine $M = (A + B + C + D) + \phi(A + B + C + D)$, and the phenotypic perturbation is not inherited. For the case where the genotypic perturbation affects the description, G, again, if the perturbation affects the description of A, B or C, the automaton will most likely be rendered infertile. If the perturbation affects the description of D in G, creating a machine $M' = (A + B + C + D) + \phi(A + B + C + D')$, M' will proceed to decode and copy G' under the decoding function $\psi()$ to create a new machine, $M'' = (A + B + C + D') + \phi(A + B + C + D')$, where $M'' = (P' + G')$. This case of mutation allow for the offspring of a machine M to be modified and eventually produce M'' which is itself self reproducing, but M'' *may* be more complex than its ancestor M. So there is a potentially densely connected network of mutational pathways linking arbitrarily simple to arbitrarily complex machines, having just $(A + B + C) + \phi(A + B + C)$ in common.

It is worth noting that when a perturbation occurs within P, the perturbation is not inherited in further generations, furthermore, when the perturbation occurs within G, there is a generation delay between when the perturbation occurs in the genotype

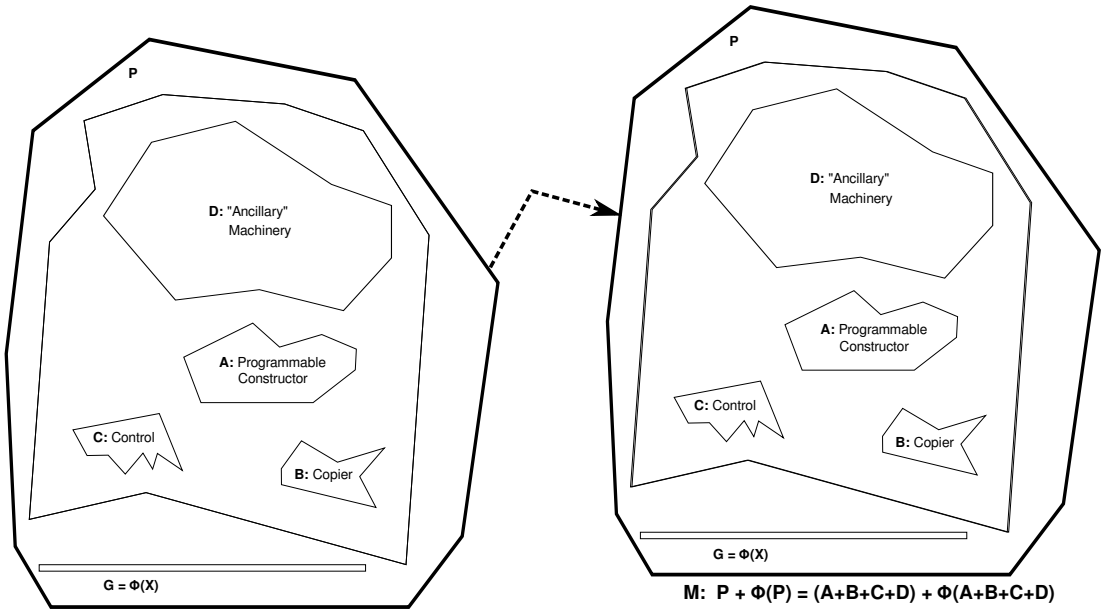

Figure 3.1: A schematic of the von Neumann style architecture of machine self reproduction. Adapted from McMullin (2012).

and when it is expressed in the phenotype.

Should a random perturbation occur while copying the description of A to an offspring, which results in $\phi(A' + B + C + D)$, then the machine $(A + B + C + D) + \phi(A' + B + C + D)$ will produce $(A' + B + C + D) + \phi(A' + B + C + D)$. It is possible that this machine will now have an altered general constructive automaton. Von Neumann/Burks stated "If the change is in A, B or C, the next generation will be sterile." (Von Neumann & Burks, 1966, p. 86), however it is conceivable that this statement is untrue, and does not take into account perturbations within the description of A which incrementally modify the decoding function $\psi()$ without completely breaking its reproductive functionality. Accordingly, this machine $(P' + G')$ may not be sterile, but implement an altered decoding function $\psi^*()$ where $\psi^*(G') = \psi^*(\phi(P')) = (P')^*$, to construct the machine $((P')^* + G')$. If there are changes in the mapping which allow $(P')^* = P'$, then the machine $(P' + G')$ will self reproduce successfully while implementing a different genotype-phenotype mapping (McMullin, 2000).

3.4 Interpretations of von Neumann's design

One specific aspect of von Neumann's design, which may have led to some confusion, was the fact the representation of von Neumann's machine within the cellular automata framework did not allow for spontaneous or indeterministic perturbations of the cell states. Therefore, when represented in such a deterministic system, the designed machine cannot actually undergo inheritable mutations, hence, this representation is strictly unable to support evolutionary change. Due to this, it might be supposed that

von Neumann's problem was limited only to the logic or architecture of self reproduction. Burks himself stated that von Neumann was interested in the following general question: "What kind of logical organization is *sufficient* for an automaton to be able to reproduce itself?" (Burks, 1970).

In order to rule out trivial self reproducing systems, such as crystal growth, it has generally been required that any relevant self reproducing configuration must be capable of universal computation, however, it is unlikely that the earliest self replicating molecules, from which all present living organisms have evolved had this capability (Langton, 1984).

In an attempt to build a platform simpler than von Neumann's which is still capable of accommodating self reproducing machines, British computer scientist Edgar F. Codd asked "What kind of logical organisation is *necessary* for an automaton to be able to reproduce" (Codd, 1968). Codd set out to reduce the complexity of von Neumann's CA and was able to demonstrate a configuration which required only 8 states per cell rather than von Neumann's 27. Both von Neumann and Codd's machines reproduce in a similar manner, however, Codd's machine is more heavily influenced by the physiology of the nervous system of animals, where instructions are transferred about the machine via *data paths*.

The data path consists of a string of cells in state 1 (core cells). This string of cells is surrounded on either side by cells in state 2 (sheath cells). Signals in the data path consists of a packet of two co-travelling states, the signal state (state 4, 5, 6 or 7) followed by the state 0. A data path may branch and fan out, or turn at right angles.

Computer scientist Christopher Langton recognised that if one of these data paths closed in on itself, it could form a dynamic loop which could allow the storage of a dynamic instruction rather than a static description in the form of a tape. Langton proceeded to design a different machine based on this principle, the Langton Loop, Figure 3.2.

Within von Neumann's machine, the general constructor must decipher a static symbol string (genotype) in order to acquire the instructions which will guide a constructor arm in building the offspring phenotype. With Langton's Loop, rather then having a static symbol string which is interpreted as a series of instructions, the unencoded instruction itself is simply propagating within a dynamic loop so that there is no need for a general constructor, or a decoding mechanism, as the decoded signal is already "active" within the loop.

Langton focused on designing a loop which can self reproduce, but cannot perform any further behaviour such as construction of arbitrary machines. If von Neumann's problem was one of purely machine self reproduction, than this alternative method shows that von Neumann's machine was overly complex, and the same result could be achieved by far simpler means. However, if we understand that von Neumann's problem was in understanding the evolutionary growth of complexity of arbitrary machines, then Langton did not provide a simpler alternative, as it does not incorporate a decoding

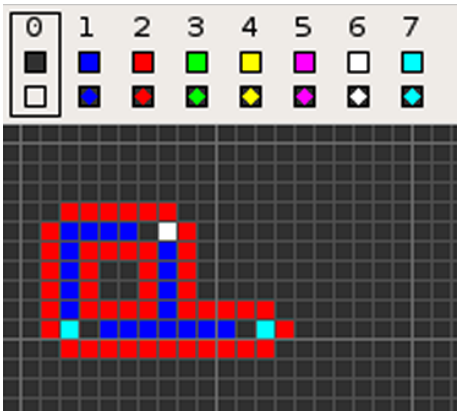

Figure 3.2: Screen capture of Langton's dynamic loop simulated in Golly 2.6 for Linux, (Trevorrow & Rokicki, 2015), an open source cellular automata platform. The dynamic loop can be seen on the left, with the data path/construction arm protruding to the right.

mechanism between genotype and phenotype, which is a fundamental aspect to both von Neumann's automaton, and all known forms of organic life on Earth which display an evolutionary growth of complexity.

3.5 Conclusion

This chapter explained in detail one possible architecture which is capable of self reproduction. This architecture, proposed by John von Neumann, incorporates a distinction between genotype and phenotype where a general constructor is required to decode the genotype, under some shared genotype-phenotype mapping, and construct the offspring phenotype. As the aim of this thesis is to study the evolution of symbol systems, von Neumann's architecture provides an appropriate foundation in which we can follow in designing such a system.

One particular candidate platform which may be used to host this architecture is the Tierra platform. The Tierran operating system has memory protection, and is already equipped with a number of operators, which will allow the introduction of random perturbations to the agents within the system. Furthermore, reproduction rates within Tierra are faster than those of typical cellular automata-based simulators. For example, Ray's initial experiment allows a population of approximately 500 to have a turnover of several generations[3] per second when run on a typical current personal computer hardware, vastly reducing the wall-clock simulation time.

Using these foundations, it should be possible to examine if potential mutational pathways may exist where the decoding mechanism is altered, resulting in an evolution of the genotype-phenotype mapping, and examine what properties and potential

[3]Generation time in Tierra refers to the time period in which it takes for sum of accumulated births and deaths in the soup, to be equal the twice the average population of the previous generation.

advantages these new lineages may possess.

Chapter 4

Tierra: A Platform For Artificial Life

4.1 Chapter Overview

The preceding chapter discusses the specific architecture, as described by John von Neumann, of a self reproducing machine which incorporates a mutable genotype-phenotype mapping. If realised on an appropriate artificial life platform, this architecture may provide proof-of-principle examples in which the evolution of a genotype-phenotype mapping may result in additional evolutionary lineages which are only possible with a mutable mapping. A possible candidate for this task is the Tierra virtual computer as it provides a platform which allows self reproducing programs in machine code to reproduce with random perturbations affecting the contents of memory, introducing noise and allowing for inheritable variation. In order to understand how the von Neumann architecture may be implemented within this platform, this chapter discusses the organisation of the Tierra virtual computer, its default instruction set, and the various possible operators which perturb the contents of memory.

4.2 The Tierra 6.02 Virtual Computer

Tierra 6.02 is an virtual computer developed by ecologist Tom Ray with the design objective to "support the evolution of self replicating machine code" (Ray, 1991).

The memory space in Tierra in which the machine code resides is called *the soup*. The soup is a circular memory of configurable size. Each memory location within the soup is an 8 bit, integer word, allowing a maximum of 256 distinct symbols to establish the machine code instruction set in which programs in Tierra can be built.

Tierra is a Single Instruction, Multiple Data (SIMD) class of parallel computer where multiple central processing units may perform operations on data in a shared simultaneously. Each CPU includes seven registers; Ax, Bx, Cx, Dx, Ex, Fx, and IP, each a 32 bit signed integer word. One register is reserved specifically for the instruction pointer (IP), while the remaining six are general purpose registers. The maximum soup size is limited by the size of the instruction pointer register which is 2^{32} memory locations.

A circular stack of size 10 is also included within the CPU, where each memory location is also a 32 bit wide signed integer word. The Tierra interface which displays its registers and stack is shown in Figure 4.1

Upon initiation of a simulation in Tierra, certain configurable parameters can be set via the `soup_in` file. This is a file which contain a list of "environmental" and "observational" parameters which affect the output of a simulation. Environmental parameters affect aspects such as the length of a simulation, soup size, mutation rates, number of creatures to inoculate a run etc., which actually affect the evolutionary trajectories of the creatures within the soup. Observational parameters do not affect evolution, but affect how the data is saved to the computer which is running Tierra.

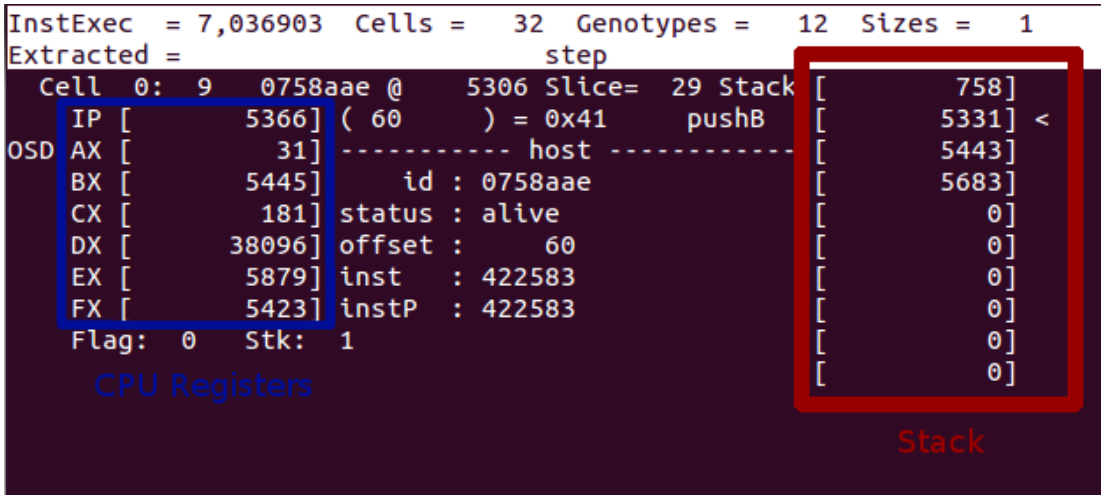

Figure 4.1: A screen capture of Tierra 6.02 highlighting the seven registers and stack of one CPU.

4.2.1 The Default Instruction Set

When designing the default instruction set for Tierra, Figure 4.2, Ray based his system on the Intel x86 instruction set. However, in order to simulate what he thought to be the properties of life, he adjusted this set to optimise for digital evolution.

The Tierra virtual computer allows for a maximum of 256 low-level machine instructions. The instruction set for a specific run is configurable in size and content, drawn from a library of pre-programmed potential instructions.

The default instruction set as configured by Ray used only 32 instructions. The classes of instructions which were implemented in the default instruction set fall into five categories; no operation, memory movement, instruction pointer manipulation, calculation, and biological and sensory.

The no operation instructions, `nop0` and `nop1`, are instructions that have no effect if executed directly, however, within Tierra these instructions are utilised in template addressing (as already described in Chapter 2). In Tierra, templates are used to specify memory locations rather than more conventional absolute or relative addressing. Templates are an idea borrowed from molecular biology, where molecules "address" one another by having complementary shapes. Templates in Tierra are complementary patterns of `nop` sequences, where `nop0` is complementary to `nop1` and vice versa, so the instruction sequence `jmpo nop1 nop0 nop1`, for example, causes execution of the code to jump to the nearest occurrence of the instruction sequence `nop0 nop1 nop0`.

The memory movement instructions include a series of `push` and `pop` instructions which copy data from the registers to the stack and vice versa. The stack is 10 memory locations long which is initialised with zeros, and is cyclic, so a `push` to a full stack will cause the stack pointer to loop back to the lowest memory location in the stack.

Default Tierra Instruction Set

No Operations: 2

`nop0`
`nop1`

Memory Movement: 11

`pushA` (push Ax onto stack)
`pushB` (push Bx onto stack)
`pushC` (push Cx onto stack)
`pushD` (push Dx onto stack)
`popA` (pop from stack into Ax)
`popB` (pop from stack into Bx)
`popC` (pop from stack into Cx)
`popD` (pop from stack into Dx)
`movCD` (Dx = Cx)
`movAB` (Bx = Ax)
`movii` (move from ram [Bx] to ram [Ax])

Calculation: 9

`subCAB` (Cx = Ax - Bx)
`subAAC` (Ax = Ax - Cx)
`incA` (increment Ax)
`incB` (increment Bx)
`incC` (increment Cx)
`decC` (decrement Cx)
`zero` (zero Cx)
`not0` (flip low order bit of Cx)
`shl` (shift left all bits of Cx)

Instruction Pointer Manipulation: 5

`ifz` (if Cx == 0 execute next instruction)
`jmpo` (jump to template)
`jmpb` (jump backwards to template)
`call` (push IP onto the stack, jump to template)
`ret` (pop the stack into the IP)

Biological and Sensory: 5

`adro` (search outward for template, put address in Ax)
`adrb` (search backward for template, put address in Ax)
`adrf` (search forward for template, put address in Ax)
`mal` (allocate amount of space specified in Cx)
`divide` (cell division)

Total: 32

Figure 4.2: The default Tierra instruction set as configured by Tom Ray.

Furthermore, executing a `pop` to an empty stack will cause the stack pointer to point to the top of the stack. There are also a set of `mov` instructions which are used to copy data between registers, from the soup to a register, and from a register to the soup.

Arithmetic in Tierra's registers allow for addition, subtraction, incrementing, decrementing, shifting left, and bitwise operations such as flipping the lowest order bit or replacing a number with zero.

The "biological/sensory" instructions can search for a specific address location via the address instructions `adro, adrb` and `adrf`, which will search outwards in either direction for a matching template, specifically backwards for a matching template, or specifically forward. Once found, the absolute memory address of the memory location, directly following the matching template will be copied to the Ax.

The memory allocation instruction `mal` will attempt to allocate exclusive write access to a block of memory whose size is determined by the number stored in the Cx register. The instruction `divide` will deallocate any write access that a CPU might have on a block of memory, and it will assign a new CPU to start executing at the start of that block of memory. The CPU which executed the divide will then continue to the next instruction in memory.

It should be noted that there is some level of ambiguity with the semantics of the term "instructions" when referring to the assembler language used within Tierra 6.02. To understand this, what is referred to as "the `opcode.map`" must be introduced. When utilising the maximum of an 8 bit word size, the Tierran CPU can operate upon 256 distinct binary numbers. How these numbers are interpreted when executed as instructions by the CPU is defined by the preconfigured `opcode.map` file, so the same 8 bit low-level machine code may result in a different action when executed, depending on the configuration of the `opcode.map` file. Furthermore, data in memory may not always be interpreted as an executable instruction. In general, the contents of a memory location within the soup may be interpreted as an executable instruction, or interpreted as numerical data.

Von Neumann style self reproducers require a passive genotype which is never executed, so the term "instruction" is not ideal in describing the contents of these memory locations. In traditional descriptions of Tierra, all memory location content/values are referred to as instructions, even though they may never be executed. To avoid confusion, *from now forth* we will refer to the arbitrary data stored within the Tierran memory locations as *symbols* where a symbol may be interpreted as either passive data (represented as a number) or an active instruction, and the underlying symbol alphabet will normally be labelled with its underlying numerical representation.

4.2.2 Self Reproduction by Self Inspection (Self Copiers)

Tierra was designed as an artificial life platform where populations of self reproducing programs can compete with each other for both CPU time and memory space.

Upon initiation of a run, the `soup_in` file is read which instructs Tierra which creature or creatures within the Tierran genebank to use to inoculate the soup. Other information such as where the seed ancestors are to be placed within the soup, number of generation to run the simulation for, and the seed for the pseudorandom number generator which controls the "random" perturbations are also included.

The default self copier, *0080aaa*, as designed by Ray is illustrated in Figure 4.3. This creature is comprised of a string of symbols, which consists of four separate functional blocks or subroutines in order to facilitate self reproduction, namely: *initiation, replication, separation* and *reset*. During the initiation stage, the creature examines its length, and the Tierra operating system inspects the surrounding memory space for a single contiguous section of memory of equal size and write permission is allocated to that block of memory. During replication, the contents of the parent creature's memory space is copied in succession to the allocated memory space. Upon separation, the Tierra operating will assign a new CPU to the newly allocated memory space, the parent loses write access to this memory space and an offspring is born. Reset is the final stage of reproduction, where the parent resets its registers and stack pointer back to their starting position, and the reproduction cycle may begin again.

4.2.3 Darwinian Operating System

In order to support Darwinian evolution, Tierra has certain built in functionalities which allows it to simulate inheritable variation and the struggle for existence.

Time Allocation

Upon birth, each creature's identity is entered into a circular queue of processes called the slicer. The slicer allocates CPU time to each creature. The slicer may operate under one of three possible mechanisms for allocating time, which is configurable within the `soup_in` file via the `SliceStyle`. The slice size may be fixed, in which each consecutive creature receives an equal amount of CPU time. The slice size may vary, in which a random fluctuation may be applied which varies the time slice randomly within a certain window (the default setting). Finally, the amount of CPU time allocated may be a function of creature length, where larger creatures are allocated larger shares of CPU time in order to allow for longer more complex creatures. A duration of a simulation in Tierra may be measured in terms of CPU instruction executions, which is the total number of instructions which are executed in a single run, (regardless of the number of CPUs)

Death

Upon birth, a creature's identity is entered onto the end of a linear reaper queue. If the proportion of memory allocated within the soup exceeds a certain threshold which is configurable within the `soup_in` file, then the creature at the top of the reaper queue

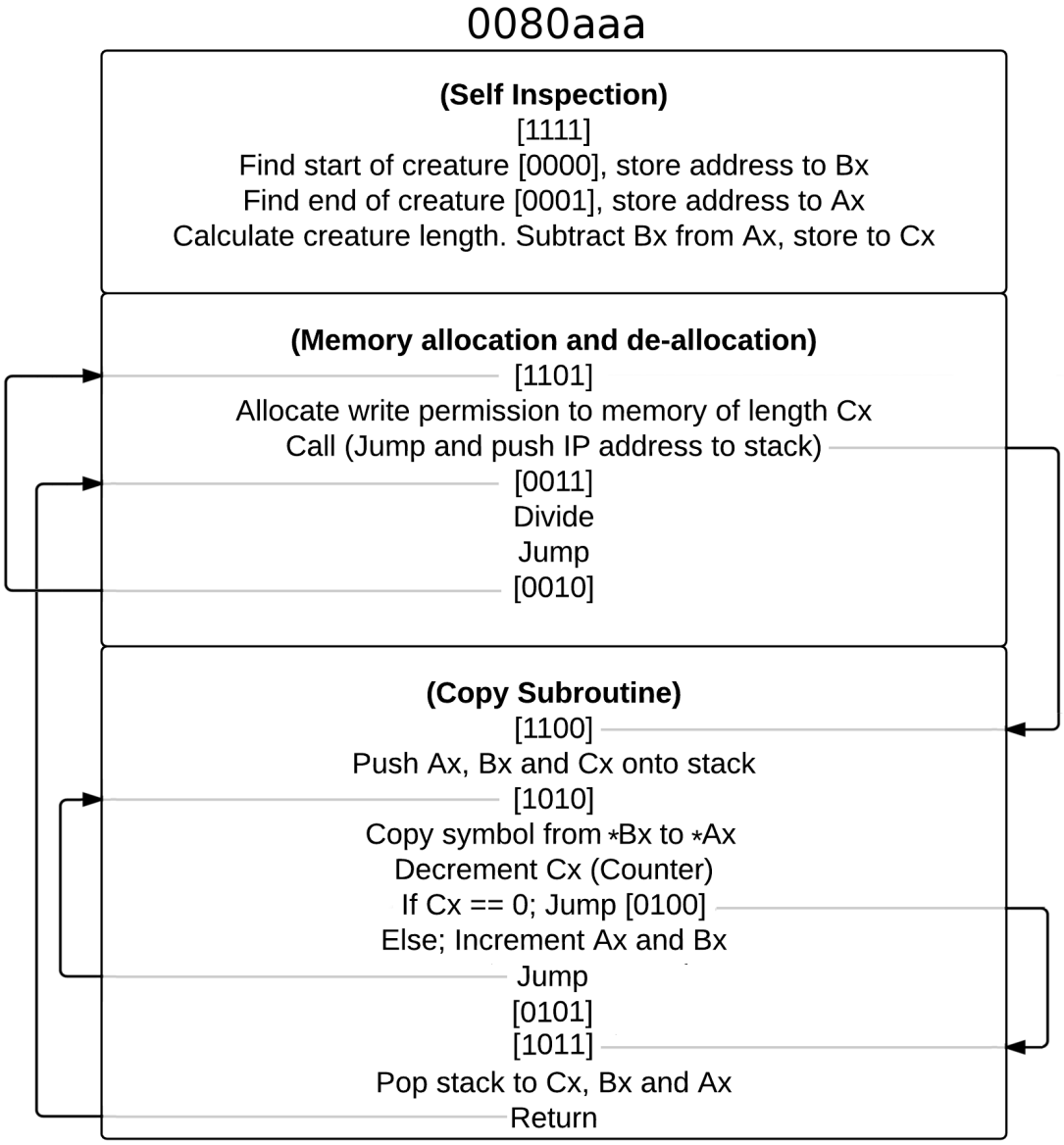

Figure 4.3: This schematic diagram illustrates the structure of the Ray style self copier. Template addresses are denoted within square brackets. Registers with an asterisk superscript represent situations where the data within the register is not interpreted directly as numerical data, but as a pointer to data at address location within the soup.

will be "killed", at which point it is removed from the queue and its CPU is removed from the system. Certain actions within Tierra are considered "errors", for example, a creature trying to write data to a memory location in which it does not have write access. Every million CPU executions the error count of the creatures in the queue are analysed. If a creature has generated an error count greater than that of the creature above it in the queue, then the two creatures will trade places, moving error prone creatures one position higher in the queue and therefore resulting in an earlier death. Conversely, a successful birth of an offspring will result in the parent creature swapping places with the creature below it in the queue, extending the lifetime of reproducing creatures. However, as a successful birth coincides with a new creature being inserted into the bottom of the reaper queue, shifting the entire contents of the queue up one step, the act of reproduction will effectively result in the parent creature remaining in the same position while all other creatures move up the queue.

Memory Allocation

By default, all memory locations within the soup have write protection. In order for a creature to reproduce it must seek write permission on a block of memory within the soup. If no suitable contiguous blocks of memory are available, than the reaper is activated and creatures are reaped until a suitable block of memory becomes available. A creature may only have write permission to one contiguous block of memory at any one time. Once a parent and offspring successfully divide, write permission is removed and neither parent nor offspring can modify the content of the offspring's memory.

Perturbations

There are three classes of random perturbations which may disrupt the functioning of creatures within Tierra: point perturbations, splicing, and flaws.

- Point Perturbations:

  Point perturbations affect the memory locations within the soup. This perturbation may either be a bit flip, where a single bit is flipped, or a replacement, where the number is replaced by a random number within the available symbol alphabet.

  There are two actions which may trigger this class of perturbation. Each time a CPU executes an instruction, there is a certain probability that a random memory location within the soup will be perturbed and its contents will be altered. Secondly, upon birth of an offspring, a random memory location within the offspring creature may be targeted, and a point perturbation may be applied.

- Splicing:

  There are three splicing mechanisms implemented in Tierra and they may only occur within a creature at the time of birth. Splicing may occur at random

locations within a creature, or between segment boundaries marked by template addresses.

Deletions. There are two types of deletions, point deletions and segment deletions. Either a single symbol, or a segment up to half of the offspring's total length may be deleted. During a deletion, the Tierra operating system effectively overwrites the symbols which are to be "deleted" with the offspring's memory image immediately to the right of the segment to be deleted. Next, a section of memory to the very right of the creature, of equal size to that which was to be deleted, is freed and memory protection is removed to create a shorter creature. For example, if a hypothetical offspring's memory image was the sequence of numbers `[1,2,3,4,5,6,7,8,9]`, and the segment `[4,5]` was to be deleted, the memory image `[6,7,8,9]` will be copied by the Tierran operating system, and written to memory, starting at the memory location `[4]`, leaving `[1,2,3,6,7,8,9,8,9]`. As the memory image to be deleted was 2 symbols long, the last two symbols in the memory image lose their write protection, so the memory image will now look like `[1,2,3,6,7,8,9]8,9` where the final two symbols still remain in memory but do not have any write protection, leaving the offspring 2 symbols shorter.

Insertions. Similar to deletions, there are point and segment insertions. A random creature within the soup is chosen and a segment of memory up to half of its total length is selected. A random location within the offspring is targeted and selected segment of the random creature is written to memory, followed by the memory image of the offspring immediately to the right of the target location, overwriting any underlying memory locations. For example, if an offspring with a memory image `[1,2,3,4,5,6,7]` is selected for an insertion, a random point within the creature is selected, e.g, after the fifth symbol. The entire offspring's memory image directly to the right of this point, `[6,7]`, is copied by the Tierra operating system. A segment of a random creature within the soup, e.g. `[8,9,10]`, is also selected and copied by the Tierra operating system. The random creature's copied segment is now written to the offspring's memory image starting after the fifth location, and the copied segment of the offspring is written next in memory. As this "inserted" segment is three symbols long, the size of the offspring's write protected memory image is extended by three symbols and the final offspring's memory image will be `[1,2,3,4,5,8,9,10,6,7]`.

Crossover. A random creature within the soup is targeted and a segment within its memory is selected. A random crossover point within the offspring is targeted which divides the offspring into two segments. The smaller segment of the offspring is replaced by the previously selected segment from the random creature.

- Flaws:

  Flaws are perturbations which affect the contents of the registers within a CPU.

When performing arithmetic such as adding or subtracting data, flaws may be introduced which vary the outcome by +/- 1.

4.2.4 Conclusion

Tierra is a possible candidate for supporting the von Neumann architecture as it allows for inheritable variation for self reproducing programs. However, Tierra was initially designed with self copying programs in mind. Significantly more complex programs must be designed in order to display von Neumann style reproduction which implements a mutable genotype-phenotype mapping. The default instruction set is very limited and only allows for very specific registers to be utilised for memory manipulation. For example, the only instruction which copied data from one memory location within the soup to another, is the `movii` instruction, which copied the contents of the address pointed at by register Bx to the address pointed at by register Ax. There is no general instruction for copying the contents of a memory address to a register, or for copying the contents of a register to a location in memory. There are only `push` and `pop` instructions to move data between four out of the possible six general purpose registers, further limiting the functionality of the instruction set. Finally, there are only two instructions which perform arithmetic, `subcaab` and `subaac`, which perform subtraction and no instruction for addition. Out of a possible six available registers, these two instructions only allow subtraction between Ax and Bx, and Ax and Cx. A series of `push` and `pop` instruction must be performed if arithmetic must be performed on any other registers, making both the task of designing of programmes significantly more difficult, and also limiting the amount of registers which arithmetic may be performed, therefore decreasing the evolvability of these programs. Many changes must be made to the Tierra source code, and to the default configuration options of Tierra before the von Neumann architecture can be straightforwardly implemented within Tierra.

Chapter 5

Implementation of von Neumann's Architecture Within Tierra

5.1 Chapter Overview

The previous chapter described the Tierra operating system in detail and explained why it might be a suitable evolutionary platform for hosting von Neumann style self reproducers. It also discussed some of the limitations of the Tierra platform in relation to programming a von Neumann type ancestor. Certain changes must be made to the Tierra default instruction set in order to allow it to host such an ancestor. In this chapter, specific details of possible methods in which such an architecture may be implemented within Tierra are also discussed and solutions are proposed.

5.2 Implementation of von Neumann's Architecture for Machine Self Reproduction Within The Tierra Platform

5.2.1 Implementation of a mutable genotype-phenotype mapping within Tierra

The genetic code is a nearly universal feature of life on earth, (Bedau, 2003) and yet it is difficult to understand how it could evolve to the high level of complexity to which it exists today. Although von Neumann's architecture allowed for heritable mutations to all aspects of the general constructor, which includes the genotype-phenotype mapping, von Neumann himself stated that a mutation which affects the functionality of the genotype-phenotype mapping, will result in a sterile offspring:

> If there is a change in the description $\phi(A + B + C + D)$, then the system will produce, not itself, but a modification of itself. Whether the next generation can produce anything or not depends on where the change is. If the change is in A, B, or C, then the next generation will be sterile. —(Von Neumann & Burks, 1966, p. 86)

This hypothesis had not been experimentally tested, so this thesis aims to explore the mutational space of digital organisms with a von Neumann architecture to build a case for, or against, the possibility of a mutable genotype-phenotype mapping.

Typically, self reproduction within Tierra is accomplished via self copying, where a creature must inspect its entire memory image in order to construct an identical offspring. This mechanism is loosely analogous to the reproduction process which occurs in the RNA world hypothesis which posits that at the earlier stages of evolution, RNA acted as both template and template-directed polymerase, and there existed no distinction between genotype and phenotype. In order to implement the von Neumann architecture within the platform of Tierra, the seed automaton must enforce a division of labour between the storage of genetic information and the catalytic functionality, hence recognising the roles of genotype and phenotype. Furthermore, the phenotype

must consist of three[1] subcomponents of a von Neumann self reproducer's phenotype; the general constructor A, the copier B and the control mechanism C.

5.2.2 The Instruction Set

While initially designing the von Neumann architecture within Tierra, it came to light that for many reasons the default instruction set configuration, and furthermore, the set of instructions available within the Tierra instruction library, was unsuitable and it was necessary to expand the instruction library in order to construct the new ancestor. Code revisions can be seen in Appendix D.

Memory Manipulation

The general constructor (or programmable constructor) within a von Neumann self reproducer must be able to decode (or be programmed to decode) *any* computable function. A Turing complete general constructor allows for any possible genotype to be decoded during construction of an offspring phenotype. Tom Ray states that the Tierran language is Turing complete, as:

> Sets of machine instructions similar to those used in the Tierra simulator have been shown to be capable of 'universal computation'. This suggests that evolving machine codes should be able to generate any level of complexity —(Ray, 1991).

However, there is one crucial difference between his default instruction set, and those of other assembler languages. The default Tierra instruction set does not include a generic memory read or write instruction. Assembler languages usually have an instruction for reading from memory, and a separate instruction for writing to memory, however, the default Tierra instruction set combines these two actions into a single instruction, `movii`, which copies the contents of the absolute memory location pointed at by the Bx address, to the contents of the absolute memory location pointed at by the Ax register. In doing so, the reading and writing of data is performed in a single instruction and therefore, it is impossible to directly inspect and perform arithmetic manipulation on the contents of memory address.

A much more practical approach would be to split the `movii` instruction into a read and write instruction, which would allow a creature to examine data within its own memory space. The Tierra instruction library does contain two pre-programmed instructions, `movdi` (read) and `movid` (write), which can be included within the instruction set of a particular run, which allow data to be copied from memory to a register, and from a register to memory. `movdi` will copy the contents of the absolute memory location pointed at by the Bx address to the Ax register, and `movid` will copy the

[1] We will ignore the ancillary machinery D at this point as it is not immediately relevant to the process of self reproduction in the Tierra framework.

contents of the Bx register to the memory location pointed at by the Ax register. However, there are there is no choice between choosing the source and destination registers, which further increases the convoluted task of programming with Tierra, furthermore, the mnemonic `movdi` does not relay any information to the programmer as to which are the source and destination registers.

A set of instructions of the form `movaB` was therefore added to the Tierra source code. This instruction allows the CPU to read the numerical data in the location pointed at by the Bx register, and store this number within the Ax register. This instruction mnemonic takes the form `[instruction type][destination][source]`, where the letters in the operands highlight which registers will be affected. For upper case letters, the data within the specified register acts as a pointer to the corresponding absolute memory address in the soup, and the data within that memory location is accessed. For lower case letters, the data within the specific register itself is accessed. A complete set of `mov` instructions were added to the instruction set to allow data to be copied freely between any of the registers and the soup, which allowed for easier programming of the creatures.

Tierra was designed with 6 general purpose registers, however, the initial instruction set only includes `push` and `pop` instructions for 4 of these registers. This limits the functionality of the creature and makes it more difficult to develop increasingly complex creatures. `push` and `pop` instructions were accordingly added for all available registers.

Calculation

The Tierran instruction library is also very limited by the small number of pre-programmed instructions which can perform calculations upon data. For addition, there was only one instruction, `addbbc`. For subtraction, there were only two options, `subcab` or `subaac`. This mnemonic takes the form `[operation][destination][source][source]` so for example, it is possible only to add the data in Cx and Bx together and place the sum in Bx. This limitation on available registers for performing arithmetic poses difficulty when designing more complex creatures, as each arithmetic operation becomes increasingly convoluted as they must be preceded and followed by a series of pop and push instructions in order to move the data within the registers to the appropriate positions temporarily in order to perform the calculations.

There was only an increment, `inc`, instruction available for Ax, Bx and Cx, and a decrement instruction, `dec`, for Cx, which means that in order to increment or decrement any other register, a series of push and pop instructions must also be executed. The instruction set was therefore expanded to include a wider variety of `add`, `sub`, `inc` and `dec` instructions to simplify the programming of more complex creatures.

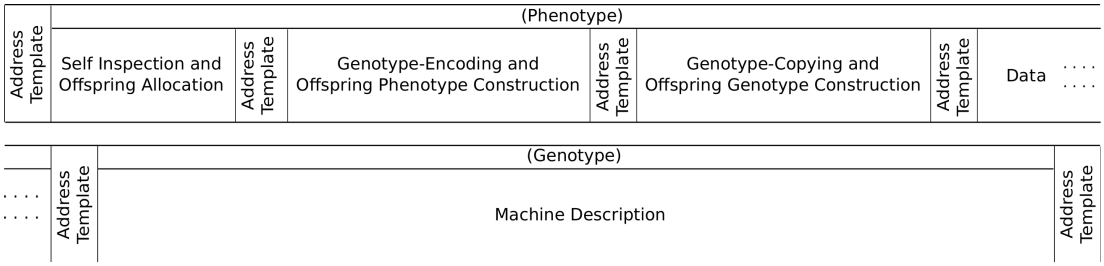

Figure 5.1: Schematic diagram representing the memory image of a Von Neumann style ancestor in Tierra.

Instruction Pointer Manipulation

The initial instruction library included a conditional jump instruction `ifz`, which checks if the number stored within the Cx register equals zero. If true, the next instruction would be executed, otherwise the next instruction would be skipped.

There was no conditional jump instruction which checks if the number within Cx *is not* equal to zero, `ifnz`. This limitation somewhat complicates the procedure of creating conditional jumps and loops in Tierra, so the instruction `ifnz` was created and inserted into the instruction library.

5.2.3 The von Neumann Ancestor Structure in Tierra

A von Neumann style architecture that reproduces via implementation of a genotype-phenotype mapping was designed within Tierra as outlined in Figure 5.1. Prior to reproduction, this seed creature first inspects its genome and calculates the appropriate offspring size and allocates a block of memory. While constructing an offspring phenotype, the CPU incrementally steps through each memory location within the parent genotype and the symbol stored at each address is inspected. The creature then decodes the genotype under some arbitrary genotype-phenotype mapping, and uses this description to construct the offspring phenotype. Following the construction of the offspring phenotype, the parent's genotype is then copied to the offspring space and the connection between parent and offspring is severed. At this point the parent loses write access to the offspring's memory block and a new CPU is created and allocated to the offspring. While copying the genotype, should a random perturbation occur which affects the encoded description of the general constructor (or otherwise modify the decoding process), then the creature's offspring will incorporate a mutated genotype-phenotype mapping. This is the particular phenomenon which is initially to be investigated.

5.2.4 Conclusion

This chapter discussed the relevance of investigating symbol systems which reproduce via implementation of a mutable genotype-phenotype mapping. How such a system may be designed and implemented within Tierra was presented. The default instruction set for Tierra was severely limited so many new instructions were added to increase the functionality of the Tierran instruction set, and also allow easier programmability. A possible ancestor structure was described which could possibly implement a mutable genotype-phenotype mapping. The next chapter describes the experimental procedures which were carried out when implementing a von Neumann ancestor within Tierra. The resultant data which was produced from the experiments is documented and analysis of these results is presented.

Chapter 6

Experimental Procedure, Results and Discussions

6.1 Chapter Overview

The previous two chapters described the Tierra platform in detail, von Neumann's self reproducing architecture, and a possible mechanism in which a von Neumann style creature may be implemented within Tierra. This chapter introduces the designed creatures and documents a series of experiments using these creatures. The particular phenomena which were observed during these experiments are presented, analysed and discussed. The experiments within this chapter were run on Tierra 6.02, however it was necessary to make several revisions to the initial source code in order to achieve a suitable platform. The revisions which were applied are documented and explained in Appendix D. The revised source code, experimental data, analysis tools and designed creatures can be found at `http://alife.rince.ie/db_phd_2015/`. Furthermore, in order to analysis the data produced from evolutionary runs, several analysis tools were created, written in the Python programming language. These tools were developed to analyse data, specific to these experiments, e.g., searching for change in the genotype-phenotype mapping of a lineage, however, useful data analysis tools which can be applied to any Tierra run were also created, such as tracing the lineage from a specific creature to the seed ancestor, and graphing the population of a specific strain from its emergence to its extinction. The created data analysis tools are documented in Appendix E.

6.2 Experimental Procedure, Results and Discussions I

At the highest level, a von Neumann creature is comprised of two distinct components, a passive genotype, which acts exclusively as an information store for a creature's description, and an active phenotype, which is responsible for all dynamic functionality of the creature including the ability to decode the genotype and construct the described creature to implement self reproduction. Each word value or symbol in the Tierra memory may be interpreted as either (numerical) data, or a functional instruction, depending on how the symbol is used in practice. Symbols situated within the genotype will be labelled *g-symbols*, and symbols situated in the phenotype will be labelled *p-symbols*. Both g-symbols and p-symbols share the same alphabet, which is a subset of the maximum available word alphabet within Tierra.

In order to encode the phenotype of such a creature, some particular genotype-phenotype mapping must be implemented. The evolutionary dynamics of a lineage seeded from such a creature will be significantly affected by the nature of this arbitrarily selected initial mapping.

For the purpose of this study, a simple bijective, mono-alphabetic substitution mapping was initially chosen. This was loosely based on the biological genetic code, in which an mRNA, consisting of a one-dimensional string of symbols (nucleotides), is transcribed into a different one-dimensional string of symbols (amino acids). If a single

letter in an mRNA codon gets perturbed, then the affected codon may result in the incorporation of a different amino acid in the constructed protein.

If such an architecture is implemented which allows perturbations to the genotype which may alter the description of the general constructor, specifically, altering the genotype-phenotype mapping function $\psi()$, then new evolutionary trajectories may arise where creatures implement an altered genotype-phenotype mapping.

This mapping process can be implemented via a *look-up table* within the general constructor. This is only one specific mechanism which allows for the implementation of this particular mapping and is by no means a fundamental requirement, as there exists countless other formalisms which would satisfy the conditions for von Neumann style self reproduction, so we must be aware that any phenomenon observed may be specific and characteristic to the specific substitution mapping system which is implemented.

The look-up table consists of a simple one-dimensional symbol string representation of all the available p-symbols in the intended phenotype space under an arbitrary permutation. The permutation of p-symbols within the look-up table will determine the genotype-phenotype mapping which is implemented within this substitution mapping. During construction of an offspring phenotype, the parent's genotype is incrementally examined by the general constructor in order to decode each g-symbol. A g-symbol is read and stored in a dedicated CPU register. This symbol is then used as a numerical offset for a look-up table pointer, changing the look-up table pointer to an address in a corresponding position within the look-up table. The symbol stored in this location is now accessed and can be copied and written to the offspring phenotype where it may subsequently function as an instruction or as numerical data. This activity facilitates the mapping of passive g-symbols which are stored within the parent genotype, to active p-symbols incorporated in the offspring phenotype.

Because the decoding mechanism which includes the look-up table is situated within the phenotype, and therefore itself has an encoded description within the genotype, it is subject to heritable mutations. Random perturbations within the genotype which alter the description of the decoding mechanism may result in a mutation of its offspring's genotype-phenotype mapping. This may have the effect of introducing new mutational pathways for the creature, which were not possible under the previous genotype-phenotype mapping.

For the initial investigation, an arbitrary substitution mapping from genotype to phenotype was chosen, where the g-symbols were mapped onto arbitrary p-symbols. This initial substitution function $\psi()$ is determined by the permutation of symbols within the look-up table. It is worth noting however, that regardless of the initial permutation chosen for the (phenotypic) look-up table, the initial description of the look-up table within the genotype will always be the same consecutive sequence of symbols. If we let S represent the non-permutated list of symbols which exist in the particular Tierra configuration[1], then the permutation of the look-up table depicts how

[1]In this case, the non-permutated list of symbols is represented by a list of consecutive binary

each individual element within this non-permutated list of symbols is decoded under $\psi()$[2]. The look-up table can now be described as $\psi(S)$. If a phenotype in encoded to generate a seed creature genotype, the encoded look-up table would be represented by $\phi(\psi(S)) = \psi^{-1}(\psi(S)) = S$. Therefore, regardless of the initial permutation of the look-up table, the encoding of the initial look-up table in the genotype will always take the form of S. A series of look-up table based creatures were designed and a schematic flowchart of the final prototype is shown in Figure 6.1.

6.2.1 Classifications of Emergent Behaviour

When discussing the emergent behaviour which may arise from evolutionary models such as Tierra, it is worth first classifying the various types of behaviour which we might expect to see. Cariani (1991) described "emergence-relative-to-a-model" where the model constitutes the observer's expectations of how the system will behave in the future. If the system evolves such that the model no longer describes the system, we have emergence in this sense.

Cariani recognised three different types of such emergence: syntactic, semantic and pragmatic. Syntactic (symbolic or non symbolic) operations are those of computation. Semantic operations are those of measurement and control. Pragmatic operations are those of performance-measuring and relates to criteria which controls the selection (Ray, 1991).

In relation to Tierra, symbolic emergence (syntactic), takes place within the CPU registers where arithmetic operations take place, while non-symbolic emergence takes place within the memory locations of the soup. Symbolic emergence arises via non-hereditary perturbations to mathematical operations such as a CPU performing an `add` or `subtract` incorrectly. Non-symbolic emergence relates to any change in the executable code and data of the creature.

Semantic emergence affects the creature's sensory subroutines, and how it "communicates" with itself or its environment. In Tierra, this type of emergence generally occurs via the modification of the template addresses. There are two different functions of templates address in Tierra, which are distinguished here as *source templates*, and *destination templates*. A source template is a template address which informs the CPU of the `nop0/nop1` sequence which must be searched for. This form of template generally exist within the memory image of the creature which owns the executing CPU (unless the CPU has been captured by another creature). A destination template may exist anywhere within the soup, be it within the creature itself, within a neighbouring creature, or within unallocated memory previously allocated to a deceased creature[3].

numbers from 00000 to 11111 when using a 5 bit symbol alphabet (32 symbols).

[2]The first location in the look-up table represents which p-symbol is mapped onto by 00000. The second position in the look-up table represents which p-symbol is mapped onto by 00001 etc.

[3]When a creature is reaped, write protection is removed from its memory space, and its CPU is removed, however, its contents is not deleted, so unallocated space within the soup will generally contain the fossil remains of reaped creatures.

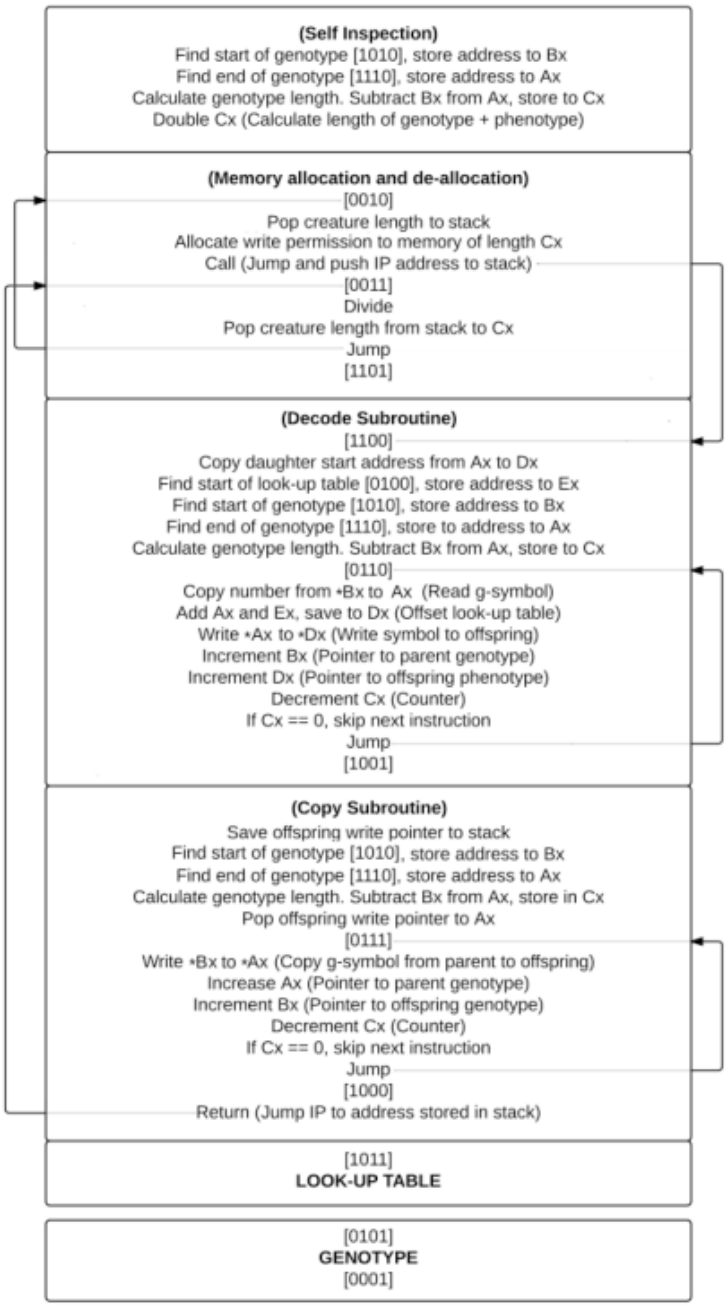

Figure 6.1: This schematic diagram illustrates the structure of the *look-up table* based von Neumann style self reproducer which was designed. Template addresses are denoted within square brackets. Registers proceeded with an asterisk represent situations where the data within the register is not interpreted directly as numerical data, but as a pointer to data at address location within the soup.

When the CPU processes a source template, the Tierra operating system will scan the soup for the nearest occurrence of its complement, the destination template. When a destination template is located, its absolute address location within the soup is identified and stored in a CPU register. If a source template is perturbed, the "meaning" of the template changes and a different destination template will be sought, which may or may not exist within the creature's own memory image. Similarly, if a destination template is perturbed, its "meaning" is also changed, so it may come to match a different source template which uses the complementary `nop0/nop1` pattern.

Pragmatic emergence, is the least common form of emergence and corresponds to any Darwinian selection event which results in an increase in the fitness of the lineage. Fitness in Tierra is implicit and is determined by the creatures themselves and their ability to self reproduce, therefore, pragmatic emergence in Tierra would be recognised as any change in the creature's phenotypic behaviour which reduce the reproduction time.

In order to model evolutionary behaviour and build tools which analyse the resulting data, one should ideally know what sort of behaviour is expected. By highlighting the possible forms of emergence, we can now move forward and run evolutionary experiments where we can predict and search for the particular forms of emergence which are being investigated and be aware of other forms of emergence which may also emerge but are not central to our research question, namely investigating the effects of implementing a mutable genotype-phenotype mapping.

6.2.2 Degeneration to Self Copying

The soup was inoculated with the initial design of the von Neumann ancestor. Tierra's internal naming mechanism labels this creature as `0344aaa`, which only tells us that this creature is 344 symbols long. For clarity, a separate, complementary naming scheme will be used in this text which will supply a brief description of the creature, thus: `vn_lut32_344`. This indicates that it is a von Neumann style creature (`vn`), its genotype-phenotype mapping incorporates a look-up table (`lut`), with a maximum alphabet size of 32 symbols, and the entire creature is `344` symbols long, which is documented in Appendix A.1. The applicable `opcode.map` files and `soup_in` files are found in Appendix B.1 and Appendix C.1 respectively.

In the first set of experiments `vn_lut32_344` reproduced effectively and populated the memory to form a stable ecosystem provided all random perturbations are disabled. This demonstrated that `vn_lut32_344` is a stable self reproducer in the absence of disruption. However, when random perturbations are switched on, a short period of stasis would consistently be followed by a rapid change in population, where the population of von Neumann ancestors would revert to a population of self copiers. A self copier will generally reproduce at a greater rate than the von Neumann style reproducer as it can simply copy its memory image during reproduction rather than

performing a comparatively time expensive decoding function. The emergence of self copiers quickly drives the population of von Neumann style reproducers to extinction, hence preventing any further study on the evolutionary trajectory of lineages with the von Neumann architecture. Under investigation by comparing the memory space of this self copier with that of the seed ancestor it was found that the ancestor had undergone a single point mutation which resulted in the alteration of a source template. In order to construct an offspring, the seed ancestor calculates its entire length, and allocates a block of memory of equal size to construct its offspring. Next it calculates the length of the offspring phenotype and then proceed to decode and construct an offspring phenotype. Finally, it measures the genotype length, and proceeds to copy the genotype to the offspring. When the genotype is copied, the parent divides and resets its registers so that the process can start again.

However, the relative address #201 within `vn_lut32_344`, which is situated within the genotype, was mutated from a `3` to a `2`. The `3` and `2` g-symbols translate to the p-symbols `0` and `1` respectively under the arbitrary mapping chosen. This resulted in an instruction sequence `call nop0 nop0 nop1 nop1` being changed to `call nop0 nop1 nop1 nop1`. Within `vn_lut32_344`, the instruction sequence `call nop0 nop0 nop1 nop1` is located directly after the offspring's memory image has been allocated, and directly before the genotype decoding subroutine. `call nop0 nop0 nop1 nop1` causes the current instruction pointer's absolute address location to be pushed to the stack, and the instruction pointer will then jump to its complement, `nop1 nop1 nop0 nop0` which marks the start of the genotype decoding subroutine. The creature then decodes the genotype and creates the offspring phenotype. However, when the mutated instruction sequence "`call nop0 nop1 nop1 nop1`" is executed, the instruction pointer jumps to the template `nop1 nop0 nop0 nop0` which marks the start of the creature's genotype copy loop and completely skips the decoding subroutine and the instruction sequence which fetches the start address of the genotype and calculates the length of the genotype.

At this point, the destination register, Ax, contains the starting address of the offspring. The source register, Bx, contains the starting address of the parent, and the count register, Cx, contains the length of the entire creature. Therefore the mutant will proceed to copy its entire memory image to the allocated memory block and divide, completing the reproduction cycle. As the creature no longer requires a time consuming decoding process, it will reproduce faster. Furthermore, unused symbols will occupy the majority of the memory image of these newly created self copiers. These are symbols associated with what previously acted as the decoding subroutine and the entire genotype which are no longer used. The length of the creatures dominating the soup immediately reduced following the emergence of a self copiers due to random deletions removing these unused regions, creating shorter, faster reproducing creatures. In any case, the von Neumann style reproducers, which are comparatively expensive on CPU time will be selected against and become extinct.

This emergence of self copiers in these experiments impeded the study of the evolution of the genotype-phenotype mapping, which was the intended aim of this study. In order to tackle this problem, the initial creature design was revised. It was noted that although the ancestor exhibited "genetic reproduction", it was in at least one sense not a full representation of the "von Neumann architecture". The von Neumann architecture includes a general constructor, which when provided with a description of any arbitrary machine, can decode the description, and construct an offspring machine. However, this seed creature did not follow this logical formalism as it was designed to only construct creatures of equal length to itself. If the genotype were perturbed to one which describes a creature of longer or shorter length than itself, then it could not reproduce successfully. This is due to the fact that in order to allocate space for an offspring, the creature would inspect its entire memory image and allocate a memory block of equal size to write its offspring too. This method uses the length of the parent's phenotype plus genotype to calculate the offspring length. A true von Neumann self reproducer should only examine the genotype in order to construct an offspring. A new technique was designed and implemented, in which the ancestor calculates the required space for its offspring's memory image via inspection of the genotype alone.

In order to simplify the coding of creatures, the substitution mapping was modified to be a simple identity map, and the instruction set was increased from 32 to 64, to allow for extra functionality when coding.

6.2.3 Pathological Construction

Using the mentioned amendments, a new creature, `vn_lut32_311`[4], was designed. The creature's code, `opcode.map` file and `soup_in` file are found in Appendix A.2, Appendix B.2 and Appendix C.2 respectively.

When all random perturbations are disabled, `vn_lut32_311` reproduced effectively and populated the memory to form a stable ecosystem of identical creatures, proving that this creature was indeed, self reproducing effectively. However, when all random perturbations are switched on a particular phenomenon was consistently observed where what is here termed *pathological constructors* quickly emerged, which then typically leads to catastrophic ecosystem collapse. Pathological constructors are categorised as creatures which repeatedly (and rapidly) construct multiple short, malfunctioning, offspring. Pathological constructors are evolutionary dead ends in themselves (as they do not self reproduce) but they are a hindrance to an ecosystem because their offspring, although sterile, still occupy both memory space and CPU time. If several pathological constructors coincide in time, their production rate can be so high that their non-functional offspring stochastically displace the entire population of functional self reproducing creatures, resulting in ecosystem collapse. This is obviously a problem as it prevents any long term evolutionary trajectories from being studied.

[4]Labelled 0311aaa by Tierra's internal naming mechanism.

Under a series of simulations where each source of random perturbation was individually disabled, the disabling of the segment deletion perturbation showed an apparent barrier to the emergence of pathological constructors. The conjectured interpretation of this is as follows. When a large segment deletion occurs while copying the genotype from parent to offspring, the resultant creature will typically consists of a functional phenotype, but paired with a short, partial, genotype. This creature continues to produce offspring rapidly, (due to the drastically shortened genotype) but these offspring are usually non-functional as they consist of a short, corrupt, phenotype, encoded by the corrupted genotype.

For von Neumann style reproducers, all perturbations which affects the genotype will result in a constructor which will create at least one offspring which may or may not be functional. Genotypes which experienced a segment deletion will generally result in pathological constructors which can construct many non-functional offspring before being killed by the reaper. Therefore, under the possibility of segment deletions, von Neumann style creatures, which differentiate between genotype and phenotype, regardless of any other details including the specific choice of genotype-phenotype mapping, will be particularly prone to giving rise to pathological constructors. This is not generally the case with non von Neumann reproducers such as self copiers which do not differentiate between genotype and phenotype.

An example can be shown for one specific example where pathological constructors emerge relatively early in the runs. For each Tierra simulation, a pseudorandom number generator is used to generate "random" perturbations. Within the `soup_in` file, the seed to this pseudorandom number generator can be specified by the "seed" parameter. Here the system was inoculated with `vn_lut32_311` with a seed parameter of 50. By 11.5 million CPU cycles (each generation is approximately 1 million CPU cycles) a new population of creatures emerge, which Tierra labels `0035aaa` (Appendix A.6). This creature is the offspring of the pathological constructor `0669aaa` (Appendix A.7), where an instance of this pathological constructor is located at absolute address location zero. By 14.5 million CPU cycles, there are 3672 creatures in the system. 2348 creatures are instances of `0035aaa`, a product of a pathological constructor. 121 creatures are instances of `0034aaa`, the product of another newly emerged pathological constructor. Only 11 instances of `vn_lut32_311` remain. The other 1192 creatures appear to be non self reproducing as their individual population count does not exceed 1. At approximately 16 million CPU cycles, `vn_lut32_311` is driven to extinction. As `vn_lut32_311` was the only strain remaining in the soup which was self reproducing,eventually all remaining creatures are reaped and the ecosystem collapses.

This analysis concludes that the mechanism which results in ecosystem collapse due to pathological constructors appears to depend critically on the inclusion of segment deletion perturbations. This factor results in a propensity for segment deletions to lead to very short genotypes, while still leaving a functioning phenotype.

By contrast, in order for a pathological constructor to emerge from a classic Tier-

ran self copier, relatively much more specific, multiple, coordinated mutations must occur upon very specific locations, which will alter, but not corrupt the reproductive functionality. This suggests that the probability of pathological constructors emerging within a population of von Neumann reproducers in Tierra is much higher than that of a population of self copiers.

The Heterogeneity of the Tierra Memory Space

Previously, experiments within Tierra were under the assumption that the memory space is homogeneous. It was assumed that each location within the memory system was identical so a creature's chances of survival were not affected by its memory location, but only by its reproductive capability and the actions of the other creatures residing within the soup. However, it was discovered that the Tierra platform is actually heterogeneous, and this property can significantly affect the outcome of an evolutionary run, occasionally contributing to ecosystem collapse.

As already described, Tierra incorporates a template addressing mode. Thus, for example, the `jmp` instruction destination is specified by matching complementary sequences of `nops`. However, each memory location is also assigned an absolute underlying, numeric address, and the `call` and `ret` instructions indirectly rely on this absolute addressing mode to position the instruction pointer in memory.

If a CPU executes the `call` instruction, the instruction pointer's current absolute address location is pushed to the stack. When `ret` is executed, the number at the top of the stack is popped to the instruction pointer register, returning the instruction pointer to the corresponding absolute address. Upon birth, the stack is initialised with zeros. It follows that if a malfunctioning creature executes `ret` before pushing a value to the stack, then this will redirect the CPU instruction pointer to the absolute address zero, and execute whatever code is located there (either an active creature or fossil code). This absolute addressing, implicitly employed by `call` and `ret`, introduces a level of heterogeneity to the system, where within an ecosystem which includes a number of malfunctioning creatures, address zero is a preferential location for a functioning creature to exist. Any creature located at address zero can effectively "harvest" CPU's from a significant variety of malfunctioning creatures throughout the soup. In all investigated instances, where ecosystem collapse was observed due to the exploitation of pathological constructors, the creature type dominating the soup immediately prior to collapse was a malfunctioning product of a pathological constructor which is located at address zero. It seems that in order for pathological constructors to result in ecosystem collapse, a relatively large population of pathological constructors may have to exist within the soup simultaneously, specifically including a pathological constructor located at address zero. It is apparent that the cumulative effects of multiple malfunctioning creatures throughout the soup, relocating their CPU to address zero, may indeed result in ecosystem collapse.

Furthermore, rare instances have been observed, where a pathological constructor (`0669aaa`) creates offspring (`0035aaa`, Appendix A.6) which immediately executes the `ret` instruction. When such a pathological constructor is positioned at location zero, the CPU of each offspring it creates will immediately return to the parent at address zero, resulting in exponential growth of the number of CPUs executing the pathological constructor's code, and thus exponential growth in the number of malfunctioning offspring created. This is in contrast to a pathological constructor positioned at any other location, where the number of malfunctioning offspring would grow only linearly in time. A single instance of such a pathological constructor, located at address zero, has been shown to have the effect of completely exhausting the system of resources, resulting in catastrophic ecosystem collapse.

6.2.4 The Emergence of Pathological Constructors from Self Copiers

Previous reported work with Tierra has been performed exclusively using populations of self copiers, and the phenomenon of pathological constructors has not been reported explicitly under these circumstances. However, there is evidence that suggests that they may have been observed in previously documented experiments under the standard Tierra distribution.

Simulations experimenting with so called "macro-evolution" were performed by Tierra creator, Ray (1991), where CPU time is allocated to each creature as a function of creature length in order to study the evolutionary dynamics of larger creatures. Thus, creatures of greater length are allocated a greater amount of CPU time. Under these conditions, self reproducing creatures with lengths of up to 10 times that of the standard Tierra ancestor are easily observed. Under these circumstances the following has been recorded:

> Two communities have been observed to die after long periods. In one community, a chaotic period led to a situation where only a few replicating creatures were left in the soup, and *these were producing sterile offspring.* When these last replicating creatures died (presumably from an accumulation of mutations) the community was dead. (Ray, 1991, p. 16 emphasis added).

The direct origin and function of these creatures which caused ecosystem collapse were not investigated and it was simply assumed that "Under these circumstances it is probably difficult for any genotype to breed true, and the genotypes may simply have 'melted"' (Ray, 1991). However, in retrospect, this description of ecosystem collapse correlates at least partially to our observations of ecosystem collapse following the emergence of pathological constructors.

These experiments by Ray were performed with creatures which were initially 80 symbols long, and the minimum creature size allowed was 12. If a creature of length 80

reproduces on average once in its lifetime, then a similar sized creature with the same lifespan should only be able to create at most 6 full malfunctioning offspring of length 12 before it is reaped. However a population where the average creature size is 10 times longer potentially allows for 66 malfunctioning offspring of length 12 to be created before the parent creature is reaped. Without access to the original experimental data, it is possible to speculate that this increase in creature length and reproduction time may be one of the reasons why behaviour characteristic of pathological constructors was observed here, but not on previous experiments. Moreover, the greater number of potential malfunctioning offsprings possible will allow more time for a creature positioned at location zero to harvest the CPU's of malfunctioning creatures and eventually trigger ecosystem collapse before it is reaped.

6.2.5 Discussion

In order to study the evolutionary trajectory of the genotype-phenotype mapping of von Neumann style self reproducers in Tierra, it is important that we address the problem of pathological constructors.

Interestingly enough, in the course of a number of experimental runs natural selection occasionally provided us with a new class of creatures, represented here by an examplar denoted `vn_lut32_413`, which seemed to be less vulnerable to generating pathological constructors, and also would drive the family of `vn_lut32_311` to extinction. This creature's memory image is found in Appendix A.3. This showed a distinctive method of increasing the mutational robustness of a population of von Neumann style reproducers, by effectively causing the abortion of pathological constructors before they are assigned a CPU. This occurred from a single point mutation to a source template, within the genotype inspection subroutine. Prior to reproduction, `vn_lut32_311` measures the length of its genotype, and then doubles this figure in order to calculate the size of the offspring memory image which will host both genotype and its described phenotype. This technique is functional as the chosen genotype-phenotype mapping has a 1:1 mapping between symbols in the genotype and symbols in the phenotype. It follows that the genotype and phenotype are of equal lengths.

A mutation affecting a source template within this subroutine causes the creature to calculate its genotype as being 50 symbols longer than it actually is. The resulting offspring, `vn_lut32_413`, is now approximately 33% longer, increasing the overall creature length from 311 symbols to 413 symbols. The creature proceeds to successfully construct an offspring phenotype and genotype as before; however the final 100 symbols are not actually written into the offspring segment by the parent. They simply contain unused data, fossil remains of creatures that have previously been reaped. This mutation ensures that regardless of the length of the genotype, the offspring will always contain 100 symbols of unused, essentially "junk" data.

It is stated in the Tierra documentation (Ray et al., 2000, Tierra Manual,

p. 17), that when a creature attempts to divide, a configurable `soup_in` parameter *MovPropThrDiv* is checked. If the parent creature has written less than *MovPropThrDiv* times the size of mother's memory image into the daughter's memory image, then the division will abort. The default *MovPropThrDiv* value of .75 (as used in current experiments) means that the mother must write data to at least 75% of the daughter's memory image. The reason this parameter exists is that without it, mutant creatures may allocate space for an offspring, and immediately execute the `divide` instruction without copying any data at all to the offspring, hence rapidly creating non-functioning, sterile offspring, which may eventually lead to ecosystem collapse, i.e., an extreme form of pathological construction. This problem was evidently recognised and addressed by implementing a defensive feature, where in order for a creature to be assigned a CPU upon birth, it must have at least 75% of its data written to it directly from the parent. If over 25% of an offspring's memory image does not contain data written to it directly by its parent, then the division will be aborted, the parent loses writing permission to the memory space and the space will not be assigned a CPU.

By ensuring that an offspring "must" contain 100 unused symbols which were not written directly by its parent, `vn_lut32_413`, ensures that any mutated offspring which experiences a segment deletion to the genotype, resulting in an offspring with less than 400 symbols, will not progress past the fetal stage, as less than 75% of its memory image was written to by `vn_lut32_413`. In summary, an offspring of `vn_lut32_413` whom experiences a segment deletion to the genotype, will only be allocated a CPU if the segment deletions is 13 symbols or less. However, a segment deletion of this size is generally not large enough to constitute an effective pathological constructor.

This mutated creature, `vn_lut32_413` has the same gestation time as its predecessor, `vn_lut32_311`, as no CPU time is spent copying the extra 100 symbols. However it essentially cannot give rise to pathological constructors once its lineage appears in the soup, less CPU time is "wasted" on pathological constructors and their malfunctioning offspring, and more CPU time is allocated to the self reproducing creatures. Consistently, `vn_lut32_413` comes to dominate the soup, driving the initial von Neumann ancestor to extinction.

A set of experiments were run, where a soup was inoculated with both `vn_lut32_311` and `vn_lut32_413`. The experiment was run 100 times with all perturbations are switched off, until one of the strains went extinct. Neither strain appeared to have a consistent advantage over the other, and each strain displaced the other approximately 50% of the time. However, with perturbations switched on, `vn_lut32_311` was consistently driven to extinction. 100 more simulations were run where perturbations were switched off and the soup was inoculated with `vn_lut32_311`. Tierra was run until the soup was at maximum capacity with instances of `vn_lut32_311`. A single instance of `vn_lut32_413` was introduced to the soup and perturbations were switched on. Even under these circumstances, where the population of `vn_lut32_311` greatly outnumbers the population of `vn_lut32_413`, in 76% of the runs the strain `vn_lut32_311` was still

driven to extinction. These experiments demonstrates that both `vn_lut32_311` and `vn_lut32_413` are equally fit when mutations are not a factor, however, in the presence of mutations, (specifically the segment deletion) `vn_lut32_413` has a distinctly higher fitness and even in a case where a soup is entirely populated by `vn_lut32_311`, if just one instance of `vn_lut32_413` emerges, there is a high probability that it will drive `vn_lut32_311` to extinction.

6.3 Experimental Procedure, Results and Discussions II

6.3.1 Modifications to the Tierra source code and configuration file

The previous section described the phenomenon of pathological constructors and the emergence of the `vn_lut32_413` lineage of mutants which prevented the emergence of such a phenomenon. However, this in itself does not provide an adequate strategy for resuming the investigation of genotype-phenotype mapping evolution for the following reason. When a `vn_lut32_413` style creature reproduces successfully, its offspring, though functionally identical, will not in general be identical in memory image to the parent. This is because the 100 extra unused symbols contain the fossil remains of the creature which previously resided in that position in memory. After a successful reproduction event, as the offspring may not contain the identical symbol sequence as the parent, the Tierra system will classify this as a different strain and it will be assigned a different label than its parent. This will make it very difficult at best, to track the lineages of such self reproducing creatures. Furthermore, the space of possible names for new creature strains will be rapidly exhausted. Tierra uses a three letter label as an identifier for each distinct strain of creature at any given length, eg., "aaa" in `0311aaa`. This allows for 26^3, or 17576 distinctive labels to be saved to the genebank for each length[5]. As every reproduction event of a creature such as `vn_lut32_413` may cause a new identifier to be allocated, the name space will be exhausted rapidly. Therefore, in order to prevent ecosystem collapse by pathological constructors, a number of modifications were made, both to the Tierra source code and to Tierra's configurable input parameters.

The previously mentioned flaw with the `ret` instructions led to a heterogeneous memory space, and specifically allowed effectively exponential growth by a single pathological constructor. A correctly functioning creature should only execute the `ret` instruction after executing a corresponding `call` instruction which pushes the next instruction pointer memory location to the stack. Now, although the soup is designed to be a circular memory system, the virtual computer is actually built upon an underlying linear memory system. A Tierran CPU may execute freely across the boundary between the end and the start of the soup to implement a circular memory space. Thus, the only legitimate reason why a `ret` to address zero should be executed is if a `call` instruction was situated at the last memory location in the soup, causing the subsequent memory address, address zero, to be pushed to the stack. However, the memory allocation instruction `mal` does not actually treat the soup as a single circular memory system, but rather as as a single linear memory space, so cannot allocate a single block of memory that crosses over the start/end border. As it is impossible for a single creature to be situated in such a way as to span this border, there are no legitimate reasons why a properly functioning creature should execute a `ret` to zero. Accordingly, the

[5]Due to a flaw in the Tierra code, non-alphabetic, and also non-printable ASCII characters are also used. This is addressed in more detail in Chapter 7.

`ret` instruction was modified to address this problem. Any instance where a creature's CPU attempts to jump to address location zero following a `ret` instruction is treated like a `nop`; the CPU does not execute anything and the instruction pointer is simply incremented to the next location in memory.

Furthermore, another flaw was also found in the Tierra source code which resulted in the instruction pointer jumping to address zero via a flaw in the `jmp` instruction. The `jmp` instruction causes the instruction pointer to jump to a location in memory determined by the subsequent `nop` sequence. However, if a malfunctioning creature has a `jmp` instruction that is not directly followed by a `nop` sequence, then the instruction pointer will jump to the address location, pointed at by the Bx register. As the initial state of the registers is zero, if a malfunctioning creature executes a `jmp` without a template address, before any data is pushed to Bx, then the instruction pointer will jump to zero. This idiosyncratic function is clearly a flaw in the code, and was fixed so that a `jmp` instruction simply behaves like a `nop` if it is not followed by a template address.

These modifications were effective in preventing ecosystem collapse by pathological constructors, and allowed experiments in which much more extended evolutionary trajectories could be simulated and analysed.

Although the change to the `ret` and `jmp` instructions prevented ecosystem collapse, pathological constructors can still easily arise within the soup and can still represent a significant drain on CPU time, so further changes were made in an attempt to further reduce the incidence of emergence of pathological constructors. This would allow the evolution of the genotype-phenotype mapping to be better investigated without this pathological constructor overhead.

Further changes were made to the `soup_in` configuration file which contains values for the observational parameters and environmental variables that control each specific run. All random perturbations with the exception of the single point perturbation and the copy perturbation were disabled. As the segment delete perturbation was primarily responsible for the emergence of pathological constructors, removal of this perturbation greatly reduced the number of sterile creatures existing and consuming resources in any given run.

In an attempt to simplify data analysis, the `soup_in` configuration file was further set so that only creatures of the same size as the seed ancestor could be born. This was achieved by switching on the *DivSameSiz* parameter in the configuration files which ensures the when a `divide` is executed, a CPU is only allocated to the new memory image if it is exactly the same size as the parent. This meant that for every self reproducing creature to appear within the soup, the relative positions of all their subroutines would typically be identical. This allowed for tools to be developed which automated the analysis of the thousands of strains which emerge within an evolutionary run. The look-up table (if present) would start and finish in the exact same location within each creature, so data analysis tools could be written which easily located the look-up table

and compared it with that of the initial ancestor to see if it has changed. It must be noted that mutations to the look-up table are not the only potential mechanism which can affect the genotype-phenotype mapping. It is possible that alternative mutations to the general constructor may affect the decoding mechanism and alter the mapping. However, as the size of the creatures are fixed, and a changes of this nature may require multiple, coordinated, simultaneous mutations, mutations of this nature are neglected in the current analysis, and data analysis tools were written focussing specifically on changes to the look-up table as a mechanism which can result in a change to the genotype-phenotype mapping. It must also be noted that restricting the size of potential offspring may also, in turn, reduce the evolutionary search space for potential offspring. Evolutionary trends which are common in Tierra such as reduction of creature size and parasitism will be eliminated with this change in the configuration file. However, these phenomena have already been studied and documented in detail, (Ray, 1991), so are of limited concern here. This investigation aims to provide proof-of-principle examples where evolution results in a change in the genotype-phenotype mapping, so the evolutionary search space of single point perturbations was chosen to be investigated as a starting point.

The redesigned creature, `vn_lut64_316`, its `opcode.map` file and `soup_in` file are located at Appendix A.4 ,Appendix B.3 and Appendix C.3 respectively.

6.3.2 Redesigning the von Neumann ancestor and introducing redundancy; vn_lut64_316

The redesigned ancestor `vn_lut64_316` used a minimum of 28 distinct p-symbols in order to self reproduce. However, the mRNA-amino acid genetic code of natural biology consists of a genotype space of 64 different codons and a significantly smaller phenotype space of 22 amino acids, plus reserved start and stop codons. For the next set of experiments, in an attempt to mirror the redundancy of the genetic code and introduce a greater potential for an evolution of the genotype-phenotype mapping, a genotype and phenotype space of 64 was implemented rather than 32. This corresponds to 64 distinct g-symbols and 64 distinct p-symbols, only 28 of which have an active function relied on for self reproduction, and 36 of which have no active function at all and are not employed in order to self reproduce. To facilitate this, 36 extra `nops` were introduced to the instruction set. This mapping is presented in the `opcode.map` file in Appendix B.3, (see also Figure 6.2.)

6.3.3 Experimental Procedure

The Tierran soup was inoculated with `vn_lut64_316`. Point perturbations which affect random memory locations throughout the soup (cosmic rays) and perturbations which occur exclusively to symbols that are being written to memory locations in the soup (copy perturbations) were enabled and the system was run for 100 billion CPU

UUU	(Phe/F)	UCU	(Ser/S)	UAU	(Tyr/Y)	UGU	(Cyc/C)
UUC		UCC		UAC		UGC	
UUA	(Leu/L)	UCA		UAA	Stop(Ochre)	UGA	Stop (Opal)
UUG		UCG		UAG	Stop(Amber)	UGG	(Trp/W)
CUU		CCU	(Pro/P)	CAU	(His/H)	CGU	(Arg/R)
CUC		CCC		CAC		CGC	
CUA		CCA		CAA	(Gln/Q)	CGA	
CUG		CCG		CAG		CGG	
AUU	(Ile/I)	ACU	(Thr/T)	AAU	(Asn/N)	AGU	(Ser/S)
AUC		ACC		AAC		AGC	
AUA		ACA		AAA	(Lys/K)	AGA	(Arg/R)
AUG	(Met/M)	ACG		AAG		AGG	
GUU	(Val/V)	GCU	(Ala/A)	GAU	(Asp/D)	GGU	(Gly/G)
GUC		GCC		GAC		GGC	
GUA		GCA		GAA	(Glu/E)	GGA	
GUG		GCG		GAG		GGG	

000000	0 (nop0)	010000	16 (dec)	100000	32 (popC)	110000	48 (movAb)
000001	1 (nop1)	010001	17 (nop10)	100001	33 (nop19)	110001	49 (nop29)
000010	2 (nop2)	010010	18 (incD)	100010	34 (popD)	110010	50 (movda)
000011	3 (nop3)	010011	19(nop11)	100011	35 (nop20)	110011	51 (nop30)
000100	4 (ifnz)	010100	20 (pushA)	100100	36 (popE)	110100	52 (movAb)
000101	5 (nop4)	010101	21 (nop12)	100101	37 (nop21)	110101	53 (nop31)
000110	6 (addAAE)	010110	22 (nop13)	100110	40 (nop22)	110110	54 (ret)
000111	7 (nop5)	010111	23 (nop14)	100111	41 (nop23)	110111	55 (nop32)
001000	8 (subCAB)	011000	24 (pushC)	101000	42 (jmpb)	111000	56 (nop33)
001001	9 (nop6)	011001	25 (nop15)	101001	43 (nop24)	111001	57 (shlA)
001010	10 (subAAC)	011010	26 (pushD)	101010	44 (adrf)	111010	58 (nop34)
001011	11 (nop7)	011011	27 (nop16)	101011	45 (nop25)	111011	59 (nop35)
001100	12 (incA)	011100	28 (popA)	101100	46 (nop26)	111100	60 (mal)
001101	13 (nop8)	011101	29 (nop17)	101101	47 (nop27)	111101	61 (nop36)
001110	14 (incB)	011110	30 (popB)	101110	48 (cal)	111110	62 (nop37)
001111	15 (nop9)	011111	31 nop18)	101111	49 (nop28)	111111	63 (divide)

Figure 6.2: The upper figure presents the mapping from mRNA to amino acid. The lower figure presents the initial mapping from g-symbols to p-symbols, implemented with vn_lut64_316. Orange symbols highlight those which are non-employed and grey highlights symbols which initially map onto non-employed p-symbols.

cycles, which corresponded to approximately 250 thousand generations[6]. Each distinct creature to emerge throughout the run was captured and the number of employed and non-employed p-symbols within the look-up table for each creature was counted. Employed p-symbols refer to those which have a functional role in the process of reproduction, while non-employed p-symbols are included to introduce a potential for redundancy in the genotype-phenotype mapping and do not actively contribute towards the reproduction process. If a specific p-symbol exists in the look-up table, then there must exist a specific g-symbol which maps onto it, otherwise construction of an offspring look-up table would not be possible. If a p-symbol is absent from the look-up table then it is lost from the genotype-phenotype mapping. With our current mapping system, a substitution mapping, it is impossible for a p-symbol which is absent from a parent's look-up table to be included in its offspring's phenotype (with the exception, of course, of random phenotypic perturbations introducing random p-symbols to an offspring).

The population of non-employed p-symbols in the look-up table of each newly emerging strains was then plotted against the time of emergence of that strain, and this process was repeated 4 times. This result can be seen in Figure 6.3.

6.3.4 Results

Standard evolutionary Behaviour

The first set of experiments showed evolutionary behaviour qualitatively similar to that documented in Ray's initial experiments (Ray, 1991). Informational parasitism[7] quickly emerged due to segment deletions resulting in the description of the look-up table being omitted from the genotype. The resulting creature will redirect its CPU to a neighbouring host to facilitate the construction of its phenotype and therefore expend less CPU time per reproduction cycle due to its reduced length. Another evolutionary phenomenon typical of Ray's experiments is the reduction of creature size by reducing template addresses where possible. While the programmer creating the creature may use an initial template size of four `nop` instructions, evolution will typically reduce the template size where ever possible, creating shorter and more efficient offspring. For further experiments, measures were taken to eliminate the distraction of these phenomena, such as only allowing point perturbations and only allowing offspring of a specific length to be created.

[6] A generation in Tierra is a calculated time interval, which is determined by a rolling average of the number of CPU cycles required for each creature present in the soup to reproduce once and die.

[7] Informational parasitism refers to a form of parasitism which accesses and reads a host's memory contents, but does not directly interfere with its functionality.

Evolution of the genotype-phenotype mapping

The aforementioned evolutionary behaviours have already been studied and documented, and therefore are not of primary concern, so for the remaining experiments the system parameters were configured so that the creature size cannot change. This will prevent the distraction of the discussed known phenomena occurring and allow us to focus on the specific evolutionary changes which arise as a direct result of a change in the genotype-phenotype mapping. Specifically, we will focus on changes affecting the look-up table.

Initially, non-fatal inheritable silent perturbations of the genotype can occur in the description the look-up table. A "silent" perturbation is here defined as one which alters the genotypic sequence but does not affect the functioning of the phenotype. This alters the genotype-phenotype mapping and allows previously silent g-symbols (by silent g-symbols, we refer to g-symbols which were initially mapped onto non-employed p-symbols) to be mapped onto employed p-symbols. This allows single employed p-symbols to be mapped onto by multiple g-symbols.

The initial ancestor, `vn_lut64_316`, has 36 silent g-symbols, which are mapped onto 36 different non-employed p-symbols. As neither the silent g-symbols nor the non-employed p-symbols functionally contribute to the reproduction of offspring, the silent mutations that affect which p-symbol the silent g-symbols are mapped onto are random and arbitrary. However it was found that there was a strong bias towards the mapping of silent g-symbols onto employed p-symbols. During an evolutionary run, we see a sharp decrease in the number of non-employed p-symbols within the look-up tables of newly emerging strains. Eventually, all 36 non-employed p-symbols are eliminated from the creatures within the soup, and the 64 positions in the look-up tables will consist entirely of employed p-symbols.

6.3.5 Discussion

In this particular "toy" model, the evolution of the genotype-phenotype mapping is initially driven predominantly by the underlying dynamics of the coding system. The nature of the substitution mapping mechanism employed means that certain perturbations of the look-up table are not directly reversible. This results in a systematic evolutionary change of the genotype-phenotype mapping, eventually eliminating all non-employed p-symbols from the phenotype by ensuring that they are not mapped onto by any elements of the genotype space.

Figure 6.4 demonstrates a small section of the look-up table and its description. By studying a creature's look-up table one can deduce the genotype-phenotype mapping that is implemented by that creature. For this small section of the mapping between the g-symbols and p-symbols, we see a set of four g-symbols, `0`, `1`, `2` and `3`, which are mapped onto four p-symbols which may be interpreted as numerical data or, as the instructions `nop0`, `nop1`, `nop2` and `nop3` respectively. The red symbols within the look-up table

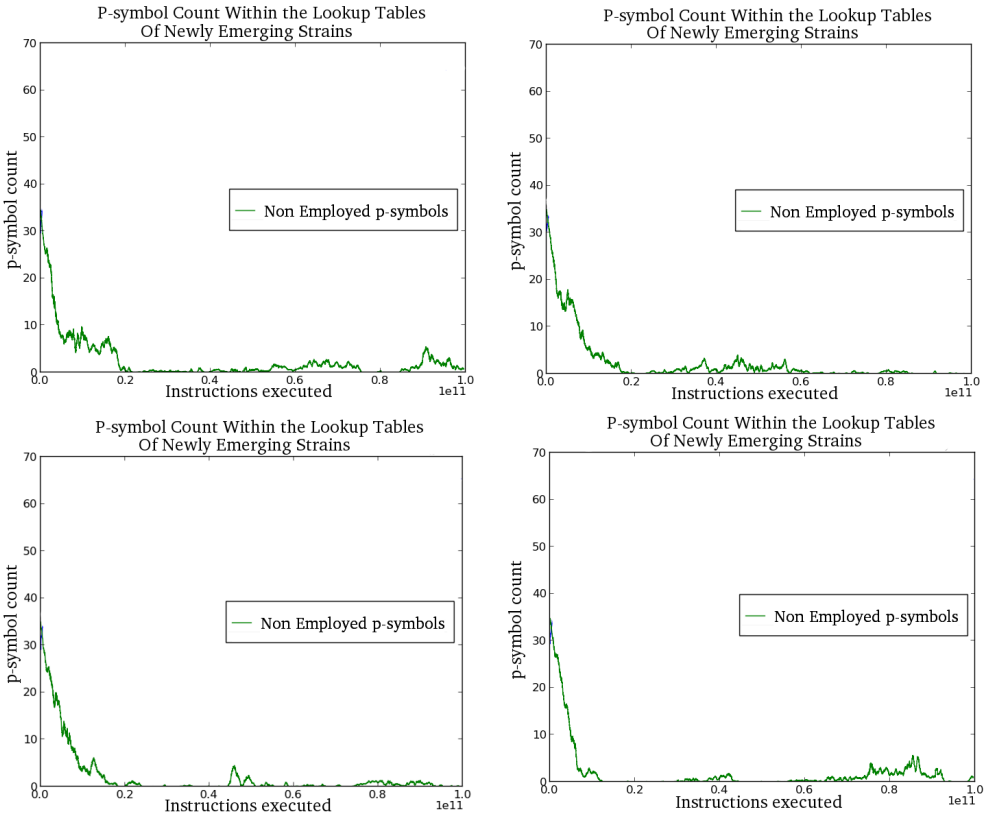

Figure 6.3: Four evolutionary simulations displaying the number of non-employed p-symbols present in the look-up table of strains of newly emerging lineages. Out of a maximum look-up table symbol size of 64, the initial creature has 36 distinct non-employed symbols which quickly are lost from the look-up tables of newly emerging creatures.

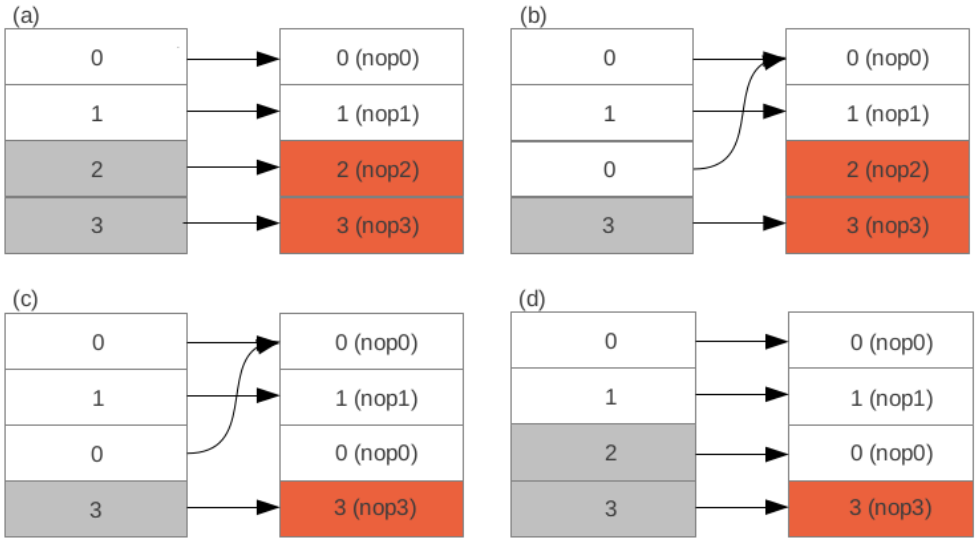

Figure 6.4: Schematic diagram to demonstrate how specific perturbations to a look-up table's description may lead to the loss of a symbol mapping when expressed in the phenotype. The left hand column represents a section of the genotype encoding of a look-up table, and the right hand column represents the corresponding look-up table. (a), (b), (c), and (d) represents the subsequent generations of a creature after it experienced a mutation.

represent non-employed p-symbols, while the grey symbols within the look-up table's genetic encoding represent the g-symbols which initially map onto a non-employed p-symbol. Figure 6.4(a) demonstrates the initial 1st generation ancestor's look-up table and description. We can see here that the initial mapping which was chosen is both injective and surjective (bijective), as each element of the genotype space is mapped onto a different element of the phenotype space. This mapping is also invertible, as it is possible to determine a unique genotypic sequence corresponding to any arbitrary phenotype. This can be denoted by saying that $P \triangleq \phi(G)$ and $G \triangleq \psi(P)$ where $\psi() = \phi^{-1}()$.

Figure 6.4(b) represents an offspring which experienced a mutation to the 3rd position in the look-up table's description, changing symbol `2` to `0`. For this particular implementation of von Neumann reproduction, there is normally a one-generation delay between when a perturbation occurs in a genotype and when the perturbation is expressed in the phenotype. When this 2nd generation creature attempts to reproduce, it must first copy its exact genotype to the 3rd generation offspring, Figure 6.4(c). The 2nd generation creature must then decode its own genotype, and construct the 3rd generation creature's phenotype. However, under construction of the phenotype, when

decoding the 3rd symbol in the look-up table description, the employed p-symbol `0`, (`nop0`), is written to the third position in the look-up table, and the previous non-employed p-symbol `2` (`nop2`), is therefore lost from the genotype-phenotype mapping.

Even if the perturbed genomic position in the look-up table *description* gets perturbed back to the previous state, Figure 6.4(d), the non-employed p-symbol cannot be re-instated to the phenotype. This is because the genotype-phenotype mapping has been changed, and now the silent g-symbol, which initially was mapped onto a non-employed p-symbol, `2` (`nop2`), is now mapped onto an employed p-symbol, `0` (`nop0`).

We also see that the mapping is now non-injective and non-surjective, as an element of the phenotype alphabet, `0` (`nop0`), is mapped onto by more than one element of the genotype alphabet, `0` and `2`. Furthermore, another element of the phenotype alphabet, `2` (`nop2`), is not mapped onto by any element of the genotype space. This renders the mapping non-invertible, as it is now impossible to determine a unique genotypic sequence corresponding to any arbitrary phenotype as $\psi() \neq \phi^{-1}()$ as now $G = \psi^*(P)$.

The only mechanism in which this mutation (which completely removes a p-symbol from the look-up table) can be reversed, is via a genotypic perturbation, which returns the look-up table description to its previous state, followed by a phenotypic perturbation, which directly introduces the lost p-symbol to the look-up table. Due to the ease with which a non-employed p-symbol can be lost, and the level of difficulty required to re-introduce the non-employed p-symbol to the mapping, there is a strong immediate bias present which quickly eliminates all non-employed p-symbols from the look-up table.

This feature of the implemented mapping system does incidentally demonstrate a particular mechanism whereby *phenotypic* perturbations may be inheritable under the circumstance that the perturbation affects the function $\psi()$. If we have a machine $X = (A'+B+C+D)+\phi(A+B+C+D)$, where A' represents a general constructor with a changed genotype-phenotype mapping $\psi^{'}()$, then this perturbation will be inherited to the offspring phenotype only if $\psi^{'}(G) = P'$, where $G = \phi(A+B+C+D)$ and $P' = (A'+B+C+D)$. In other words, this demonstrates an instance of Lamarkian inheritance, where a perturbation of the phenotype is passed down to future generations without any change to the genotype. However, in order for this to occur, the perturbation must affect the component of the phenotype which decodes the genotype, such that that same genotype will now be decoded to give rise to the new perturbed phenotype. In general, this would be an extremely unlikely (yet not quite impossible) combination of circumstances, and may relate to the points in Figure 6.3 in which we see brief periods where new strains emerge, which include non-employed symbols within the lookup table.

Mutational robustness and Darwinian selection.

In such a situation where redundancy is introduced to the look-up table and many g-symbols map onto the same p-symbol, Darwinian selection may also affect the genotype-phenotype mapping, and create a more mutationally robust genotype. In particular, the allocation in which silent g-symbols are mapped onto employed p-symbols may be subject to Darwinian selection. Following a perturbation to a g-symbol, a phenotype may still preserve form if both g-symbols maps to the same p-symbol. Conversely, as long as there is no redundancy, then the mapping cannot incorporate any inherent mechanism to ensure stability to perturbations and help the phenotype preserve its form under inheritable variation. Every p-symbol will have the same robustness to mutation, no matter how frequently its description occurs in the genotype, or how imperative it is to the correct operation of the phenotype.

For these experiments, there are only have 28 employed p-symbols, but 64 g-symbols. Darwinian selection may select how the silent g-symbols are mapped upon the employed p-symbols. A p-symbol which is very common within the phenotype, has a higher probability of having some of its description perturbed within the genotype. If a large percentage of the silent g-symbols are mapped upon the most frequent, employed p-symbols, then the phenotype will have an increased probability of remaining unchanged following an inheritable perturbation to the genotype.

To test this hypothesis, a creature was engineered with a non-surjective genotype-phenotype mapping. All silent g-symbols were mapped onto the employed p-symbol, `0` (`nop0`). `0` is very frequent throughout the phenotype, as it is used for template addressing. The mutational robustness of `0`'s description has now greatly increased, as there are 36 possible genotypic perturbations which will still allow the phenotype to remain unchanged. The Tierra soup was inoculated with two von Neumann self reproducers, the original ancestor with a surjective genotype-phenotype mapping, and the engineered ancestor with the non-surjective genotype-phenotype mapping. The two creatures used distinctly different start address templates, so that the descendants of each ancestor could be distinguished from each other. This simulation was run for 100 billion instructions and the experiment was repeated 100 times with only point perturbations activated. It was found that in 76 instances the initial ancestor was driven to extinction, while in only 24 instances the engineered ancestor with the non-surjective genotype-phenotype mapping was driven to extinction. If both creatures were equally fit you should expect that drift will eventually cause either of the creatures to go extinct with an equal probability. These preliminary tests show that there may be a selective advantage for distributing the silent g-symbols amongst the most frequently occurring employed p-symbols, and therefore room for Darwinian selection to guide the evolution of the genotype-phenotype mapping.

6.4 Experimental Procedure, Results and Discussions III

6.4.1 Alternative Implementation of the Genotype-Phenotype Mapping

For the next set of experiments, a different implementation of the same genotype-phenotype mapping was designed in order to compare and contrast different systems to explore further the scope of phenomena which may arise when implementing a mutable genotype-phenotype mapping.

The previously implemented genotype-phenotype mapping mechanism which included a look-up table relies on a *indexed addressing* system where a symbol in the genotype can be interpreted as numerical data which is used to direct a pointer to a specific location within the look-up table. The look-up table address is specified by indicating its distance from another address, the *base address*, which is the first location in the look-up table. The symbol within that specific memory address is then accessed and used as the output from the mapping.

In order to design an alternative mechanism for implementing the genotype-phenotype mapping, a system which utilises *associative memory* may be implemented.

For the look-up table mechanism, a g-symbol is interpreted as a number, which is used as an index into a "value array". The symbol at this address location is then interpreted as the output to the mapping. For the associative memory mechanism, a "key-value array" or "associative memory" was used in the form of a *translation table*, where a g-symbol is interpreted as a key, and the creature must scan through the list of entries, or key-value pairs within the translation table to find an entry where the key-symbol is identical to the inspected symbol in the genotype. The associated value-symbols will be accessed and returned as the output to the mapping.

The translation table therefore consists of two columns, representing of all the possible symbols which can be used as inputs to the mapping (keys), and all the possible symbols which can be returned as outputs (key-values). Each symbol within the input column will map onto a single symbol within the second column. The biological motivation for the implementation of a translation table is an attempt to mimic tRNA, which acts as the physical link between a nucleotide sequence (genotype) and an amino acid sequence of proteins (phenotype) according to the genetic code, by specifying which sequence of nucleotides correspond to which amino acid.

6.4.2 Comparing and Contrasting the Different Mapping Implementations

Disadvantages of the Translation Table

The processing speed of performing a genotype-phenotype mapping via a translation table will be significantly slower than that with a look-up table. In order to translate a g-symbol to a p-symbol, the creature must potentially scan through the entire translation

table, and at row in the table perform a check to see if the input g-symbol is identical to key symbol within that row.

If the input g-symbol does not match the key in any row then the mapping will fail and result in an indeterministic offspring; unless this is tested and handled in some deterministic way. However due to the limited functionality of programming within Tierra, adding this feature would be quite complicated to implement, and further increase the gestation time of each creature, so was ignored for the purposes of these experiments. If a creature does not find a matching key-value within the translation table, it will continue to scan through memory until it eventually detects a matching symbol elsewhere, and use this symbol as a key and perform a symbol mapping. In doing so, matching symbols which are not located within the creatures translation table but elsewhere within it's memory image may be used. Furthermore, if no matching symbol is detected within the creatures memory image, the search will extend beyond the creature itself, and may find matching symbols within the unallocated memory locations of the soup or within other creatures. This process is not only indeterministic, but potentially causes a very significant increase in gestation time, so it is likely, although not for definitely, that any such creature will have low fitness and be selectively displaced.

Advantages of the Translation Table

Throughout a run, the size of the g-symbol alphabet is fixed, therefore this puts a limit on the minimum possible size of the look-up table. Under evolution, as symbols are lost from the mapping the size of the look-up table might also decrease to reduce the overall creature size and result in a more efficient self reproducer. However, as the look-up table ancestor uses relative addressing to locate the output to a genotype-phenotype mapping, the length of the look-up table must always be large enough to decode for the employed g-symbol with the largest underlying numerical value. For example, if there are only 28 employed g-symbols, but one employed g-symbol has an underlying numerical value of 127, then the look-up table must be a minimum of 127 symbols long in order to contain the relative address 127. The look-up table itself cannot reduce in size any further and will be unnecessarily inefficient if it is constrained to contain many unemployed p-symbols.

By contrast, when using an associative memory approach, the size of the translation table might more easily be reduced as unemployed symbols are removed from the mapping. As only the employed symbols are necessary in the translation table columns, the memory image of the genotype-phenotype mapping routine can reduce in size to only include the table rows necessary for reproduction, increasing the efficiency of the self reproducer. This is a potential advantage to implementing a mapping system which utilized an associative memory rather than a value array (look-up table). It must be noted that it is possible that this "advantage" may be offset by the possibility

of Darwinian selection resulting in a genotype-phenotype mapping where the employed p-symbols are mapped onto by g-symbols with a lower underlying numerical value, therefore allowing a reduction in size of the look-up table.

However, the preliminary experiments to be described here are not directly focusing on analysing the change in fitness of creatures due to a change in the genotype-phenotype mapping, and so to simplify the analysis of these experiments, only offspring of equal size to its parent will actually be allocated CPUs. Under these circumstances, this possible advantage could not occur as any creature whose translation table may have been reduced would be reaped at birth.

6.4.3 The Structure of the Translation Table.

For this set of experiments it was decided to use the maximum memory symbol alphabet size allowed by Tierra, to increase the possible space of potential redundancy in the genotype-phenotype mapping. The underlying 8 bit data words which comprise the Tierran soup allows for a maximum alphabet size of 256 distinct symbols. However, as the data words are numerically typed as signed integers, the numerical range spans from -128 to +127. The use of negative numerical values introduces unnecessary complication to designing and programming the mapping system so a 7 bit alphabet size was used which only incorporated positive numerical values. Each data word can be interpreted as either a g-symbol or a p-symbol so our initial mapping system will require a translation table with 128 rows, containing the complete set of g-symbols in the first column and the complete set of p-symbols in the second column.

The initial mapping will be a permutation mapping where the array of data words which make up the set of g-symbols in the translation table, and the array of data words which make up the set of p-symbols are permutations of each other, so every symbol within the shared alphabet will appear exactly once in each column.

Within the translation table, the first column which contains an ordered set of data words which represent g-symbols will be referred to as "K" (keys) and the second column which contains an ordered set of data words which are interpreted as p-symbols will be referred to as "V" (key-values). The translation table, which is situated within the phenotype can therefore be represented by $[K, V]$. The encoded description of the translation table in the genotype can be represented by $\phi([K, V]) = [\phi(K), \phi(V)]$

6.4.4 The Redesigned Ancestor, vn_tt128_758

By definition, an arbitrary translation table is a table with 2 columns, and (in general) an indefinite number of rows. The initial translation table which was designed for this creature contained 128 rows. Each row contains a key for a specific g-symbol, an an associated value for the corresponding p-symbol. In order to construct the translation table, the table is stored sequentially by row in memory. Therefore the first two memory location in the translation table's memory image contains a key its associated value

respectively. This pattern repeats throughout the translation table until every possible g-symbol and its associated p-symbol under the implemented genotype-phenotype mapping are accounted for. This creature required 28 distinct p-symbols in order to reproduce successfully, therefore, there were initially 100 unemployed p-symbols in the genotype-phenotype mapping.

The redesigned creature, `vn_tt128_758`, its `opcode.map` file and `soup_in` file are located at Appendix A.5, Appendix B.4 and Appendix C.4 respectively and is schematically represented in Figure 6.5.

During reproduction, when `vn_tt128_758` is attempting to decode a g-symbol, it is first copied to a general purpose register. The first cell in the first row in the translation table is then inspected. If the two symbols are not equal then the translation table pointer is incremented by two, and the next key in the translation table inspected. If the two symbols are equal, the translation table pointer is incremented by one, and the corresponding value is interpreted as the p-symbol to be used as the output to the genotype-phenotype mapping. `vn_tt128_758` is significantly longer than the look-up table based ancestor, `vn_lut64_316`, and its rate of reproduction is much slower, taking approximately 75 times more CPU instruction cycles to complete one reproduction event than the `vn_lut64_316` as its decoding process is more expensive on CPU time.

Nonetheless, with all perturbations switched off, `vn_tt128_758` successfully self reproduced and a population of descendent creatures proceeded to fill the soup, demonstrating that it is a functional self reproducer. In an attempt to compare the two different mapping mechanisms, simulations were run in order to detect if this alternative implementation of a substitution mapping also experiences a quasi-deterministic loss of symbols from the genotype-phenotype mapping, which would be indicative again of mutational irreversibility.

This experiment was run for 100 billion CPU instructions, as were the simulations in Figure 6.3. However, this amounted to little over 2200 generations, as opposed to 50,000 generations for the look-up table simulations. Initial experiments showed that approximately 50 out of 100 possible distinct unemployed p-symbols had been removed from the translation table. This was concluded as the symbols were absent from the V column in the translation table, so there were no longer any g-symbols which mapped onto these particular p-symbols. However, it was uncertain whether or not the system had plateaued yet, or if the system simply was not given enough time to plateau. It became clear that due to the significantly slower reproduction rate, and with the available experiment hardware, it was no longer practical to run the experiment long enough for all unemployed p-symbols to be lost from the mapping. In real time, previous experiments were run for up to seven days, however with reproduction rates approximately 75 times slower, this was no longer feasible.

In order to observe the evolutionary trajectory of this new mapping system, it was therefore more practical to first analyse the space of possible heritable perturbations which are possible with such a system, and then manually introduce perturbations

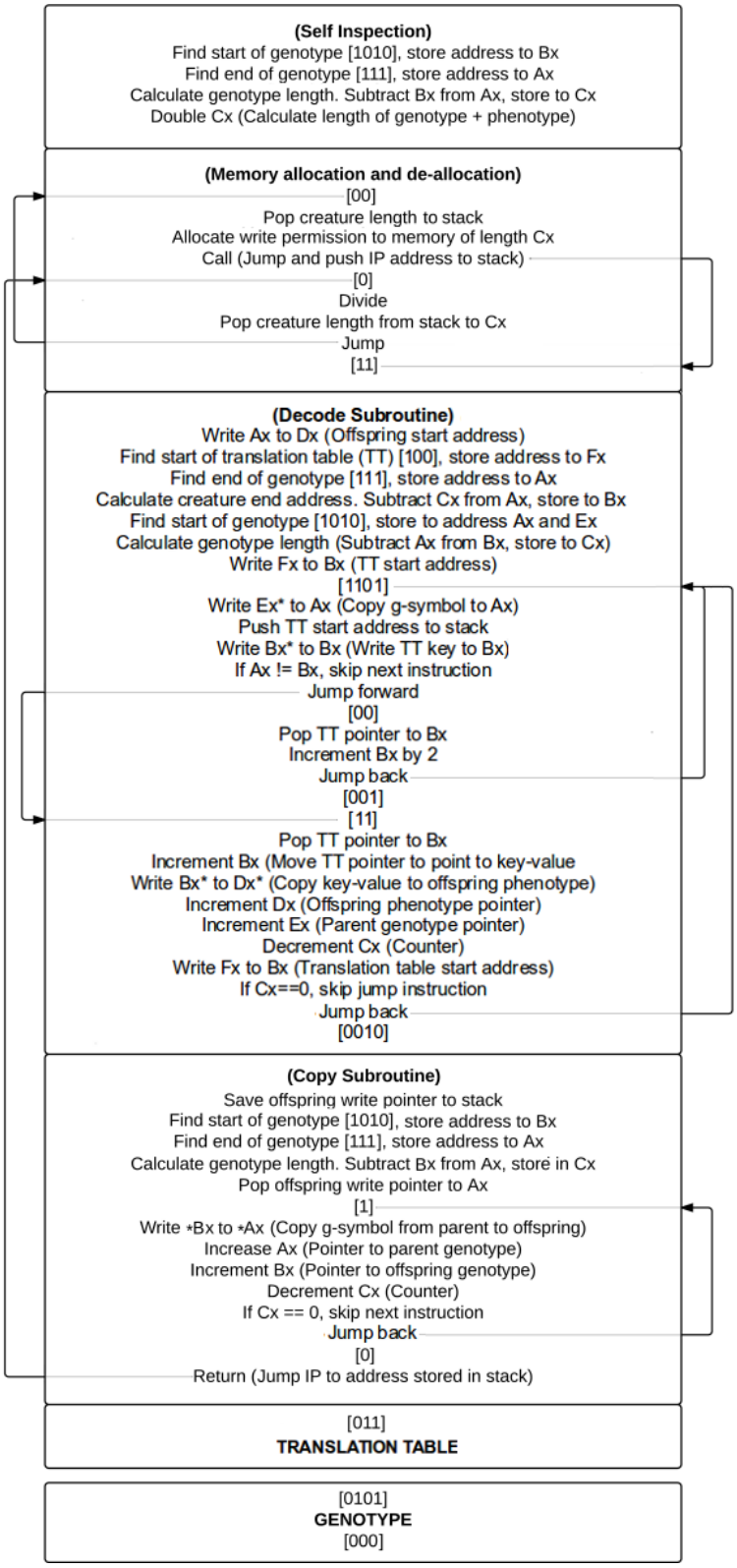

Figure 6.5: This schematic diagram illustrates the structure of the *translation table* based von Neumann style self reproducer which was designed. Template addresses are denoted within square brackets. Registers with an asterisk, represent situations where the data within the register is not interpreted directly as numerical data, but as a pointer to data at address location within the soup.

to observe the expected resultant behaviours, and see is it possible for single point perturbations to result in the change of a genotype-phenotype mapping while still rendering the creature a stable, deterministic self reproducer.

To study the possible heritable perturbations which are possible with such a system, a methodical approach must be taken, where perturbations are applied to the various conceivable configurations in which such a translation table may exist. First the possibility for single point perturbations to result in a loss or introduction of symbol mappings to the translation table is investigated. Secondly, the possibility of a change to the mapping, while keeping the total symbol count constant is investigated

6.4.5 Investigation of Loss or Introduction of Symbol Mappings

Phenotypic Perturbations Affecting The V Column

The effects of single point perturbations on a translation table with an identity mapping was first investigated as it is the simplest configuration that the translation table may occur in. This mapping will be a bijection where every distinct symbol in the genotype space will be mapped to a distinct symbol in the phenotype space. Both symbol sets use the same alphabet of 128 symbols. Whether they act as a p-symbol or g-symbol is determined by how they are interpreted by the general constructor. The genotype-phenotype mapping for this system will be a permutation mapping, where each of the 128 symbols will appear once and only once in both V, and K.

If a phenotypic perturbation affects V, then this will inevitably result in an indeterministic offspring, as demonstrated in Figure 6.6 that represents a fragment of a possible translation table, and of its genomic description. The left column which contains K and V represents the translation table within the phenotype, and the right column represents its description within the genotype. As it implements the identity mapping, each key symbol is mapped onto the identical key-value symbol.

First the effects of perturbing the V column was studied, so the 1 symbol within V was changed to a 2 in the second generation. This altered creature, although retaining the same genotypic description, would execute a different genotype-phenotype mapping as the g-symbol 1 now maps onto the p-symbol 2, so instances of 1 and 2 in the genotype will both result in a 2 in the phenotype of the third generation. This creature has now lost the ability to decode the g-symbol 1, so if 1 appears anywhere in the genotype the creature will malfunction as it no longer possesses the ability to reliably decode this g-symbol. In practice the offspring behaviour will be indeterministic as we cannot predict an output to the g-symbol 1 as it does not exist within the K column of the translation table.

This creature will step through the entire K column searching for the key symbol 1. As this key is not found within K, it will continue to search through the soup until it eventually encounters an indeterminate matching symbol (or until the creature is reaped). If a matching symbol is found, then an indeterminate symbol in the memory

Generation 1

Phenotype		Genotype	
K	V	$\phi(K)$	$\phi(V)$
1	1	1	1
2	2	2	2
3	3	3	3
4	4	4	4

Generation 2

Phenotype		Genotype	
K	V	$\phi(K)$	$\phi(V)$
1	2	1	1
2	2	2	2
3	3	3	3
4	4	4	4

Generation 3

Phenotype		Genotype	
K	V	$\phi(K)$	$\phi(V)$
2	2	1	1
2	2	2	2
3	3	3	3
4	4	4	4

Generation 4

Phenotype		Genotype	
K	V	$\phi(K)$	$\phi(V)$
?	?	1	1
2	2	2	2
3	3	3	3
4	4	4	4

Figure 6.6: Schematic representation of a section the translation table within the phenotype, and the corresponding description within the genotype for a lineage over four generations where a perturbation was introduced to the V column in the second generation. Highlighted symbols represent those which are different to that of the previous generation.

location directly following this symbol will be copied to the offspring phenotype. This not only drastically decreases the reproduction speed of such a creature, but results in a creature with an indeterminate genotype-phenotype mapping (indeterminate offspring strain). Thus, for this architecture, there are no cases of a single point perturbation that affects the V column of a permutation mapping with no redundancy, that will result in a deterministic self reproducer.

Consequently, every distinct symbol must appear twice in the translation table, and also twice in its description $\phi(K)$ and $\phi(V)$. For this translation table, every g-symbol available to the mapping must be located within the column K. However, K is located within the phenotype, therefore if a distinct symbol is removed entirely from the phenotype, then that symbol cannot be interpreted as a g-symbol. If a distinct symbol is removed from the phenotype alphabet, and therefore removed from the mapping, every instance of it must also be removed from the genotype to achieve a deterministic self reproducer. Therefore, every distinct symbol must appear exactly once in both columns V and K.

If a symbol is lost from the mapping via a single phenotypic perturbation, then that symbol will still exist in the genotype. It is impossible to remove a symbol from the mapping of such a system and result in a stable, deterministic self reproducer without also removing every instance of that symbol from the genotype. Therefore at least 2 more point perturbations to the genotype would be needed to remove the symbol from both $\phi(K)$ and $\phi(V)$.

Phenotypic Perturbations Affecting the K Column

For the next case, the effects of introducing a phenotypic point perturbation which only affects the V column of an identity mapping was investigated, Figure 6.7

The K column is perturbed in the second generation so that the symbol with an underlying numerical value of 1 is perturbed to 2. This perturbation results in the loss of the key-value pair for the g-symbol 1. When the creature attempts to reproduce, it is not able to decode the g-symbol 1 and an indeterministic key-value pair is produced in the third generation. Furthermore, we note that the g-symbol 2 appears to have two distinct possible symbol mappings in the second generation; it may be mapped onto 1 or 2 as it is located both in the first and second rows of the translation table. However, with this specific implementation, the first matching key will be selected as the active symbol mapping and any following symbol mappings which include the same key will not be accessed or used. When this creature attempts to reproduce (assuming that the indeterministic key-value pair has not rendered the creature sterile or affected any other key-value pairs), the g-symbol 1 cannot be decoded, and furthermore, the g-symbol 2 (rather than 1, as previously) will be decoded to the symbol 1.

If the third generation reproduces, the key 1 is now re-introduced to the mapping of the fourth generation, however the key 2 is lost, so this creature in turn will not be a

Generation 2

Phenotype		Genotype	
K	V	$\phi(K)$	$\phi(V)$
2	1	1	1
2	2	2	2
3	3	3	3
4	4	4	4

Generation 3

Phenotype		Genotype	
K	V	$\phi(K)$	$\phi(V)$
?	?	1	1
1	1	2	2
3	3	3	3
4	4	4	4

Generation 4

Phenotype		Genotype	
K	V	$\phi(K)$	$\phi(V)$
1	1	1	1
?	?	2	2
3	3	3	3
4	4	4	4

Figure 6.7: Schematic representation of a section the translation table within the phenotype, and the corresponding description within the genotype for a lineage over three generations, where a perturbation was introduced to the K column in the second generation. Highlighted symbols represent those which are different to that of the previous generation.

Generation 2

Phenotype		Genotype	
K	V	$\phi(K)$	$\phi(V)$
1	1	1	1
2	2	2	2
2	3	3	3
4	4	4	4

Generation 3

Phenotype		Genotype	
K	V	$\phi(K)$	$\phi(V)$
1	1	1	1
2	2	2	2
?	?	3	3
4	4	4	4

Figure 6.8: Schematic representation of a section the translation table within the phenotype, and the corresponding description within the genotype for a lineage over two generations, where a perturbation was introduced to the K column in the second generation. Highlighted symbols represent those which are different to that of the previous generation.

deterministic reproducer. From this example it can be seen that it was not the symbol which was initially perturbed which was lost from the mapping, but the symbol *it was perturbed to* which was lost. A 1 was perturbed to a 2, however, it was the symbol 2 which was then lost from the mapping. This is due to the importance of location of symbol mappings in the translation table. If a symbol in K is perturbed to a symbol which already exists below it, then the symbol it was changed to will be lost as it will effectively "deactivate" the symbol mapping below it.

Figure 6.8 is an example of a perturbation to K in which a key is perturbed to a key which is located *above it*. Here a *3* was perturbed to a 2 in the second generation. Under this new genotype-phenotype mapping, the g-symbol 2 will still be decoded into the p-symbol 2 as this symbol mapping is unaffected. However, the g-symbol 3 can not now be decoded which results in an indeterministic offspring.

From these examples it is concluded that a single phenotypic point perturbation to the K column cannot result in a stable, deterministic creature with a different genotype-phenotype mapping. If a symbol is perturbed, and the symbol in which it is perturbed to exists lower down in the translation table, then the symbol in which it was perturbed to is lost from the mapping, other wise the symbol which was initially perturbed will be lost from the mapping.

Genotypic Perturbations Affecting The $\phi(V)$ Column

A single point genotypic perturbation was now applied to the description of the translation table, within the $\phi(V)$ column, as demonstrated in the second generation in Figure 6.9. In this situation, a 1 was perturbed to a 2.

Generation 2

Phenotype		Genotype	
K	V	$\phi(K)$	$\phi(V)$
1	1	1	2
2	2	2	2
3	3	3	3
4	4	4	4

Generation 3

Phenotype		Genotype	
K	V	$\phi(K)$	$\phi(V)$
1	2	1	2
2	2	2	2
3	3	3	3
4	4	4	4

Generation 4

Phenotype		Genotype	
K	V	$\phi(K)$	$\phi(V)$
2	2	1	2
2	2	2	2
3	3	3	3
4	4	4	4

Generation 5

Phenotype		Genotype	
K	V	$\phi(K)$	$\phi(V)$
?	2	1	2
2	2	2	2
3	3	3	3
4	4	4	4

Figure 6.9: Schematic representation of a section the translation table within the phenotype, and the corresponding description within the genotype for a lineage over four generations, where a perturbation was introduced to the description of V within the genotype in the second generation. Highlighted symbols represent those which are different to that of the previous generation.

As the genotype-phenotype mapping of the second generation has not yet changed, this creature will reproduce using the initial mapping and decode the altered genotype to produce the third generation with a translation table where the V column no longer contains the 1 key-value symbol. The third generation will now implement a different mapping where both g-symbols 1 and 2 will be mapped to the p-symbol 2. As it reproduces, the fourth generation no longer possesses the ability to decode the symbol which was perturbed, 1, however as this symbol still remains once in the genotype as it was not removed from the $\phi(K)$ column, the creature will be indeterministic.

Genotypic Perturbations Affecting The $\phi(K)$ Column

A single point genotypic perturbation was applied to the $\phi(K)$ column within the translation table's description. In this case, the 1 symbol was changed to a 2 which already appears *below* it in the translation table, which can be seen in Figure 6.10

The second generation creature will proceed to construct an offspring under the unaltered genotype-phenotype mapping. This will produce a third generation whose translation table no longer contains the key symbol 1 within the K column. The third generation no longer possess the ability to decode the g-symbol 1 as it does not appear in K. Furthermore, the g-symbol 2 is now mapped to the p-symbol 1. As this reproduces, the fourth generation will have lost the ability to decode the g-symbol 2, furthermore, the 1 g-symbol is mapped onto an indeterminate p-symbol, denoted by "?". The ability to decode for the symbol *which was perturbed*, 1, and the symbol *which it was perturbed to*, 2, has now been lost from the mapping. As these two symbols still exist within the genotype, this creature is indeterministic.

Finally, a different single point genotypic perturbation was applied to the $\phi(K)$ column within the translation table's description, Figure 6.11. In this case, a mutation results in a key symbol 3 being changed to a 2 which already ready appears *above* it in the translation table.

By the third generation when this mutation is expressed in the phenotype, the creature has lost the ability to decode for the g-symbol 3, however, as the perturbed symbol was changed to a symbol that already exists above it in the $\phi(K)$ column, then the edited row will not affect the mapping of the symbol which it was perturbed to, as it appears higher up in the translation table. The g-symbol 2 will still map onto the p-symbol 2, and the g-symbol which was perturbed 3 is not mapped onto any p-symbol. This creature has only lost one symbol from its mapping, the symbol which was perturbed, and not the symbol which it was perturbed to so the fourth generation will be indeterministic.

To summarise, with this specific substitution mapping, it is possible to lose symbol mappings from a genotype-phenotype mapping via a single point perturbation. Some single point perturbations result in two symbol mappings being lost from the genotype-phenotype mapping. However, the offspring will always be indeterministic.

Generation 2

Phenotype		Genotype	
K	V	$\phi(K)$	$\phi(V)$
1	1	2	1
2	2	2	2
3	3	3	3
4	4	4	4

Generation 3

Phenotype		Genotype	
K	V	$\phi(K)$	$\phi(V)$
2	1	2	1
2	2	2	2
3	3	3	3
4	4	4	4

Generation 4

Phenotype		Genotype	
K	V	$\phi(K)$	$\phi(V)$
1	?	2	1
1	1	2	2
3	3	3	3
4	4	4	4

Generation 5

Phenotype		Genotype	
K	V	$\phi(K)$	$\phi(V)$
?	?	2	1
?	?	2	2
3	3	3	3
4	4	4	4

Figure 6.10: Schematic representation of a section the translation table within the phenotype, and the corresponding description within the genotype for a lineage over four generations, where a perturbation was introduced to the description of K within the genotype in the second generation. Highlighted symbols represent those which are different to that of the previous generation.

Generation 2

Phenotype		Genotype	
K	V	$\phi(K)$	$\phi(V)$
1	1	1	1
2	2	2	2
3	3	2	3
4	4	4	4

Generation 3

Phenotype		Genotype	
K	V	$\phi(K)$	$\phi(V)$
1	1	1	1
2	2	2	2
2	3	2	3
4	4	4	4

Generation 4

Phenotype		Genotype	
K	V	$\phi(K)$	$\phi(V)$
1	1	1	1
2	2	2	2
2	?	2	3
4	4	4	4

Figure 6.11: Schematic representation of a section the translation table within the phenotype, and the corresponding description within the genotype for a lineage over three generations, where a perturbation was introduced to the description of K within the genotype in the second generation. The targeted symbol was changed to a symbol which already exists higher up in the table. Highlighted symbols represent those which are different to that of the previous generation.

This indeterministic creature may still find a matching symbol within itself or elsewhere in the soup and reproduce, however, this indeterministic symbol mapping, may affect the symbol mappings of employed p-symbols lower down in the translation table, and therefore result in sterile offspring, further decreasing the probability that a perturbation of this nature will produce a stable reproducer. As each symbol occurs at least twice in the genotype, within the description of the translation table, in order to ensure a stable reproducer, at least one more consecutive mutation must occur in the genotype, which replaces the indeterministic g-symbol, with one for which there is a matching key in the translation table.

Similarly, in order to introduce a new symbol mapping to the translation table of a stable, deterministic self reproducer, a minimum of two point perturbations must occur within the genotype to introduce this g-symbol to $\phi(K)$ and $\phi(V)$. Furthermore, a further two phenotypic perturbation must occur to introduce the p-symbol to the K and V columns. Therefore, I conjecture that it is impossible for a single point perturbation to result in a symbol mapping being added or removed to the genotype-phenotype mapping and still result in a stable, deterministic self reproducer. However, it is possible that indeterministic creatures may reproduce successfully provided the indeterministic symbol mapping does not affect the symbol mappings of employed p-symbols, therefore, it is possible that a second mutation may occur within this indeterministic creature, which removes the indeterministic g-symbol from the genotype, and renders it a deterministic, stable, self reproducer. However, in order to introduce a symbol mapping to the translation table, a minimum of 4 consecutive perturbations must occur, making the probability of this occurring significantly less. Therefore, with the translation table implementation of a substitution mapping, we should still expect to see an evolutionary bias to where creatures experiences a quasi-deterministic loss of unemployed symbols from the genotype-phenotype mapping, which would be indicative again of mutational irreversibility, as was observed with the look-up table

Furthermore, the situation is complicated even further if a permutation mapping is used instead of an identity mapping. For a permutation mapping, if for example, a key-value is lost from the translation table, then because the g-symbols and p-symbols share the same alphabet, somewhere else in the translation table, that same symbol, which is acting as a key, will be also lost. The key-value which this key mapped onto will be indeterministic in the next generation, and this may cause a chain reaction, until an employed symbol is finally lost from the mapping, rendering the lineage infertile.

6.4.6 Investigating a Change in Mapping, Without the Loss or Addition of Symbol Mappings

The nature of this translation table requires the active genotype-phenotype mappings to take the form of a permutation mapping between two symbol sets which use the same alphabet. In general this permutation mapping can be decomposed into several

cycles, or ordered sets. For example, Figure 6.12 presents an example of an identity permutation mapping. This initial mapping can be thought of five different cycles, each with a cycle length one, where 1 maps onto 1 is a single cycle. 2 mapping onto 2 is a second cycle etc. The different cycles are indicated by different colours.

Phenotype		Genotype	
K	V	$\phi(K)$	$\phi(V)$
1	1	1	1
2	2	2	2
3	3	3	3
4	4	4	4
5	5	5	5

Figure 6.12: Schematic representation of a section of creature's translation table and its description which implements a permutation mapping which is decomposed into five cycles, each of cycle length 1.

For the next example, Figure 6.13, the cycle length is increased so there is only one cycle, in which all symbols are included, with a cycle length of five. For example, the g-symbol 1 maps to 2, 2 maps to 3, 3 maps to 4, 4 maps onto 5, and 5 maps back onto 1.

Phenotype		Genotype	
K	V	$\phi(K)$	$\phi(V)$
1	2	5	1
2	3	1	2
3	4	2	3
4	5	3	4
5	1	4	5

Figure 6.13: Schematic representation of a section of creature's translation table and description, which implements a permutation mapping which is decomposed into a single cycle, with a cycle length of 5.

This mapping can be edited to create two cycles with a cycle length of three and two, rather than one single cycle with a cycle length of five, as shown in Figure 6.14. The first cycle is as follows: 1 maps onto 2, 2 maps onto 3 and 3 maps back onto 1. For the second cycle, 4 is mapped onto 5, and 5 is mapped back onto 4.

Phenotype		Genotype	
K	V	$\phi(K)$	$\phi(V)$
1	2	3	1
2	3	1	2
3	1	2	3
4	5	5	4
5	4	4	5

Figure 6.14: Schematic representation of a section of creature's translation table and description, which implements a permutation mapping which is decomposed into two cycles, with cycle lengths of 3 and 2.

In order to change the mapping of a translation table type system without either introducing or losing symbols from the mapping, then the cycles within the permutation mapping must be changed while keeping the total number of symbols consistent. This can theoretically be achieved by either increasing the cycle length of one permutation mapping and reducing the cycle length of another, joining two cycles together to create one larger cycle, or dividing a cycle up into two smaller cycles.

A permutation mapping is effectively cyclic, for example in Figure 6.14, in the first cycle, 1 is mapped onto 2, 2 is mapped onto 3, and 3 is mapped back onto 1. In a closed sequence of symbol mappings, we will refer to the final symbol mapping in the cycle as the *wrap around* symbol mapping, as the final symbol to be mapped in a sequence must be mapped onto the first symbol in the sequence. In order to break up a large cycle into two smaller cycles, a division point within the cycle must be chosen. At that division point, a symbol mapping must be converted to a wrap around symbol mapping, and it must now map to the first symbol in the cycle. Furthermore, the symbol mapping which was initially the wrap around symbol mapping, must also be changed to map onto the symbol which marks the first symbol in the new cycle.

Similarly, if you wish to join two cycles, e.g., the two cycles in Figure 6.14, the first wrap around symbol mapping $3-1$, must be edited to map onto the first symbol the second cycle, $3-4$. Furthermore, the wrap around symbol mapping of the second cycle $5-4$ must be changed to now map onto the first symbol in the first cycle, $5-1$. Now both cycles will be joined in one contiguous cycle, of cycle length 5, identical to the initial translation table in Figure 6.13.

From this reasoning, in can be seen that in order to change the genotype-phenotype mapping in a creature which implements this form of mapping, two symbol mappings must simultaneously be changed in the translation table at once. This is impossible to achieve with a single point perturbation in a translation table where every key and key-value is only represented once. However this *it is* possible if we add extra key-value pairs to the translation table, which will be demonstrated in the next section.

Due to the nature of this translation table, it is actually impossible to introduce

redundancy. In order for redundancy to occur, the set of possible g-symbols must be greater than the set of allowed p-symbols. With a look-up table, it is possible for multiple g-symbols to map into the same p-symbol, however, with this translation table, each distinct g-symbol must only map onto one distinct p-symbol. It is possible however, to have what will be referred to here, as *dormant* symbol mappings within the translation table. The translation table may consist of a number of *active* and *dormant* symbol mappings. A dormant symbol mapping refers to a row which exists in the translation table that will not be accessed by the creature during reproduction. As this ancestor uses key-value pairs to search for entries in the translation table, once a matching entry is found, the creature performs the mapping and proceeds to decode the next g-symbol in the genotype. If more than one identical key exists in the translation table matching any given g-symbol, then only the first key in the sequence will be accessed and contribute towards an active symbol mapping. It must also be highlighted that within a stable self reproducer, there cannot exist any keys or key-values within the dormant rows of the translation table, which do not already exist within the active rows, as for a symbol to appear *anywhere* in the phenotype, the p-symbol must be mapped onto by some g-symbol, and hence, appear within an active mapping. Therefore this genotype-phenotype mapping is a permutation mapping, where every active entry within the K column must also exist within the V column. Furthermore, as the dormant key-value pairs also exist within the creature's phenotype, any entry which is located within a dormant key-value pair, whether it be a key or an associated key-value, must also exist within an active key-value pair. It is impossible to have a stable, deterministic self reproducer which has a symbol within a dormant key-value pair which is not also the key-value symbol within some active key-value pair.

Introducing change to a permutation mapping

As described earlier, in order to change the genotype-phenotype mapping in a creature which implements this form of mapping, two key-value pairs must simultaneously be changed in the translation table at once. It is possible to achieve this via a single point perturbation, which has the effect of converting the status of certain key-value pairs from active to dormant, and similarly, convert dormant key-value pairs to active, having the final effect of a translation table with two changed symbol mappings.

When the translation table is perturbed, this may have the effect of changing the relative positions of different key-value pairs, and therefore have the effect of making a previously dormant mapping active, making a previously active mapping dormant, or both. If an active row is perturbed, then the key it is changing from must have been situated above any other instance of it in K. The key that it changed to may be situated either above or below. If a key is perturbed to a key which already exists above it, than the perturbed row in the translation table will become dormant. Furthermore, if the perturbed key within an active row already exists in a row further down in K,

than that row will switch from dormant to active as demonstrated in Figure 6.15. Here is an example of a possible translation table with two cycles, one with a cycle length of four, in which all symbol mappings are active, and the second cycle with a cycle length of one in which this mapping is currently dormant, 3 mapped onto 4. This mapping is a mutation from the the initial mapping illustrated in in Figure 6.13, in which the key 4 now maps onto 1 following a single point mutation. A minimum of two further mutations allows the final key-value pair to be lost from the mapping and is rendered a dormant mapping. This dormant mapping has by chance been permutated to resemble an already existing, active key-value pair within the translation table.

For clarity the genotype has been omitted from the following diagrams and a "status" column has been included which highlights if a row in the translation table is active, "*A*", or dormant, "*D*".

Generation 1

K	V	status
1	2	A
2	3	A
3	4	A
4	1	A
3	4	D

Generation 2

K	V	status
1	2	A
2	3	A
1*	4	D
4	1	A
3	4	A

Figure 6.15: Schematic representation of a section of creature's translation table. The separate colours represent separate permutation mappings. A number followed by an asterisk *, represents a symbol which is different from the previous generation.

In the mapping depicted in Figure 6.15, the symbol 3 appears twice in K, therefore the first key-value pair where 3 is mapped onto 4 is active, and the second key-value pair where 3 is also mapped onto 4 is dormant. A mutation is introduced which changes the first instance of key 3 to a 1. As key 1 already exists further up the K column, this row now becomes dormant. The second instance of key 3 is now the only instance of 3 within K, so this mapping now becomes active.

In this example a point perturbation to the translation table did not result in any change to the genotype-phenotype mapping. When a perturbation to K changes a key to one which already exists higher in K, then this will have the effect of switching an active symbol mapping dormant, while activating a dormant symbol mapping. As the

newly active symbol mapping was actually identical to a perturbed one, the genotype-phenotype mapping remained the same.

The second possible perturbation is if the symbol is perturbed to a symbol which already exists below it in K, as shown in Figure 6.16.

Generation 1

K	V	status
1	2	A
2	3	A
3	4	A
4	1	A
2	1	D

Generation 2

K	V	status
1	2	A
4*	3	A
3	4	A
4	1	D
2	1	A

Figure 6.16: Schematic representation of a section of creature's translation table. The separate colours represent separate permutation mappings. A number followed by an asterisk, represents a symbol which is different from the previous generation.

In this mapping, the symbol 2 appears twice in K, therefore the first symbol mapping where 2 is mapped onto 3 is active, and the second symbol mapping where 2 is mapped onto 1 is dormant. A perturbation is introduced which changes the first instance of 2 to a 4, where 4 already exists lower down in the K column. This perturbed row still stays active, however the lower row containing 4 becomes dormant while the bottom row which contains 2 is activated.

In this case, because the affected key is perturbed to a key which exists below it, then the mapping *will* change. Rather than having one large active cycle with a cycle length of four, this point mutation had the effect of creating 2 smaller active cycles of cycle length two, where 1 is mapped onto 2 and 2 is mapped back onto 1, and also 3 is mapped onto 4 and 4 is mapped back onto 3. This is a definite proof-of-principle example of single point perturbation to a stable, deterministic self reproducer, resulting in a stable, deterministic self reproducer with a different genotype-phenotype mapping.

Therefore, when an active row is perturbed in a stable, deterministic reproducer, if the perturbed row stays active, the mapping will change. If a dormant row is perturbed and the affected key symbol is perturbed to a key symbol which already exists above it, then the perturbed row will stay dormant and it will not affect the mapping. However, if the affected symbol is perturbed to a symbol which already exists below it, then

it will change from a dormant mapping to an active mapping. This may cause the mapping to change.

If it changes from dormant to active, then a row below will change from active to dormant. If the newly active row does is not equal to the newly dormant row, then it will not be a permutation mapping. Therefore, the only method in which a perturbation to a dormant symbol can give rise to a stable reproducer, is if the mapping is unchanged afterwards.

The only method in which the mapping can change is if an active row is perturbed, and it stays active by changing the symbol to one which already exists below. This method can be used to break up a single permutation mapping into two smaller permutations, or join two permutations together to form one large one. For example, the following is a mapping with two different permutation maps as shown in Figure 6.17

K	V	status
1	2	A
2	3	A
3	1	A
4	5	A
5	4	A

Figure 6.17: Schematic representation of a section of creature's translation table. The separate colours represent separate permutation mappings.

In order to join these two permutation mappings, two individual symbol mappings must be changed. The rows $3-1$ and $5-4$ must be changed to $3-4$ and $5-1$. If dormant symbol mappings are present in the translation table, then it is possible for this to be accomplished via one perturbation by changing one active mapping to a dormant one.

Figure 6.17 demonstrated a possible translation table which holds the same mapping as Figure 6.18, although the location of the rows have been changed.

K	V	status
1	2	A
2	3	A
5	4	A
3	1	A
4	5	A

Figure 6.18: Schematic representation of a section of creature's translation table. The separate colours represent separate permutation mappings.

If a dormant symbol mapping $5-1$ is introduced, generation one in Figure 6.19,

then it is possible for these two permutation maps can be joined together with only one perturbation.

Generation 1

K	V	status
1	2	A
2	3	A
5	4	A
3	1	A
4	5	A
5	1	D

Generation 2

K	V	status
1	2	A
2	3	A
3	4	A
3*	1	D
4	5	A
5	1	A

Figure 6.19: Schematic representation of a section of creature's translation table. The separate colours represent separate permutation mappings. A number followed by an asterisk *, represents a symbol which is different from the previous generation.

A perturbation is introduced in generation 2 of Figure 6.19, which changes the active mapping from $5-4$ to $3-4$, which is demonstrated.

This has the effect of turning the lower down $3-1$ mapping dormant, and activating the $5-1$ mapping. This is now a single active permutation mapping rather than two smaller active mappings. This is an example of *a single point perturbation changing the genotype-phenotype mapping* when space for redundancy is introduced.

This example changed the genotype-phenotype mapping by joining up two smaller permutation mappings into one larger one. This next example attempts to change the genotype-phenotype mapping by splitting a single permutation mapping into two smaller mappings.

In order to split the above permutation mapping into two smaller mappings, a random "break point" must be chosen in which the single mapping separates into two individual mappings. If between the symbol 3 and 4 in the K column is chosen as the break point, then the $3-4$ and the $5-1$ mappings must be removed, and a $3-1$ and a $5-4$ mapping must be introduced in a single point perturbation. This can be achieved by introducing the dormant mapping $5-4$ and changing the position of the active mapping $5-1$ in the translation table shown in Figure 6.20.

Generation 1

K	V	status
1	2	A
2	3	A
5	1	A
3	4	A
4	5	A
5	4	D

Generation 2

K	V	status
1	2	A
2	3	A
3*	1	A
3	4	D
4	5	A
5	4	A

Figure 6.20: Schematic representation of a section of creature's translation table. The separate colours represent separate permutation mappings. A number followed by an asterisk *, represents a symbol which is different from the previous generation.

When we have a dormant mapping $5-4$, the perturbation which changes the $5-1$ mapping to a $3-1$ mapping was applied.

The $3-4$ mapping was rendered dormant so it had been removed from the active mapping, while the $5-1$ mapping has been perturbed, so it had been lost. This action also created the $3-1$ mapping, and activated the dormant $5-4$ mapping. By removing the $3-4$ and the $5-1$ mappings and introducing a $3-1$ and a $5-4$ mapping it was possible to break up the single active permutation mapping into two smaller ones, where 1 is mapped onto 2, 2 is mapped onto 3, and 3 is mapped back onto 1, and also 4 is mapped onto 5, and 5 is mapped back onto 4. This example shows that it is possible for a single point mutation to change a mapping by both splitting a single active permutation mapping up into two smaller active permutation mappings, or by joining up two smaller permutation mappings into one larger one, however dormant mappings must also exist in the translation table.

6.4.7 Discussions

For this experiment it became apparent that the redesigned creatures, `vn_tt128_758`, needs substantially more processing time in order to reproduce. This meant that performing simulations which allowed us to obtain data sets of the same size as previous experiments would prove infeasible. An initial simulation was run which showed a reduction in the number of unemployed symbols. However the relatively small data set obtained was not substantial evidence to confirm that the same mutational ratcheting

phenomena, which was observed with previous experiments, was being displayed here.

Rather than stochastically introducing pseudorandom perturbations to the genotype-phenotype mapping, the mapping was manually tested, and each possible single point perturbation which may give rise to a change in the mapping was tested. For an initial mapping which included no dormant key-value pairs, it was discovered that there are no single point mutations which can lead to a deterministic, self reproducing offspring with a different genotype-phenotype mapping. A creature would have to experience a minimum of two consecutive mutations in order to remove a symbol mapping from the genotype-phenotype mapping. In order to re-introduce a symbol mapping to the translation table, the creature must experience at least 2 phenotypic perturbations and 2 genotypic perturbations, which shows that this associative memory mechanism of implementing a substitution mapping also displays the same mutational ratcheting phenomena as the previous experiments.

It was also found that in order to change a "permutation mapping" by a single point perturbation, it is necessary to either join two cycles together to form one large cycle, or break one cycle apart to form two smaller cycles.

To join two permutation mappings together, the last "wrap around" mapping for both permutations must be edited (in the last case $3 - 1$ and $5 - 4$). This means two mappings being lost and two mappings are created. This is the maximum number of changes that is possible with one perturbation, so it is only possible to join up a maximum of two permutation mapping with a single point perturbation. Furthermore, when breaking up a permutation mapping into two, both end wrap around mappings must be created. this means that $3 - 1$ and $5 - 4$ must be created and $3 - 4$ and $5 - 1$ must be lost. So it is also only possible to break up a single permutation mapping into a maximum of two different mappings with a single point perturbation.

6.5 Conclusion

This chapter documented the experimental procedure and results for three separate sets of experiments, each demonstrating a different mechanism for implementing a mutable genotype-phenotype mapping within a von Neumann style self reproducer. The first set of experiments implemented a random substitution mapping, carried out via inclusion of a look-up table. These experiments saw the degeneration to self copiers and the emergence of pathological constructors, which may eventually result in total ecosystem collapse.

Both the Tierra operating system and ancestor prototype was redesigned in order to eliminate these phenomena in order to focus on the evolution of the genotype-phenotype mapping. An identity mapping which included several "unemployed" symbols was chosen for the look-up table. When this new von Neumann style ancestor was implemented, a particular phenomenon was observed in which the genotype-phenotype mappings of the lineages which propagated throughout the soup changed so that all unemployed

symbol mappings were removed from the look-up table. This bias in the evolutionary trajectory of the genotype-phenotype mapping was not due to Darwinian selection, or drift, but due to mutations to the genotype-phenotype mapping which were not directly reversible, therefore, a mutational ratcheting effect was observed. In order to reverse these "not directly reversible" mutations, not only must the change to the genotype must be reversed, but this change must also be coincidentally coordinated with a phenotypic perturbation which returns the phenotype to its previous state. This is an example of how, in very exceptional circumstances, a form of Lamarkian inheritance may in fact occur, if the phenotypic perturbation is such that it changes the decoding mechanism of the creature. The creature now decodes its genotype under a different genotype-phenotype mapping, and the newly constructed phenotype matches exactly that of the parent creature after it experienced a perturbation to the phenotype.

A different implementation of a substitution mapping was implemented via the use of a translation table. Through investigation it was discovered that this mechanism is also subject to similar phenomena to those observed when examining the evolution in the genotype-phenotype mapping carried out via the look-up table.

The results of the first set of experiments were accepted for publication in the proceedings of three international conferences in 2012, European Conference on Complex Systems (Baugh & McMullin, 2012b), Simulation on Adaptive Behaviour (Baugh & McMullin, 2012a), and the Frontiers of Natural Computing Workshop (Baugh & McMullin, 2012c). The results of the second set of experiments were published and orally presented in 2013 at the European Conference of Artificial Life (Baugh & McMullin, 2013a), and at the European Conference on Complex Systems (Baugh & McMullin, 2013b).

Chapter 7

Conclusions and Future Work

7.1 Revisiting the Original Research Question

The implementation of a mutable genotype-phenotype mapping is a phenomenon which appears to be common to all forms of life as we know it. However, while devising an architecture which supports the evolutionary growth of machine complexity, John von Neumann and Arthur Burks stated that "If the change is in A, B or C, the next generation will be sterile." (Von Neumann & Burks, 1966, p. 86). In this context, A, B and C refer to the set of components which are directly, and solely responsible for decoding and copying a genotype, and constructing an offspring genotype and phenotype. This mechanism must therefore contain a programmable constructor, A, which inspects a genotype, and under some genotype-phenotype mapping, constructs an offspring phenotype. Although von Neumann stated that if a change happens to A then the offspring will be sterile, it is apparent that through evolution, the biological genotype-phenotype mapping must have experienced numerous and consecutive changes over time to evolve into its current state. There must therefore be some situations in which a mutated genotype-phenotype mapping may give rise to functional, self reproducing offspring. Although the genetic code appears to be universal throughout the entire phylogenetic tree of life, genotype-phenotype mappings are quite diversified. For example, a genotype-phenotype mapping of a particular organism might perform error correction when decoding for specific attributes of the phenotype, making it less variable than other attributes, or it might use a single gene to determine two attributes ensuring that the two can only vary together (Webb & Knowles, 2014). Furthermore, as the genotype-phenotype mappings are so diverse throughout life, the phenotype of one organism can not map a genotype belonging to a distinctly different genus.

This thesis aimed to develop an exploratory model, using a modified version of an existing artificial life platform, and the architecture which was devised by Von Neumann, to provide proof-of-principle examples where self reproducing agents, with a mutable genotype-phenotype mapping, give rise to functional offspring which implemented a mutated mapping.

7.2 Thesis Summary

The field of artificial life investigates the logic of living systems in artificial environments, in order to gain a deeper understanding of the complex information processing that defines such systems. In order to develop an agent based system which simulates the emergent properties of life, it is important to have a deep understanding of the fundamental criterion to which bear on the nature of life as we know it. Chapter 1 investigated the question of "What is Life?" and a list of the fundamental properties of life were proposed. One such property, *information storage of a self representation* was noted as never being fully incorporated into the most widely used artificial life systems such as *Tierra*.

Chapter 2 described the developments in artificial life, from the publication of, *On The Origin Of Species*, to the development of artificial environments such as cellular automata and core world type systems. In order to develop a system which can implement a self reproducing agent that contains a self representation within an information storage, core world type systems appeared to be a suitable option, as specifically, compared to cellular automata, they are relativity fast in wall-clock time, and easier to configure/program. A possible candidate for a machine architecture which separated genotype from phenotype had already been put forward by John von Neumann, and Chapter 3 described this architecture in detail, highlighting the fact that such an architecture would naturally incorporate a mutable genotype-phenotype mapping. The focus of this thesis concentrated on designing an exploratory model which implements a mutable genotype-phenotype mapping, and analysed the evolutionary trajectory and phenomena which emerged from the system.

7.3 Experimental Results Overview

A von Neumann style self reproducer was designed and implemented within Tierra. In Chapter 5, the evolutionary trajectory of this reproductive mechanism was stochastically and deterministically explored for different implementations of this architecture.

Amendments were made to the Tierra system itself to correct flaws, add extra functionality to the system, and simplify data analysis. A number of tools were also written in Python to analyse the output.

Preliminary experiments saw the emergence of *self copiers*, where a single point mutation resulted in the loss of the genotype-phenotype mapping mechanism completely. The mutated strain reverted to simply copying its code directly to its offspring without performing any decoding of the genotype. This alludes to the possibility that there are at least conceivable situations where the reverse may occur, in which a single point mutation can result in a self copier giving rise to a creature which reproduces via the inclusion of a genotype-phenotype mapping.

Further experiments saw the emergence of *pathological constructors*, which were at least partially a result of a specific feature within Tierra, where a flaw in the code resulted in a selective advantage for creatures to be located at a specific location in memory. The separation of genotype and phenotype also resulted in an increased level of ease with which these creatures can emerge, and hence, increased the probability that such a creature will be randomly located at this position within the soup.

The specific genotype-phenotype mapping implemented was a substitution mapping, which was implemented by two different mechanisms; via a look-up table, and a translation table. By analysing the look-up and translation tables of the strains which are produced through evolution, the change in mappings can be studied, and a similar set of phenomena was observed for both implementations. The first phenomenon observed was that of a mutational ratcheting effect, where symbols which were originally

included in the mapping, but not employed specifically for reproduction, were systematically lost. Mutations which resulted in the loss of a symbol from the mapping were not directly reversible (i.e. via the result of a genotypic change) so this resulted in a non-Darwinian evolutionary bias, leading to cumulative loss of these non-employed symbol mapping.

A second phenomenon which was observed, was an example of Lamarkian inheritance, where perturbations directly within the phenotype, could in exceptional circumstances, be inherited without any change to the genotype. This was due to a perturbation which affected the genotype-phenotype mapping function, which is located within the phenotype itself. Although the genotype was unaltered, its interpretation by the parent was changed such that the genotype now decodes to exactly match that of the "new" perturbed phenotype, so effectively, the change to the phenotype is inherited to its offspring. This form of Lamarkian inheritance proved to be the only mechanism which allowed p-symbols, which were not present in the parent, to be introduced to the mapping of the offspring, expanding the set of potential p-symbols which could be coded for in the offspring phenotype.

Although the phenomenon observed may be true for other mappings, from these experiments, one can only say for certain that this set of phenomena is true for this specific implementation of this specific mapping. There are an infinite of different mapping mechanisms to choose from, rather than a substitution mapping, and even within the possibilities of a substitution mapping, there are a large number of possible substitution mappings which may be implemented. Furthermore, these experiments were performed on only one artificial life platform, Tierra. A completely different, non core world architecture, such as cellular automata, may have been used which could significantly affect the outcome of evolutionary runs.

Overall, I believe that choosing Tierra as a platform, von Neumann's architecture and a substitution mapping was a useful starting point. Tierra is a well known platform with a wealth of literature available, so results can be compared and contrasted to work which is already performed in the past. Using von Neumann's architecture supplied a definite research question which could be answered, "is it possible to demonstrate how a mutable genotype-phenotype mapping may give rise to new strains, which are self reproducing, but realise a different mapping?", which was achieved and answered.

In hindsight, an identity mapping may not have been the best option. Although it greatly simplified the process of programming ancestors, and creating data analysis tools which would search the look-up table/translation table and their descriptions in order to detect changes in the mappings, it also made it easier for "stray CPUs" to start executing the symbols *within the genotype*, and interpreting it as meaningful code. This may have had significant effects on the outcome of some evolutionary runs. If an arbitrary initial substitution mapping was chosen which was not the identity mapping, there would be a decreased probability that sections of a creature's genotype could be interpreted as functional strings of instructions.

7.4 Future Work

7.4.1 Development of Tierra

Tierra is a widely documented platform for investigating and simulating evolutionary phenomena in silico. However, over the course of this thesis, some deficiencies became apparent in the Tierra Platform, which could be improved upon.

The first fault is in relation to the strain labelling mechanism in which is implemented. When a new strain emerges, a label is generated where the creature length is concatenated with a three letter string. For example, the initial Ray self copier is labelled *0080aaa*. This three letter mnemonic only allows a default of 26^3 or 17,567 distinct mnemonics for each creature size, or an increased space of 52^3 or 140,608 if the `BIGNAMES` parameter is configured. `BIGNAMES` allows the three letter mnemonic to use upper case as well as lower case letters to increase the available namespace. However, the source code which generates labels to strains is unreliable. If the namespace is exhausted, then the leading/most significant character in the mnemonic will increment past 'z' and non-printable ASCII characters may be used. When `BIGNAMES` is not configured, once a mnemonic character increments past ASCII 'z', or numeric 122, it will increment up to numeric 127, and wrap around back to numeric 0. Numeric 123 - 126 relates to the ASCII characters `{`, `|`, `}`, and ~, while numeric 127 related to ASCII `DELL` which is non-printable. Alternatively, if `BIGNAMES` is configured, the leading charterer will increment up immediately past 'Z' in ASCII; initially that would be `[`, `\`, `]`, `^`, `_`, and `` ` ``; but that is followed by the start of the lower-case alphabetics - 'a', 'b', 'c', etc., so it would actually start generating labels that it had already generated previously, but now referring to completely unrelated strains. Either way, once the three letter alphabetic character namespace is exhausted, the genebanker functionality is no longer reliable or even really usable. Tierra could be amended to included defensive coding to raise an exception and terminate the run (with a suitable diagnostic message) if this happens. However it is likely that when Tierra was initially developed, it just seemed inconceivably (on then available hardware) that a run would go on long enough to actually trigger this, so it was just simply ignored.

Additionally, for long simulations, the three letter string method of labelling creatures is still not a viable mechanism. When a creature is born that is not identical to its parent, Tierra must check if this creature previously exists in the genebank. If an identical creature is found in the genebank than the new creature will be assigned the same label. If Tierra searches every same length strain within the genebank and no identical creature is found, then a new label will be created and applied to that strain. As the genebank increases in size, this process becomes painstakingly slow and actually results in a bottleneck where Tierra must sequentially search the genebank of creatures each time a new creature is born, significantly slowing down the wall-clock speed of experiments.

A far more effective labelling mechanism would be to use a hash function to label the

creatures saved to the genebank. A hash function is any function that can be used to map digital data of arbitrary size to digital data of fixed size. When a creature is born, its entire code can be fed as a "key" into a hash function in order to produce a "hash value" which will be a code of fixed size, such as a numerical value. This numerical value can will act as a label for the corresponding strain and can be stored in a hash table data structure. Hash tables are widely used in computer software for rapid data look-up. When a creature is born which differs from its parent, the creature's raw code can be fed into the hash function. If the hash value which is produced is not already within the hash table, than the creature is a new distinct strain whose hash value will be saved to the hash table. By using this mechanism, when a new creature emerges, Tierra needs only to calculate the hash value, and seek this value within the hash table. There is no need to search the genebank and compare the code of every creature of the same length with that of the newly emerged. Including this feature in Tierra would allow for longer simulations without severe performance loss as the genebank becomes populated. Furthermore, this will significantly diminish the problem of exhausting the namespace and increase the maximum number of distinct creatures which can be saved to the genebank. Technically, a limit will still exist, which is defined by the range of the hash function. For experiments of this size we can assume that this limit is beyond that which will affect any currently conceivable experiments.

7.4.2 Investigation of Alternative Mappings

Firstly, the space of possible mappings which were explored was very limited. Within the space of substitution mappings, there are a large number of symbol mappings which may occur. The choice of the selected initial mapping may affect the possible fitness of the descendants. For example including redundancy within the genotype-phenotype mapping introduces a scope for evolutionary enhancement of mutational robustness, where certain changes to the genotype do not lead to any changes in the decoded phenotype. However, including a larger symbol alphabet may result in larger creatures which increases the gestation period. There may be an optimum redundancy size which will result in lineages of the greatest fitness. Furthermore, a substitution mapping is not the only possible mechanism available. As the Tierra language is proven to be "Turing complete" Maley (1994), then any conceivable computable function may be used when decoding the genotype. However, the Tierra instruction set was created with the design goal of encouraging evolution, and not to simplify human programming. Manually designing a mapping which uses, for example, a compression coding, would be complicated, and would significantly increase the size of the programmable constructor, and the time taken to decode a genotype. In Tierra, reproduction time is the dominating factor which affects a strain's fitness, therefore strains with more complex mappings will be seen as decreasingly fit, and will increase the evolutionary pressure for alternative creatures such as self copiers to be retained.

Two specific types of reproduction have been demonstrated within the Tierra world, self copying, and genetic reproduction. However, there may be alternative reproduction mechanisms available which can be explored. In Tierra, a creature must be fully developed before it is allocated a CPU. Due to this, more complex creatures will have a larger gestation time and its fitness will decrease. For simple single celled lifeforms such as bacteria, once a fully grown cell divides, both cells are fully functioning and fully developed. However, for multicellular, more complex lifeforms such as plant and animal life, an offspring starts functioning on its own as a separate individual organism long before it is fully developed. Introducing creatures who are only "partially developed" upon birth, would decrease the gestation time. For example, when a creature is born, it contains a full genome, and partial phenotype and is still "developing" after it is born. Such a creature may start decoding its genotype and constructing its offspring phenotype, while simultaneously constructing the parts of its phenotype which are not required to be in use yet.

7.4.3 Spontaneous Emergence of a Genotype-Phenotype Mapping

The original emergence of a genotype-phenotype mapping is a subject which requires further investigation. It was demonstrated how a single point mutation can result in a self copier emerging from a lineage of von Neumann style reproducers. This opens up the conceptual possibility that it may be possible for the reverse, a von Neumann style reproducer emerging from a lineage of self copiers.

According to the RNA world hypothesis, in the early origin of life, molecules of RNA reproduced via social reproduction. A single molecule of RNA cannot self reproduce in isolation but might function as an enzyme to replicate a separate molecule of RNA. Within this system, there is no division of labour between the passive storage of information and the functional catalytic component. The entire strand of RNA can act as either a functional component which catalyses the reproduction event, or act as passive data of a self representation in order to be replicated by another RNA. As there is no distinction between genotype and phenotype, this form of reproduction does not include a (mutable) genotype-phenotype mapping. At some point, environmental pressures resulted in a world in which there was a separation between genotype and phenotype, and a passive information storage (DNA) and an active catalyst (proteins). However the exact process of how an RNA world transitioned to a DNA-Protein world is still unknown. This topic still requires much investigation and it may be useful to attempt to replicate the RNA world, where creatures do not self reproduce, but replicate the memory images of neighbouring creatures. However, an evolutionary simulation platform capable of operating in this manner, where all programs can in principle function as both "rules" (functional enzymes) and "strings" (passive data) has already been developed by Decraene & McMullin (2011). It was also shown that *Cellular Information Processing Networks*, capable of distinct information processing could be evolved

in this manner, so it may be beneficial to investigate if this idea can be adapted to observe if the complex, information processing system of a genotype-phenotype mapping may emerge/evolve using this similar methods.

In order to have a separation of genotype and phenotype, the phenotype must also include a mechanism for decoding the genotype and constructing an offspring phenotype. Complex information processing is not a property which is selectively advantageous within Tierra, as it slows down reproduction time and hence the creature appears to be "unfit" as Tierra favours rapid reproduction above everything else, so this behaviour is not selectively favoured.

The question of "what are the environmental pressures that could steer the emergence or evolution of the genotype-phenotype mapping?" is very important question that requires investigation. This thesis only looked at the possibility of a mutable genotype-phenotype mapping, but not how different mappings might be favoured. If the overall aim is to investigate how to exploit genotype-phenotype evolution in artificial systems, one would need to have a theoretical understanding of the change of genotype-phenotype mappings and understanding of evolutionary dynamics, and what environmental factors influence these changes.

In the case of Tierra, this could be achieved by making changes to the creature architecture itself, or by amending the Tierra platform. As Tierra favours creatures with rapid reproduction times, creatures which spend extra time on information processing are selectively displaced. This problem has already been noted and a possible solution has been put in place with the artificial life platform *Cosmos*, which was designed with the intention "to encourage the evolution of diversity and complexity of the competing programs" (Taylor, 1999). Within Cosmos, a program has a store of energy tokens which it collects from the environment. In order to function it must pay an energy token to the processor for each instruction it executes. Avida is another Artificial life platform which implements the idea of programmes being rewarded for performing specific tasks, however, social reproduction cannot be performed on either Cosmos or Avida, as the programs within these platforms cannot directly read the memory image of their neighbours, so it would be impossible to implement an RNA type world on these systems without significant change to the underlying source code.

By adapting this idea within Tierra, the system could be modified so that creatures who perform a decoding of symbols during reproduction are allocated "bonus" slices of CPU time, or are moved down the reaper queue so as to increase their life span.

If these evolutionary pressures are introduced, it may be possible to observe and identify concrete examples of lineages of self copiers that evolve into lineages which reproduce via the implementation of a genotype-phenotype mapping. This would not however, shed any light on the natural selectional pressures that might favour such pathways actually being followed in the absence of such artificial selection, but would simply be a proof-of-principle example of the possibility of evolutionary pressures resulting in the emergence of a genotype-phenotype mapping in the digital medium.

One specific method which could be investigated, would be to study the effects of using an instruction set which includes some form of "decoding" instruction, which may be implemented and executed during reproduction, as opposed to the copy instructions. Creatures can be rewarded in CPU time for executing this particular instruction and might be one possible mechanism to introduce some environmental pressure to encourage the emergence of a genotype-phenotype mapping.

7.5 Closing Statement

The overall aim of this thesis was to develop an exploratory model which simulated machine self reproduction, which facilitates the evolution of a mutable genotype-phenotype mapping, to provide proof-of-principle examples that this task could be realised in silico, and to investigate the set of phenomena which may emerge from such a system. This was accomplished via the implementation of the von Neumann architecture for machine self reproduction within the artificial life platform of Tierra.

Phenomena which emerged from this system, was investigated, characterised and published at various international conferences. Despite this preliminary work only addressing a very small, particular, piece of the puzzle in the mechanisms at play during the emergence and evolution of a biological genotype-phenotype mapping, this thesis adds to the base of knowledge of the field, and poses significant future work which is imperative to fully understanding the mystery of the origin of life.

Bibliography

Adami, C., & C. Titus, B. (1994). Evolutionary learning in the 2D artificial life system Avida. California Institute of Technology Pasadena, CA 91125. http://arxiv.org/abs/adap-org/9405003.

Adami, C., Ofria, C., & Collier, T. C. (2000). Evolution of biological complexity. *Proceedings of the National Academy of Sciences*, *97*(9), 4463. http://www.pnas.org/content/97/9/4463.full.

Altenberg, L. (1995). Genome growth and the evolution of the genotype-phenotype map. Evolution and Biocomputation, Lecture Notes in Computer Science Volume 899, pp 205-259, Springer. http://link.springer.com/chapter/10.1007/3-540-59046-3_11.

Barricelli, N. (1957). Numerical testing of evolution theories. (pp. 97–127). J. Statistical Computation and Simulation, Vol. 1.

Barricelli, N. (1962). Numerical testing of evolution theories part 1, theoretical introduction and basic tests. (pp. 69–98). Acta Biotheoretica, Vol 16, Issue 1-2.

Baugh, D., & McMullin, B. (2012a). The emergence of pathological constructors when implementing the Von Neumann architecture for self reproduction in Tierra. In *12th International Conference on the Simulation on Adaptive Behaviour*, (pp. 240–248). University of Southern Denmark: From Animals to Animats, Volume 7426. http://link.springer.com/chapter/10.1007%2F978-3-642-33093-3_24.

Baugh, D., & McMullin, B. (2012b). The emergence of pathological constructors when implementing the Von Neumann architecture within Tierra. In *Proceedings of the European Conference on Complex Systems (ECCS)*, (pp. 165–169). Springer Proceedings in Complexity 2013. http://link.springer.com/chapter/10.1007%2F978-3-319-00395-5_25.

Baugh, D., & McMullin, B. (2012c). The emergence of pathological constructors when implementing the Von Neumann architecture within Tierra. In *Proceedings of the Frontiers of Natural Computing Workshop*. University of York. http://www.academia.edu/4422290/Frontiers_of_Natural_Computing_2012_-_Workshop_abstract.

Baugh, D., & McMullin, B. (2013a). Evolution of g-p mapping in a Von Neumann self reproducer within Tierra. In *Advances in Artificial Life - European Conference of Artificial Life (ECAL) 2013, Proceedings of the Twelfth European Conference on the Synthesis and Simulation of Living Systems*, (pp. 210–217). University of Southern Denmark: The MIT Press. http://mitpress.mit.edu/sites/default/files/titles/content/ecal13/978-0-262-31709-2-ch032.pdf.

Baugh, D., & McMullin, B. (2013b). Evolution of genotype-phenotype mapping of Von Neumann style self reproduction within the platform of Tierra. In *Proceedings of the European Conference on Complex Systems (ECCS)*, (pp. 165–170). Springer. https://books.google.ie/books?id=gIu4BAAAQBAJ&lpg=PR1&dq=Proceedings%20of%20the%20European%20Conference%20on%20Complex%20Systems%202013&pg=PR1#v=onepage&q=Proceedings%20of%20the%20European%20Conference%20on%20Complex%20Systems%202013&f=false.

Bedau, M. A. (2003). Artificial life: organization, adaptation and complexity from the bottom up. *Trends in Cognitive Sciences*, 7(11), 505–512. http://people.reed.edu/~mab/publications/papers/BedauTICS03.pdf.

Benner, S. (2010). Defining life. (pp. 1021–1030). Astrobiology, 2010 Dec; 10(10). http://www.ncbi.nlm.nih.gov/pmc/articles/PMC3005285/.

Burks, A. (1970). *Essays on Cellular Automata*. University of Illinois Press. http://www.amazon.com/Essays-Cellular-Automata-Arthur-Burks/dp/0252000234/ref=sr_1_1?s=books&ie=UTF8&qid=1433524759&sr=1-1&keywords=Essays+on+Cellular+Automata.

Cariani, P. (1991). Emergence and articial life. (pp. 775–798). In: Artificial Life II. Sante Fe Inst. Studies in Science of Comlexity. http://www.cariani.com/CarianiNewWebsite/Publications_files/CarianiArtificialLife-II-1991.pdf.

Chen, T. M., & Robert, J. (2004). The evolution of viruses and worms. In W. W. Chen (Ed.) *Statistical Methods in Computer Security*, (pp. 265–285). https://books.google.ie/books?hl=en&lr=&id=lPDKBQAAQBAJ&oi=fnd&pg=PA265&dq=Chen,+T.+M.,+%26+Robert,+J.+(2004).+The+evolution+of+viruses+and+worms.&ots=8WKW5eBAxU&sig=5DuP4iHjwtzf9sCkTCd_TTJwuBs&redir_esc=y#v=onepage&q&f=false.

Codd, E. (1968). *Cellular Automata*. Academic Press, New York. http://www.amazon.com/Cellular-Automata-E-F-Codd/dp/1483211746.

Darwin, C. (1859). *On the Origin of Species by Means of Natural Selection, or the Preservation of Favoured Races in the Struggle for Life*. http://www.amazon.com/

Origin-Species-150th-Anniversary/dp/0451529065/ref=sr_1_1?s=books&ie=UTF8&qid=1433523584&sr=1-1&keywords=on+the+origin+of+species.

De Beule, J. (2011). Evolution, organization and function in the biological system. Proc. of the Fields Institute workshop on Semiotics, Cognitive Science and Mathematics. http://ai.vub.ac.be/publications/438.

Decraene, J., & McMullin, B. (2011). The evolution of complexity in self-maintaining cellular information processing networks. (pp. 55–75). Advances in Complex Systems, Vol, 14, No. 1. http://doras.dcu.ie/16292/1/decraene-ACS-09.pdf.

Deutsch, D. (1997). *The Fabric of Reality*. Viking Adult. http://www.amazon.com/The-Fabric-Reality-Universes-Implications/dp/0713990619.

Dewdney, A., & Jones, D. (1984). Core War guidlines. Department of Computer Science, The University of Western Ontario. http://corewar.co.uk/cwg.txt.

Eigen, M. (1971). Selforganization of matter and the evolution of biological macromolecules. (pp. 465–523). Die Naturwissenschaften, Vol. 58, No. 10. http://www.physik.uzh.ch/groups/aegerter/teaching/Biophys/eigen.pdf.

Emmeche, C. (1992). Life as an abstract phenomenon: Is artificial life possible? (pp. 466–474). Toward a Practice of Autonomous Systems. Proceedings of the First European Conference on Artificial Life, The MIT Presss. http://www.nbi.dk/~emmeche/cePubl/92a.lifabsphe.html.

Farmer, J., & Belin, A. (1991). Artificial life: The coming evolution. (pp. 815–840). Artificial Life II, Santa Fe Institute studies in the Sciences of Complexity, proceedings Vol. X. Working Paper. http://www.santafe.edu/media/workingpapers/90-003.pdf.

Fogel, L., Owens, A. J., & Walsh, M. J. (1966). *Artificial Intelligence through Simulated Evolution*. John Wiley & Sons. http://www.amazon.com/Artificial-Intelligence-through-Simulated-Evolution/dp/B0000CNARU.

Gilbert, W. (1986). Origin of life: The RNA world. Nature Publishing Group 319, 618 (1986) doi:10.1038/319618a0. http://www.nature.com/nature/journal/v319/n6055/abs/319618a0.html.

Heudin, J. (1995). Artificial life and evolutionary computing in machine perception. (pp. 418–428). Computer Architectures for Machine Perception, 1995. www.computer.org/csdl/proceedings/camp/1995/7134/00/71340418.pdf.

Holland, J. (1975). *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. University of Michigan Press. http://www.amazon.com/

Adaptation-Natural-Artificial-Systems-Introductory/dp/0262581116/ref=sr_1_1?s=books&ie=UTF8&qid=1433523944&sr=1-1&keywords=Holland%2C+J.+%281975%29.+Adaptation+in+Natural+and+Artificial+Systems.

Huxley, J. (1942). *Evolution: The Modern Synthesis*. Harper & Brothers Publishers. http://www.amazon.com/Evolution-Modern-Synthesis-Julian-Huxley/dp/0262513668.

Joyce, G. (2007). Forty years of in vitro evolution. (pp. 6420–6436). Angewandte Chemie Int. Ed 2007, 46. http://www.ncbi.nlm.nih.gov/pubmed/17634987.

Joyce, G. F., Deamer, D., & Fleischaker, G. (1994). *In Origins of Life: The Central Concepts*. Jones and Bartlett Publishers, Boston.

Koza, J. (1990). Genetic programming: A paradigm for genetically breeding populations of computer programs to solve problems. Stanford University Computer Science Department technical report STAN-CS-90-1314. http://www.genetic-programming.com/jkpdf/tr1314.pdf.

Langton, C. G. (1984). Self-reproduction in cellular automata. (pp. 135–144). Physica D: Nonlinear Phenomena 10(1-2). http://deepblue.lib.umich.edu/bitstream/handle/2027.42/24968/0000395.pdf?sequence=1&isAllowed=y.

Langton, C. G. (1989). Artificial Life. In C. G. Langton (Ed.) *Artificial life: the proceedings of an Interdisciplinary Workshop on the Synthesis and Simulation of Living Systems, held September, 1987, in Los Alamos, New Mexico. Santa Fe Institute studies in the sciences of complexity*, (pp. 1–48). Addison-Wesley, Redwood City, CA. http://www.amazon.com/Artificial-Life-Proceedings-Interdisciplinary-Simulation/dp/0201093561/ref=sr_1_3?s=books&ie=UTF8&qid=1433525422&sr=1-3&keywords=Artificial+Life+santa+fe.

Langton, C. G. (1995). *Artificial life: An Overview (Complex Adaptive Systems)*. The MIT Press. http://www.amazon.com/Artificial-Life-Overview-Complex-Adaptive/dp/0262621126/ref=sr_1_1?s=books&ie=UTF8&qid=1433524052&sr=1-1&keywords=langton+artificial+life+an+overview.

Lenski, R. (2011). Evolution in action: a 50,000-generation salute to Charles Darwin. (pp. 30–33). Microbe Vol. 6, Number 1. https://www.microbemagazine.org/index.php?option=com_content&view=article&id=3085:evolution-in-action-a-50000-generation-salute-to-charles-darwin&catid=697&Itemid=925.

Lukas, K., & Schmeck, H. (2009). A completely evolvable genotype-phenotype mapping for evolutionary robotics. Self-Adaptive and Self-Organizing Systems, 2009. SASO '09. http://www.aifb.kit.edu/images/0/09/PID937965.pdf.

Maley, C. C. (1994). The computational completeness of Ray's Tierran assembly language. (pp. 503–514). In: Artificial Life 3, ed. C. G. Langton. Addison-Wesley. ftp://theory.csail.mit.edu/people/cmaley/tierra/completeness.ps.Z.

Maynard Smith, J., & Szathmáry, E. (1995). *The Major Transitions in Evolution*. Oxford University Press. http://www.amazon.com/Major-Transitions-Evolution-Maynard-Smith/dp/019850294X/ref=sr_1_1?s=books&ie=UTF8&qid=1433525491&sr=1-1&keywords=The+Major+Transitions+in+Evolution.

McMullin, B. (2000). The Von Neumann self reproducing architecture, genetic relativism and evolvability. In *Evolvability Workshop at Artificial Life VII*. http://www.eeng.dcu.ie/~alife/talks/alife7/vn-evolvability/html-multi/.

McMullin, B. (2010). Evosym: Emergence and evolution of biological symbol systems project proposal. http://www.nwo.nl/en/research-and-results/research-projects/33/2300165233.html.

McMullin, B. (2012). Architectures for self reproduction: Abstractions, realisations and a research program. (pp. 83–90). University of Southern Denmark: Artificial Life 13. The MIT Press. https://mitpress.mit.edu/sites/default/files/titles/content/alife13/ch012.html.

Mitchell, M., & Forrest, S. (1994). Genetic algorithms and artificial life. (pp. 267–289). Artificial Life 1 (3). http://pdxscholar.library.pdx.edu/cgi/viewcontent.cgi?article=1003&context=compsci_fac.

Pesavento, U. (1995). An implementation of Von Neumann's self reproducing machine. *Artificial Life*, *2*(4), 337–354. http://www-users.york.ac.uk/~gt512/BIC/pesavento95.pdf.

Rasmussen, S. (1990). The coreworld: Emergence and evolution of cooperative structures in a computational chemistry. (pp. 111–134). Physica D: Nonlinear Phenomena, volume 42, Issues 1-3. http://www.sciencedirect.com/science/article/pii/0167278990900706.

Ray, T. (1991). Artificial life II, santa fe institute studies in the sciences of complexity, vol. XI, 371-408. Redwood City, CA: Addison-Wesley. *Artificial life II*, *10*, 371–408. http://www.amazon.com/Artificial-Institute-Sciences-Complexity-Proceedings/dp/0201525712/ref=sr_1_1?s=books&ie=UTF8&qid=1433524283&sr=1-1&keywords=artificial+life+II.

Ray, T. (1994). Evolution, complexity, entropy, and artificial reality. (pp. 239–263). Physica D: Nonlinear Phenomena. Volume 75, Issues 1-3. http://beaconcourse.pbworks.com/f/Ray1994+-+Tierra.pdf.

Ray, T. (1999). Some thoughts on evolvability. (Unpublished draft manuscrift)http://life.ou.edu/pubs/evolvability/.

Ray, T., Xu, C., Charrel, A., Kimezawa, T., Yoshikawa, T., Chaland, M., & Uffner, T. (2000). Tierra v6.02 documentation. Working Paper. Document (Tierra.doc) located within tgz file at http://life.ou.edu/tierra/source/Tierra6_02.tgz.

Sayama, H. (1999). Toward the realization of an evolving ecosystem on cellular automata. (pp. 254–257). Proceedings of the Fourth International Symposium on Artificial Life and Robotics. http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=DA9331850B8618964E408B2C3F1B50D6?doi=10.1.1.40.391&rep=rep1&type=pdf.

Schrödinger, E. (1944). *What is Life: With Mind and Matter and Autobiographical Sketches*. Cambridge University Press. http://www.amazon.com/What-Life-Autobiographical-Sketches-Classics/dp/1107604664/ref=sr_1_1?s=books&ie=UTF8&qid=1433524384&sr=1-1&keywords=what+is+life+schrodinger.

Shannon, C. (1948). A mathematical theory of communication. University of Illinois Press. Reprinted with corrections from The Bell System Technical Journal, Vol. 27, pp. 379–423, 623–656. http://worrydream.com/refs/Shannon%20-%20A%20Mathematical%20Theory%20of%20Communication.pdf.

Stuart, Y., Campbell, T., Hohenlohe, P., Reynolds, R., Revell, L., & Losos, J. (2014). Rapid evolution of a native species fllowing invasion by a congener. (pp. 463–466). Science Vol. 346 no. 6208. http://www.sciencemag.org/content/346/6208/463.abstract.

Takeuchi, N., Hogeweg, P., & Koonin, E. V. (2011). On the origin of DNA genomes: Evolution of the division of labor between template and catalyst in model replicator systems. PLoS Computational Biology 7(3): e1002024. doi:10.1371/journal.pcbi.1002024. http://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1002024.

Taylor, T. (1999). The Cosmos artificial life system. Department of Artificial Intelligence, University of Edinburgh. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.42.9606&rep=rep1&type=pdf.

Thatcher, J. (1964). Universality in the Von Neumann cellular model. (p. 100). University of Michigan, College of Literature, Science and the Arts. Techni-

cal Report. http://deepblue.lib.umich.edu/bitstream/handle/2027.42/7923/bad2800.0001.001.pdf?sequence=5&isAllowed=y.

Trevorrow, A., & Rokicki, T. (2015). Golly. http://golly.sourceforge.net.

Trut, L. (1999). Early canid domestication: The farm-fox experiment. American Scientist, Volume 87, Number 2, Page: 160. http://www.americanscientist.org/issues/pub/early-canid-domestication-the-farm-fox-experiment.

Turing, A. (1936). On computable numbers, with an application to the entscheidungsproblem. Proceedings of the London Mathematical Society, (Ser. 2, Vol. 43, 1937). http://www.dna.caltech.edu/courses/cs129/caltech_restricted/Turing_1936_IBID.pdf.

Von Neumann, J. (1948). *The General and Logical Theory of Automata*. Oxford, England: Wiley, xiv, 311 pp. https://www.cs.ucf.edu/~dcm/Teaching/COP5611Spring2010/vonNeumannSelfReproducingAutomata.pdf.

Von Neumann, J., & Burks, A. (1966). *Theory of Self reproducing Automata*. University of Illinois Press. http://www.amazon.com/Theory-Self-Reproducing-Automata-John-Neumann/dp/B0000CNFSD/ref=sr_1_1?s=books&ie=UTF8&qid=1433524616&sr=1-1&keywords=john+von+neumann+the+theory+of+Self+reproducing+Automata.

Webb, A., & Knowles, J. (2014). Studying the evolvability of self-encoding genotype-phenotype maps. Proceedings of the Fourteenth International Conference on the Synthesis and Simulation of Living Systems. http://mitpress.mit.edu/sites/default/files/titles/content/alife14/978-0-262-32621-6-ch014.pdf.

Wikipedia (2014). Tree data structure. http://en.wikipedia.org/wiki/Tree_data_structure. [Online; accessed 01-June-2015].

Appendix A

Creature Design

This section contains the raw data for the code of the creatures which are discussed throughout the body of this thesis.

A.1 vn_lut32_344 Code

Listing A.1: Von Neumann Creature: vn_lut32_344 code

```
1  nop1  ;Creature start template
2  nop1
3  nop1
4  nop1
5  nop1
6  adrb  ;Find creature start template address
7  nop0
8  nop0
9  nop0
10 nop0
11 nop0
12 subAAC   ;Subtract template size from start template address
13 pushA ;Push start address to stack
14 popB  ;Pop start address to Bx
15 adrf  ;Find  creature end template
16 nop0
17 nop0
18 nop0
19 nop0
20 nop1
21 incA  ;Increment to allow for dummy instruction
22 subCAB   ;Calculate total creature length
23 nop1
24 nop1
25 nop1
26 nop0
27 nop1
28 pushC ;Save creature length to stack
29 mal   ;Allocate offspring memory space
30 call  ;Save current IP location. Jump to decode subroutine
31 nop0
32 nop0
```

```
nop1
nop1
divide   ;Divide. Remove write protection to offspring memory image
popC  ;Pop creature length to Cx
jmpo  ;Jump to daughter allocation subroutine
nop0
nop0
nop0
nop1
nop0
ifz
nop1
nop1
nop1
nop0
nop0
pushA ;Push offspring start address in stack
popD  ;Pop offspring start address to Dx
adrf  ;Find start of LUT
nop0
nop0
nop1
nop0
nop1
pushA ;Push LUT start address to stack
popE  ;Pop LUT start address to Ex
adrf  ;Find genotype start address
nop0
nop0
nop1
nop0
nop0
pushA ;Pushg genotype start address to stack
popB  ;Popo genotype start address to Bx
adrf  ;Find creature end address
nop0
nop0
nop0
nop0
nop1
subAAC   ;Calculate genotype end address
subCAB   ;Calculate genotype length
nop1
nop1
nop1
nop0
nop0
movdi ;Copy g-symbol to Ax
add   ;Add LUT start address to Ax
movii2   ;Write symbol poinetd at by Ax to phenotype
incB  ;Increment genotype pointer
incC  ;Increment offspring phenotype pointer
decC  ;Decrement counter
ifz   ;If C!=0 jump to start of decode loop
jmpb  ;Else skip jmpb instruction
nop0
nop0
nop0
```

```
nop1
nop1
pushD ;Push offspring phenotype pointer to stack
adrf  ;Find genotype start address
nop0
nop0
nop1
nop0
nop0
pushA ;Push genotype start address to stack
popB  ;Popo genotype start address to Bx
adrf  ;Find creature end template
nop0
nop0
nop0
nop0
nop1
incA  ;Increment to include dummy instruction
subCAB   ;Calculate genotype length
popA  ;Pop offspring pointer to Ax
nop0
nop1
nop0
nop0
nop0
movii ;Copy g-symbol from parent to offspring
incA  ;Increment offspring pointer
incB  ;Increment parent genotype pointer
decC  ;Decrement counter
ifz   ;If C!=0 jump to start of copy loop
jmpb  ;Else skip jmpb instruction
nop1
nop0
nop1
nop1
nop1
ret   ;Return IP to address stored in stack
nop1  ;Look-up table (LUT) start template
nop1
nop0
nop1
nop0
pushA ;LOOK-UP TABLE
subCAB
nop1
nop0
popE
ret
popB
divide
subAAC
popC
incA
mal
pushD
movii2
jmpb
add
```

```
ifz
pushB
adrf
adrb
movdi
adro
movii
incC
call
decC
movBA
incB
popA
pushC
popD
jmpo  :LOOK-UP TABLE END
nop1  ;Genotype start template
nop1
nop0
nop1
nop1
add
add
add
add
add
jmpo
ifz
ifz
ifz
ifz
ifz
decC
nop0
incA
popE
ifz
ifz
ifz
ifz
add
pushA
nop1
add
add
add
ifz
add
adrf
pushB
movdi
ifz
ifz
add
add
incB
incC
divide
```

```
ifz
ifz
ifz
add
ifz
popC
add
add
add
ifz
ifz
nop0
mal
popE
ifz
ifz
add
ifz
add
nop0
subCAB
popE
ifz
ifz
add
ifz
ifz
nop0
incA
popE
ifz
ifz
ifz
ifz
add
decC
nop1
add
add
add
ifz
ifz
jmpb
popB
pushD
adro
movBA
movii2
popC
popA
ifz
ifz
ifz
add
add
pushC
popE
ifz
```

```
ifz
add
ifz
ifz
nop0
incA
popE
ifz
ifz
ifz
ifz
add
pushA
nop1
adrb
ifz
add
ifz
ifz
ifz
ret
pushA
adro
movii2
popC
popA
add
ifz
add
add
add
subAAC
add
add
ifz
add
ifz
nop0
nop1
add
ifz
subCAB
subAAC
incA
incB
decC
incC
pushA
pushB
pushC
pushD
popA
popB
popC
popD
popE
jmpo
jmpb
```

```
call
ret
movBA
movdi
movii2
movii
adro
adrb
adrf
mal
divide
add
add
ifz
add
add   ;GENOTYPE END
nop1  ;Genotype end template
nop1
nop1
nop1
nop0
ifz   ;Dummy instruction
```

A.2 vn_lut32_311 Code

Listing A.2: Von Neumann Creature: vn_lut32_311 code

```
adrf  ; Search for start of genotype
nop1  ; Creature start template
nop0
nop1
nop0
pushA ; Push genotype start address to stack
popB  ; Pop genotype start address to Bx
adrf  ; Find  creature end template
nop1
nop1
nop1
nop0
pushC ; Push creature end template size to stack
subCAB   ; Subtract to find the length of the genotype + end template
pushC ; Push genotype length to stack
popA  ; Pop genotype length to Ax
add2  ; Double the value in Ax
incA  ; Increment to allow for dummy instruction
popB  ; Pop creature end template to Bx
subCAB   ; Subtract Bx from Ax to find length of twice the genotype, plus one
    end template
nop0
nop0
nop1
nop0
pushC ; Save length to stack
mal   ; Allocate offspring memory space
call  ; Save current IP location. Jump to decode subroutine
nop0
nop0
nop1
nop1
divide   ; Divide. Remove write protection to offspring memory image
popC  ; Pop creature length to Cx
jmpo  ; Jump to daughter allocation subroutine
nop1
nop1
nop0
nop1
ifz
nop1
nop1
nop0
nop0
pushA ; Push offspring start address in stack
popD  ; Pop offspring start address to Dx
adrf  ; Find start of LUT
nop0
nop1
nop0
nop0
pushA ; Push LUT start address to stack
popE  ; Pop LUT start address to Ex
adrf  ; Find genotype start address
```

```
nop1
nop0
nop1
nop0
pushA ; Push genotype start address to stack
popB  ; Pop genotype start address to Bx
adrf  ; Find creature end address
nop1
nop1
nop1
nop0
subAAC   ; Calculate genotype end address
subCAB   ; Calculate genotype length
nop0
nop1
nop1
nop0
movdi ; Copy g-symbol to Ax
add   ; Add LUT start address to Ax
movii2   ; Write symbol poinetd at by Ax to phenotype
incB  ; Increment genotype pointer
incC  ; Increment offspring phenotype pointer
decC  ; Decrement counter
ifz   ; If C!=0 jump to start of decode loop
jmpb  ; Else skip jmpb instruction
nop1
nop0
nop0
nop1
pushD ; Push offspring phenotype pointer to stack
adrf  ; Find genotype start address
nop1
nop0
nop1
nop0
pushA ; Push genotype start address to stack
popB  ; Popo genotype start address to Bx
adrf  ; Find creature end template
nop1
nop1
nop1
nop0
incA  ; Increment to include dummy instruction
subCAB   ; Calculate genotype length
popA  ; Pop offspring pointer to Ax
nop0
nop1
nop1
nop1
movii ; Copy g-symbol from parent to offspring
incA  ; Increment offspring pointer
incB  ; Increment parent genotype pointer
decC  ; Decrement counter
ifz   ; If C!=0 jump to start of copy loop
jmpb  ; Else skip jmpb instruction
nop1
nop0
nop0
```

```
nop0
ret   ; Return IP to address stored in stack
nop1  ; Look-up table (LUT) start template
nop0
nop1
nop1
pushA ; LOOK-UP TABLE
subCAB
nop1
nop0
popE
ret
popB
divide
subAAC
popC
incA
mal
pushD
movii2
jmpb
add
ifz
pushB
adrf
adrb
movdi
adro
movii
incC
call
decC
add2
incB
popA
pushC
popD
jmpo  : LOOK-UP TABLE END
nop0  ; Genotype start template
nop1
nop0
nop1
popE  ; GENOTYPE
add
ifz
add
ifz
nop0
incA
popE
add
add
add
ifz
adrf
nop1
adrf
adrb
```

```
movii
pushA
incA
nop1
ifz
ifz
add
ifz
adrf
pushB
movdi
ifz
ifz
add
add
incB
incC
divide
add
add
ifz
add
popC
add
add
ifz
ifz
nop0
mal
popE
ifz
add
ifz
ifz
nop0
subCAB
popE
add
ifz
add
ifz
nop0
incA
popE
add
add
add
ifz
decC
nop1
ifz
add
add
ifz
jmpb
popB
pushD
adro
```

```
add2
movii2
popC
popA
add
ifz
ifz
add
pushC
popE
add
ifz
add
ifz
nop0
incA
popE
add
add
add
ifz
pushA
nop1
adrb
ifz
add
add
add
ret
pushA
adro
movii2
popC
popA
add
ifz
ifz
ifz
subAAC
add
ifz
add
add
nop0
nop1
add
ifz
subCAB
subAAC
incA
incB
decC
incC
pushA
pushB
pushC
pushD
popA
```

```
popB
popC
popD
popE
jmpo
jmpb
call
ret
add2
movdi
movii2
movii
adro
adrb
adrf
mal
divide
ifz
add
ifz
add   ;GENOTYPE END
nop0  ;Genotype end template
nop0
nop0
nop1
ifz   ;Dummy instruction
```

A.3 vn_lut32_413 Code

Listing A.3: PC Immune Creature: vn_lut32_413 code

```
adrf  ; Search for start of genotype
nop1  ; Creature start template
nop0
nop0
nop0
pushA ; Push genotype start address to stack
popB  ; Pop genotype start address to Bx
adrf  ; Find  creature end template
nop1
nop1
nop1
nop0
pushC ; Push creature end template size to stack
subCAB   ; Subtract to find the length of the genotype + end template
pushC ; Push genotype length to stack
popA  ; Pop genotype length to Ax
add2  ; Double the value in Ax
incA  ; Increment to allow for dummy instruction
popB  ; Pop creature end template to Bx
subCAB   ; Subtract Bx from Ax to find length of twice the genotype, plus one
    end template
nop0
nop0
nop1
nop0
pushC ; Save length to stack
mal   ; Allocate offspring memory space
call  ; Save current IP location. Jump to decode subroutine
nop0
nop0
nop1
nop1
divide   ; Divide. Remove write protection to offspring memory image
popC  ; Pop creature length to Cx
jmpo  ; Jump to daughter allocation subroutine
nop1
nop1
nop0
nop1
ifz
nop1
nop1
nop0
nop0
pushA ; Push offspring start address in stack
popD  ; Pop offspring start address to Dx
adrf  ; Find start of LUT
nop0
nop1
nop0
nop0
pushA ; Push LUT start address to stack
popE  ; Pop LUT start address to Ex
adrf  ; Find genotype start address
```

```
nop1
nop0
nop1
nop0
pushA ; Push genotype start address to stack
popB  ; Pop genotype start address to Bx
adrf  ; Find creature end address
nop1
nop1
nop1
nop0
subAAC   ; Calculate genotype end address
subCAB   ; Calculate genotype length
nop0
nop1
nop1
nop0
movdi ; Copy g-symbol to Ax
add   ; Add LUT start address to Ax
movii2   ; Write symbol poinetd at by Ax to phenotype
incB  ; Increment genotype pointer
incC  ; Increment offspring phenotype pointer
decC  ; Decrement counter
ifz   ; If C!=0 jump to start of decode loop
jmpb  ; Else skip jmpb instruction
nop1
nop0
nop0
nop1
pushD ; Push offspring phenotype pointer to stack
adrf  ; Find genotype start address
nop1
nop0
nop1
nop0
pushA ; Push genotype start address to stack
popB  ; Popo genotype start address to Bx
adrf  ; Find creature end template
nop1
nop1
nop1
nop0
incA  ; Increment to include dummy instruction
subCAB   ; Calculate genotype length
popA  ; Pop offspring pointer to Ax
nop0
nop1
nop1
nop1
movii ; Copy g-symbol from parent to offspring
incA  ; Increment offspring pointer
incB  ; Increment parent genotype pointer
decC  ; Decrement counter
ifz   ; If C!=0 jump to start of copy loop
jmpb  ; Else skip jmpb instruction
nop1
nop0
nop0
```

```
nop0
ret   ; Return IP to address stored in stack
nop1  ; Look-up table (LUT) start template
nop0
nop1
nop1
pushA ; LOOK-UP TABLE
subCAB
nop1
nop0
popE
ret
popB
divide
subAAC
popC
incA
mal
pushD
movii2
jmpb
add
ifz
pushB
adrf
adrb
movdi
adro
movii
incC
call
decC
add2
incB
popA
pushC
popD
jmpo  : LOOK-UP TABLE END
nop0  ; Genotype start template
nop1
nop0
nop1
popE  ; GENOTYPE
add
ifz
ifz
ifz
nop0
incA
popE
add
add
add
ifz
adrf
nop1
adrf
adrb
```

```
movii
pushA
incA
nop1
ifz
ifz
add
ifz
adrf
pushB
movdi
ifz
ifz
add
add
incB
incC
divide
add
add
ifz
add
popC
add
add
ifz
ifz
nop0
mal
popE
ifz
add
ifz
ifz
nop0
subCAB
popE
add
ifz
add
ifz
nop0
incA
popE
add
add
add
ifz
decC
nop1
ifz
add
add
ifz
jmpb
popB
pushD
adro
```

```
add2
movii2
popC
popA
add
ifz
ifz
add
pushC
popE
add
ifz
add
ifz
nop0
incA
popE
add
add
add
ifz
pushA
nop1
adrb
ifz
add
add
add
ret
pushA
adro
movii2
popC
popA
add
ifz
ifz
ifz
subAAC
add
ifz
add
add
nop0
nop1
add
ifz
subCAB
subAAC
incA
incB
decC
incC
pushA
pushB
pushC
pushD
popA
```

```
popB
popC
popD
popE
jmpo
jmpb
call
ret
add2
movdi
movii2
movii
adro
adrb
adrf
mal
divide
ifz
add
ifz
add   ;GENOTYPE END
nop0  ;Genotype end template
nop0
nop0
nop1
ifz   ;Dummy Instruction
nop0  ; JUNK DATA
nop0
nop0
nop0
nop0
nop0
nop0
nop0
nop0
nop0
nop0
nop0
nop0
nop0
nop0
nop0
nop0
nop0
nop0
nop0
nop0
nop0
nop0
nop0
nop0
nop0
nop0
nop0
nop0
nop0
nop0
nop0
```

```
344 nop0
345 nop0
346 nop0
347 nop0
348 nop0
349 nop0
350 nop0
351 nop0
352 nop0
353 nop0
354 nop0
355 nop0
356 nop0
357 nop0
358 nop0
359 nop0
360 nop0
361 nop0
362 nop0
363 nop0
364 nop0
365 nop0
366 nop0
367 nop0
368 nop0
369 nop0
370 nop0
371 nop0
372 nop0
373 nop0
374 nop0
375 nop0
376 nop0
377 nop0
378 nop0
379 nop0
380 nop0
381 nop0
382 nop0
383 nop0
384 nop0
385 nop0
386 nop0
387 nop0
388 nop0
389 nop0
390 nop0
391 nop0
392 nop0
393 nop0
394 nop0
395 nop0
396 nop0
397 nop0
398 nop0
399 nop0
400 nop0
401 nop0
```

```
nop0
nop0
nop0
nop0
nop0
nop0
nop0
nop0
nop0
nop0
nop0
nop0
```

A.4 vn_lut64_316 Code

Listing A.4: Von Neumann Creature: vn_lut64_316 code

```
adrf  ; Search for start of genotype
nop1  ; Creature start template
nop0
nop1
nop0
pushA ; Push genotype start address to stack
popB  ; Pop genotype start address to Bx
adrf  ; Find  creature end template
nop1
nop1
nop1
pushC ; Push creature end template size to stack
subCAB   ; Subtract to find the length of the genotype + end template
pushC ; Push genotype length to stack
popA  ; Pop genotype length to Ax
shlA  ; Double the value in Ax
incA  ; Increment to allow for dummy instruction
popB  ; Pop creature end template to Bx
subCAB   ; Subtract Bx from Ax to find length of twice the genotype, plus one
    end template
nop0
nop0
pushC ; Save length to stack
mal   ; Allocate offspring memory space
call  ; Save current IP location. Jump to decode subroutine
nop0
divide   ; Divide. Remove write protection to offspring memory image
popC  ; Pop creature length to Cx
jmpb  ; Jump to daughter allocation subroutine
nop1
nop1
pushA ; Push offspring start address in stack
popD  ; Pop offspring start address to Dx
adrf  ; Find start of LUT
nop1
nop0
nop0
pushA ; Push LUT start address to stack
popE  ; Pop LUT start address to Ex
adrf  ; Find genotype start address
nop1
nop0
nop1
nop0
pushA ; Push genotype start address to stack
popB  ; Pop genotype start address to Bx
adrf  ; Find creature end address
nop1
nop1
nop1
subAAC   ; Calculate genotype end address
subCAB   ; Calculate genotype length
nop0
movAb ; Copy g-symbol to Ax
```

```
addAAE   ; Add LUT start address to Ax
movda ; Write symbol pointed at by Ax to phenotype
incB  ; Increment genotype pointer
incD  ; Increment offspring phenotype pointer
decC  ; Decrement counter
ifz   ; If C!=0 jump to start of decode loop
jmpb  ; Else skip jmpb instruction
nop1
pushD ; Push offspring phenotype pointer to stack
adrf  ; Find genotype start address
nop1
nop0
nop1
nop0
pushA ; Push genotype start address to stack
popB  ; Popo genotype start address to Bx
adrf  ; Find creature end template
nop1
nop1
nop1
incA  ; Increment to include dummy instruction
subCAB   ; Calculate genotype length
popA  ; Pop offspring pointer to Ax
nop1
movab ; Copy g-symbol from parent to offspring
incA  ; Increment offspring pointer
incB  ; Increment parent genotype pointer
decC  ; Decrement counter
ifnz  ; If C=0 jump to start of copy loop
jmpb  ; Else skip jmpb instruction
nop0
ret   ; Return IP to address stored in stack
nop0  ; Look-up table (LUT) start template
nop1
nop1
nop0  ; LOOK_UP TABLE START
nop1
nop2
nop3
ifnz
nop4
addAAE
nop5
subCAB
nop6
subAAC
nop7
incA
nop8
incB
nop9
decC
nop10
incD
nop11
pushA
nop12
nop13
```

```
nop14
pushC
nop15
pushD
nop16
popA
nop17
popB
nop18
popC
nop19
popD
nop20
popE
nop21
nop22
nop23
jmpb
nop24
adrf
nop25
nop26
nop27
call
nop28
movAb
nop29
movda
nop30
movab
nop31
ret
nop32
nop33
shlA
nop34
nop35
mal
nop36
nop37
divide   ; LOOK-UP TABLE END
nop0  ; Genotype start template
nop1
nop0
nop1
adrf  ; GENOTYPE
nop1
nop0
nop1
nop0
pushA
popB
adrf
nop1
nop1
nop1
pushC
subCAB
```

```
pushC
popA
shlA
incA
popB
subCAB
nop0
nop0
pushC
mal
call
nop0
divide
popC
jmpb
nop1
nop1
pushA
popD
adrf
nop1
nop0
nop0
pushA
popE
adrf
nop1
nop0
nop1
nop0
pushA
popB
adrf
nop1
nop1
nop1
subAAC
subCAB
nop0
movAb
addAAE
movda
incB
incD
decC
ifnz
jmpb
nop1
pushD
adrf
nop1
nop0
nop1
nop0
pushA
popB
adrf
nop1
```

```
nop1
nop1
incA
subCAB
popA
nop1
movab
incA
incB
decC
ifnz
jmpb
nop0
ret
nop0
nop1
nop1
nop0
nop1
nop2
nop3
ifnz
nop4
addAAE
nop5
subCAB
nop6
subAAC
nop7
incA
nop8
incB
nop9
decC
nop10
incD
nop11
pushA
nop12
nop13
nop14
pushC
nop15
pushD
nop16
popA
nop17
popB
nop18
popC
nop19
popD
nop20
popE
nop21
nop22
nop23
jmpb
```

```
nop24
adrf
nop25
nop26
nop27
call
nop28
movAb
nop29
movda
nop30
movab
nop31
ret
nop32
nop33
shlA
nop34
nop35
mal
nop36
nop37
divide
nop0
nop1
nop0
nop1  ; GENOTYPE END
nop0  ; Creature end template
nop0
nop0
ifz   ;Dummy instruction
```

A.5 vn_tt128_758 Code

Listing A.5: Von Neumann Creature: vn_tt128_758 code

```
adrf  ; Search for start of genotype (SELF INSPECTION)
nop1  ; Creature start template
nop0
nop1
nop0
pushA ; Push genotype start address to stack
popB  ; Pop genotype start address to Bx
adrf  ; Find creature end template
nop1
nop1
nop1
pushC ; Push creature end template size to stack
subCAB   ; Subtract to find the length of the genotype + end template
pushC ; Push genotype length to stack
popA  ; Pop genotype length to Ax
addAAA   ; Double the value in Ax
incA  ; Increment to allow for dummy instruction
popB  ; Pop creature end template to Bx
subCAB   ; Subtract Bx from Ax to find length of twice the genotype, plus one
    end template
nop0  ; (MEMORY ALLOCATE AND DE-ALLOCATE)
nop0
pushC ; Save length to stack
mal   ; Allocate offspring memory space
call  ; Save current IP location. Jump to decode subroutine
nop0
divide   ; Divide. Remove write protection to offspring memory image
popC  ; Pop creature length to Cx
jmpb  ; Jump to daughter allocation subroutine
nop1
nop1
pushA ; Push offspring start address in stack (DECODE SUBROUTINE)
popD  ; Pop offspring start address to Dx
adrf  ; Find start of TT
nop1
nop0
nop0
pushA ; Push TT start address to stack
popF  ; Pop TT start address to Fx
adrf  ; Find genotype end address
nop1
nop1
nop1
subAAC   ; Calculate genotype end address
pushA ; Push genotype end address to stack
popB  ; Pop genotype end address to Bx
adrf  ; Search for start of genotype
nop1
nop0
nop1
nop0
pushA ; Push genotype start address to stack
popE  ; Pop genotype start address to Ex
subCBA   ; Calculate genotype length, put in Cx
```

```
pushF ; Push TT start address to stack
popB  ; Popo TT start address to Bx
nop1
nop1
nop0
nop1
movAe ; Copy g-symbol numerical value to Ax
pushB ; Push TT start address to stack
movBb ; Copy number in TT to Bx
ifequal  ; If Ax != Bx skip next instruction
jmpf  ; Else jump to template [11]
nop0
nop0
popB  ; Pop TT start address to Bx
incB  ; Increase Bx (pointer to TT)
incB  ; Increase Bx (pointer to TT)
jmpb  ; Jumb back to start of decoding subroutine
nop0
nop0
nop1
ifz
nop1
nop1
popB  ; Pop current TT address to Bx
incB  ; Increment to point to associated key-value
movdb ; Copy key value to offspring phenotype
incD  ; Increase offspring phenotype pointer
incE  ; Increase parent genotype pointer
decC  ; Decrease counter
pushF ; Push TT start to stack
popB  ; Popo TT start to Bx
ifnz  ; If Cx=0, skip jump instruction
jmpb  ; Else, jump back to start of decode subroutine
nop0
nop0
nop1
nop0  ; (COPY SUBROUTINE)
pushD ; Push offspring pointer to stack
adrf  ; Search for start of genotype
nop1
nop0
nop1
nop0
pushA ; Push genotype start address to stack
popB  ; Pop genotype start address to Bx
adrf  ; Find genotype end template
nop1
nop1
nop1
incA  ; Increment to include dummy instruction
subCAB   ; Subtract to find genotype length
popA  ; Pop offspring pointer to Ax
nop1
movab ; Copy g-symbol to offspring
incA  ; Increment offspring pointer
incB  ; Increment g-symbol pointer
decC  ; Decrement counter
ifnz  ;If Cx!=0, skip jump instruction
```

```
jmpb  ; Else, jump to start of copy subroutine
nop0
ret   ; Return IP to address stored in stack
nop0  ; Translation table (TT) start template [011]
nop1
nop1
nop1  ; (TRANSLATION TABLE START)
nop1
nop0
nop0
popB
popB
adrf
adrf
pushA
pushA
incB
incB
jmpb
jmpb
subCAB
subCAB
pushC
pushC
incA
incA
pushF
pushF
decC
decC
ifnz
ifnz
popA
popA
ifz
ifz
pushD
pushD
ifequal
ifequal
movab
movab
movBb
movBb
movAe
movAe
subCBA
subCBA
popF
popF
jmpf
jmpf
incD
incD
popE
popE
subAAC
subAAC
```

```
incE
incE
mal
mal
addAAA
addAAA
popD
popD
popC
popC
ret
ret
movdb
movdb
call
call
pushB
pushB
divide
divide
decF
decF
moveA
moveA
movac
movac
subCEB
subCEB
subBFC
subBFC
subCDB
subCDB
movAb
movAb
movda
movda
movce
movce
movbc
movbc
addCCF
addCCF
subBBC
subBBC
jmpo
jmpo
pushE
pushE
movdf
movdf
addAAD
addAAD
subBDC
subBDC
movcd
movcd
addBBF
addBBF
```

```
subBEA
subBEA
movbd
movbd
subCFB
subCFB
subABC
subABC
movae
movae
addCCA
addCCA
movBa
movBa
incC
incC
movdA
movdA
subAAB
subAAB
subAFC
subAFC
movAd
movAd
movaB
movaB
subADC
subADC
subACB
subACB
subADB
subADB
decA
decA
subCFA
subCFA
movbe
movbe
subAEC
subAEC
moveB
moveB
movcA
movcA
movad
movad
subCEA
subCEA
movde
movde
adrb
adrb
addBBE
addBBE
subAFB
subAFB
subAEB
subAEB
```

```
addAAF
addAAF
movcb
movcb
subBBA
subBBA
decB
decB
movbf
movbf
addBBD
addBBD
addAAE
addAAE
movBc
movBc
subCCA
subCCA
movca
movca
movcB
movcB
incF
incF
addBBC
addBBC
subCDA
subCDA
movbA
movbA
zero
zero
movAf
movAf
movBd
movBd
movaf
movaf
subCCB
subCCB
addCCB
addCCB
movfA
movfA
subBCA
subBCA
movAa
movAa
addBBB
addBBB
adro
adro
decD
decD
movdc
movdc
addCCE
addCCE
```

```
movBe
movBe
movba
movba
movdB
movdB
addBBA
addBBA
movfB
movfB
addAAC
addAAC
decE
decE
movBf
movBf
addCCD
addCCD
subBAC
subBAC
movAc
movAc
addAAB
addAAB
subBEC
subBEC
subBDA
subBDA
movcf
movcf ; TRANSLATION TABLE END
nop0  ; Genotype start template
nop1
nop0
nop1
adrf  ; (GENOTYPE)
nop1
nop0
nop1
nop0
pushA
popB
adrf
nop1
nop1
nop1
pushC
subCAB
pushC
popA
addAAA
incA
popB
subCAB
nop0
nop0
pushC
mal
call
```

```
nop0
divide
popC
jmpb
nop1
nop1
pushA
popD
adrf
nop1
nop0
nop0
pushA
popF
adrf
nop1
nop1
nop1
subAAC
pushA
popB
adrf
nop1
nop0
nop1
nop0
pushA
popE
subCBA
pushF
popB
nop1
nop1
nop0
nop1
movAe
pushB
movBb
ifequal
jmpf
nop0
nop0
popB
incB
incB
jmpb
nop0
nop0
nop1
ifz
nop1
nop1
popB
incB
movdb
incD
incE
decC
```

```
460 pushF
461 popB
462 ifnz
463 jmpb
464 nop0
465 nop0
466 nop1
467 nop0
468 pushD
469 adrf
470 nop1
471 nop0
472 nop1
473 nop0
474 pushA
475 popB
476 adrf
477 nop1
478 nop1
479 nop1
480 incA
481 subCAB
482 popA
483 nop1
484 movab
485 incA
486 incB
487 decC
488 ifnz
489 jmpb
490 nop0
491 ret
492 nop0
493 nop1
494 nop1
495 nop1
496 nop1
497 nop0
498 nop0
499 popB
500 popB
501 adrf
502 adrf
503 pushA
504 pushA
505 incB
506 incB
507 jmpb
508 jmpb
509 subCAB
510 subCAB
511 pushC
512 pushC
513 incA
514 incA
515 pushF
516 pushF
517 decC
```

```
decC
ifnz
ifnz
popA
popA
ifz
ifz
pushD
pushD
ifequal
ifequal
movab
movab
movBb
movBb
movAe
movAe
subCBA
subCBA
popF
popF
jmpf
jmpf
incD
incD
popE
popE
subAAC
subAAC
incE
incE
mal
mal
addAAA
addAAA
popD
popD
popC
popC
ret
ret
movdb
movdb
call
call
pushB
pushB
divide
divide
decF
decF
moveA
moveA
movac
movac
subCEB
subCEB
subBFC
```

```
subBFC
subCDB
subCDB
movAb
movAb
movda
movda
movce
movce
movbc
movbc
addCCF
addCCF
subBBC
subBBC
jmpo
jmpo
pushE
pushE
movdf
movdf
addAAD
addAAD
subBDC
subBDC
movcd
movcd
addBBF
addBBF
subBEA
subBEA
movbd
movbd
subCFB
subCFB
subABC
subABC
movae
movae
addCCA
addCCA
movBa
movBa
incC
incC
movdA
movdA
subAAB
subAAB
subAFC
subAFC
movAd
movAd
movaB
movaB
subADC
subADC
subACB
```

```
subACB
subADB
subADB
decA
decA
subCFA
subCFA
movbe
movbe
subAEC
subAEC
moveB
moveB
movcA
movcA
movad
movad
subCEA
subCEA
movde
movde
adrb
adrb
addBBE
addBBE
subAFB
subAFB
subAEB
subAEB
addAAF
addAAF
movcb
movcb
subBBA
subBBA
decB
decB
movbf
movbf
addBBD
addBBD
addAAE
addAAE
movBc
movBc
subCCA
subCCA
movca
movca
movcB
movcB
incF
incF
addBBC
addBBC
subCDA
subCDA
movbA
```

```
movbA
zero
zero
movAf
movAf
movBd
movBd
movaf
movaf
subCCB
subCCB
addCCB
addCCB
movfA
movfA
subBCA
subBCA
movAa
movAa
addBBB
addBBB
adro
adro
decD
decD
movdc
movdc
addCCE
addCCE
movBe
movBe
movba
movba
movdB
movdB
addBBA
addBBA
movfB
movfB
addAAC
addAAC
decE
decE
movBf
movBf
addCCD
addCCD
subBAC
subBAC
movAc
movAc
addAAB
addAAB
subBEC
subBEC
subBDA
subBDA
movcf
```

```
750 movcf
751 nop0
752 nop1
753 nop0
754 nop1   ; GENOTYPE END
755 nop0   ; Creature end template
756 nop0
757 nop0
758 ifz    ;Dummy instruction
```

A.6 0035aaa Code

This creature is the result of a pathological constructor, which simply executes the `ret` instruction and redirects its CPU to location zero in the soup. The remaining instructions are never executed.

Listing A.6: Pathological Creature: 0035aaa code

```
ret   ; Redirect instruction pointer to absolute address zero.
nop1  ; The instructions from this point onwards are never accessed.
ret
divide
popC
adrf
pushD
adrb
adrf
pushA
pushA
subCAB
pushA
subCAB
subCAB
movdi
add
movii2
incB
incC
call
pushC
mal
call
nop0
nop0
nop1
nop0
nop1
nop1
nop0
nop0
nop0
nop1
nop1
```

A.7 0669aaa Code

This pathological constructor is an offspring of vn_lut32_311, which received quite a large segment insertion. A segment 358 symbols long was inserted into the middle of the look-up table. This segment insertion contained strings of `nops`. As the creature attempted to search for the addresses templates which usually mark the start and finish of the genotype, the creature instead finds the addressed of templates within the segment insertion. The creature then calculates the offspring as being of length 35 and proceeds to create malfunctioning creatures of that length.

Listing A.7: Pathological Constructor: 0669aaa code

```
1  adrf  ; Search for template [0101]
2  nop1
3  nop0
4  nop1
5  nop0
6  pushA ; Push [0101] address to stack
7  popB  ; Pop [0101] address to Bx
8  adrf  ; Search for template [0001]
9  nop1
10 nop1
11 nop1
12 nop0
13 pushC ; Push [0001] template size to stack
14 subCAB   ; Subtract to find the length between template [0101] and [0001].
15 pushC ; Push length to stack (This length is 20 instead of genotype length)
16 popA  ; Pop length to Ax
17 add2  ; Double value in Ax
18 incA  ; Increment Ax
19 popB  ; Pop [0001] template size to Bx
20 subCAB   ; Subtract Bx from Ax
21 nop0
22 nop0
23 nop1
24 nop0
25 pushC ; Save length to stack
26 mal   ; Allocate offspring memory space
27 call  ; Save current IP location. Jump to decode subroutine
28 nop0
29 nop0
30 nop1
31 nop1
32 divide   ; Divide. Remove write protection to offspring memory image
33 popC  ; Pop creature length to Cx
34 jmpo  ; Jump to daughter allocation subroutine
35 nop1
36 nop1
37 nop0
38 nop1
39 ifz
40 nop1
41 nop1
42 nop0
43 nop0
44 pushA ; Push offspring start address in stack
```

```
popD  ; Pop offspring start address to Dx
adrf  ; Find start of LUT
nop0
nop1
nop0
nop0
pushA ; Push LUT start address to stack
popE  ; Pop LUT start address to Ex
adrf  ; Find genotype start address
nop1
nop0
nop1
nop0
pushA ; Push [0101] address to stack
popB  ; Pop [0101] address to Bx
adrf  ; Find [0001] address
nop1
nop1
nop1
nop0
subAAC   ; Subtract the template size (4) from [0101]
subCAB   ; Subtract the position of template [0001] from Ax.
nop0
nop1
nop1
nop0
movdi ; Copy g-symbol to Ax
add   ; Add LUT start address to Ax
movii2   ; Write symbol poinetd at by Ax to phenotype
incB  ; Increment genotype pointer
incC  ; Increment offspring phenotype pointer
decC  ; Decrement counter
ifz   ; If C!=0 jump to start of decode loop
jmpb  ; Else skip jmpb instruction
nop1
nop0
nop0
nop1
pushD ; Push offspring phenotype pointer to stack
adrf  ; Find genotype start address
nop1
nop0
nop1
nop0
pushA ; Push genotype start address to stack
popB  ; Popo genotype start address to Bx
adrf  ; Find creature end template
nop1
nop1
nop1
nop0
incA  ; Increment to include dummy instruction
subCAB   ; Calculate genotype length
popA  ; Pop offspring pointer to Ax
nop0
nop1
nop1
nop1
```

```
movii ; Copy g-symbol from parent to offspring
incA  ; Increment offspring pointer
incB  ; Increment parent genotype pointer
decC  ; Decrement counter
ifz   ; If C!=0 jump to start of copy loop
jmpb  ; Else skip jmpb instruction
nop1
nop0
nop0
nop0
ret   ; Return IP to address stored in stack
nop1  ; Look-up table (LUT) start template
nop0
nop1
nop1
pushA ; LOOK-UP TABLE
subCAB
nop1
nop0
popE
ret
popB
divide
subAAC
popC
incA
mal
pushD
movii2
jmpb
add
ifz
pushB; ***  This is an insertion perturbation of 358 symbols ***
nop0; These symbols never get executed,
popA; however, the string of nops affect the
popE; template addressing and result in
adrf; the creature calculating its offspring as 35 symbols long
nop1
nop0
ret
ret
movdi
movii2
movii
adro
adrb
adrf
mal
divide
ifz
add
ifz
add
nop0
popB
add
popE
pushA
```

```
pushA
subCAB
pushA
pushD
popD
adro
pushA
pushA
subCAB
incA
nop1
ifz
ifz
decC
add2
incB
popA
pushC
popD
jmpo
nop0; Template [0101]
nop1
nop0
nop1
movdi
add
movii2
incB
incC
call
pushC
mal
call
nop0
nop0
nop1
nop0
nop1
nop1
nop0; Template [0001]
nop0
nop0
nop1
nop1
nop0
nop0
nop1
nop0
nop0
nop0
popD
adrf
nop0
nop1
nop0
nop0
pushA
popE
```

```
popE
popE
add
adrf
movii
nop0
nop1
nop1
nop1
nop0
pushC
nop1
nop1
nop0
nop0
adrf
nop1
nop1
nop1
popB
nop1
nop0
nop0
nop0
nop1
nop0
adrf
ifz
adrf
ifz
nop1
pushC
nop0
adrf
adrf
adrf
ifz
nop0
nop1
nop0
pushC
adrb
movdi
adro
movii
incC
call
decC
add
add
incB
incC
divide
add
add
ifz
add
popC
```

```
add
add
ifz
ifz
nop0
ifz
nop1
nop1
nop0
nop0
pushA
popD
adrf
nop0
nop1
nop0
nop0
pushA
popE
adrf
nop1
nop0
nop1
nop0
pushA
popB
adrf
nop1
nop1
nop1
nop0
subAAC
subCAB
nop0
nop1
nop1
nop0
movdi
adro
add
ret
popE
adrf
nop1
adrf
adrb
call
nop1
decC
divide
popC
subAAC
nop0
movdi
subCAB
pushA
pushA
subCAB
```

```
movii2
pushC
subCAB
nop1
add
ifz
ifz
add
pushC
popE
add
ifz
popD
jmpo
nop0
popB
nop0
nop0
nop0
nop1
decC
nop0
nop1
nop0
nop0
nop1
nop1
nop0
mal
call
nop1
nop1
nop0
pushA
nop0
nop1
nop1
nop0
nop0
popB
nop0
nop1
nop0
nop1
pushA
nop0
nop1
nop1
nop1
nop1
nop1
mal
divide
ifz
add
ifz
add
nop0
```

```
nop0
nop1
ifz
ifz
adrf
nop1
nop0
nop1
nop0
pushA
popB
adrf
nop1
nop1
nop1
nop0
pushC
subCAB
pushC
popA
add2
incA
popB
subCAB
nop0
nop0
nop1
nop0
pushC
mal
call
nop0
nop0
nop1
nop0
nop1
nop1
nop0
nop0
nop0
nop1
nop1
nop0
nop0
nop1
nop0
nop0
nop0
nop0
nop1
nop1
pushA
nop0
nop1
nop1
nop1
nop1
popB
```

```
451 add
452 adrf
453 movii
454 add
455 nop0
456 incA
457 adrf
458 popD
459 adrf
460 add
461 adrf
462 add
463 add
464 adrf
465 popC
466 popE
467 adrf
468 add
469 adrf
470 add
471 adrf
472 add
473 popE
474 adrf
475 adrf
476 adrf
477 add
478 popE
479 adrf
480 nop1
481 adrf
482 adrb
483 call
484 nop1
485 decC
486 divide
487 popC
488 subAAC
489 nop0
490 movdi
491 subCAB
492 pushA
493 pushA; *** Insertion Ends ***
494 adrf
495 adrb
496 movdi
497 adro
498 movii
499 incC
500 call
501 decC
502 add2
503 incB
504 popA
505 pushC
506 popD
507 jmpo  : LOOK-UP TABLE END
508 nop0  ; Genotype start template
```

```
509 nop1
510 nop0
511 nop1
512 popE  ; GENOTYPE
513 add
514 ifz
515 add
516 ifz
517 nop0
518 incA
519 popE
520 add
521 add
522 add
523 ifz
524 adrf
525 nop1
526 adrf
527 adrb
528 movii
529 pushA
530 incA
531 nop1
532 ifz
533 ifz
534 add
535 ifz
536 adrf
537 pushB
538 movdi
539 ifz
540 ifz
541 add
542 add
543 incB
544 incC
545 divide
546 add
547 add
548 ifz
549 add
550 popC
551 add
552 add
553 ifz
554 ifz
555 nop0
556 mal
557 popE
558 ifz
559 add
560 ifz
561 ifz
562 nop0
563 subCAB
564 popE
565 add
566 ifz
```

```
add
ifz
nop0
incA
popE
add
add
add
ifz
decC
nop1
ifz
add
add
ifz
jmpb
popB
pushD
adro
add2
movii2
popC
popA
add
ifz
ifz
add
pushC
popE
add
ifz
add
ifz
nop0
incA
popE
add
add
add
ifz
pushA
nop1
adrb
ifz
add
add
add
ret
pushA
adro
movii2
popC
popA
add
ifz
ifz
ifz
subAAC
```

```
625 add
626 ifz
627 add
628 add
629 nop0
630 nop1
631 add
632 ifz
633 subCAB
634 subAAC
635 incA
636 incB
637 decC
638 incC
639 pushA
640 pushB
641 pushC
642 pushD
643 popA
644 popB
645 popC
646 popD
647 popE
648 jmpo
649 jmpb
650 call
651 ret
652 add2
653 movdi
654 movii2
655 movii
656 adro
657 adrb
658 adrf
659 mal
660 divide
661 ifz
662 add
663 ifz
664 add   ;GENOTYPE END
665 nop0  ;Genotype end template
666 nop0
667 nop0
668 nop1
669 ifz   ;Dummy instruction
```

Appendix B

Opcode Map Files

This section contains the `opcode.map` files for the simulations which were run and discussed throughout the body of this thesis. The `opcode.map` files determines which instructions out of the pre-programmed Tierra instruction library are used within the instruction set of a particular run and each row represents a different instruction.

The second column[1] instructs Tierra how many CPU cycles to spend on this instruction.

The third column represents the assembler mnemonic associated with that instruction.

The fourth column is a pointer to the function which will actually carry out the operation associated with this instruction. Each function may carry out a number of different instructions, for example, the `math` function is responsible for the subtraction and addition instructions. These functions are defined in the instruct.c source module.

The fifth column is a pointer to the function which will decode the operation associated with this instruction, for example, the `dec1d2s` function which is responsible for the `sabAAC` instruction will decode one destination register and two source values. These functions are defined in the decode.c source module.

Many of the executable instructions operate on source values from registers and/or generate destination values that will be placed in registers. For example, add and subtract take two source values and generate one destination value. The fifth column specifies the registers that will be associated with these source and destination values. The opcode.map file contains a string in this field, in which the letters specify the registers which are affected. For example, the instruction subAAC uses the following string, "aac". The first "a" specifies the destination register, and the next two letters "ac" specify the two source registers.

Finally the fifth column contains the flags bitfield. There are some situations in which the relationship of the source and destination values to registers in not simple

[1] Note that neither the source code commentary or the Tierra documentation explains the purpose of the first column. However, it appears that this column must contain a zero in order for the associated instruction to be executed properly by a Tierran CPU.

enough to be handled by the mechanisms implemented through the standard procedures. In these cases a flag is set, to indicate that special handling is necessary. Flag conditions are triggered by the presence of characters in the register assignments field which are upper case letters. For example, the divide() instruction creates a new CPU whose stack and general purpose registers are all initialised as zero, however the instruction pointer register must contain the absolute address location of the first memory location in the newly created offspring. By setting a "C" flag in this field it is possible to write to "speCial" registers which to not belong to the CPU which is executing the instruction.

B.1 vn_lut32_344 opcode.map

Listing B.1: vn_lut32_344 opcode.map file

```
1     {0, 1, "nop0", nop, pnop, "", ""},
2     {0, 1, "nop1", nop, pnop, "", ""},
3     {0, 1, "add", add, dec1d2s, "", {0}},  /* "rrr" */
4     {0, 1, "ifz", ifz, dec2s, "cc", ""},        /* "r#" */
5     {0, 1, "subCAB", math, dec1d2s, "cab", ""}, /* "cab" */
6     {0, 1, "subAAC", math, dec1d2s, "aac", ""}, /* "aac" */
7     {0, 1, "incA", math, dec1d1s, "aa", ""},    /* "aa" */
8     {0, 1, "incB", math, dec1d1s, "bb", ""},    /* "bb" */
9     {0, 1, "decC", math, dec1d1s, "cc", ""},    /* "cc" */
10    {0, 1, "incC", math, dec1d1s, "dd", ""},    /* "cc" */
11    {0, 1, "pushA", push, dec1s, "a", ""},      /* "a" */
12    {0, 1, "pushB", push, dec1s, "b", ""},      /* "b" */
13    {0, 1, "pushC", push, dec1s, "c", ""},      /* "c" */
14    {0, 1, "pushD", push, dec1s, "d", ""},      /* "d" */
15    {0, 1, "popA", pop, dec1d, "a", ""},        /* "a" */
16    {0, 1, "popB", pop, dec1d, "b", ""},        /* "b" */
17    {0, 1, "popC", pop, dec1d, "c", ""},        /* "c" */
18    {0, 1, "popD", pop, dec1d, "d", ""},        /* "d" */
19    {0, 1, "popE", pop, dec1d, "e", ""},        /* "e" */
20    {0, 1, "jmpo", adr, decjmp, "b", ""},       /* "r" */
21    {0, 1, "jmpb", adr, decjmp, "b", ""},       /* "r" */
22    {0, 1, "call", tcall, ptcall, "", ""},      /* no decode args */
23    {0, 1, "ret", pop, dec1d, "", ""},          /* no decode args */
24    {0, 1, "movBA", movdd, dec1d1s, "ba", ""},  /* "ba" */
25    {0, 1, "movdi", movdi, pmovdi, "ab", ""},    /* "rr" */
26   {0, 1, "movii2", movii, pmovii, "da", ""},   /* "rr" */
27    {0, 1, "movii", movii, pmovii, "ab", ""},   /* "rr" */
28    {0, 1, "adro", adr, decadr, "ac  ", ""},    /* "rr  " */
29    {0, 1, "adrb", adr, decadr, "ac  ", ""},    /* "rr  " */
30    {0, 1, "adrf", adr, decadr, "ac  ", ""},    /* "rr  " */
31    {0, 1, "mal", malchm, dec1d3s, "ac a", ""}, /* "rr r" */
32    {0, 1, "divide", divide, dec2s, "ac", "C"},  /* "rr" */
```

B.2 vn_lut32_311 opcode.map

Listing B.2: vn_lut32_311 opcode.map file

```
    {0, 1, "nop0", nop, pnop, "", ""},
    {0, 1, "nop1", nop, pnop, "", ""},
    {0, 1, "add", add, dec1d2s, "", {0}},  /* "rrr" */
    {0, 1, "ifz", ifz, dec2s, "cc", ""},        /* "r#" */
    {0, 1, "subCAB", math, dec1d2s, "cab", ""}, /* "cab" */
    {0, 1, "subAAC", math, dec1d2s, "aac", ""}, /* "aac" */
    {0, 1, "incA", math, dec1d1s, "aa", ""},    /* "aa" */
    {0, 1, "incB", math, dec1d1s, "bb", ""},    /* "bb" */
    {0, 1, "decC", math, dec1d1s, "cc", ""},    /* "cc" */
    {0, 1, "incC", math, dec1d1s, "dd", ""},    /* "cc" */
    {0, 1, "pushA", push, dec1s, "a", ""},      /* "a" */
    {0, 1, "pushB", push, dec1s, "b", ""},      /* "b" */
    {0, 1, "pushC", push, dec1s, "c", ""},      /* "c" */
    {0, 1, "pushD", push, dec1s, "d", ""},      /* "d" */
    {0, 1, "popA", pop, dec1d, "a", ""},        /* "a" */
    {0, 1, "popB", pop, dec1d, "b", ""},        /* "b" */
    {0, 1, "popC", pop, dec1d, "c", ""},        /* "c" */
    {0, 1, "popD", pop, dec1d, "d", ""},        /* "d" */
    {0, 1, "popE", pop, dec1d, "e", ""},        /* "e" */
    {0, 1, "jmpo", adr, decjmp, "b", ""},       /* "r" */
    {0, 1, "jmpb", adr, decjmp, "b", ""},       /* "r" */
    {0, 1, "call", tcall, ptcall, "", ""},      /* no decode args */
    {0, 1, "ret", pop, dec1d, "", ""},          /* no decode args */
    {0, 1, "add2", add, dec1d2s, "", {0}},  /* "rrr" */
    {0, 1, "movdi", movdi, pmovdi, "ab", ""},    /* "rr" */
   {0, 1, "movii2", movii, pmovii, "da", ""},   /* "rr" */
    {0, 1, "movii", movii, pmovii, "ab", ""},   /* "rr" */
    {0, 1, "adro", adr, decadr, "ac  ", ""},    /* "rr  " */
    {0, 1, "adrb", adr, decadr, "ac  ", ""},    /* "rr  " */
    {0, 1, "adrf", adr, decadr, "ac  ", ""},    /* "rr  " */
    {0, 1, "mal", malchm, dec1d3s, "ac a", ""}, /* "rr r" */
    {0, 1, "divide", divide, dec2s, "ac", "C"},  /* "rr" */
```

B.3 vn_lut64_316 opcode.map

Listing B.3: vn_lut64_316 opcode.map file

```
    {0, 1, "nop0", nop, pnop, "", ""},
    {0, 1, "nop1", nop, pnop, "", ""},
    {0, 1, "add", add, dec1d2s, "", {0}},  /* "rrr" */
    {0, 1, "ifz", ifz, dec2s, "cc", ""},        /* "r#" */
    {0, 1, "subCAB", math, dec1d2s, "cab", ""}, /* "cab" */
    {0, 1, "subAAC", math, dec1d2s, "aac", ""}, /* "aac" */
    {0, 1, "incA", math, dec1d1s, "aa", ""},    /* "aa" */
    {0, 1, "incB", math, dec1d1s, "bb", ""},    /* "bb" */
    {0, 1, "decC", math, dec1d1s, "cc", ""},    /* "cc" */
    {0, 1, "incC", math, dec1d1s, "dd", ""},    /* "cc" */
    {0, 1, "pushA", push, dec1s, "a", ""},      /* "a" */
    {0, 1, "pushB", push, dec1s, "b", ""},      /* "b" */
    {0, 1, "pushC", push, dec1s, "c", ""},      /* "c" */
    {0, 1, "pushD", push, dec1s, "d", ""},      /* "d" */
    {0, 1, "popA", pop, dec1d, "a", ""},        /* "a" */
    {0, 1, "popB", pop, dec1d, "b", ""},        /* "b" */
    {0, 1, "popC", pop, dec1d, "c", ""},        /* "c" */
    {0, 1, "popD", pop, dec1d, "d", ""},        /* "d" */
    {0, 1, "popE", pop, dec1d, "e", ""},        /* "e" */
    {0, 1, "jmpo", adr, decjmp, "b", ""},       /* "r" */
    {0, 1, "jmpb", adr, decjmp, "b", ""},       /* "r" */
    {0, 1, "call", tcall, ptcall, "", ""},      /* no decode args */
    {0, 1, "ret", pop, dec1d, "", ""},          /* no decode args */
    {0, 1, "add2", add, dec1d2s, "", {0}},  /* "rrr" */
    {0, 1, "movdi", movdi, pmovdi, "ab", ""},    /* "rr" */
   {0, 1, "movii2", movii, pmovii, "da", ""},   /* "rr" */
    {0, 1, "movii", movii, pmovii, "ab", ""},   /* "rr" */
    {0, 1, "adro", adr, decadr, "ac  ", ""},    /* "rr  " */
    {0, 1, "adrb", adr, decadr, "ac  ", ""},    /* "rr  " */
    {0, 1, "adrf", adr, decadr, "ac  ", ""},    /* "rr  " */
    {0, 1, "mal", malchm, dec1d3s, "ac a", ""}, /* "rr r" */
    {0, 1, "divide", divide, dec2s, "ac", "C"},  /* "rr" */
```

B.4 vn_tt128_758 opcode.map

Listing B.4: vn_tt128_758 opcode.map file

```
    {0, 1, "nop0", nop, pnop, "", ""},
    {0, 1, "nop1", nop, pnop, "", ""},  /* no decode args */
    {0, 1, "zero", movdd, dec1d1s, "", {0}},  /* "rr" */
    {0, 1, "ifequal", skip, dec2s, "ab", ""},            /* "rr" */
    {0, 1, "ifz", skip, dec2s, "cc", ""},              /* "rr" */
    {0, 1, "ifnz", skip, dec2s, "cc", ""},             /* "rr" */  */
    {0, 1, "addAAA", add, dec1d2s, "aaa", ""},
    {0, 1, "addAAB", add, dec1d2s, "aab", ""},
    {0, 1, "addAAC", add, dec1d2s, "aac", ""},
    {0, 1, "addAAD", add, dec1d2s, "aad", ""},
    {0, 1, "addAAE", add, dec1d2s, "aae", ""},
    {0, 1, "addAAF", add, dec1d2s, "aaf", ""},
    {0, 1, "addBBA", add, dec1d2s, "bba", ""},
    {0, 1, "addBBB", add, dec1d2s, "bbb", ""},
    {0, 1, "addBBC", add, dec1d2s, "bbc", ""},
    {0, 1, "addBBD", add, dec1d2s, "bbd", ""},
    {0, 1, "addBBE", add, dec1d2s, "bbe", ""},
    {0, 1, "addBBF", add, dec1d2s, "bbf", ""},
    {0, 1, "addCCA", add, dec1d2s, "cca", ""},
    {0, 1, "addCCB", add, dec1d2s, "ccb", ""},
    {0, 1, "addCCD", add, dec1d2s, "ccd", ""},
    {0, 1, "addCCE", add, dec1d2s, "cce", ""},
    {0, 1, "addCCF", add, dec1d2s, "ccf", ""},
    {0, 1, "subCBA", add, dec1d2s, "cba", ""},
    {0, 1, "subCCA", add, dec1d2s, "cca", ""},
    {0, 1, "subCDA", add, dec1d2s, "cda", ""},
    {0, 1, "subCEA", add, dec1d2s, "cea", ""},
    {0, 1, "subCFA", add, dec1d2s, "cfa", ""},
    {0, 1, "subCAB", add, dec1d2s, "cab", ""},
    {0, 1, "subCCB", add, dec1d2s, "ccb", ""},
    {0, 1, "subCDB", add, dec1d2s, "cdb", ""},
    {0, 1, "subCEB", add, dec1d2s, "ceb", ""},
    {0, 1, "subCFB", add, dec1d2s, "cfb", ""},
    {0, 1, "subAAC", add, dec1d2s, "aac", ""},
    {0, 1, "subABC", add, dec1d2s, "abc", ""},
    {0, 1, "subADC", add, dec1d2s, "adc", ""},
    {0, 1, "subAEC", add, dec1d2s, "aec", ""},
    {0, 1, "subAFC", add, dec1d2s, "afc", ""},
    {0, 1, "subAAB", add, dec1d2s, "aab", ""},
    {0, 1, "subACB", add, dec1d2s, "acb", ""},
    {0, 1, "subADB", add, dec1d2s, "adb", ""},
    {0, 1, "subAEB", add, dec1d2s, "aeb", ""},
    {0, 1, "subAFB", add, dec1d2s, "afb", ""},
    {0, 1, "subBAC", add, dec1d2s, "bac", ""},
    {0, 1, "subBBC", add, dec1d2s, "bbc", ""},
    {0, 1, "subBDC", add, dec1d2s, "bdc", ""},
    {0, 1, "subBEC", add, dec1d2s, "bec", ""},
    {0, 1, "subBFC", add, dec1d2s, "bfc", ""},
    {0, 1, "subBBA", add, dec1d2s, "bba", ""},
    {0, 1, "subBCA", add, dec1d2s, "bca", ""},
    {0, 1, "subBDA", add, dec1d2s, "bda", ""},
    {0, 1, "subBEA", add, dec1d2s, "bea", ""},
    {0, 1, "incA", math, dec1d1s, "aa", ""},       /* "aa" */
    {0, 1, "incB", math, dec1d1s, "bb", ""},       /* "bb" */
```

```
55      {0, 1, "incD", add, dec1d1s, "dd", ""},
56      {0, 1, "incC", add, dec1d1s, "cc", ""},
57      {0, 1, "incE", add, dec1d1s, "ee", ""},
58      {0, 1, "incF", add, dec1d1s, "ff", ""},
59      {0, 1, "decA", add, dec1d1s, "aa", ""},
60      {0, 1, "decB", add, dec1d1s, "bb", ""},
61      {0, 1, "decC", add, dec1d1s, "cc", ""},
62      {0, 1, "decD", add, dec1d1s, "dd", ""},
63      {0, 1, "decE", add, dec1d1s, "ee", ""},
64      {0, 1, "decF", add, dec1d1s, "ff", ""},
65      {0, 1, "pushA", push, dec1s, "a", ""},
66      {0, 1, "pushB", push, dec1s, "b", ""},
67      {0, 1, "pushC", push, dec1s, "c", ""},
68      {0, 1, "pushD", push, dec1s, "d", ""},
69      {0, 1, "pushE", push, dec1s, "e", ""},
70      {0, 1, "pushF", push, dec1s, "f", ""},
71      {0, 1, "popA", pop, dec1d, "a", ""},            /* "a" */
72      {0, 1, "popB", pop, dec1d, "b", ""},            /* "b" */
73      {0, 1, "popC", pop, dec1d, "c", ""},            /* "c" */
74      {0, 1, "popD", pop, dec1d, "d", ""},            /* "d" */
75      {0, 1, "popE", pop, dec1d, "e", ""},            /* "e" */
76      {0, 1, "popF", pop, dec1d, "f", ""},            /* "e" */       /* "e" */
77      {0, 1, "jmpf", adr, decjmp, "b", ""},           /* "r" */
78      {0, 1, "jmpb", adr, decjmp, "b", ""},           /* "r" */
79      {0, 1, "jmpo", adr, decjmp, "b", ""},           /* "r" */
80      {0, 1, "adrf", adr, decadr, "ac  ", ""},        /* "rr  " */
81      {0, 1, "adrb", adr, decadr, "ac  ", ""},        /* "rr  " */
82      {0, 1, "adro", adr, decadr, "ac  ", ""},        /* "rr  " */
83      {0, 1, "call", tcall, ptcall, "", ""},          /* no decode args */
84      {0, 1, "movAa", movdi, pmovdi, "aa", ""},  /* "rr" */
85      {0, 1, "movAb", movdi, pmovdi, "ab", ""},  /* "rr" */
86      {0, 1, "movAc", movdi, pmovdi, "ac", ""},  /* "rr" */
87      {0, 1, "movAd", movdi, pmovdi, "ad", ""},  /* "rr" */
88      {0, 1, "movAe", movdi, pmovdi, "ae", ""},  /* "rr" */
89      {0, 1, "movAf", movdi, pmovdi, "af", ""},  /* "rr" */
90      {0, 1, "movBa", movdi, pmovdi, "ba", ""},  /* "rr" */
91      {0, 1, "movBb", movdi, pmovdi, "bb", ""},  /* "rr" */
92      {0, 1, "movBc", movdi, pmovdi, "bc", ""},  /* "rr" */
93      {0, 1, "movBd", movdi, pmovdi, "bd", ""},  /* "rr" */
94      {0, 1, "movBe", movdi, pmovdi, "be", ""},  /* "rr" */
95      {0, 1, "movBf", movdi, pmovdi, "bf", ""},  /* "rr" */
96      {0, 1, "movbA", movid, pmovid, "ba", ""},  /* "rr" */
97      {0, 1, "movcA", movid, pmovid, "ca", ""},  /* "rr" */
98      {0, 1, "movdA", movid, pmovid, "da", ""},  /* "rr" */
99      {0, 1, "moveA", movid, pmovid, "ea", ""},  /* "rr" */
100     {0, 1, "movfA", movid, pmovid, "fa", ""},  /* "rr" */
101     {0, 1, "movaB", movid, pmovid, "ab", ""},  /* "rr" */
102     {0, 1, "movcB", movid, pmovid, "cb", ""},  /* "rr" */
103     {0, 1, "movdB", movid, pmovid, "db", ""},  /* "rr" */
104     {0, 1, "moveB", movid, pmovid, "eb", ""},  /* "rr" */
105     {0, 1, "movfB", movid, pmovid, "fb", ""},  /* "rr" */
106     {0, 1, "movab", movii, pmovii, "ab", ""},  /* "rr" */
107     {0, 1, "movac", movii, pmovii, "ac", ""},  /* "rr" */
108     {0, 1, "movad", movii, pmovii, "ad", ""},  /* "rr" */
109     {0, 1, "movae", movii, pmovii, "ae", ""},  /* "rr" */
110     {0, 1, "movaf", movii, pmovii, "af", ""},  /* "rr" */
111     {0, 1, "movba", movii, pmovii, "ba", ""},  /* "rr" */
112     {0, 1, "movbc", movii, pmovii, "bc", ""},  /* "rr" */
```

```
113     {0, 1, "movbd", movii, pmovii, "bd", ""},  /* "rr" */
114     {0, 1, "movbe", movii, pmovii, "be", ""},  /* "rr" */
115     {0, 1, "movbf", movii, pmovii, "bf", ""},  /* "rr" */
116     {0, 1, "movca", movii, pmovii, "ca", ""},  /* "rr" */
117     {0, 1, "movcb", movii, pmovii, "cb", ""},  /* "rr" */
118     {0, 1, "movcd", movii, pmovii, "cd", ""},  /* "rr" */
119     {0, 1, "movce", movii, pmovii, "ce", ""},  /* "rr" */
120     {0, 1, "movcf", movii, pmovii, "cf", ""},  /* "rr" */
121     {0, 1, "movda", movii, pmovii, "da", ""},  /* "rr" */
122     {0, 1, "movdb", movii, pmovii, "db", ""},  /* "rr" */
123     {0, 1, "movdc", movii, pmovii, "dc", ""},  /* "rr" */
124     {0, 1, "movde", movii, pmovii, "de", ""},  /* "rr" */
125     {0, 1, "movdf", movii, pmovii, "df", ""},  /* "rr" */
126     {0, 1, "ret", pop, dec1d, "", ""},             /* no decode args */
127     {0, 1, "mal", malchm, dec1d3s, "ac a", ""},    /* "rr r" */
128     {0, 1, "divide", divide, dec2s, "ac", "C"},    /* "rr" */
```

Appendix C

Soup_in Files

This section contains the `soup_in` files for the simulations which were run and discussed throughout the body of this thesis. The `soup_in` file contains a set of configurable parameters which can be set before each run.

C.1 vn_lut32_344 soup_in

Listing C.1: vn_lut32_344 soup_in file

```
# tierra core: 14-12-93 INST == 0

# observational parameters:

BrkupSiz = 0      size of output file in K, named break.1, break.2 ...
CumGeneBnk = 0    Use cumulative gene files, or overwrite
debug = 0               0 = off, 1 = on, printf statements for debugging
DiskBank = 1    turn disk-genebanker on and off
DiskOut = 1             output data to disk (1 = on, 0 = off)
FindTimeM = 0    to set trap at a certain InstExe time, for debugging
FindTimeI = 0    to set trap at a certain InstExe time, for debugging
GeneBnker = 1   turn genebanker on and off
GenebankPath = 1/  path for genebanker output
hangup = 0            0 = exit on error, 1 = hangup on error for debugging
MaxFreeBlocks = 800     initial number of structures for memory allocation
SaveFreq = 50     frequency of saving core_out, soup_out and list
SavRenewMem = 0   free and renew dynamic memory after saving to disk
SavMinNum = 1     minimum number of individuals to save genotype
SavThrMem = .02  threshold memory occupancy to save genotype
SavThrPop = .02  threshold population proportion to save genotype
TierraLog = 1    0 = no log file, 1 = write log file
WatchExe = 0     mark executed instructions in genome in genebank
WatchMov = 0     set mov bits in genome in genebank
WatchTem = 0     set template bits in genome in genebank

# environmental variables:

alive = 0       how many generations will we run, 0 = infinite
DistFreq = -.3     frequency of disturbance, factor of recovery time
DistProp = .2      proportion of population affected by distrubance
DivSameGen = 0 cells must produce offspring of same genotype, to stop evolution
```

```
DivSameSiz = 0 cells must produce offspring of same size, to stop size change
DropDead = 10 stop system if no reproduction in the last x million instructions
EjectRate = 0 rate at which random ejections from soup occur
GenPerBkgMut = 16 mutation rate control by generations ("cosmic ray")
GenPerFlaw = 16       flaw control by generations
GenPerMovMut = 16    mutation rate control by generations (copy mutation)
GenPerDivMut = 16
GenPerCroInsSamSiz = 16
GenPerInsIns = 16
GenPerDelIns = 16
GenPerCroIns = 16
GenPerDelSeg = 16
GenPerInsSeg = 16
GenPerCroSeg = 16
MutBitProp = .2   proportion of mutations that are bit flips
IMapFile = opcode.map map of opcodes to instructions, file in GenebankPath
JmpSouTra = 0.   source track switches per average size
JumpTrackProb = .2 probability of switching track during a jump of the IP
LazyTol = 8 tolerance for non-reproductive cells
MalMode = 1 0 = first fit, 1 = better fit, 2 = random preference,
# 3 = near mother's address, 4 = near bx address
# 5 = near top of stack address, 6 = suggested address (parse dependant)
MalReapTol = 1  0 = reap by queue, 1 = reap oldest creature within MalTol
MalSamSiz = 0  force memory alloc to be same size as parent (stop evolution)
MalTol = 20 multiple of avgsize to search for free block
MateSizeEp = 2 size epsilon for potential mate
MaxCpuPerCell = 16   maximum number of CPUs allowed per cell
MaxIOBufSiz = 32      maximum size for IOS buffer
MaxGetBufSiz = 16      maximum size for get IO buffer
MaxPutBufSiz = 16      maximum size for put IO buffer
MaxSigBufSiz = 32      maximum size for signal buffer
MemModeFree = 0  read, write, execute protection for free memory
MemModeMine = 0  rwx protection for memory owned by a creature
MemModeProt = 2  rwx protection for memory owned by another creature
# rwx protect mem: 1 bit = execute, 2 bit = write, 4 bit = read
MinCellSize = 12       minimum size for cells
MinGenMemSiz = 12       minimum size for genetic memory of cells
MinTemplSize = 1       minimum size for templates
MovPropThrDiv = .7       minimum proportion of daughter cell filled by mov
new_soup = 1          1 = this a new soup, 0 = restarting an old run
NumCells = 1      number of creatures and gaps used to inoculate new soup
PhotonPow = 1.5         power for photon match slice size
PhotonWidth = 8   amount by which photons slide to find best fit
PhotonWord = chlorophill  word used to define photon
PutLimit = 10 distance for intercellular communication, mult of avg creat siz
ReapRndProp = .3 top prop of reaper que to reap from
SearchLimit = 5 distance for template matching, mult of avg creat siz
seed = 32 seed for random number generator, 0 uses time to set seed
SizDepSlice = 0  set slice size by size of creature
SlicePow = 1    set power for slice size, use when SizDepSlice = 1
SliceSize = 25    slice size when SizDepSlice = 0
SliceStyle = 2   choose style of determining slice size
SlicFixFrac = 0   fixed fraction of slice size
SlicRanFrac = 2   random fraction of slice size
SoupSize = 300000 size of soup in instructions
AliveGen = 0
```

```
90 0344aaa
```

C.2 vn_lut32_311 soup_in

Listing C.2: vn_lut32_311 soup_in file

```
# tierra core: 14-12-93 INST == 0

# observational parameters:

BrkupSiz = 0      size of output file in K, named break.1, break.2 ...
CumGeneBnk = 0    Use cumulative gene files, or overwrite
debug = 1               0 = off, 1 = on, printf statements for debugging
DiskBank = 1    turn disk-genebanker on and off
DiskOut = 1             output data to disk (1 = on, 0 = off)
FindTimeM = 0    to set trap at a certain InstExe time, for debugging
FindTimeI = 0    to set trap at a certain InstExe time, for debugging
GeneBnker = 0   turn genebanker on and off
GenebankPath = gb0/  path for genebanker output
hangup = 0            0 = exit on error, 1 = hangup on error for debugging
MaxFreeBlocks = 800     initial number of structures for memory allocation
SaveFreq = 100     frequency of saving core_out, soup_out and list
SavRenewMem = 0   free and renew dynamic memory after saving to disk
SavMinNum = 10     minimum number of individuals to save genotype
SavThrMem = .02  threshold memory occupancy to save genotype
SavThrPop = .02  threshold population proportion to save genotype
TierraLog = 0    0 = no log file, 1 = write log file
WatchExe = 0     mark executed instructions in genome in genebank
WatchMov = 0     set mov bits in genome in genebank
WatchTem = 0     set template bits in genome in genebank

# environmental variables:

alive = 0       how many generations will we run, 0 = infinite
DistFreq = -.3     frequency of disturbance, factor of recovery time
DistProp = .2      proportion of population affected by distrubance
DivSameGen = 0 cells must produce offspring of same genotype, to stop evolution
DivSameSiz = 0 cells must produce offspring of same size, to stop size change
DropDead = 5 stop system if no reproduction in the last x million instructions
EjectRate = 0 rate at which random ejections from soup occur
GenPerBkgMut = 32 mutation rate control by generations ("cosmic ray")
GenPerFlaw = 32   flaw control by generations
GenPerMovMut = 32   mutation rate control by generations (copy mutation)
GenPerDivMut = 32
GenPerCroInsSamSiz = 32
GenPerInsIns = 32
GenPerDelIns = 32
GenPerCroIns = 32
GenPerDelSeg = 32
GenPerInsSeg = 32
GenPerCroSeg = 32
MutBitProp = .2   proportion of mutations that are bit flips
IMapFile = opcode.map map of opcodes to instructions, file in GenebankPath
JmpSouTra = 0.   source track switches per average size
JumpTrackProb = .2 probability of switching track during a jump of the IP
LazyTol = 7 tolerance for non-reproductive cells
MalMode = 1 0 = first fit, 1 = better fit, 2 = random preference,
# 3 = near mother's address, 4 = near bx address
# 5 = near top of stack address, 6 = suggested address (parse dependant)
MalReapTol = 1  0 = reap by queue, 1 = reap oldest creature within MalTol
```

```
MalSamSiz = 0 force memory alloc to be same size as parent (stop evolution)
MalTol = 20 multiple of avgsize to search for free block
MateSizeEp = 2 size epsilon for potential mate
MaxCpuPerCell = 16   maximum number of CPUs allowed per cell
MaxIOBufSiz = 32      maximum size for IOS buffer
MaxGetBufSiz = 16      maximum size for get IO buffer
MaxPutBufSiz = 16      maximum size for put IO buffer
MaxSigBufSiz = 32      maximum size for signal buffer
MemModeFree = 0  read, write, execute protection for free memory
MemModeMine = 0  rwx protection for memory owned by a creature
MemModeProt = 2  rwx protection for memory owned by another creature
# rwx protect mem: 1 bit = execute, 2 bit = write, 4 bit = read
MinCellSize = 12       minimum size for cells
MinGenMemSiz = 12       minimum size for genetic memory of cells
MinTemplSize = 1       minimum size for templates
MovPropThrDiv = .7       minimum proportion of daughter cell filled by mov
new_soup = 1          1 = this a new soup, 0 = restarting an old run
NumCells = 1      number of creatures and gaps used to inoculate new soup
PhotonPow = 1.5         power for photon match slice size
PhotonWidth = 8   amount by which photons slide to find best fit
PhotonWord = chlorophill  word used to define photon
PutLimit = 10 distance for intercellular communication, mult of avg creat siz
ReapRndProp = .3 top prop of reaper que to reap from
SearchLimit = 5 distance for template matching, mult of avg creat siz
seed = 24 seed for random number generator, 0 uses time to set seed
SizDepSlice = 0  set slice size by size of creature
SlicePow = 1    set power for slice size, use when SizDepSlice = 1
SliceSize = 25    slice size when SizDepSlice = 0
SliceStyle = 2   choose style of determining slice size
SlicFixFrac = 0   fixed fraction of slice size
SlicRanFrac = 2   random fraction of slice size
SoupSize = 300000 size of soup in instructions
AliveGen = 0


0311aaa
```

C.3 vn_lut64_316 soup_in

Listing C.3: vn_lut64_316 soup_in file

```
# tierra core: 14-12-93 INST == 0

# observational parameters:

BrkupSiz = 0      size of output file in K, named break.1, break.2 ...
CumGeneBnk = 0    Use cumulative gene files, or overwrite
debug = 0               0 = off, 1 = on, printf statements for debugging
DiskBank = 0    turn disk-genebanker on and off
DiskOut = 0             output data to disk (1 = on, 0 = off)
FindTimeM = 0    to set trap at a certain InstExe time, for debugging
FindTimeI = 0    to set trap at a certain InstExe time, for debugging
GeneBnker = 0   turn genebanker on and off
GenebankPath = gb0/  path for genebanker output
hangup = 0            0 = exit on error, 1 = hangup on error for debugging
MaxFreeBlocks = 800     initial number of structures for memory allocation
SaveFreq = 100     frequency of saving core_out, soup_out and list
SavRenewMem = 0   free and renew dynamic memory after saving to disk
SavMinNum = 100     minimum number of individuals to save genotype
SavThrMem = .8  threshold memory occupancy to save genotype
SavThrPop = .8  threshold population proportion to save genotype
TierraLog = 0    0 = no log file, 1 = write log file
WatchExe = 0     mark executed instructions in genome in genebank
WatchMov = 0     set mov bits in genome in genebank
WatchTem = 0     set template bits in genome in genebank

# environmental variables:

alive = 0       how many generations will we run, 0 = infinite
DistFreq = -.3     frequency of disturbance, factor of recovery time
DistProp = .2      proportion of population affected by distrubance
DivSameGen = 0 cells must produce offspring of same genotype, to stop evolution
DivSameSiz = 0 cells must produce offspring of same size, to stop size change
DropDead = 5 stop system if no reproduction in the last x million instructions
EjectRate = 0 rate at which random ejections from soup occur
GenPerBkgMut = 16 mutation rate control by generations ("cosmic ray")
GenPerFlaw = 0   flaw control by generations
GenPerMovMut = 16   mutation rate control by generations (copy mutation)
GenPerDivMut = 16
GenPerCroInsSamSiz = 16
GenPerInsIns = 16
GenPerDelIns = 10
GenPerCroIns = 16
GenPerDelSeg = 10
GenPerInsSeg = 0
GenPerCroSeg = 0
MutBitProp = .2   proportion of mutations that are bit flips
IMapFile = opcode.map map of opcodes to instructions, file in GenebankPath
JmpSouTra = 0.   source track switches per average size
JumpTrackProb = .2 probability of switching track during a jump of the IP
LazyTol = 5 tolerance for non-reproductive cells
MalMode = 1 0 = first fit, 1 = better fit, 2 = random preference,
# 3 = near mother's address, 4 = near bx address
# 5 = near top of stack address, 6 = suggested address (parse dependant)
MalReapTol = 1  0 = reap by queue, 1 = reap oldest creature within MalTol
```

```
MalSamSiz = 0 force memory alloc to be same size as parent (stop evolution)
MalTol = 20 multiple of avgsize to search for free block
MateSizeEp = 2 size epsilon for potential mate
MaxCpuPerCell = 16   maximum number of CPUs allowed per cell
MaxIOBufSiz = 32      maximum size for IOS buffer
MaxGetBufSiz = 16      maximum size for get IO buffer
MaxPutBufSiz = 16      maximum size for put IO buffer
MaxSigBufSiz = 32      maximum size for signal buffer
MemModeFree = 0  read, write, execute protection for free memory
MemModeMine = 0  rwx protection for memory owned by a creature
MemModeProt = 2  rwx protection for memory owned by another creature
# rwx protect mem: 1 bit = execute, 2 bit = write, 4 bit = read
MinCellSize = 12       minimum size for cells
MinGenMemSiz = 12       minimum size for genetic memory of cells
MinTemplSize = 1       minimum size for templates
MovPropThrDiv = .7       minimum proportion of daughter cell filled by mov
new_soup = 1          1 = this a new soup, 0 = restarting an old run
NumCells = 1      number of creatures and gaps used to inoculate new soup
PhotonPow = 1.5         power for photon match slice size
PhotonWidth = 8   amount by which photons slide to find best fit
PhotonWord = chlorophill  word used to define photon
PutLimit = 10 distance for intercellular communication, mult of avg creat siz
ReapRndProp = .3 top prop of reaper que to reap from
SearchLimit = 5 distance for template matching, mult of avg creat siz
seed = 50    seed for random number generator, 0 uses time to set seed
SizDepSlice = 0  set slice size by size of creature
SlicePow = 1    set power for slice size, use when SizDepSlice = 1
SliceSize = 36    slice size when SizDepSlice = 0
SliceStyle = 2   choose style of determining slice size
SlicFixFrac = 0   fixed fraction of slice size
SlicRanFrac = 2   random fraction of slice size
SoupSize = 300000 size of soup in instructions
AliveGen = 0


0316aaa
```

C.4 vn_tt128_758 soup_in

Listing C.4: vn_tt128_758 soup_in file

```
1
2
3  # tierra core: 14-12-93 INST == 0
4
5  # observational parameters:
6
7  BrkupSiz = 0      size of output file in K, named break.1, break.2 ...
8  CumGeneBnk = 0    Use cumulative gene files, or overwrite
9  debug = 0               0 = off, 1 = on, printf statements for debugging
10 DiskBank = 1    turn disk-genebanker on and off
11 DiskOut = 0            output data to disk (1 = on, 0 = off)
12 FindTimeM = 0    to set trap at a certain InstExe time, for debugging
13 FindTimeI = 0    to set trap at a certain InstExe time, for debugging
14 GeneBnker = 1   turn genebanker on and off
15 GenebankPath = gb1/  path for genebanker output
16 hangup = 0            0 = exit on error, 1 = hangup on error for debugging
17 MaxFreeBlocks = 800     initial number of structures for memory allocation
18 SaveFreq = 0     frequency of saving core_out, soup_out and list
19 SavRenewMem = 0   free and renew dynamic memory after saving to disk
20 SavMinNum = 20     minimum number of individuals to save genotype
21 SavThrMem = 0.02  threshold memory occupancy to save genotype
22 SavThrPop = 0.02  threshold population proportion to save genotype
23 TierraLog = 0    0 = no log file, 1 = write log file
24 WatchExe = 0     mark executed instructions in genome in genebank
25 WatchMov = 0     set mov bits in genome in genebank
26 WatchTem = 0     set template bits in genome in genebank
27
28 # environmental variables:
29
30 alive = 10       how many generations will we run, 0 = infinite
31 DistFreq = -.3     frequency of disturbance, factor of recovery time
32 DistProp = .2      proportion of population affected by distrubance
33 DivSameGen = 0 cells must produce offspring of same genotype, to stop evolution
34 DivSameSiz = 1 cells must produce offspring of same size, to stop size change
35 DropDead = 1000 stop system if no reproduction in the last x million
       instructions
36 EjectRate = 0 rate at which random ejections from soup occur
37 GenPerBkgMut = 10  mutation rate control by generations ("cosmic ray")
38 GenPerFlaw = 0   flaw control by generations
39 GenPerMovMut = 10   mutation rate control by generations (copy mutation)
40 GenPerDivMut = 0
41 GenPerCroInsSamSiz = 0
42 GenPerInsIns = 0
43 GenPerDelIns = 0
44 GenPerCroIns = 0
45 GenPerDelSeg = 0
46 GenPerFactor = 16
47 GenPerInsSeg = 0
48 GenPerCroSeg = 0
49 MutBitProp = .2   proportion of mutations that are bit flips
50 IMapFile = opcode.map map of opcodes to instructions, file in GenebankPath
51 JmpSouTra = 0.   source track switches per average size
52 JumpTrackProb = .2 probability of switching track during a jump of the IP
53 LazyTol = 178 tolerance for non-reproductive cells
```

```
MalMode = 1 0 = first fit, 1 = better fit, 2 = random preference,
# 3 = near mother's address, 4 = near bx address
# 5 = near top of stack address, 6 = suggested address (parse dependant)
MalReapTol = 1  0 = reap by queue, 1 = reap oldest creature within MalTol
MalSamSiz = 1 force memory alloc to be same size as parent (stop evolution)
MalTol = 5 multiple of avgsize to search for free block
MateSizeEp = 2 size epsilon for potential mate
MaxCpuPerCell = 1   maximum number of CPUs allowed per cell
MaxIOBufSiz = 32      maximum size for IOS buffer
MaxGetBufSiz = 16      maximum size for get IO buffer
MaxPutBufSiz = 16      maximum size for put IO buffer
MaxSigBufSiz = 32      maximum size for signal buffer
MemModeFree = 0  read, write, execute protection for free memory
MemModeMine = 0  rwx protection for memory owned by a creature
MemModeProt = 2  rwx protection for memory owned by another creature
# rwx protect mem: 1 bit = execute, 2 bit = write, 4 bit = read
MinCellSize = 758       minimum size for cells
MinGenMemSiz = 758       minimum size for genetic memory of cells
MinTemplSize = 1       minimum size for templates
MovPropThrDiv = 1       minimum proportion of daughter cell filled by mov
new_soup = 1          1 = this a new soup, 0 = restarting an old run
NumCells = 1      number of creatures and gaps used to inoculate new soup
PhotonPow = 1.5         power for photon match slice size
PhotonWidth = 8   amount by which photons slide to find best fit
PhotonWord = chlorophill  word used to define photon
PutLimit = 5 distance for intercellular communication, mult of avg creat siz
ReapRndProp = .3 top prop of reaper que to reap from
SearchLimit = 10 distance for template matching, mult of avg creat siz
seed = 1  seed for random number generator, 0 uses time to set seed
SizDepSlice = 0  set slice size by size of creature
SlicePow = 1    set power for slice size, use when SizDepSlice = 1
SliceSize = 36    slice size when SizDepSlice = 0
SliceStyle = 2   choose style of determining slice size
SlicFixFrac = 0   fixed fraction of slice size
SlicRanFrac = 2   random fraction of slice size
SoupSize = 1000000 size of soup in instructions
AliveGen = 0

0758aaa
```

Appendix D

Tierra Source Code Revisions

This section contains the update file which was created and must be applied to the Tierra source code in order to perform the experiments in this Thesis.

D.1 Tierra 6.02 update file

Listing D.1: Final Source Code

```
# ////////////////////////////////////////////////////////////
#        17th December 2014
# Patch file to update Tierra to our specifications.
# Created by Declan Baugh, Dublin City University,
# Contact details : declanbaugh@gmail.com
# Index:
#  c: Ammend the tierra.run output file
#  b: Remove comma "," in InstExe:
#  a: Add Nops()
#  0: Makefile Debug no
#  1: GenperFactor   ()
#  2: arg 'h' input parameter
#  3:  Display Seed and start time
#  5:  clock()
#  6:  shlA()
#  7:  IncD()
#  8:  addEAA()
#  9:  ifnz()
#  10:  movAb()
#  11:  movda()
#  12: movab()
#  13: jump()
#  14: return()
#
#
# ////////////////////////////////////////////////////////////
#
#
# h: rambank_logger()
diff --git a/tierra/tierra.c b/tierra/tierra.c
--- a/tierra/tierra.c
+++ b/tierra/tierra.c
```

```
@@ -743,0 +743,1 @@
+  rambank_logger(); /* barry.mcmullin@dcu.ie 2013-06-02: See rambank.c */
# g: void rambank_logger
diff --git a/tierra/prototyp.h b/tierra/prototyp.h
--- a/tierra/prototyp.h
+++ b/tierra/prototyp.h
@@ -913,0 +913,2 @@
+/* */
+void rambank_logger(void);
# f: rambank_logger
diff --git a/tierra/rambank.c b/tierra/rambank.c
--- a/tierra/rambank.c
+++ b/tierra/rambank.c
@@ -1860,0 +1860,39 @@
+/*
+  Log a summary of the rambank, at suitable intervals.
+
+  barry.mcmullin@dcu.ie 2013-06-02
+*/
+void rambank_logger(void)
+{
+  I32s DumpLogFreq = 1;
+  I32s DumpLogMin = 0;
+  I32s si; /* size index? */
+  I32s gi;
+  GList ** gl;
+  Genotype gen;
+
+  if ((InstExe.m % DumpLogFreq) == 0) {
+    fprintf(stderr, "Timestamp: %10d,%06d\n", InstExe.m, InstExe.i);
+    //fprintf(stderr, "rl> InstExe.m: %d siz_sl: %d\n", InstExe.m, siz_sl);
+    for (si = 0; si < siz_sl; si++) {
+      if (!TNULL(sl[si])) {
+  //fprintf(stderr, " rl> si: %d num_g: %d a_num: %d\n", si, sl[si]->num_g, sl[
    si]->a_num);
+  gl = sl[si]->g;
+  if (!TNULL(gl)) {
+    for (gi = 0; gi < sl[si]->a_num; gi++) {
+      if (!TNULL(gl[gi])) {
+        if (gl[gi]->pop > DumpLogMin) {
+     gen = gl[gi]->gen;
+     fprintf(stderr, "%.4d%3s: %4d\n", gen.size, gen.label, gl[gi]->pop);
+     //fprintf(stderr, "  rl> gi: %d gen.size: %d gen.lable: %3s pop: %d\n",
+     //  gi, gen.size, gen.label, gl[gi]->pop);
+     //fprintf(stderr, "  rl> gi: %d pop: %d\n", gi, 666);
+        }
+      }
+    }
+  }
+      }
+    }
+    fflush(stderr);
+  }
+}
# e: Add extinction time
diff --git a/tierra/rambank.c b/tierra/rambank.c
--- a/tierra/rambank.c
+++ b/tierra/rambank.c
```

```
@@ -276,1 +276,7 @@ void ReapGenBook(cp, mutflag)
-    {   if(!tgl->origpop)
+        {
+        FILE *extinct_file;
+        tsprintf((char *)(&(Buff[0])),"%sextinct.log", GenebankPath);
+        extinct_file = tfopen(&(Buff[0]), (I8s *)"a");
+        fprintf(extinct_file,"%d%6.6d %4.4d%3s extinct \n", InstExe.m , InstExe
    .i, cp->mm.s, Int2Lbl(cp->d.gi) );
+        fclose(extinct_file);
+        if(!tgl->origpop)
# d: Add emergence time
diff --git a/tierra/rambank.c b/tierra/rambank.c
--- a/tierra/rambank.c
+++ b/tierra/rambank.c
@@ -162,1 +162,7 @@ void DivGenBook(cp, InstExe, reaped, mom, same, disk,
    mutflag)
-    {   NumGenotypes++;
+  {
+        FILE *extinct_file;
+        tsprintf((char *)(&(Buff[0])),"%sextinct.log", GenebankPath);
+        extinct_file = tfopen(&(Buff[0]), (I8s *)"a");
+        fprintf(extinct_file,"%d%6.6d %4.4d%3s emerge \n", InstExe.m, InstExe.i
    , cp->mm.s, Int2Lbl(cp->d.gi) );
+        fclose(extinct_file);
+       NumGenotypes++;
# c: Ammend the tierra.run output file
diff --git a/tierra/bookeep.c b/tierra/bookeep.c
--- a/tierra/bookeep.c
+++ b/tierra/bookeep.c
@@ -353,1 +353,1 @@ void OutDisk(bd, size, label)
-                BrkupCum += 1 + tfprintf(oufr, " %s\n", label);
+                BrkupCum += 1 + tfprintf(oufr, "%s\n", label);
@@ -363,1 +363,1 @@ void OutDisk(bd, size, label)
-            if (lo.bd != bd)
+
@@ -365,1 +365,1 @@ void OutDisk(bd, size, label)
-            if (lo.size != size)
+
@@ -367,3 +367,3 @@ void OutDisk(bd, size, label)
-            if (GeneBnker && strcmp((const char *)(&(lo.label[0])),
-                (const char *)(&(label[0]))))
-            {
+
+
+
@@ -373,2 +373,2 @@ void OutDisk(bd, size, label)
-                BrkupCum += tfprintf(oufr, " %s", label);
-            }
+                BrkupCum += tfprintf(oufr, "%s", label);
+
# b: Remove comma "," in InstExe:
diff --git a/tierra/genio.c b/tierra/genio.c
--- a/tierra/genio.c
+++ b/tierra/genio.c
@@ -1872,1 +1872,1 @@ WritAscFile()
-        tfprintf(fp, "Origin: InstExe: %d,%.6d",
+        tfprintf(fp, "Origin: InstExe: %d%.6d",
# a: Add Nops() instruction
```

```
diff --git a/tierra/soup_in.h b/tierra/soup_in.h
--- a/tierra/soup_in.h
+++ b/tierra/soup_in.h
@@ -208,0 +208,350 @@ InstDef idt[] =
+    {0, 1, "addAAA", add, dec1d2s, "fff", {0}},  /* "rrr" */
+    {0, 1, "addAAB", add, dec1d2s, "ffa", {0}},  /* "rrr" */
+    {0, 1, "addAAC", add, dec1d2s, "ffb", {0}},  /* "rrr" */
+    {0, 1, "addAAD", add, dec1d2s, "ffc", {0}},  /* "rrr" */
+    {0, 1, "addAAF", add, dec1d2s, "ffe", {0}},  /* "rrr" */
+    {0, 1, "addBBA", add, dec1d2s, "aaf", {0}},  /* "rrr" */
+    {0, 1, "addBBB", add, dec1d2s, "aaa", {0}},  /* "rrr" */
+    {0, 1, "addBBC", add, dec1d2s, "aab", {0}},  /* "rrr" */
+    {0, 1, "addBBD", add, dec1d2s, "aac", {0}},  /* "rrr" */
+    {0, 1, "addBBE", add, dec1d2s, "aad", {0}},  /* "rrr" */
+    {0, 1, "addBBF", add, dec1d2s, "aae", {0}},  /* "rrr" */
+    {0, 1, "addCCA", add, dec1d2s, "bbf", {0}},  /* "rrr" */
+    {0, 1, "addCCB", add, dec1d2s, "bba", {0}},  /* "rrr" */
+    {0, 1, "addCCD", add, dec1d2s, "bbc", {0}},  /* "rrr" */
+    {0, 1, "addCCE", add, dec1d2s, "bbd", {0}},  /* "rrr" */
+    {0, 1, "addCCF", add, dec1d2s, "bbe", {0}},  /* "rrr" */
+    {0, 1, "incE", add, dec1d1s, "ee", {0}},  /* "cc" */
+    {0, 1, "incF", add, dec1d1s, "ff", {0}},  /* "cc" */
+    {0, 1, "movAa", movdi, pmovdi, "aa", {0}},  /* "rr" */
+    {0, 1, "movAc", movdi, pmovdi, "ac", {0}},  /* "rr" */
+    {0, 1, "movAd", movdi, pmovdi, "ad", {0}},  /* "rr" */
+    {0, 1, "movAe", movdi, pmovdi, "ae", {0}},  /* "rr" */
+    {0, 1, "movAf", movdi, pmovdi, "af", {0}},  /* "rr" */
+    {0, 1, "movBa", movdi, pmovdi, "ba", {0}},  /* "rr" */
+    {0, 1, "movBb", movdi, pmovdi, "bb", {0}},  /* "rr" */
+    {0, 1, "movBc", movdi, pmovdi, "bc", {0}},  /* "rr" */
+    {0, 1, "movBd", movdi, pmovdi, "bd", {0}},  /* "rr" */
+    {0, 1, "movBe", movdi, pmovdi, "be", {0}},  /* "rr" */
+    {0, 1, "movBf", movdi, pmovdi, "bf", {0}},  /* "rr" */
+    {0, 1, "movbA", movid, pmovid, "bA", {0}},  /* "rr" */
+    {0, 1, "movcA", movid, pmovid, "cA", {0}},  /* "rr" */
+    {0, 1, "movdA", movid, pmovid, "dA", {0}},  /* "rr" */
+    {0, 1, "moveA", movid, pmovid, "eA", {0}},  /* "rr" */
+    {0, 1, "movfA", movid, pmovid, "fA", {0}},  /* "rr" */
+    {0, 1, "movaB", movid, pmovid, "aB", {0}},  /* "rr" */
+    {0, 1, "movcB", movid, pmovid, "cB", {0}},  /* "rr" */
+    {0, 1, "movdB", movid, pmovid, "dB", {0}},  /* "rr" */
+    {0, 1, "moveB", movid, pmovid, "eB", {0}},  /* "rr" */
+    {0, 1, "movfB", movid, pmovid, "fB", {0}},  /* "rr" */
+    {0, 1, "movac", movii, pmovii, "ac", {0}},  /* "rr" */
+    {0, 1, "movad", movii, pmovii, "ad", {0}},  /* "rr" */
+    {0, 1, "movae", movii, pmovii, "ae", {0}},  /* "rr" */
+    {0, 1, "movaf", movii, pmovii, "af", {0}},  /* "rr" */
+    {0, 1, "movba", movii, pmovii, "ba", {0}},  /* "rr" */
+    {0, 1, "movbc", movii, pmovii, "bc", {0}},  /* "rr" */
+    {0, 1, "movbd", movii, pmovii, "bd", {0}},  /* "rr" */
+    {0, 1, "movbe", movii, pmovii, "be", {0}},  /* "rr" */
+    {0, 1, "movbf", movii, pmovii, "bf", {0}},  /* "rr" */
+    {0, 1, "movca", movii, pmovii, "ca", {0}},  /* "rr" */
+    {0, 1, "movcb", movii, pmovii, "cb", {0}},  /* "rr" */
+    {0, 1, "movcd", movii, pmovii, "cd", {0}},  /* "rr" */
+    {0, 1, "movce", movii, pmovii, "ce", {0}},  /* "rr" */
+    {0, 1, "movcf", movii, pmovii, "cf", {0}},  /* "rr" */
+    {0, 1, "movdb", movii, pmovii, "db", {0}},  /* "rr" */
```

```
203 +     {0, 1, "movdc", movii, pmovii, "dc", {0}},  /* "rr" */
204 +     {0, 1, "movde", movii, pmovii, "de", {0}},  /* "rr" */
205 +     {0, 1, "movdf", movii, pmovii, "df", {0}},  /* "rr" */
206 +     {0, 1, "shlB", shl, dec1d, "b", {0}},  /* "r" */
207 +     {0, 1, "shlC", shl, dec1d, "c", {0}},  /* "r" */
208 +     {0, 1, "shlD", shl, dec1d, "", {0}},  /* "r" */
209 +     {0, 1, "decA", add, dec1d1s, "aa", {0}},  /* "cc" */
210 +     {0, 1, "decB", add, dec1d1s, "bb", {0}},  /* "cc" */
211 +     {0, 1, "decD", add, dec1d1s, "dd", {0}},  /* "cc" */
212 +     {0, 1, "decE", add, dec1d1s, "ee", {0}},  /* "cc" */
213 +     {0, 1, "decF", add, dec1d1s, "ff", {0}},  /* "cc" */
214 +     {0, 1, "subCCA", add, dec1d2s, "cca", {0}},  /* "cab" */
215 +     {0, 1, "subCDA", add, dec1d2s, "cda", {0}},  /* "cab" */
216 +     {0, 1, "subCEA", add, dec1d2s, "cea", {0}},  /* "cab" */
217 +     {0, 1, "subCFA", add, dec1d2s, "cfa", {0}},  /* "bac" */
218 +     {0, 1, "subCCB", add, dec1d2s, "ccb", {0}},  /* "cab" */
219 +     {0, 1, "subCDB", add, dec1d2s, "cdb", {0}},  /* "cab" */
220 +     {0, 1, "subCEB", add, dec1d2s, "ceb", {0}},  /* "cab" */
221 +     {0, 1, "subCFB", add, dec1d2s, "cfb", {0}},  /* "bac" */
222 +     {0, 1, "subABC", add, dec1d2s, "abc", {0}},  /* "bac" */
223 +     {0, 1, "subADC", add, dec1d2s, "adc", {0}},  /* "bac" */
224 +     {0, 1, "subAEC", add, dec1d2s, "aec", {0}},  /* "bac" */
225 +     {0, 1, "subAFC", add, dec1d2s, "afc", {0}},  /* "bac" */
226 +     {0, 1, "subAAB", add, dec1d2s, "aab", {0}},  /* "bac" */
227 +     {0, 1, "subACB", add, dec1d2s, "acb", {0}},  /* "bac" */
228 +     {0, 1, "subADB", add, dec1d2s, "adb", {0}},  /* "bac" */
229 +     {0, 1, "subAEB", add, dec1d2s, "aeb", {0}},  /* "bac" */
230 +     {0, 1, "subAFB", add, dec1d2s, "afb", {0}},  /* "bac" */
231 +     {0, 1, "subBBC", add, dec1d2s, "bbc", {0}},  /* "bac" */
232 +     {0, 1, "subBDC", add, dec1d2s, "bdc", {0}},  /* "bac" */
233 +     {0, 1, "subBEC", add, dec1d2s, "bec", {0}},  /* "bac" */
234 +     {0, 1, "subBFC", add, dec1d2s, "bfc", {0}},  /* "bac" */
235 +     {0, 1, "subBBA", add, dec1d2s, "bba", {0}},  /* "bac" */
236 +     {0, 1, "subBCA", add, dec1d2s, "bca", {0}},  /* "bac" */
237 +     {0, 1, "subBDA", add, dec1d2s, "bda", {0}},  /* "bac" */
238 +     {0, 1, "subBEA", add, dec1d2s, "bea", {0}},  /* "bac" */
239 +     {0, 1, "subCCD", add, dec1d2s, "ccd", {0}},  /* "ccd" */
240 +     {0, 1, "nop2", nop, pnop, "", {0}},  /* no decode args */
241 +     {0, 1, "nop3", nop, pnop, "", {0}},  /* no decode args */
242 +     {0, 1, "nop4", nop, pnop, "", {0}},  /* no decode args */
243 +     {0, 1, "nop5", nop, pnop, "", {0}},  /* no decode args */
244 +     {0, 1, "nop6", nop, pnop, "", {0}},  /* no decode args */
245 +     {0, 1, "nop7", nop, pnop, "", {0}},  /* no decode args */
246 +     {0, 1, "nop8", nop, pnop, "", {0}},  /* no decode args */
247 +     {0, 1, "nop9", nop, pnop, "", {0}},  /* no decode args */
248 +     {0, 1, "nop10", nop, pnop, "", {0}},  /* no decode args */
249 +     {0, 1, "nop11", nop, pnop, "", {0}},  /* no decode args */
250 +     {0, 1, "nop12", nop, pnop, "", {0}},  /* no decode args */
251 +     {0, 1, "nop13", nop, pnop, "", {0}},  /* no decode args */
252 +     {0, 1, "nop14", nop, pnop, "", {0}},  /* no decode args */
253 +     {0, 1, "nop15", nop, pnop, "", {0}},  /* no decode args */
254 +     {0, 1, "nop16", nop, pnop, "", {0}},  /* no decode args */
255 +     {0, 1, "nop17", nop, pnop, "", {0}},  /* no decode args */
256 +     {0, 1, "nop18", nop, pnop, "", {0}},  /* no decode args */
257 +     {0, 1, "nop19", nop, pnop, "", {0}},  /* no decode args */
258 +     {0, 1, "nop20", nop, pnop, "", {0}},  /* no decode args */
259 +     {0, 1, "nop21", nop, pnop, "", {0}},  /* no decode args */
260 +     {0, 1, "nop22", nop, pnop, "", {0}},  /* no decode args */
```

```
261 +    {0, 1, "nop23", nop, pnop, "", {0}},  /* no decode args */
262 +    {0, 1, "nop24", nop, pnop, "", {0}},  /* no decode args */
263 +    {0, 1, "nop25", nop, pnop, "", {0}},  /* no decode args */
264 +    {0, 1, "nop26", nop, pnop, "", {0}},  /* no decode args */
265 +    {0, 1, "nop27", nop, pnop, "", {0}},  /* no decode args */
266 +    {0, 1, "nop28", nop, pnop, "", {0}},  /* no decode args */
267 +    {0, 1, "nop29", nop, pnop, "", {0}},  /* no decode args */
268 +    {0, 1, "nop30", nop, pnop, "", {0}},  /* no decode args */
269 +    {0, 1, "nop31", nop, pnop, "", {0}},  /* no decode args */
270 +    {0, 1, "nop32", nop, pnop, "", {0}},  /* no decode args */
271 +    {0, 1, "nop33", nop, pnop, "", {0}},  /* no decode args */
272 +    {0, 1, "nop34", nop, pnop, "", {0}},  /* no decode args */
273 +    {0, 1, "nop35", nop, pnop, "", {0}},  /* no decode args */
274 +    {0, 1, "nop36", nop, pnop, "", {0}},  /* no decode args */
275 +    {0, 1, "nop37", nop, pnop, "", {0}},  /* no decode args */
276 +    {0, 1, "nop38", nop, pnop, "", {0}},  /* no decode args */
277 +    {0, 1, "nop39", nop, pnop, "", {0}},  /* no decode args */
278 +    {0, 1, "nop40", nop, pnop, "", {0}},  /* no decode args */
279 +    {0, 1, "nop41", nop, pnop, "", {0}},  /* no decode args */
280 +    {0, 1, "nop42", nop, pnop, "", {0}},  /* no decode args */
281 +    {0, 1, "nop43", nop, pnop, "", {0}},  /* no decode args */
282 +    {0, 1, "nop44", nop, pnop, "", {0}},  /* no decode args */
283 +    {0, 1, "nop45", nop, pnop, "", {0}},  /* no decode args */
284 +    {0, 1, "nop46", nop, pnop, "", {0}},  /* no decode args */
285 +    {0, 1, "nop47", nop, pnop, "", {0}},  /* no decode args */
286 +    {0, 1, "nop48", nop, pnop, "", {0}},  /* no decode args */
287 +    {0, 1, "nop49", nop, pnop, "", {0}},  /* no decode args */
288 +    {0, 1, "nop50", nop, pnop, "", {0}},  /* no decode args */
289 +    {0, 1, "nop51", nop, pnop, "", {0}},  /* no decode args */
290 +    {0, 1, "nop52", nop, pnop, "", {0}},  /* no decode args */
291 +    {0, 1, "nop53", nop, pnop, "", {0}},  /* no decode args */
292 +    {0, 1, "nop54", nop, pnop, "", {0}},  /* no decode args */
293 +    {0, 1, "nop55", nop, pnop, "", {0}},  /* no decode args */
294 +    {0, 1, "nop56", nop, pnop, "", {0}},  /* no decode args */
295 +    {0, 1, "nop57", nop, pnop, "", {0}},  /* no decode args */
296 +    {0, 1, "nop58", nop, pnop, "", {0}},  /* no decode args */
297 +    {0, 1, "nop59", nop, pnop, "", {0}},  /* no decode args */
298 +    {0, 1, "nop60", nop, pnop, "", {0}},  /* no decode args */
299 +    {0, 1, "nop61", nop, pnop, "", {0}},  /* no decode args */
300 +    {0, 1, "nop62", nop, pnop, "", {0}},  /* no decode args */
301 +    {0, 1, "nop63", nop, pnop, "", {0}},  /* no decode args */
302 +    {0, 1, "nop64", nop, pnop, "", {0}},  /* no decode args */
303 +    {0, 1, "nop65", nop, pnop, "", {0}},  /* no decode args */
304 +    {0, 1, "nop66", nop, pnop, "", {0}},  /* no decode args */
305 +    {0, 1, "nop67", nop, pnop, "", {0}},  /* no decode args */
306 +    {0, 1, "nop68", nop, pnop, "", {0}},  /* no decode args */
307 +    {0, 1, "nop69", nop, pnop, "", {0}},  /* no decode args */
308 +    {0, 1, "nop70", nop, pnop, "", {0}},  /* no decode args */
309 +    {0, 1, "nop71", nop, pnop, "", {0}},  /* no decode args */
310 +    {0, 1, "nop72", nop, pnop, "", {0}},  /* no decode args */
311 +    {0, 1, "nop73", nop, pnop, "", {0}},  /* no decode args */
312 +    {0, 1, "nop74", nop, pnop, "", {0}},  /* no decode args */
313 +    {0, 1, "nop75", nop, pnop, "", {0}},  /* no decode args */
314 +    {0, 1, "nop76", nop, pnop, "", {0}},  /* no decode args */
315 +    {0, 1, "nop77", nop, pnop, "", {0}},  /* no decode args */
316 +    {0, 1, "nop78", nop, pnop, "", {0}},  /* no decode args */
317 +    {0, 1, "nop79", nop, pnop, "", {0}},  /* no decode args */
318 +    {0, 1, "nop80", nop, pnop, "", {0}},  /* no decode args */
```

```
319 +    {0, 1, "nop81", nop, pnop, "", {0}},  /* no decode args */
320 +    {0, 1, "nop82", nop, pnop, "", {0}},  /* no decode args */
321 +    {0, 1, "nop83", nop, pnop, "", {0}},  /* no decode args */
322 +    {0, 1, "nop84", nop, pnop, "", {0}},  /* no decode args */
323 +    {0, 1, "nop85", nop, pnop, "", {0}},  /* no decode args */
324 +    {0, 1, "nop86", nop, pnop, "", {0}},  /* no decode args */
325 +    {0, 1, "nop87", nop, pnop, "", {0}},  /* no decode args */
326 +    {0, 1, "nop88", nop, pnop, "", {0}},  /* no decode args */
327 +    {0, 1, "nop89", nop, pnop, "", {0}},  /* no decode args */
328 +    {0, 1, "nop90", nop, pnop, "", {0}},  /* no decode args */
329 +    {0, 1, "nop91", nop, pnop, "", {0}},  /* no decode args */
330 +    {0, 1, "nop92", nop, pnop, "", {0}},  /* no decode args */
331 +    {0, 1, "nop93", nop, pnop, "", {0}},  /* no decode args */
332 +    {0, 1, "nop94", nop, pnop, "", {0}},  /* no decode args */
333 +    {0, 1, "nop95", nop, pnop, "", {0}},  /* no decode args */
334 +    {0, 1, "nop96", nop, pnop, "", {0}},  /* no decode args */
335 +    {0, 1, "nop97", nop, pnop, "", {0}},  /* no decode args */
336 +    {0, 1, "nop98", nop, pnop, "", {0}},  /* no decode args */
337 +    {0, 1, "nop99", nop, pnop, "", {0}},  /* no decode args */
338 +    {0, 1, "nop100", nop, pnop, "", {0}},  /* no decode args */
339 +    {0, 1, "nop101", nop, pnop, "", {0}},  /* no decode args */
340 +    {0, 1, "nop102", nop, pnop, "", {0}},  /* no decode args */
341 +    {0, 1, "nop103", nop, pnop, "", {0}},  /* no decode args */
342 +    {0, 1, "nop104", nop, pnop, "", {0}},  /* no decode args */
343 +    {0, 1, "nop105", nop, pnop, "", {0}},  /* no decode args */
344 +    {0, 1, "nop106", nop, pnop, "", {0}},  /* no decode args */
345 +    {0, 1, "nop107", nop, pnop, "", {0}},  /* no decode args */
346 +    {0, 1, "nop108", nop, pnop, "", {0}},  /* no decode args */
347 +    {0, 1, "nop109", nop, pnop, "", {0}},  /* no decode args */
348 +    {0, 1, "nop110", nop, pnop, "", {0}},  /* no decode args */
349 +    {0, 1, "nop111", nop, pnop, "", {0}},  /* no decode args */
350 +    {0, 1, "nop112", nop, pnop, "", {0}},  /* no decode args */
351 +    {0, 1, "nop113", nop, pnop, "", {0}},  /* no decode args */
352 +    {0, 1, "nop114", nop, pnop, "", {0}},  /* no decode args */
353 +    {0, 1, "nop115", nop, pnop, "", {0}},  /* no decode args */
354 +    {0, 1, "nop116", nop, pnop, "", {0}},  /* no decode args */
355 +    {0, 1, "nop117", nop, pnop, "", {0}},  /* no decode args */
356 +    {0, 1, "nop118", nop, pnop, "", {0}},  /* no decode args */
357 +    {0, 1, "nop119", nop, pnop, "", {0}},  /* no decode args */
358 +    {0, 1, "nop120", nop, pnop, "", {0}},  /* no decode args */
359 +    {0, 1, "nop121", nop, pnop, "", {0}},  /* no decode args */
360 +    {0, 1, "nop122", nop, pnop, "", {0}},  /* no decode args */
361 +    {0, 1, "nop123", nop, pnop, "", {0}},  /* no decode args */
362 +    {0, 1, "nop124", nop, pnop, "", {0}},  /* no decode args */
363 +    {0, 1, "nop125", nop, pnop, "", {0}},  /* no decode args */
364 +    {0, 1, "nop126", nop, pnop, "", {0}},  /* no decode args */
365 +    {0, 1, "nop127", nop, pnop, "", {0}},  /* no decode args */
366 +    {0, 1, "nop128", nop, pnop, "", {0}},  /* no decode args */
367 +    {0, 1, "nop129", nop, pnop, "", {0}},  /* no decode args */
368 +    {0, 1, "nop130", nop, pnop, "", {0}},  /* no decode args */
369 +    {0, 1, "nop131", nop, pnop, "", {0}},  /* no decode args */
370 +    {0, 1, "nop132", nop, pnop, "", {0}},  /* no decode args */
371 +    {0, 1, "nop133", nop, pnop, "", {0}},  /* no decode args */
372 +    {0, 1, "nop134", nop, pnop, "", {0}},  /* no decode args */
373 +    {0, 1, "nop135", nop, pnop, "", {0}},  /* no decode args */
374 +    {0, 1, "nop136", nop, pnop, "", {0}},  /* no decode args */
375 +    {0, 1, "nop137", nop, pnop, "", {0}},  /* no decode args */
376 +    {0, 1, "nop138", nop, pnop, "", {0}},  /* no decode args */
```

```
377 +     {0, 1, "nop139", nop, pnop, "", {0}},  /* no decode args */
378 +     {0, 1, "nop140", nop, pnop, "", {0}},  /* no decode args */
379 +     {0, 1, "nop141", nop, pnop, "", {0}},  /* no decode args */
380 +     {0, 1, "nop142", nop, pnop, "", {0}},  /* no decode args */
381 +     {0, 1, "nop143", nop, pnop, "", {0}},  /* no decode args */
382 +     {0, 1, "nop144", nop, pnop, "", {0}},  /* no decode args */
383 +     {0, 1, "nop145", nop, pnop, "", {0}},  /* no decode args */
384 +     {0, 1, "nop146", nop, pnop, "", {0}},  /* no decode args */
385 +     {0, 1, "nop147", nop, pnop, "", {0}},  /* no decode args */
386 +     {0, 1, "nop148", nop, pnop, "", {0}},  /* no decode args */
387 +     {0, 1, "nop149", nop, pnop, "", {0}},  /* no decode args */
388 +     {0, 1, "nop150", nop, pnop, "", {0}},  /* no decode args */
389 +     {0, 1, "nop151", nop, pnop, "", {0}},  /* no decode args */
390 +     {0, 1, "nop152", nop, pnop, "", {0}},  /* no decode args */
391 +     {0, 1, "nop153", nop, pnop, "", {0}},  /* no decode args */
392 +     {0, 1, "nop154", nop, pnop, "", {0}},  /* no decode args */
393 +     {0, 1, "nop155", nop, pnop, "", {0}},  /* no decode args */
394 +     {0, 1, "nop156", nop, pnop, "", {0}},  /* no decode args */
395 +     {0, 1, "nop157", nop, pnop, "", {0}},  /* no decode args */
396 +     {0, 1, "nop158", nop, pnop, "", {0}},  /* no decode args */
397 +     {0, 1, "nop159", nop, pnop, "", {0}},  /* no decode args */
398 +     {0, 1, "nop160", nop, pnop, "", {0}},  /* no decode args */
399 +     {0, 1, "nop161", nop, pnop, "", {0}},  /* no decode args */
400 +     {0, 1, "nop162", nop, pnop, "", {0}},  /* no decode args */
401 +     {0, 1, "nop163", nop, pnop, "", {0}},  /* no decode args */
402 +     {0, 1, "nop164", nop, pnop, "", {0}},  /* no decode args */
403 +     {0, 1, "nop165", nop, pnop, "", {0}},  /* no decode args */
404 +     {0, 1, "nop166", nop, pnop, "", {0}},  /* no decode args */
405 +     {0, 1, "nop167", nop, pnop, "", {0}},  /* no decode args */
406 +     {0, 1, "nop168", nop, pnop, "", {0}},  /* no decode args */
407 +     {0, 1, "nop169", nop, pnop, "", {0}},  /* no decode args */
408 +     {0, 1, "nop170", nop, pnop, "", {0}},  /* no decode args */
409 +     {0, 1, "nop171", nop, pnop, "", {0}},  /* no decode args */
410 +     {0, 1, "nop172", nop, pnop, "", {0}},  /* no decode args */
411 +     {0, 1, "nop173", nop, pnop, "", {0}},  /* no decode args */
412 +     {0, 1, "nop174", nop, pnop, "", {0}},  /* no decode args */
413 +     {0, 1, "nop175", nop, pnop, "", {0}},  /* no decode args */
414 +     {0, 1, "nop176", nop, pnop, "", {0}},  /* no decode args */
415 +     {0, 1, "nop177", nop, pnop, "", {0}},  /* no decode args */
416 +     {0, 1, "nop178", nop, pnop, "", {0}},  /* no decode args */
417 +     {0, 1, "nop179", nop, pnop, "", {0}},  /* no decode args */
418 +     {0, 1, "nop180", nop, pnop, "", {0}},  /* no decode args */
419 +     {0, 1, "nop181", nop, pnop, "", {0}},  /* no decode args */
420 +     {0, 1, "nop182", nop, pnop, "", {0}},  /* no decode args */
421 +     {0, 1, "nop183", nop, pnop, "", {0}},  /* no decode args */
422 +     {0, 1, "nop184", nop, pnop, "", {0}},  /* no decode args */
423 +     {0, 1, "nop185", nop, pnop, "", {0}},  /* no decode args */
424 +     {0, 1, "nop186", nop, pnop, "", {0}},  /* no decode args */
425 +     {0, 1, "nop187", nop, pnop, "", {0}},  /* no decode args */
426 +     {0, 1, "nop188", nop, pnop, "", {0}},  /* no decode args */
427 +     {0, 1, "nop189", nop, pnop, "", {0}},  /* no decode args */
428 +     {0, 1, "nop190", nop, pnop, "", {0}},  /* no decode args */
429 +     {0, 1, "nop191", nop, pnop, "", {0}},  /* no decode args */
430 +     {0, 1, "nop192", nop, pnop, "", {0}},  /* no decode args */
431 +     {0, 1, "nop193", nop, pnop, "", {0}},  /* no decode args */
432 +     {0, 1, "nop194", nop, pnop, "", {0}},  /* no decode args */
433 +     {0, 1, "nop195", nop, pnop, "", {0}},  /* no decode args */
434 +     {0, 1, "nop196", nop, pnop, "", {0}},  /* no decode args */
```

```
435 +     {0, 1, "nop197", nop, pnop, "", {0}},  /* no decode args */
436 +     {0, 1, "nop198", nop, pnop, "", {0}},  /* no decode args */
437 +     {0, 1, "nop199", nop, pnop, "", {0}},  /* no decode args */
438 +     {0, 1, "nop200", nop, pnop, "", {0}},  /* no decode args */
439 +     {0, 1, "nop201", nop, pnop, "", {0}},  /* no decode args */
440 +     {0, 1, "nop202", nop, pnop, "", {0}},  /* no decode args */
441 +     {0, 1, "nop203", nop, pnop, "", {0}},  /* no decode args */
442 +     {0, 1, "nop204", nop, pnop, "", {0}},  /* no decode args */
443 +     {0, 1, "nop205", nop, pnop, "", {0}},  /* no decode args */
444 +     {0, 1, "nop206", nop, pnop, "", {0}},  /* no decode args */
445 +     {0, 1, "nop207", nop, pnop, "", {0}},  /* no decode args */
446 +     {0, 1, "nop208", nop, pnop, "", {0}},  /* no decode args */
447 +     {0, 1, "nop209", nop, pnop, "", {0}},  /* no decode args */
448 +     {0, 1, "nop210", nop, pnop, "", {0}},  /* no decode args */
449 +     {0, 1, "nop211", nop, pnop, "", {0}},  /* no decode args */
450 +     {0, 1, "nop212", nop, pnop, "", {0}},  /* no decode args */
451 +     {0, 1, "nop213", nop, pnop, "", {0}},  /* no decode args */
452 +     {0, 1, "nop214", nop, pnop, "", {0}},  /* no decode args */
453 +     {0, 1, "nop215", nop, pnop, "", {0}},  /* no decode args */
454 +     {0, 1, "nop216", nop, pnop, "", {0}},  /* no decode args */
455 +     {0, 1, "nop217", nop, pnop, "", {0}},  /* no decode args */
456 +     {0, 1, "nop218", nop, pnop, "", {0}},  /* no decode args */
457 +     {0, 1, "nop219", nop, pnop, "", {0}},  /* no decode args */
458 +     {0, 1, "nop220", nop, pnop, "", {0}},  /* no decode args */
459 +     {0, 1, "nop221", nop, pnop, "", {0}},  /* no decode args */
460 +     {0, 1, "nop222", nop, pnop, "", {0}},  /* no decode args */
461 +     {0, 1, "nop223", nop, pnop, "", {0}},  /* no decode args */
462 +     {0, 1, "nop224", nop, pnop, "", {0}},  /* no decode args */
463 +     {0, 1, "nop225", nop, pnop, "", {0}},  /* no decode args */
464 +     {0, 1, "nop226", nop, pnop, "", {0}},  /* no decode args */
465 +     {0, 1, "nop227", nop, pnop, "", {0}},  /* no decode args */
466 +     {0, 1, "nop228", nop, pnop, "", {0}},  /* no decode args */
467 +     {0, 1, "nop229", nop, pnop, "", {0}},  /* no decode args */
468 +     {0, 1, "nop230", nop, pnop, "", {0}},  /* no decode args */
469 +     {0, 1, "nop231", nop, pnop, "", {0}},  /* no decode args */
470 +     {0, 1, "nop232", nop, pnop, "", {0}},  /* no decode args */
471 +     {0, 1, "nop233", nop, pnop, "", {0}},  /* no decode args */
472 +     {0, 1, "nop234", nop, pnop, "", {0}},  /* no decode args */
473 +     {0, 1, "nop235", nop, pnop, "", {0}},  /* no decode args */
474 +     {0, 1, "nop236", nop, pnop, "", {0}},  /* no decode args */
475 +     {0, 1, "nop237", nop, pnop, "", {0}},  /* no decode args */
476 +     {0, 1, "nop238", nop, pnop, "", {0}},  /* no decode args */
477 +     {0, 1, "nop239", nop, pnop, "", {0}},  /* no decode args */
478 +     {0, 1, "nop240", nop, pnop, "", {0}},  /* no decode args */
479 +     {0, 1, "nop241", nop, pnop, "", {0}},  /* no decode args */
480 +     {0, 1, "nop242", nop, pnop, "", {0}},  /* no decode args */
481 +     {0, 1, "nop243", nop, pnop, "", {0}},  /* no decode args */
482 +     {0, 1, "nop244", nop, pnop, "", {0}},  /* no decode args */
483 +     {0, 1, "nop245", nop, pnop, "", {0}},  /* no decode args */
484 +     {0, 1, "nop246", nop, pnop, "", {0}},  /* no decode args */
485 +     {0, 1, "nop247", nop, pnop, "", {0}},  /* no decode args */
486 +     {0, 1, "nop248", nop, pnop, "", {0}},  /* no decode args */
487 +     {0, 1, "nop249", nop, pnop, "", {0}},  /* no decode args */
488 +     {0, 1, "nop250", nop, pnop, "", {0}},  /* no decode args */
489 +     {0, 1, "nop251", nop, pnop, "", {0}},  /* no decode args */
490 +     {0, 1, "nop252", nop, pnop, "", {0}},  /* no decode args */
491 +     {0, 1, "nop253", nop, pnop, "", {0}},  /* no decode args */
492 +     {0, 1, "nop254", nop, pnop, "", {0}},  /* no decode args */
```

```
+    {0, 1, "nop255", nop, pnop, "", {0}},  /* no decode args */
+    {0, 1, "nop256", nop, pnop, "", {0}},  /* no decode args */
+    {0, 1, "nop257", nop, pnop, "", {0}},  /* no decode args */
+    {0, 1, "nop258", nop, pnop, "", {0}},  /* no decode args */
+    {0, 1, "nop259", nop, pnop, "", {0}},  /* no decode args */
+    {0, 1, "nop260", nop, pnop, "", {0}},  /* no decode args */
#0 Makefile.in change to non debug mode
diff --git a/tierra/Makefile.in b/tierra/Makefile.in
--- a/tierra/Makefile.in
+++ b/tierra/Makefile.in
@@ -10,1 +10,1 @@
-DEBUG=yes
+DEBUG=no
#1 GenPerFactor. Patch to include GenPerFactor as a soup_in parameter which edits the
# factor at which segment delete perturbations occur.
diff --git a/tierra/operator.c b/tierra/operator.c
--- a/tierra/operator.c
+++ b/tierra/operator.c
@@ -737,1 +737,1 @@ void DeletionSeg()
-        ODaughtNumSegs  = CountSegments(ODaughtGenStart, ODaughtGenSize);
+        ODaughtNumSegs  = CountSegments(ODaughtGenStart, ODaughtGenSize/GenPerFactor); /* Edit by Declan Baugh 2012 */
diff --git a/tierra/soup_in b/tierra/soup_in
--- a/tierra/soup_in
+++ b/tierra/soup_in
@@ -43,0 +43,1 @@ # environmental variables:
+GenPerFactor = 1
diff --git a/tierra/soup_in.h b/tierra/soup_in.h
--- a/tierra/soup_in.h
+++ b/tierra/soup_in.h
@@ -70,0 +70,3 @@ # environmental variables:
+I32s GenPerFactor = 1;     /*Edit by Declan Baugh 13th Nov 2012*/
+I32s DumpLogFreq = 1;
+I32s DumpLogMin = 0;
diff --git a/tierra/globals.h b/tierra/globals.h
--- a/tierra/globals.h
+++ b/tierra/globals.h
@@ -214,0 +214,3 @@
+extern I32s GenPerFactor; /*Edit by Declan Baugh 13th Nov 2012*/
+extern I32s DumpLogFreq;
+extern I32s DumpLogMin;
diff --git a/tierra/tsetup.c b/tierra/tsetup.c
--- a/tierra/tsetup.c
+++ b/tierra/tsetup.c
@@ -917,0 +917,14 @@ I8s GetAVar(data, alterflag, MonReq, buflen)
+        if (!strncmp((const char *)data, (const char *)"GenPerFactor", 12))
+        {   if (MonReq)
+            {   strcpy(((char *)&(vqu.name[0])),
+                    (const char *)"GenPerFactor");
+                tsprintf((char *)(&((vqu.value)[0])), "%d", GenPerFactor);
+            }
+            else if (alterflag)
+                sscanf((const char *)data,
+                    (const char *)"GenPerFactor = %d", &GenPerFactor);
+            vqu.type = 'i';
+            vqu.i = GenPerFactor;
+            rtncode=1;
```

```
+            break;
+        }   /* Edited by Declan Baugh declanbaugh@gmail.com 13th Nov 2012.
    Include arg 'h' parameter*/
@@ -3909,0 +3909,3 @@ void WriteSoup(close_disk)
+    tfprintf(ouf, "GenPerFactor = %d\n", GenPerFactor); /*Edit by Declan Baugh
    13th Nov 2012*/
+    tfprintf(ouf, "DumpLogFreq = %d\n", DumpLogFreq);   /*Edit by Declan Baugh
    13th Nov 2012*/
+    tfprintf(ouf, "DumpLogMin = %d\n", DumpLogMin);  /*Edit by Declan Baugh 13
    th Nov 2012*/
#2 Patch to create a 'h' parameter in the arg utility, to allow to dissassemble
    a creature in hex format
diff --git a/tierra/genio.c b/tierra/genio.c
--- a/tierra/genio.c
+++ b/tierra/genio.c
@@ -1988,0 +1988,205 @@ void WritHexFile()
+/*   Edit By Declan Baugh declanbaugh@gmail.com 13th Nov 2012.
+ * WritHexFile - write Hex instruction mnemonic list ("source file")
+ *               of genome ("disassembly" listing)
+ *
+ * g - pointer to GList structure for genome to be listed
+ * file - pointer to "source" file name
+ * tarpt -   0 - do not include thread analysis report in listing header
+ *         <>0 - include thread analysis report in listing header
+ * sucsiznslrat - size class migration destination node selection
+ *                "success" ratio
+ * sucsiznsl - size class migration destination node selection
+ *             "success" count
+ * siznsl    - size class migration destination node selection
+ *             attempt count
+ *
+ * detailrpt - 0 - include thread analysis summary
+ *           <>0 - include thread analysis detailed report
+ *
+ * clstrfmt  - 0 - no cluster analysis
+ *             1 - cluster analysis format
+ *
+ * symclstranafmt - 0 - asymmetric cluster analysis
+ *                  1 - symmetric cluster analysis
+ *
+ * exeptrn - 0 - non-execution pattern report
+ *           1 - execution pattern report
+ *
+ * expttsarr - pointer to execution pattern report
+ *             thread/"tissue" type array
+ * expttsarrcnt - count of entries in expttsarr array
+ * genelkup - gene lookup table
+ */
+void WritHexFile(g, file, tarpt, sucsiznslrat,
+    sucsiznsl, siznsl, detailrpt, clstrfmt,
+    symclstranafmt, exeptrn, expttsarr, expttsarrcnt,
+    genelkup, gendef, spltisana)
+Pgl g;
+I8s *file;
+I32s tarpt, detailrpt, clstrfmt, symclstranafmt,  *expttsarr, expttsarrcnt;
+double sucsiznslrat;
+I32s sucsiznsl, siznsl, exeptrn, spltisana;
+I16s *genelkup;
```

```
+GeneDefArr *gendef;
+{   I8s bit[4], chm[4];
+    I32u di, t;
+#if defined(TIERRA)||defined(ARGTIE)
+    I32s i;
+#endif /* defined(TIERRA)||defined(ARGTIE) */
+  I32s j;
+    time_t tp;
+    FILE *fp;
+
+#ifdef IBM3090
+    I8s lbl[4], plbl[4], *comnts;
+#endif
+    I8s gdt_gbits;
+
+#if defined(TIERRA)||defined(ARGTIE)
+    ThdTsTyArr codthdtstyarr[PLOIDY][NSTRTISTYP];
+    I8s fstlinprt=0,svincld,incld;
+    I32s lstmrk;
+#endif /* defined(TIERRA)||defined(ARGTIE) */
+
+    if (!strcmp((const char *)file, (const char *)"-"))
+        fp = stdout;
+    else if(!(fp=tfopen(file, (I8s *)"w")))
+    {   tsprintf((char *)(&(Fbuf[0])),
+           "Tierra WritHexFile() unable to open WritHexFile file %s", file);
+        porterrmsg(617,(char *)(&(Fbuf[0])),1);
+    }
+
+#if defined(TIERRA)||defined(ARGTIE)
+    if(tarpt||exeptrn)
+        WrtThrdAnalysis(fp, g,
+#ifdef NET
+            sucsiznslrat, sucsiznsl, siznsl,
+#else
+            0.0, 0, 0,
+#endif /* NET */
+            detailrpt, clstrfmt, symclstranafmt,
+            &codthdtstyarr, exeptrn, genelkup,
+            gendef, spltisana);
+#endif /* defined(TIERRA)||defined(ARGTIE) */
+
+#ifdef ARGTIE
+    if(!clstrfmt)
+    {
+#endif /* ARGTIE */
+        WritEcoB(g->bits, Buff);
+#ifdef IBM3090
+        strcpy(lbl, g->gen.label);
+        strcpy(plbl, g->parent.label);
+        Ascii2Ebcdic(lbl);
+        Ascii2Ebcdic(plbl);
+        tfprintf(fp,
+        "%.4d%s\n",
+            g->parent.size, plbl);
+#else
+        tfprintf(fp,
+        "%.4d%s\n",
```

```
+            g->parent.size, g->parent.label);
+#endif
+        tfprintf(fp,"%d\n",g->d1.BreedTrue);
+        tp = g->originC;
+        tfprintf(fp, "%d%.6d\n",
+                g->originI.m, g->originI.i);
+        tfprintf(fp, "");
+        chm[3] = bit[3] = 0;
+
+        for (j = 0; j < PLOIDY; j++)
+        {   if (j)
+
+            ExePtrnHead(fp, exeptrn, &(expttsarr[0]), expttsarrcnt);
+
+#if defined(TIERRA)||defined(ARGTIE)
+            incld=(I8s)(-1);
+            lstmrk=(I8s)(-1);
+#endif /* defined(TIERRA)||defined(ARGTIE) */
+
+            for (t = 0; t < (I32u)(g->gen.size); t++)
+            {
+#if PLOIDY == 1
+                di = g->genome[t];
+#else /* PLOIDY > 1 */
+                di = g->genome[t][j];
+#endif /* PLOIDY > 1 */
+                bit[0]='0';
+                bit[1]='0';
+                bit[2]='0';
+                gdt_gbits=0;
+                if(tarpt||exeptrn)
+                {   if(g->glst_thrdanadat.mtad_codsegbti.mgda_segbti)
+                        if(t<((I32u)(g->glst_thrdanadat.
+                            mtad_codsegbti.mgda_ctrl.dync_elmaloc)))
+                            if(g->glst_thrdanadat.mtad_codsegbti.
+                                mgda_segbti[t])
+                                gdt_gbits=g->glst_thrdanadat.
+                                    mtad_codsegbti.mgda_segbti[t]
+#if PLOIDY == 1
+                                    ->
+#else /* PLOIDY > 1 */
+                                    [j].
+#endif /* PLOIDY > 1 */
+                                    mgdt_gbdt.gdt_gbits;
+                }
+                else
+                {
+#if PLOIDY == 1
+                                gdt_gbits=g->gbits[t];
+#else /* PLOIDY > 1 */
+                                gdt_gbits=g->gbits[t][j];
+#endif /* PLOIDY == 1 */
+                }
+
+                bit[0]=IsBit((gdt_gbits), 0) ?'1':'0';
+                bit[1]=IsBit((gdt_gbits), 1) ?'1':'0';
+                bit[2]=IsBit((gdt_gbits), 2) ?'1':'0';
+
```

```
+#if defined(TIERRA)||defined(ARGTIE)
+                svincld=incld;
+                if((incld=IncLineChk(g,t,j,&lstmrk,
+                    exeptrn,&(expttsarr[0]),
+                    expttsarrcnt,&(codthdtstyarr[j][STRTISGENE])))
+                {
+                    if(!svincld)
+                        if(fstlinprt)
+                           tfprintf(fp, "\n\n\n");
+#endif /* defined(TIERRA)||defined(ARGTIE) */
+
+                    tfprintf(fp, "%.2x,", di);
+
+#if defined(TIERRA)||defined(ARGTIE)
+                    ExePtrnMrkLine(fp,g,t,j,exeptrn,
+                        &(expttsarr[0]),expttsarrcnt,
+                        &(codthdtstyarr[j][STRTISGENE]));
+#endif /* defined(TIERRA)||defined(ARGTIE) */
+
+#if defined(TIERRA)||defined(ARGTIE)
+                    fstlinprt=1;
+                }
+#endif /* defined(TIERRA)||defined(ARGTIE) */
+            }
+        }
+#ifdef ARGTIE
+    }
+#endif /* ARGTIE */
+
+#if defined(TIERRA)||defined(ARGTIE)
+    if(tarpt||exeptrn)
+        if(detailrpt||exeptrn)
+            for(i=0;i<PLOIDY;i++)
+                for(j=0;j<((genelkup)?NSTRTISTYP:1);j++)
+                    FreeDynArr((DynArr *)&(codthdtstyarr[i][j]),2212,0);
+#endif /* defined(TIERRA)||defined(ARGTIE) */
+
+    if(fp!=stdout)
+    {
+#ifndef AMIGA
+#ifndef DECVAX
+        tfflush(fp);
+#endif /* DECVAX */
+#endif /* AMIGA */
+        tfclose(fp);
+    }
+}
diff --git a/tierra/arg.c b/tierra/arg.c
--- a/tierra/arg.c
+++ b/tierra/arg.c
@@ -48,1 +48,2 @@ int main(argc, argv)
-   arg x[a[d[l][u[s]]]][v][i=<fract>] archive [genotype [genotype...]]\n";
+   arg x[a[d[l][u[s]]]][v][i=<fract>] archive [genotype [genotype...]]\n\
+   arg h[a[d[l][u[s]]]][v][i=<fract>] archive [genotype [genotype...]]\n"; /*
    Edit by Declan Baugh 13th Nov 2012 */
@@ -109,0 +109,1 @@ int main(argc, argv)
+        case 'h':  /*Edit by Declan Baugh 13th Nov 2012 */
@@ -416,0 +416,132 @@ int main(argc, argv)
```

```
+    /*Edit by Declan Baugh 13th Nov 2012 */
+    case 'h':
+        if(!strncmp(argv[argvidx],"i=",2))
+        {   LifeCycFrct=atof(&(argv[argvidx][2]));
+            argvidx++;
+            file = (I8s *)argv[argvidx];
+        }
+        if (!(afp = fopen((const char *)file, (const char *)"rb")))
+        {   perror(argv[0]);
+            exit(9);
+        }
+        head = read_head(afp);
+        indx = read_indx(afp, &head);
+        argvidx++;
+        if(argc>argvidx)
+            for(i = argvidx; i < argc; i++)
+            {   int j;
+                if ((j = find_gen(indx, (I8s *)argv[i], head.n)) == head.n)
+                {   fprintf(stderr, "%s not in archive\n", argv[i]);
+                    continue;
+                }
+                tindx = &indx[j];
+                g = get_gen(afp, &head, tindx, j);
+
+                ReadGenDef(&gendef, &genelkup,
+                    head.size, (I8s *)"", (I8s *)"");
+
+                file = (I8s *) malloc(8);
+                sprintf((char *)file, "%.4d%3s", head.size, g->gen.label);
+
+#ifdef NET
+                WritHexFile(g, file, tarpt,
+                    head.hdsucsiznslrat, head.hdsvsucsiznsl,
+                    head.hdsvsiznsl, dtlrpt, clstrfmt,
+                    symclstranafmt, exeptrn, &(expttsarr[0]), expttsarrcnt,
+                    genelkup, &gendef, spltisana);
+#else
+                WritHexFile(g, file, tarpt, 0.0, 0, 0,
+                    dtlrpt, clstrfmt, symclstranafmt,
+                    exeptrn, &(expttsarr[0]), expttsarrcnt,
+                    genelkup, &gendef, spltisana);
+#endif /* NET */
+                if (v)
+                {   fprintf(stdout, "x - %.4d%3s %.3f %.3f", g->gen.size,
+                        g->gen.label, g->MaxPropPop, g->MaxPropInst);
+                    if(IsBit(g->bits, 0))
+                    {   WritEcoB(g->bits, Buff);
+                        fprintf(stdout, " 1 %s\n", Buff);
+                    }
+                    else
+                        fprintf(stdout, " 0\n");
+                }
+                else
+                    fprintf(stdout, "x - %3s\n", g->gen.label);
+
+                FreeGenDef(&gendef, &genelkup, head.size);
+
+                if (file)
```

```
833 +                 {   free((void *) file);
834 +                     file = NULL;
835 +                 }
836 +                 if (g)
837 +                 {   if (g->genome)
838 +                     {   free((void *) g->genome);
839 +                         g->genome = NULL;
840 +                     }
841 +                     if (g->gbits)
842 +                     {   free((void *) g->gbits);
843 +                         g->gbits = NULL;
844 +                     }
845 +                     free((void *) g);
846 +                     g = NULL;
847 +                 }
848 +             }
849 +         else
850 +             for(i = 0; i < head.n; i++)
851 +             {   tindx = &indx[i];
852 +                 g = get_gen(afp, &head, tindx, i);
853 +
854 +                 ReadGenDef(&gendef, &genelkup,
855 +                     head.size, (I8s *)"", (I8s *)"");
856 +
857 +                 file = (I8s *) malloc(12);
858 +                 sprintf((char *)file, "%.4d%3s", head.size, g->gen.label);
859 +#ifdef NET
860 +                 WritHexFile(g, file, tarpt,
861 +                     head.hdsucsiznslrat, head.hdsvsucsiznsl,
862 +                     head.hdsvsiznsl, dtlrpt, clstrfmt,
863 +                     symclstranafmt, exeptrn,
864 +                     &(expttsarr[0]), expttsarrcnt, genelkup,
865 +                     &gendef, spltisana);
866 +#else
867 +                 WritHexFile(g, file, tarpt, 0.0, 0, 0,
868 +                     dtlrpt, clstrfmt, symclstranafmt, exeptrn,
869 +                     &(expttsarr[0]), expttsarrcnt, genelkup,
870 +                     &gendef, spltisana);
871 +#endif /* NET */
872 +
873 +
874 +
875 +                 if (v)
876 +                 {   fprintf(stdout, "h - %.4d%3s %.3f %.3f",
877 +                         g->gen.size, g->gen.label,
878 +                         g->MaxPropPop, g->MaxPropInst);
879 +                     if (IsBit(g->bits, 0))
880 +                     {   WritEcoB(g->bits, Buff);
881 +                         fprintf(stdout, " 1 %s\n", Buff);
882 +                     }
883 +                     else
884 +                         fprintf(stdout, " 0\n");
885 +                 }
886 +                 else
887 +                     fprintf(stdout, "h - %3s\n", g->gen.label);
888 +                 if (file)
889 +                 {   free((void *) file);
890 +                     file = NULL;
```

```
+                }
+                if (g)
+                {   if (g->genome)
+                    {   free((void *) g->genome);
+                        g->genome = NULL;
+                    }
+                    if (g->gbits)
+                    {   free((void *) g->gbits);
+                        g->gbits = NULL;
+                    }
+                    free((void *) g);
+                    g = NULL;
+                }
+            }
+        break;
+
# 3: Patch edit the GUI. Display seed value and start time.
diff --git a/tierra/frontend.c b/tierra/frontend.c
--- a/tierra/frontend.c
+++ b/tierra/frontend.c
@@ -2100,2 +2100,4 @@ void FEPlan(tielog)
-        "InstExeC  %8d  Generation%8.0f  %11d   %s",
-        InstExe.m, Generations, tp, ctime(&tp));
+        "InstExeC  %8d  Generation%8.0f  Seed%11d   %s",
+        InstExe.m, Generations, seed, ctime(&CpuLoadLimitLstSlp));
+/* Edited by Declan Baugh, Dublin City University, 2nd Nov 2012,
    declanbaugh@gmail.com
+ * Display Seed value and start time. */
# 5: Patch to fix clock bug, diving by zero.
diff --git a/tierra/bookeep.c b/tierra/bookeep.c
--- a/tierra/bookeep.c
+++ b/tierra/bookeep.c
@@ -474,0 +474,1 @@ void plan()
+        if (FESpeed < 1) FESpeed = 1;
@@ -477,0 +477,9 @@ void plan()
+
+/* Edited by Declan Baugh, Dublin City University, 2nd Nov 2012,
    declanbaugh@gmail.com
+ * Every 1,000,000 instructions Tierra will calculate and display
+ * the speed at which instructions are executed.
+ * If over 1,000,000 instructions are executed per second
+ * than the time difference between each period of 1,000,000
+ * instructions is zero seconds. This results
+ * in a division by zero and a 'floating point error'. */
+
# 6: Add an shlA instruction which doubles the contents of ax
diff --git a/tierra/soup_in.h b/tierra/soup_in.h
--- a/tierra/soup_in.h
+++ b/tierra/soup_in.h
@@ -294,0 +294,5 @@ InstDef idt[] =
+    {0, 1, "shlA", shl, dec1d, "a", {0}},  /* "r" */
+
+/* Edited by Declan Baugh, Dublin City University, 2nd Nov 2012,
    declanbaugh@gmail.com
+ * Additional instruction shlA() which doubles the contents of ax */
+
diff --git a/tierra/opcode.map b/tierra/opcode.map
--- a/tierra/opcode.map
```

```
+++ b/tierra/opcode.map
@@ -55,0 +55,1 @@
+    {0, 1, "shlA", shl, dec1d, "a", ""},              /* "r" */
#
# 7:Add an incD instruction which increments dx by 1.
diff --git a/tierra/soup_in.h b/tierra/soup_in.h
--- a/tierra/soup_in.h
+++ b/tierra/soup_in.h
@@ -245,0 +245,5 @@ InstDef idt[] =
+    {0, 1, "incD", add, dec1d1s, "dd", {0}},  /* "cc" */
+
+/* Edited by Declan Baugh, Dublin City University, 2nd Nov 2012,
    declanbaugh@gmail.com
+ * Additional instruction incD() which increments dx by 1 */
+
diff --git a/tierra/opcode.map b/tierra/opcode.map
--- a/tierra/opcode.map
+++ b/tierra/opcode.map
@@ -25,0 +25,1 @@
+    {0, 1, "incD", add, dec1d1s, "dd", ""},          /* "cc" */
#
# 8: Add an addEAA instruction which adds contents of ex to ax, and places the
    result in ax
diff --git a/tierra/soup_in.h b/tierra/soup_in.h
--- a/tierra/soup_in.h
+++ b/tierra/soup_in.h
@@ -208,0 +208,5 @@ InstDef idt[] =
+    {0, 1, "addAAE", add, dec1d2s, "ffd", {0}},  /* "rrr" */
+
+/* Edited by Declan Baugh, Dublin City University, 2nd Nov 2012,
    declanbaugh@gmail.com
+ * Additional instruction addAAE() which places the sum of ex and ax into ax */
+
diff --git a/tierra/opcode.map b/tierra/opcode.map
--- a/tierra/opcode.map
+++ b/tierra/opcode.map
@@ -6,0 +6,1 @@
+    {0, 1, "addAAE", add, dec1d2s, "ffd", ""},          /* "rrr" */
#
# 9: Add an ifnz (if not zero) instruction which performs next instruction only
    if cx != 0,
diff --git a/tierra/decode.c b/tierra/decode.c
--- a/tierra/decode.c
+++ b/tierra/decode.c
@@ -109,0 +109,10 @@ void dec2s()
+        case 'n':  /* ifnz() */
+        {   is.sval = (is.sval != 0);
+            is.sval2 = 2;
+            break;
+        }
+
+/* Edited by Declan Baugh, Dublin City University, 2nd Nov 2012,
    declanbaugh@gmail.com
+ * Additional instruction ifnz() which performs next instruction only if cx !=
    0,
+ * If cx == 0, the instruction is skipped */
+
diff --git a/tierra/soup_in.h b/tierra/soup_in.h
```

```
--- a/tierra/soup_in.h
+++ b/tierra/soup_in.h
@@ -241,0 +241,6 @@ InstDef idt[] =
+    {0, 1, "ifnz", skip, dec2s, "", {0}},  /* "rr" */
+
+/* Edited by Declan Baugh, Dublin City University, 2nd Nov 2012,
    declanbaugh@gmail.com
+ * Additional instruction ifnz() which performs next only instruction if cx !=
    0,
+ * If cx == 0, the instruction is skipped */
+
diff --git a/tierra/opcode.map b/tierra/opcode.map
--- a/tierra/opcode.map
+++ b/tierra/opcode.map
@@ -22,0 +22,1 @@
+    {0, 1, "ifnz", skip, dec2s, "cc", ""},           /* "rr" */
# 10: Add movAb() instruction
diff --git a/tierra/soup_in.h b/tierra/soup_in.h
--- a/tierra/soup_in.h
+++ b/tierra/soup_in.h
@@ -262,0 +262,6 @@ InstDef idt[] =
+    {0, 1, "movAb", movdi, pmovdi, "ab", {0}},  /* "rr" */
+
+/* Edited by Declan Baugh, Dublin City University, 2nd Nov 2012,
    declanbaugh@gmail.com
+ * Additional instruction movAb copies the contents of the
+ * bx register, to the address pointed at by the ax register */
+
diff --git a/tierra/opcode.map b/tierra/opcode.map
--- a/tierra/opcode.map
+++ b/tierra/opcode.map
@@ -32,0 +32,1 @@
+    {0, 1, "movAb", movdi, pmovdi, "ab", {0}},  /* "rr" */
# 11: Add movda() instruction
diff --git a/tierra/soup_in.h b/tierra/soup_in.h
--- a/tierra/soup_in.h
+++ b/tierra/soup_in.h
@@ -263,0 +263,6 @@ InstDef idt[] =
+    {0, 1, "movda", movii, pmovii, "da", {0}},  /* "rr" */
+
+/* Edited by Declan Baugh, Dublin City University, 2nd Nov 2012,
    declanbaugh@gmail.com
+ * Additional instruction movda copies the contents of the
+ * ax register, to the dx register */
+
diff --git a/tierra/opcode.map b/tierra/opcode.map
--- a/tierra/opcode.map
+++ b/tierra/opcode.map
@@ -33,0 +33,1 @@
+    {0, 1, "movda", movii, pmovii, "da", {0}},  /* "rr" */
# 12: Add an movab() instruction
diff --git a/tierra/soup_in.h b/tierra/soup_in.h
--- a/tierra/soup_in.h
+++ b/tierra/soup_in.h
@@ -264,0 +264,6 @@ InstDef idt[] =
+    {0, 1, "movab", movii, pmovii, "ab", {0}},  /* "rr" */
+
```

```
+/* Edited by Declan Baugh, Dublin City University, 2nd Nov 2012,
    declanbaugh@gmail.com
+ * Additional instruction movab copies the contents of the
+ * bx register to the ax register */
+
diff --git a/tierra/opcode.map b/tierra/opcode.map
--- a/tierra/opcode.map
+++ b/tierra/opcode.map
@@ -34,0 +34,1 @@
+    {0, 1, "movab", movii, pmovii, "ab", {0}},  /* "rr" */
#
# 13: Patch to fix the jump instruction bug, to ensure a jump with not address
    will not pop Bx register into ip address.
diff --git a/tierra/decode.c b/tierra/decode.c
--- a/tierra/decode.c
+++ b/tierra/decode.c
@@ -1155,1 +1155,7 @@ void decjmp()
-    is.sval = ad(tval);  /* target for IP if s == 0 */
+    is.sval = ad(ce->c.c->ip + 1);  /* target for IP if s == 0 */
+
+/* Edited by Declan Baugh, Dublin City University, 2nd Nov 2012,
    declanbaugh@gmail.com
+ * If a jump template is missing, instruction pointer will simply continue
+ * to next instruction in the soup, and will not jump to the location pointed
+ * at by the bx register. */
+
#
# 14: Patch to ensure return instruction never pops zero to the instruction
    pointer.
diff --git a/tierra/instruct.c b/tierra/instruct.c
--- a/tierra/instruct.c
+++ b/tierra/instruct.c
@@ -1519,17 +1519,32 @@ void pop()
     if((is.dreg)==(&(ce->c.c->ip)))
-    {   *(is.dreg)=ad(adr1);
-        ce->c.c->retins=1;
+    {
+        if(ad(adr1) != 0)
+        {   *(is.dreg)=ad(adr1);
+            ce->c.c->retins=1;
+            if (!ce->c.c->sp)
+            ce->c.c->sp = STACK_SIZE - 1;
+            else
+            --ce->c.c->sp;
+        }
+        else
+            is.iip = 1;
     }
     else
+    {
         *(is.dreg)=adr1;
     if (!ce->c.c->sp)
         ce->c.c->sp = STACK_SIZE - 1;  /* decr stack pointer */
     else
         --ce->c.c->sp;
+    }
     DoMods();
     DoFlags();
```

```
1105  #if PLOIDY > 1
1106       JumpTrack();
1107  #endif /* PLOIDY > 1 */
1108  }
1109 +
1110 +/* Edited by Declan Baugh, Dublin City University, 2nd Nov 2012,
        declanbaugh@gmail.com
1111 + * Here we ensure that any return to zero acts like a nop.*/
1112 +
```

Appendix E

Python Analysing Tools

This section contains the code which was written to create a number of data analysing tools, written in Python, which were necessary to create in order to analyse the output to the experiments which were run.

E.1 Compare Population Sizes

Listing E.1: Compare Population Sizes

```
#!/usr/bin/env python
# This piece of code will count the number of creatures in the soup at the end of a
# run, which have a specific starting template address. The decendants of an ancestor
# with a unique start template address may be counted
import os
import subprocess
import sys

# --------------------------------------
# Variables and accumulators

evolved_0316aab = "2a,01,01,00,00"
initial_0316aaa = "2a,01,00,01,00"

# --------------------------------------
# Extract all creatures from the genebank and create a dictionary with {Time : Creature Name}
os.chdir("tierra/" + sys.argv[1])
subprocess.call(["../arg","h", "0316.gen"])

number1 = []
number2 = []
files = os.listdir(".")

for name in files:
    non_funct_count = 0
    if "0316" in name and ".gen" not in name and ".tmp" not in name and "virgin" not in name and "~" not in name:
        lines = open(name,"r").readlines()
```

```
        inst_exe = lines[1][0:-1]
        first_template = lines[3][0:15]
        if initial_0316aaa in first_template:
            number1.append(int(inst_exe))
        if evolved_0316aab in first_template:
            number2.append(int(inst_exe))

print sorted(number1), len(number1)
print sorted(number2), len(number2)

print len(number1)
print len(number2)

print max(number1)
print max(number2)
```

E.2 Count Employed Symbols

Listing E.2: Count Employed Symbols

```
#!/usr/bin/env python
# This piece of code will extract all the creatures from the genbank within a certain
# size range, and print out a list of the names, time of emergence and number of
# distinct employed symbols within the look-up table.

import os
import subprocess
import re
from collections import Counter

# ---------------------------------

# Variables

min_size = 316
max_size = 316
number_of_specimens = 10
os.chdir("tierra/gb0")

# ---------------------------------

# Extract all archives
files = os.listdir(".")
creature_times = {}
for creature_size in range(min_size,max_size+1):
    genotype = "0"+str(creature_size)+".gen"
    if genotype in files:
        subprocess.call(["../arg","x", genotype])
        files = os.listdir(".")
        for genome in files:
            if str(creature_size) in genome and ".gen" not in genome:
                genome_file = open(genome,"r")

                match = re.search("InstExe: \d+",genome_file.read()).group()
                creature_times[int(match[9::])] = genome
                genome_file.close()
# ---------------------------------

# Now create a list of all the times, from newest decendant to initial ancestor
# Also create a matching list of names.

times_list = sorted(creature_times.keys())[-1:-number_of_specimens-1:-1]
names_list = []
for i in range(0,len(times_list)): names_list.append(creature_times[times_list[i]])

# ---------------------------

# I now have a list of creatures and their times.
# Times list dictionary { time : creature_name }
total = []
change = ""
```

```
look_up_table ="00,01,..,..,04,..,06,..,08,..,0a,..,0c,..,0e
    ,..,10,..,12,..,14,..,..,..,18,..,1a,..,1c,..,1e
    ,..,20,..,22,..,24,..,..,..,28,..,2a,..,..,..,2e
    ,..,30,..,32,..,34,..,36,..,..,39,..,..,3c,..,..,3f"

for name in names_list:
    subprocess.call(["../arg","h", name[0:4:]+".gen", name[4::]])
    creature_code = open(name,"r")
    x = creature_code.read()
    match = re.search(look_up_table,x)

    if match:
        match = match.group()
        total += match.split(",")
        creature_code.close()

    else:
        change = change + name + ","
        creature_code.close()

print names_list
print times_list
#print Counter(total)
#print change + ","
print len(Counter(total))
```

E.3 Count Employed Symbols II

Listing E.3: Count Employed Symbols II

```
#!/usr/bin/env python
# This Piece of code will open the soup_dump file. For every creature
# that exists, it will search for it in the genebank, find its time
# of emergence, count the number of distinct p-symbols in its look-up
# table, and write two arrays to file, containing a list of each
# time of emergence, and the number of distinct p-symbols in its look-up table

from collections import Counter
import os

f = open("tierra/gb1/soup_dump","r").readlines()

files = os.listdir("tierra/gb1/.")

arr = []
lutCount = []
instExe = []
popCount = 0
for i in f[1:]:
    if "Timestamp" in i and popCount != 0:
        arr.append(sum(lutCount)/(popCount))
        lutCount = []
        popCount = 0

    else:
        if i[:7] in files:
            creature = open("tierra/gb1/"+i[:7],"r").readlines()

            if int(creature[1]) == 1:
                lutCount.append(len(Counter(creature[3][351:1118].split(",")
                    [1::2]))*int(i[8:13].lstrip()))
                instExe.append(creature[2][:-1])


                popCount = popCount + float(i[8:13].lstrip())

temp = ""
for i in str(instExe)[1:-1]:

    if "," in i:
        temp = temp + "\n"
    elif "'" in i or " " in i:
        temp = temp
    else:
        temp = temp + i
instExe = temp


temp = ""
for i in str(arr)[1:-1]:

    if "," in i:
        temp = temp + "\n"
    elif "'" in i or " " in i:
```

```
54          temp = temp
55      else:
56          temp = temp + i
57  arr = temp
58
59  g = open("TranslationTableCount","w")
60  f = open("InstructionExecuted","w")
61  g.write(arr)
62  f.write(instExe)
63  f.close()
64  g.close()
```

E.4 Average Employed vs. Non Employed Symbol Count

Listing E.4: Average Employed vs Non Employed Symbol Count

```
#!/usr/bin/env python
# This Piece of code calculates the average number of employed vs unemployed
# symbols, within the look_up tables of alive creatures at the end of a run.

import os
import subprocess
import re
from collections import Counter

# ------------------------------------

# Input Variables

min_size = 316
max_size = 316
number_of_specimens =0
os.chdir("tierra/gb0")

# ---------------------------------

# Extract all creatures from the genebank and create a dictionary with {Time :
    Genome}
files = os.listdir(".")
creature_times = {}
for creature_size in range(min_size,max_size+1):
    genotype = "0"+str(creature_size)+".gen"
    if genotype in files:
        subprocess.call(["../arg","x", genotype])
        files = os.listdir(".")
        for genome in files:
            if str(creature_size) in genome and ".gen" not in genome:
                genome_file = open(genome,"r")
                match = re.search("InstExe: \d+",genome_file.read())
                if match:
                    match = match.group()
                    creature_times[int(match[9::])] = genome
                    genome_file.close()

# --------------------------------

# Now create a list of all the times, from newest decendant to initial ancestor
# Also create a matching list of names.
times_list = sorted(creature_times.keys())[-1:number_of_specimens:-1]
names_list = []
for i in range(0,len(times_list)): names_list.append(creature_times[times_list[i
    ]])

# --------------------------------

# Define accumulators and arrays.
allLookUpTables = []
functionalInst = 0
nonFunctionalInst = 0
changed_GPmapping = ""
```

```
look_up_table ="00,01,..,..,04,..,06,..,08,..,0a,..,0c,..,0e
    ,..,10,..,12,..,14,..,..,..,18,..,1a,..,1c,..,1e
    ,..,20,..,22,..,24,..,..,..,28,..,2a,..,..,..,2e
    ,..,30,..,32,..,34,..,36,..,..,39,..,..,3c,..,..,3f"
functionalList = ['00','01','04','06','08','0a','0c','0e','10','12','14','18','1
    a','1c','1e','20','22','24','28','2a','2e','30','32','34','36','39','3c','3f
    ']

# ----------------------------------------
# Now search for all the lookup tables and append them to an initially empty
    list.
for name in names_list:
    subprocess.call(["../arg","h", name[0:4:]+".gen", name[4::]])
    creature_code = open(name,"r")
    x = creature_code.read()
    match = re.search(look_up_table,x)

    if match: # If lookup table is found, append the LUT to the accumulater list
         "allLookUpTables"
        match = match.group()
        allLookUpTables += match.split(",")
        creature_code.close()

    else: # Creatures which have a change in the functional instructions of the
        lookup table
        changed_GPmapping = changed_GPmapping + name + ","
        creature_code.close()

# ------------------------------------

# Count the number functional and nonfunctiona instructions in the list "
    allLookUpTables" which contains all the lookup tables.

total_num_creatures = len(allLookUpTables)/len(look_up_table.split(","))
instructions = sorted(Counter(allLookUpTables).keys())

for instruction in instructions:
    if instruction in functionalList:
        functionalInst += float(Counter(allLookUpTables)[instruction])
    else:
        nonFunctionalInst += float(Counter(allLookUpTables)[instruction])

print functionalInst/total_num_creatures
print nonFunctionalInst/total_num_creatures
```

E.5 Creature Population Graph I

Listing E.5: Creature Population Graph I

```
#!/usr/bin/env python
# This piece of code will take a creature name as input, and search the Tierra Run
# file, to and produce a graph to produce a graph showing its population within
# the soup from its emergence to time of extinction.

import os
import matplotlib.pyplot as plt

# ------------------------------

os.chdir("tierra/gb0")
runfile = open("tierra.run","r").readlines()

# ------------------------------

temp = []

for line in runfile:
    if "316aaa" in line:
        temp.append(line)

inst_count = []
pop_count = []
pop = 0

for line in temp:
    if "b" in line:
        pop += 1
        pop_count.append(pop)
        inst_count.append(int(line.split(" ")[0]))

    else:
        pop -= 1
        pop_count.append(pop)
        inst_count.append(int(line.split(" ")[0]))

print max(inst_count)

# Create data
x_values = inst_count
y_values = pop_count

# Plot Data
plt.plot(x_values[-400:], y_values[-400:], label = "0316aaa population" )


# Add in labels and the title

plt.xlabel("Instructions Executed")
plt.ylabel("Creature Count")
plt.title("Population of 0316aaa")

#Create legend and save image to png
```

```
54 plt.legend(loc="upper right")
55
56 os.chdir("../..")
57 plt.savefig("0316.png")
```

E.6 Creature Population Graph II

Listing E.6: Creature Population Graph II

```
#!/usr/bin/env python
# This piece of code will take a creature name as input
# and produce a graph, showing its population within
# the soup from its emergence to time of extinction.

import os
import matplotlib.pyplot as plt

# ----------------------------------

os.chdir("tierra/gb0")
runfile = open("tierra.run","r").readlines()

# ----------------------------------
temp = []

for line in runfile:
    if "316aaa" in line:
        temp.append(line)

inst_count = []
pop_count = []
pop = 0

for line in temp:
    if "b" in line:
        pop += 1
        pop_count.append(pop)
        inst_count.append(int(line.split(" ")[0]))

    else:
        pop -= 1
        pop_count.append(pop)
        inst_count.append(int(line.split(" ")[0]))


print max(inst_count)

# Create data
x_values = inst_count
y_values = pop_count


# Plot Data
plt.plot(x_values[-400:], y_values[-400:], label = "0316aaa population" )

# Add in labels and the title

plt.xlabel("Instructions Executed")
plt.ylabel("Creature Count")
plt.title("Population of 0316aaa")

#Create legend and save image to png
plt.legend(loc="upper right")
```

```
55
56 os.chdir("../..")
57 plt.savefig("0316.png")
```

E.7 Employed Symbol Graph

Listing E.7: Employed Symbol Graph

```
#!/usr/bin/env python
# This piece of code will search the genbank for every creature
# of a certain length, and produce a graph of the look-up table
# count, and time of emergence of each creature to emerge.

import os
import subprocess
import re
from collections import Counter
import sys

# ---------------------------------
# Variables and accumulators

min_size = 316
max_size = 316
number_of_specimens = 100
genebank = sys.argv[1]
smoothing_average = 50


allLookUpTables = []
functionalInst = 0
nonFunctionalInst = 0
changed_GPmapping = ""
functionalList = ['00','01','04','06','08','0a','0c','0e','10','12','14','18','1
    a','1c','1e','20','22','24','28','2a','2e','30','32','34','36','39','3c','3f
    ']

# ------------------------------

# Extract all creatures from the genebank and create a dictionary with {Time :
    Creature Name}
os.chdir("tierra/gb"+str(genebank))
files = os.listdir(".")
creature_times = {}
for creature_size in range(min_size,max_size+1):
    genotype = "0"+str(creature_size)+".gen"
    if genotype in files:
        subprocess.call(["../arg","x", genotype])
        files = os.listdir(".")
        for genome in files:
            if str(creature_size) in genome and ".gen" not in genome:
                inst_exe_line = open(genome,"r").readlines()[7][17:]
                inst_exe = re.search("\d+",inst_exe_line)
                if inst_exe:
                    inst_exe = inst_exe.group()
                    creature_times[int(inst_exe)] = genome

# ----------------------------

# Now create a list of all the times, from newest decendant to initial ancestor
# Also create a matching list of names.

```

```
52 times_list = sorted(creature_times.keys())
53 names_list = []
54 for i in range(0,len(times_list)): names_list.append(creature_times[times_list[i
       ]])
55
56 # -----------------------------
57 # Times list dictionary { time : creature_name }
58 total_instruction_count = []
59 dictionary = {}
60 i = 0
61 for name in names_list:
62
63     subprocess.call(["../arg","h", name[0:4:]+".gen", name[4::]])
64     look_up_table = open(name,"r").readlines()[0][264:455].split(",")
65     instructions = Counter(look_up_table).keys()
66
67     for instruction in instructions:
68
69         if instruction in functionalList:
70             functionalInst += float(Counter(look_up_table)[instruction])
71         else:
72             nonFunctionalInst += float(Counter(look_up_table)[instruction])
73     dictionary[times_list[i]] = [functionalInst, nonFunctionalInst]
74     functionalInst = 0
75     nonFunctionalInst = 0
76     i += 1
77
78 # -----------------------------
79 #Set up matplotlib and the figure
80 import matplotlib.pyplot as plt
81 plt.figure()
82
83 # Create data
84 x_values = times_list
85 y_series_1 = [dictionary[x][0] for x in x_values]
86 y_series_2 = [dictionary[x][1] for x in x_values]
87
88
89 # Smooting average filter
90 def smoothing_filter(y_series):
91     temp = []
92     for i in range(0,len(y_series)-smoothing_average):
93         j = 0
94         while j < smoothing_average:
95             y_series[i] += y_series[i+j]
96             j += 1
97         y_series[i] /= float(smoothing_average)
98         temp.append(y_series[i])
99     return list(temp)
100
101 x_values = x_values[:-smoothing_average:]
102 y_series_1 = smoothing_filter(y_series_1)
103 y_series_2 = smoothing_filter(y_series_2)
104
105 # Plot Data
106 plt.plot(x_values, y_series_1, label = "Functional Instructions" )
107 plt.plot(x_values, y_series_2, label = "Nonfunctional Instructions")
108
```

```
# Add in labels and the title

plt.xlabel("Instructions Executed")
plt.ylabel("Instruction Count")
plt.title("Instruction Count Within the Lookup Table of Offspring")

# Add limits to the x&y axis
#plt.xlim(0, 6)
plt.ylim(0,70)
#Create legend and save image to png
plt.legend(loc="center right")

os.chdir("../..")
plt.savefig("graph"+str(genebank)+".png")
```

E.8 Increase In Employed Symbols

Listing E.8: Increase In Employed Symbols

```
#!/usr/bin/env python
# This piece of code will leek at ever creature within the genebank.
# It will count the number of distinct symbols within both it and
# its parents look-up table. Any instance where the look-up
# table count has increased by one, is flagged for investigation
# as an example of a symbol being added to the g-p mapping

import subprocess
from collections import Counter
import sys
import os
import re

os.chdir("tierra/gb1/")
subprocess.call(["../arg","h", "0316.gen"])
files = os.listdir(".")

lut ="00,01,..,..,04,..,06,..,08,..,0a,..,0c,..,0e
    ,..,10,..,12,..,14,..,..,..,18,..,1a,..,1c,..,1e
    ,..,20,..,22,..,24,..,..,..,28,..,2a,..,..,..,2e
    ,..,30,..,32,..,34,..,36,..,..,39,..,..,3c,..,..,3f"

x = 0
y = 0
gtype = []
for name in files:
    if "316" in name and ".gen" not in name and ".tmp" not in name:
        lines = open(name,"r").readlines()
        look_up_table = lines[3][264:455].split(",")

        if re.search(lut,lines[3][264:455]):
            if len(Counter(look_up_table)) > 28:
                parent = lines[0][0:7]
                if parent == "0000god":
                    continue
                parent_lines = open(parent,"r").readlines()
                grandparent =  parent_lines[0][0:7]
                if grandparent == "0000god":
                    continue
                grandparent_lines = open(grandparent,"r").readlines()

                child_array = lines[3].split(",")
                parent_array = parent_lines[3].split(",")
                grandparent_array = grandparent_lines[3].split(",")

                a = 0
                b = 0
                pvalue = 0
                for value in child_array:

                    if value == parent_array[a]:
                        a += 1
                    else:
                        pvalue = a
```

```
52                          a += 1
53                          b += 1
54
55                  if b == 1:
56                      gvalue = 0
57                      c = 0
58                      d = 0
59
60                      for values in parent_array:
61                          if values == grandparent_array[c]:
62                              c += 1
63
64                          else:
65                              gvalue = c
66                              c += 1
67                              d += 1
68
69                      if d == 1:
70 # Now there is only a single difference between parent and daughter, and parent
       and grand parent
71
72                          if lines[3][264:455] not in parent_lines[3][264:455] and
                             parent_lines[3][732:923] not in grandparent_lines
                            [3][732:923]:
73                              print "This is an example of genotypic mutation
                                    increasing the lookup table"
74                              x += 1
75                              gtype.append(name)
76
77                              print gtype, pvalue+156, gvalue
```

E.9 Average Employed Symbol Count

Listing E.9: Average Employed Symbol Count

```
#!/usr/bin/env python
# This piece of code searches the genebank for creatures who emerged
# within a certain time frame and counts the average number
# of distinct symbols within the look-up tables
# within these creatures

import os
import subprocess
import re
from collections import Counter

# ---------------------------
# Variables and accumulators

min_size = 316
max_size = 316
number_of_specimens = 100
os.chdir("tierra/gb5")
allLookUpTables = []
functionalInst = 0
nonFunctionalInst = 0
changed_GPmapping = ""
functionalList = ['00','01','04','06','08','0a','0c','0e','10','12','14','18','1a','1c','1e','20','22','24','28','2a','2e','30','32','34','36','39','3c','3f']

# ----------------------------
# Extract all creatures from the genebank and create a dictionary with {Time : Creature Name}
files = os.listdir(".")
creature_times = {}
for creature_size in range(min_size,max_size+1):
    genotype = "0"+str(creature_size)+".gen"
    if genotype in files:
        subprocess.call(["../arg","x", genotype])
        files = os.listdir(".")
        for genome in files:
            if str(creature_size) in genome and ".gen" not in genome:
                inst_exe_line = open(genome,"r").readlines()[7][17:]
                inst_exe = re.search("\d+",inst_exe_line)
                if inst_exe:
                    inst_exe = inst_exe.group()
                    creature_times[int(inst_exe)] = genome

# -------------------------------

# Now create a list of all the times, from newest decendant to initial ancestor
# Also create a matching list of names.

times_list = sorted(creature_times.keys())[-1:-number_of_specimens-1:-1]
names_list = []
for i in range(0,len(times_list)): names_list.append(creature_times[times_list[i]])

```

```
# --------------------------------


# Times list dictionary { time : creature_name }
total_instruction_count = []
change = ""

for name in names_list:

    subprocess.call(["../arg","h", name[0:4:]+".gen", name[4::]])
    look_up_table = open(name,"r").readlines()[0][264:455]
    total_instruction_count += look_up_table.split(",")

instruct_dic = Counter(total_instruction_count)

# -----------------------------------

# Count the number functional and nonfunctional
# instructions in the list "allLookUpTables" which contains all the lookup
    tables.

# total_num_creatures = len(total_instruction_count)/len(look_up_table.split
    (","))
instructions = instruct_dic.keys()
for instruction in instructions:
    if instruction in functionalList:
        functionalInst += float(instruct_dic[instruction])
    else:
        nonFunctionalInst += float(instruct_dic[instruction])
print "For the last %d creatures to exist in the soup." %number_of_specimens
print "%s out of a possible 64 different instruction types were found within the
     phenotype space." % len(instruct_dic)

print "%f individual instructions were functional." % (functionalInst/
    number_of_specimens)
print "%f individual instructions were non functional." % (nonFunctionalInst/
    number_of_specimens)
```

E.10 Lineage Tracer

Listing E.10: Lineage Tracer

```
#!/usr/bin/env python
# This Piece of code will take a creature as an input, and find its
# complete lineage, from itself up to the seed ancestor

# >>> lineage.py [decendent_name]

import os
import subprocess
import sys

# Input Variables
decendant = sys.argv[1]
os.chdir("tierra/gb1")
print sys.argv

# Extract all creatures from the genebank
files = os.listdir(".")
for gen_file in files:
    if ".gen" in gen_file:
        subprocess.call(["../arg","x", gen_file])

# Find lineage
lineage = [decendant]
while decendant in os.listdir("."):
    parent = open(decendant,"r").readlines()[2][52:59]
    lineage.append(parent)
    decendant = parent

os.chdir("../..")
f = open(sys.argv[1]+"_lineage","w")
s = str(lineage)
f.write(s)
f.close()

print lineage
print sys.argv[1]
```

E.11 Change in Look-Up Table

Listing E.11: Change in Look-Up Table

```
#!/usr/bin/env python
#This piece of code will scan the entire genbank for a run.
# at each creature and its parent, and check if there is one,
# and only one change in the look-up table between it and its
# and it's parent. A list of every creature that had a
# change to its look-up table will be written to file to be
# Investigated further

import subprocess
from collections import Counter
import sys
import os
import re

f = open(sys.argv[1]+"_genebank","w")
os.chdir("tierra/" + sys.argv[1])
subprocess.call(["../arg","h", "0316.gen"])

files = os.listdir(".")
i = 0
a = 0
b = 0
c = 0
lut ="00,01,..,..,04,..,06,..,08,..,0a,..,0c,..,0e
    ,..,10,..,12,..,14,..,..,..,18,..,1a,..,1c,..,1e
    ,..,20,..,22,..,24,..,..,..,28,..,2a,..,..,..,2e
    ,..,30,..,32,..,34,..,36,..,..,39,..,..,3c,..,..,3f"


for name in files:
    if "316" in name and ".gen" not in name and ".tmp" not in name:
        lines = open(name,"r").readlines()
        look_up_table = lines[3][264:455].split(",")

        if len(Counter(look_up_table)) > 28:

            if re.search(lut,lines[3][264:455]):

                parent = lines[0][0:7]
                parent_lines = open(parent,"r").readlines()
                parent_look_up_table = parent_lines[3][264:455].split(",")
                if len(Counter(parent_look_up_table)) < len(Counter(
                    look_up_table)):

                    if parent_lines[3][732:923] in lines[3][732:923]:

                        string = "(A) "+str(len(Counter(parent_look_up_table)))
                            + " " + str(len(Counter(look_up_table)))+ " YES\n"
                        f.write(string)
                        a = a + 1


                    else:
                        j = 0
```

```
                        k = 0
                        parent_array = parent_lines[3].split(",")
                        child_array = lines[3].split(",")
                        for value in parent_array:
                            if value == child_array[j]:
                                j = j+1
                            else:
                                j = j+ 1
                                k = k+ 1
                        if k > 1:
                            f.write("(B) More than one simultanious
                                perturbations\n")
                            b = b + 1

                        else:
                            f.write("(C) Investigate This\n")
                            c = c + 1

string = str(a)+"\n" + str(b)+ "\n"+ str(c)

f.write(string)
f.close()
```

E.12 Translation Table Counter

Listing E.12: Translation Table Counter

```
#!/usr/bin/env python
# This piece of code will search the Tierra dump file for every creature
# alive at the end of a run, search for each creature within the genebank
# and write two arrays to file containing the time of emeregcnce and
# count of distinct p-symbols within the translation table of each creature

from collections import Counter
import os

f = open("tierra/gb1/soup_dump","r").readlines()

files = os.listdir("tierra/gb1/.")

arr = []
lutCount = []
instExe = []
popCount = 0
for i in f[1:]:
    if "Timestamp" in i and popCount != 0:
        arr.append(sum(lutCount)/(popCount))
        lutCount = []
        popCount = 0

    else:
        if i[:7] in files:
            creature = open("tierra/gb1/"+i[:7],"r").readlines()

            if int(creature[1]) == 1:
                lutCount.append(len(Counter(creature[3][351:1118].split(",")
                    [1::2]))*int(i[8:13].lstrip()))
                instExe.append(creature[2][:-1])

                popCount = popCount + float(i[8:13].lstrip())

temp = ""
for i in str(instExe)[1:-1]:

    if "," in i:
        temp = temp + "\n"
    elif "'" in i or " " in i:
        temp = temp
    else:
        temp = temp + i
instExe = temp

temp = ""
for i in str(arr)[1:-1]:

    if "," in i:
        temp = temp + "\n"
    elif "'" in i or " " in i:
        temp = temp
    else:
        temp = temp + i
```

```
54 arr = temp
55
56 g = open("TranslationTableCount","w")
57 f = open("InstructionExecuted","w")
58 g.write(arr)
59 f.write(instExe)
60 f.close()
61 g.close()
```