Summer 8-2010

DNAgents: Genetically Engineered Intelligent Mobile Agents

Jeremy Otho Kackley
University of Southern Mississippi

The University of Southern Mississippi

DNAGENTS:
GENETICALLY ENGINEERED INTELLIGENT MOBILE AGENTS

by

Jeremy Otho Kackley

Abstract of a Dissertation
Submitted to the Graduate School
of The University of Southern Mississippi
in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy

August 2010

ABSTRACT

DNAGENTS:

GENETICALLY ENGINEERED INTELLIGENT MOBILE AGENTS

by Jeremy Otho Kackley

August 2010

Mobile agents are a useful paradigm for network coding providing many advantages and disadvantages. Unfortunately, widespread adoption of mobile agents has been hampered by the disadvantages, which could be said to outweigh the advantages. There is a variety of ongoing work to address these issues, and this is discussed. Ultimately, genetic algorithms are selected as the most interesting potential avenue. Genetic algorithms have many potential benefits for mobile agents. The primary benefit is the potential for agents to become even more adaptive to situational changes in the environment and/or emergent security risks. There are secondary benefits such as the natural obfuscation of functions inherent to genetic algorithms. Pitfalls also exist, namely the difficulty of defining a satisfactory fitness function and the variable execution time of mobile agents arising from the fact that it exists on a network. DNAgents 1.0, an original application of genetic algorithms to mobile agents is implemented and discussed, and serves to highlight these difficulties. Modifications of traditional genetic algorithms are also discussed. Ultimately, a combination of genetic algorithms and artificial life is considered to be the most appropriate approach to mobile agents. This allows the consideration of agents to be organisms, and the network to be their environment. Towards this end, a novel framework called DNAgents 2.0 is designed and implemented. This framework allows the continual evolution of agents in a network without having a seperate training and deployment phase. Parameters for this new framework were defined and explored. Lastly, an experiment similar to DNAgents 1.0 is performed for comparative purposes against DNAgents 1.0 and to prove the viability of this new framework.

The University of Southern Mississippi

DNAGENTS:

GENETICALLY ENGINEERED INTELLIGENT MOBILE AGENTS

by

Jeremy Otho Kackley

A Dissertation
Submitted to the Graduate School
of The University of Southern Mississippi
in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy

Approved:

Dr. Dia Ali
Director

Dr. Joe Zhang

Dr. Clifford Burgess

Dr. Ras B. Pandey

Dr. Jean Gourd

Dr. Susan A. Siltanen
Dean of the Graduate School

August 2010

ACKNOWLEDGMENTS

In perfect honesty figuring out how to sufficiently thank everyone peripherally involved in this endeavor in anything less than a book length manuscript is incredibly difficult. Certainly if I named everyone involved, it would simply be impossible. There is one name that must be mentioned of course; without the assistance of Dr. Dia Ali this work would have never been undertaken. There is no way to quantify the myriad ways in which he has assisted and guided this effort. I am also thankful to my committee for going above and beyond the call of duty in helping me refine these ideas. My colleagues in Tec 251, past and present, were also of great help. Indeed, some of them were largely responsible for my initial decision to pursue graduate school.

On a more personal note, my family has, despite changes and hardships during this process, been extremely supportive of this decade-long detour from life called College. I owe them more than can ever be said. In particular, I am grateful to my grandmother. Although she did not live to see this work, she never doubted that I would ultimately be successful.

Lastly, there are many friends who had to put up with tangential conversations. These conversations lead to more insights than you know, so you did not suffer in vain. You all know who you are, and have my sincere thanks.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

Chapter 1

INTRODUCTION

1.1 Mobile Agents

Mobile agents are a very useful paradigm for implementing network and web centric functions. In brief, mobile agents refer to intelligent agents that suspend their execution on a platform and transmit themselves and their current state to a new platform, thereby resuming execution. There are, unfortunately, a number of problems. Chiefly, these revolve around security and robustness. Robustness refers to the ability of a mobile agent to survive in a network. Security can refer to both the agent's security and the security of the platform providing support for the agent. For the purpose of this work, the focus is primarily on agent security as opposed to the platform's security; although this work may have bearing in platform security in the future. Robustness can be difficult to achieve because network applications can be very complex, and the network situation can vary over time. Similarly, it is difficult to guarantee the security of anything, much less a mobile agent, due to the fact that the situation is constantly evolving. In other words, both robustness and security are difficult to solve and require continual work to maintain.

1.2 Genetic Algorithms

Another useful programming paradigm is the idea of genetic algorithms. In genetic algorithms, random mutations and a selection process over a series of generations lead to an emergent algorithm that is effective and efficient at a task. Interestingly enough, the result of a genetic algorithm is seldom human-comprehensible. This would lend itself naturally to security concerns; making it more difficult to discern the purpose of compromised code. Additionally, the process which achieved the results itself is what is important, and this whole process would not necessarily be evident to someone viewing a small sample of produced emergent algorithms. This naturally reduces the value of an individual "solution."

Consider the combination of genetic algorithms and mobile agents. This has a many fold benefit for mobile agents. The primary benefit is the potential for agents to become even more adaptive to situational changes in the environment and/or emergent security risks. There are secondary benefits such as the natural obfuscation of functions inherent to genetic algorithms. Additionally, one of the problems facing the adoption of mobile agents is the relative complexity of creating efficient algorithms to govern their behavior; evolving that behavior might logically be a simpler task.

1.3 Benefits of Genetic Mobile Agents

Genetic algorithms have many properties that might be of benefit to the mobile agent paradigm. These properties are mainly applicable to the security of agents, the robustness of the agent, agent efficiency, and naturally mitigate complexity.

With respect to security of agents; the genetic programming subset of genetic algorithms yields code that is naturally obfuscated. This is because genetic programming usually combines code in ways that a human programmer would not consider. This makes it a challenge to understand even simple genetic programs. Additionally, the properties conducive to making a programming language well suited to evolutionary algorithms do not necessarily yield a language conducive to human-coding. In traditional mobile agents, value is often placed upon the code of the agent. This is due to proprietary algorithms, and the value of the man hours that went into the design of this agent. In genetic programming; the candidate solutions are not as valuable; more value is put upon the mechanism and methodology used to obtain the candidate solution.

Genetic algorithms are theoretically extremely well suited to the evolution of robustness. It is, indeed, the original application of evolution: the survival of the fittest. Genetic algorithms are uniquely suited to the evolution of agents that can survive, at least in theory.

While there are many applications of genetic algorithms, many of them boil down to optimization problems. Genetic algorithms are well suited to trying two solutions and picking the more efficient of the two. While this does not directly address the efficiency concern of mobile agents, genetic algorithms could still be employed to increase the individual efficiency of the agents themselves. That is, the fitness function could be defined, hypothetically, to reward agents that used a minimum amount of memory and cycles to achieve their answer; thus increasing efficiency with respect to host utilization.

Evolving solutions through genetic algorithms provides a level of abstraction for the programmer. The programmer need not be directly involved with the agents themselves; this could reduce some of the complexity inherent to the creation of mobile agents, since the programmer need not write code themselves. The veracity of the evolved solution can be tested through benchmarks; the code does not need to be examined or evaluated directly.

1.4 Pitfalls of Applying Genetic Algorithms to Mobile Agents

Unfortunately, in some senses, the traditional approach to genetic algorithms seems poorly suited to mobile agents. The chief difficulty one encounters when trying to design a complex genetic algorithm based mobile agent, is the difficulty of defining exactly what is meant by survival. That is, it is difficult to decide how to measure the success of an agent. In the simplest case, one might measure survival. What does it mean, really, that an agent did not die? Consider the example: if you send 10 graduate students for pizza; the overwhelming likelihood is that they will all survive. Even if some fail to survive the trip, it does not necessary indicate a beneficial algorithm, perhaps they were simply lucky. In theory, this would equalize over many generations, but the inherent difficulty is distinguishing performance in any manner other than 0 or 1. This is not to say that such a task is insurmountable; simply difficult. This primarily applies to the concept of evolving survivability. Objective based performance is much easier to measure; but does not necessarily lend itself to agents more capable of surviving. This difficulty somewhat mitigates the potential complexity-reducing aspects of using genetic algorithms to evolve agents. Additionally, even if satisfactorily quantified, it must be carefully balanced with the agent's success rate in order to produce a fitness value.

Another problem facing the combination of genetic algorithms and mobile agents has to do with the inherent design of mobile agents. Mobile agents are designed to be mobile, and choose to move around a network. Networks are generally not a static environment; ergo there is some potential difficulty in training an algorithm on one network, then deploying it on an almost certainly different network. This again is not insurmountable, but it seems that mobile agents would benefit more from continual training; perhaps even after deployment. The traditional genetic algorithm framework does not lend itself to this task. They are simply not designed to operate in a distributed manner. They require global knowledge of the solution set in order to rank the potential solutions.

There are well studied adaptations of this nature: distributed genetic algorithms. They are divided into fine, and coarse grained approaches. Indeed, the fine grained approach might work, and indeed be practical, if the genetic algorithm environment were distributed as part of the agency framework. This distribution of the genetic algorithm environment is necessary because even at their most fine grained, distributed genetic algorithms require global knowledge, locally. Still, this could be workable, with some major adaptations; chiefly among them that distributed genetic algorithms are not designed for the high amount of migration inherent to mobile agent algorithms. It also does nothing to address the previously mentioned problem of quantifying survivability.

1.5 Toward a Better Solution: Artificial Life

There is a set of frameworks that do, by nature, address the primary issue of quantifying survival. This area is the area of artificial life. In artificial life, organisms are created in a "world" with very simplistic fitness models. Generally, fitness isn't explicitly defined at all. Fitness is an emergent feature of the world. There is debate that artificial life represents more closely "natural" evolution. There are additional concepts such as reproduction, and death, which are lacking from traditional and distributed genetic algorithms. In some cases, organisms are rewarded or punished for certain types of behavior. Lastly, most approaches to artificial life do not require any sort of global knowledge of the world; organisms can only interact with nearby organisms, if at all. This would lend itself to complete distribution in a much easier manner.

Consider the problem of adapting mobile agents to the paradigm of genetic algorithms from a different point of view. What if agents were organisms? The network would be their world. It would follow that agents would interact with it and each other, choosing when and how to reproduce. Additionally, various actions will be rewarding, perhaps in the nature of providing extended longevity. Through the mixture of reward, and punishment, the ecosystem will help determine the exact mix of actions to take. This mix would naturally vary with the network; agents adapting to a changing environment. The reward system could also guide evolution in a particular direction, so that certain goals are achieved.

1.6 Overview of the Dissertation

In this work a review of the genetic algorithm, mobile agent, and artificial life paradigms is presented. Chiefly, however, the focus is of course on mobile agents. Shortcomings in the current state of mobile agent literature are highlighted, as well as attempts to resolve this. Ultimately, this leads to the consideration of genetic algorithms for this task. Towards this end, several anecdotal attempts at this are presented and discussed. DNAgents 1.0, an original application of genetic algorithms to mobile agents, is also presented and discussed. This attempt more than the others highlights the shortcomings of genetic algorithms when applied to the domain of mobile agents, and artificial life is presented as an alternative to genetic algorithms after a discussion of pertinent variations on genetic algorithms. This ultimately brings the narrative to the primary purpose of this work: the introduction of the framework of DNAgents 2.0. DNAgents 2.0 are suspended betwixt the related fields of genetic algorithms and artificial life. It belongs to neither, and is well suited to the evolution of mobile agents.

1.7 Contributions of the Dissertation

Specifically, there are two main contributions in this work. The first is the application of a canonical genetic algorithm to mobile agents in DNAgents 1.0. DNAgents 1.0 was successful in evolving agents for a simple task. DNAgents 1.0 also serves to highlight the primary issues with combining genetic algorithms and mobile agents. These issues include the difficulty of defining a suitable fitness function and the uncertainty of agent execution time in a network environment. This sets the stage for the second contribution of this work: the formulation of a framework for the evolution of genetic algorithms known as DNAgents 2.0.

DNAgents 2.0 are a combination of the core idea of genetic algorithms: optimization, with the soul of artificial life: a quest for natural evolution. Chiefly it uses the analogy of artificial life; artificial organisms, and applies this to agents. In DNAgents 2.0, agents are considered organisms. The network is their world. This is a natural analogy. Towards this end, agents do not have specific goals other than interaction, survival and reproduction. In a nod towards genetic algorithms, a crossover mechanic is explicitly provided. This is a depature from artificial life, where part of the goal is ususally to spontaenously evolve a crossover mechanism using just mutation as a defined behavior. This framework provides itself naturally to distribution, and indeed could theoretically operate in a global network such as the Internet since agents are only truly concerned with their local contemporaries. In some sense, this could be considered the ultimate multi-agent system.

Agents surviving in a network based world alone are not enough to truly evolve mobile agents. This is due to the goal oriented nature of mobile agents. However, it is difficult to define fitness functions for mobile agents. DNAgents 2.0 avoids this through the the concept of reward mechanisms, which reduce the likelihood of an agent expiring. This mitigates the difficult of defining a fitness function due to the possibility of defining a variety of rewards. The agents then automatically find the proper balance of actions to optimally solve the problem. Indeed, with the introduction of communication parameters, this might involve inter-agent cooperation. Thus, the sole goal survival is expanded to the new goal of survive well. The fittest solutions will survive longer, thus having more offspring and distributing the optimal subprograms throughout the network. A final benefit of this approach is that it is designed to run continuously. This is a departure from most applications of genetic algorithms, in which there is a training period followed by a deployment period. This lends adaptability to the agents; as the network changes so too will their fitness; thus the efficiency will gradually improve over time.

1.8 Dissertation Structure

In Chapter 2 an full overview of agents and mobile agents is given, as well as their strengths, weaknesses, and the research that is ongoing in this field. In Chapter 3 an high-level review of genetic algorithms is conducted touching on some of the variations of genetic algorithms. In Chapter 4 the primary motivations of artificial life are discussed, as well as the common features of artificial life simulations. A review of notable artificial life simulations is also conducted in this chapter. In Chapter 5 a more focused review of the challenges facing mobile agents is conducted, as well as a brief analysis of the approaches attempting to solve these challenges. This chapter also introduces the idea of combining genetic algorithms and mobile agents and reviews some work in this area. Chapter 5 also introduces the first half of this work, a successful attempt at combining genetic algorithms and mobile agents (DNAgents 1.0), as well as the results of this attempt. It goes on to discuss the shortcomings of genetic algorithms when combined with mobile agents, and introduces the concept of applying a new artificial life inspired algorithm to mobile agents. This leads naturally to the second, and more important portion of this work, the proposed framework of DNAgents 2.0. DNAgents 2.0 is defined and discussed in Chapter 6. Implementation details as they affect the definition of the algorithm are discussed, and experimental results provided. Finally, in Chapter 7 the major points of this work are reviewed, as well as its contributions and implications.

Chapter 2

MOBILE AGENTS

2.1 Static Agents

In order to define a mobile agent, one first must define the concept of a static agent: one that does not move. The analogy to a travel agent is often made in the literature. This is an accurate description. An agent is an entity that performs a task on behalf of someone else. The following characteristics are ascribed to agents [49]:

- Autonomy. Capability to perform tasks without intervention.
- Social Ability. Capability to communicate with other agents.
- Reactivity. Ability to react to environmental changes and events.
- Pro-activeness. Not strictly reactive; can chose to initiate change if necessary.

There is additionally a concept of agency; which refers in the broad sense to a storehouse for agents. In terms of software, it might implement an interface to data, or provide secure mechanisms for intra-agent communication [49]. Agent's are sometimes grouped together into a hierarchy whereby higher level agents might utilize lower level agents to perform specialized tasks, whilst the user only interacts with the high level agent [49].

2.2 Mobile Agents

Mobile agents in general refer to a set of self-contained algorithms (agents) that can choose to suspend execution and move around the network, acting on behalf of an entity [90]. Similarly to agents, they are said to exhibit features such as autonomy, social ability, learning, and mobility. Autonomy refers to the fact that the agent executes within its own execution environment, making its own decisions. Social ability refers to the ability to interact with other agents and, implicitly, platforms. Learning can be more generally referred to as adaptability; but defines the ability to react to its environment. The most important characteristic of a mobile agent is mobility, and it is the defining difference between mobile agents and regular agents, and is defined by the following steps [49]:

- The mobile agent halts execution upon reaching some predetermined state or event.
- The mobile agents current state is saved.
- The mobile agent is serialized into an array for transit.
- The agent is encoded into some common format for communication.
- The encoded agent is transferred to a remote host.

Mobile agents choose to migrate. This choice is the key distinguishing feature between a mobile agent other types of mobile code such as process migration or code-on-demand. Process migration refers, in general, to applications where an operating system of chooses to execute a process on different processors; the choice to suspend execution and migrate is not taken by the process itself but by some controlling entity. Code-on-demand applications refer to applications where code is downloaded from a server to a client machine as needed upon the clients request [42]. The data being executed upon remains upon the client machine.

2.3 Advantages of Mobile Agents

There are several key advantages to the mobile agent paradigm. These advantages include:

- Reduction in network traffic
- Asynchronous
- Adaptive
- Tolerance
- Reduced Maintenance
- Portability
- Scalability

The reduction in network traffic is one of the chief motivations for exploration of the mobile agent domain. This is accomplished due to the fact that in mobile agent architecture, one only need to send the source code for the agent, and any associated data, to the server once. Additionally, upon the completion of execution, the server sends the agent and any processed data back, only once. This is in contrast to client-server models where communication is back and forth and near-continuous for the duration of the task, consisting of many messages. Additionally, due to relying only upon two transmissions per server; agents inherently achieve a level of fault tolerance difficult to obtain with a client-server model. This is due to the fact that if the network connection between client and server fail during a traditional task being executed, then the task fails. In a mobile agent, this failure has no impact upon the agent, and at the worst delays the return of the agent.

Due to the fact that mobile agents execute autonomously on a machine, their execution is by nature asynchronous. Thus, it is trivial to achieve parallel processing of remote data by simply sending agents to multiple data sources, leading to great scalability [50]. Agents due to their nature operate on many operating systems and environments, and thus can be said to be highly portable and adaptive. Lastly, the requisite maintenance is reduced because only the agent generally has to change, and not any distributed platforms.

2.4 Disadvantages of Mobile Agents

There are some drawbacks, or potential pitfalls, of mobile agents, including:

- Security
- Authentication
- Trust
- Efficiency
- Complex to setup
- Increased complexity

These largely revolve around issues of trust[136, 106], and security. Trust refers, in the broad sense, to how one determines how trustworthy an agent is. There is extensive research dedicated to trying to determine if an agent is trustworthy, and the related concept of reputation [77, 36]. Reputation refers to how reliable the community at large feels you to be, while trust, more specifically, refers to an individual's trust in you. The concept of trust and reputation is applicable to both agents and hosts. These highlight the hosts view of security; why should this remote code be allowed access to these resources [25]? There is another aspect of host security; how to ensure that in spite of what the agent does, the host remains secure and protected. There is a somewhat related area of trying to detect agents that, while not necessarily hostile, might perform in a very inefficient manner; leading to performance degradation [25]. The solutions that are proposed usually revolve around the concept of a sandbox or safe language. A sandbox [7, 67, 49] refers to a protected, encapsulated environment in which the agent runs. This environment essentially protects the host from the agent by limiting what it can do, and is logically similar to a virtual machine. There is a similar concept known as a safe language; a language designed to purposefully limit what an agent can do to only safe actions that cannot unduly affect the host [49].

While this is a serious concern, there is the related concern of ensuring the agents security, which also has significant research dedicated to it [49]. The agents security is more a question of how does the agent ensure that its data, which it is carrying into a potentially hostile environment, is safe? The code of some agents might itself be sensitive, so this is a concern as well.

It is generally accepted that protecting the agent is much harder than protecting the host. In fact, there is a feeling that the problem might be intractable, and that it is better to detect it and manage it [49]. Some ways of detecting and mitigating it include the use of multiple agent systems [99, 135, 93, 137]. Multiple agent systems allow the encapsulation of aspects of the algorithm, such that any one agent cannot give much of a view into what the whole system is trying to accomplish, or give away a significant portion of any algorithm [49]. If redundant agents are employed these systems might also employ voting to determine if any of them have been compromised, or if any of their data is suspect. An approach to protecting the data is the encryption of the current data with a throw away key, thus the data, if captured, is not easily interpretable [49, 58]. Additionally, if agents return home frequently, compromises become less severe and easier to detect [49, 58]. If the code of the agent itself is sensitive, it might be protected by encryption or obfuscation [49, 27]. While it is impossible to allow a processor to execute instructions without it gleaming something about the algorithm, if sufficiently obfuscated, this becomes quite difficult [110, 49]. Encryption would only allow authorized hardware to execute the agent. This is not in any way an exhaustive list of methods to protect the agents. With respect to preventing compromise in the first place, the best approach appears to either be the use of a set of trusted hosts [58] or secure hardware [49, 26, 27].

While it is true that mobile agents are naturally resistant to connection interruptions, this does not necessarily make them perfectly fault tolerant. There is work into the issue of fault tolerance, and agent fault tolerance [108, 115, 116]. Generally, these approaches revolve around the concept of replication [88, 94]. Replication involves the copying of an agent prior to movement. This copy resides on the host for a time, and if it doesn't hear back from its spawning agent, eventually continues execution. This is a generalization of replication approaches; they can take several forms. At any rate, there is an issue with replication: the exactly once property [108]. Basically, it is difficult to ensure that, in the presence of replication, an agent's task is executed once and only once.

In addition to the security and fault-tolerance concerns; there are issues of complexity. Not only are mobile agent systems complex to setup initially[42], the systems themselves are simply more complex in general than the existing paradigms such as client-server [42]. This hinders adoption, due primarily to the fact that there is no "killer" application that absolutely requires them. There are attempts to model agent systems such as the API calculus and its security-focused extension, API-S [49]. Among other things, these systems attempt to aid the understanding of agent systems and their interactions. Due to their complexity, it is difficult to model them directly, generally some level of abstraction is used [49].

Mobile agents have many applications; such as in resource discovery and monitoring [129, 46, 38, 31, 33, 59], information retrieval [31, 50, 85, 20], and network management [15, 95, 37, 5]. They are additionally useful in data replication, for remote data backup, and data sharing [35], and for dynamic software deployment [53, 107].

Chapter 3
GENETIC ALGORITHMS

3.1 Overview of Genetic Algorithms

Genetic algorithms (GA) refers in the general case to a process for finding solutions to optimization problems. GA's function by formulating potential solutions to a problem and testing them against the desired answer. In a sense, in GA's the desired answer or result is generally known and definable, but the exact method of achieving the answer is not. For example, consider the knapsack problem: given a series of items with a weight and a value, and a limit in weight which a knapsack can hold, determine the optimal set of items to take, maximizing value, while not exceeding the weight limit of the knapsack [80]. In the knapsack problem, the solution in a general sense is known: maximize value while not exceeding the weight limit. The set or sets of items that comprise this solution is not known.

From the solution, a fitness function may be defined. In the case of a knapsack problem, the fitness function could be the combined value of a selection of items. Not all combinations of items are valid, and in the event that the combined weight exceeds the weight threshold of the knapsack, the fitness of that selection is zero. While it is somewhat trivial to define a fitness function for the knapsack problem, for some problems it is extremely difficult or impossible to define a fitness function. In these cases interactive genetic algorithms may be used.

Now that the fitness function is defined, the solution domain must be defined in a manner conducive to genetic processes. In the case of the knapsack problem, this can be done by assigning a bit to each potential item, such that the string of bits indicate the inclusion of each item. GA's function by initially generating many random solutions, and then selecting among them based on the fitness function. The exact methods of selection, and the action taken upon the fittest solutions vary. In general GA's borrow from the biological concepts of natural selection, crossover, and mutation. Through these processes, the solutions are recombined over and over until a satisfactory fitness has been achieved.

3.2 Selection

Selection algorithms generally rank potential solutions by fitness, and provide a higher chance for the fitter solutions to be selected. (Add more info on ranking).

In some cases, only the fittest solutions are selected. These cases are referred to as either elitist selection algorithms or truncation selection algorithms because anything beyond the fittest N solutions is truncated. In practice, elitist selection algorithms are rare due to poor comparative performance, but they do exist [84]. They are mainly listed in the literature as being used for breeders [16, 21, 30]. Generally, truncation algorithms suffer from a loss of diversity in the population, this has been analyzed in the literature [16]. Additionally there are some hybrid approaches [65, 12].

More commonly, genetic algorithms employ a method in which, while fitter solutions have a higher probability to be selected, less fit solutions also have a chance to be selected. There are a variety of methods, but the most common are roulette-wheel selection [4] and tournament selection.

In roulette-wheel selection, a probability is assigned to each potential solution based off its fitness. An example formula for calculating this value is given as:

$$p_i = \frac{f_i}{\sum_{j=1}^{N} f_j}$$

This formula ensures each solutions probability is between 0 and 1. The fitter solutions will have a slightly higher probability, and less fit solutions a slightly lower probability, but all solutions still have a chance to be selected. At this point, a random selection is made of them. This is done by arranging the solutions in a line, with their corresponding probability representing their 'slice' of the line. The line ranges between 0 and 1. A random number between 0 and 1 will fall within one of the solutions associated ranges, thus selecting that solution. Figure 3.1 illustrates this process. The fact that less fit solutions can be selected is important because although the whole of the solution might be weak, some component of it might not be, and greater genetic diversity is generally a good thing. There are many derivations of this algorithm, such as Stochastic universal sampling [4], which ensures a minimum spread in addition to zero bias [4, 16].

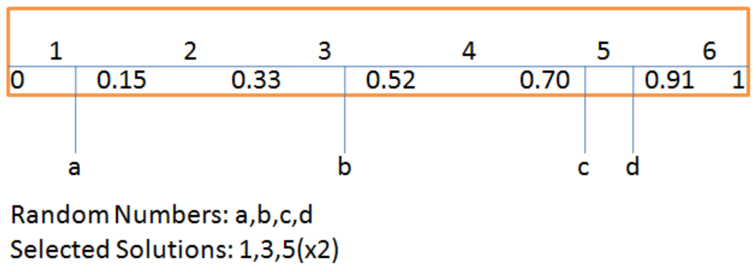

Figure 3.1: Roulette-wheel selection.

Stochastic universal sampling [4] differs from roulette-wheel selection only in that instead of selecting N random numbers from the line, it first picks a position between 0 and 1/N where N is the number of desired solutions to be selected. It then adds the value 1/N to this value to pick the next value. Figure 3.2 illustrates this process. This allows for the selection to be evenly distributed.

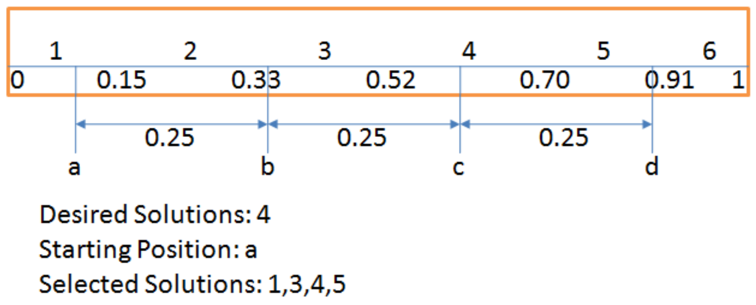

Figure 3.2: Stochastic universal sampling.

In tournament selection, a random set of individuals is chosen from the population, and subsequently the best individual in that subset is picked to reproduce [47]. This is logically similar to the method by which tournaments of two player games, such as billiards, are conducted. It is not limited to populations of size two, of course, as any game with a set number of players can be conducted in tournament fashion, by conducting a series of matches, and then conducting matches between the winners. The main variable in tournament selection is the size of the random subset of the population, sometimes referred to as tour, which can range from a minimum of 2 to the size of the population.

Additionally, there is a concept called local selection. In local selection, half the mating population is selected either randomly or by some other selection algorithm. Once half the population is selected, then neighborhoods are defined for the selected population. These mating individuals interact only with their neighborhood. Within the neighborhood the mating partner is selected either randomly, or by selecting the fittest. Because of the overlap of these neighborhoods, genetic material can propagate across the entire population. Fu et al [45] applied local selection to a Virus Evolutionary Genetic Algorithm. Virus Evolutionary Genetic Algorithms are a derivation of genetic algorithms that are discussed in the section on derivations of genetic algorithms. The properties of local selection in their work helps to prevent premature convergence and helped maintain the diversity of the host population, and indirectly caused the elimination of ineffective strains due to the local survival of the fittest host.

The neighborhoods can be defined in many ways, and Figures 3.3,3.4, and 3.5 illustrate a few. Additionally these neighborhoods can be three-dimensional and include any combination of one and two-dimensional definitions. An analysis and comparison of a form of local selection and more traditional selection methods within massively parallel genetic algorithms is given in [28].

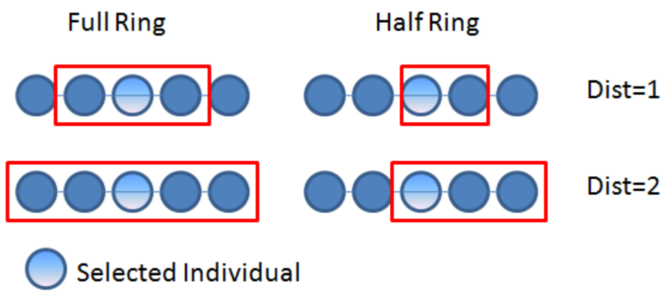

Figure 3.3: One-dimensional local population neighborhoods.

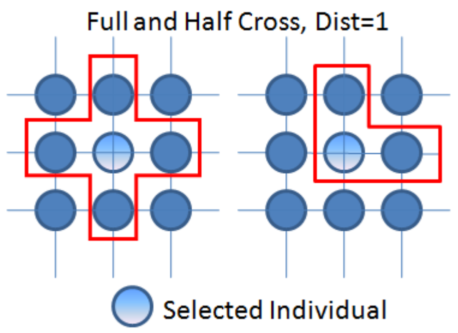

Figure 3.4: Two-dimensional local population neighborhoods 1.

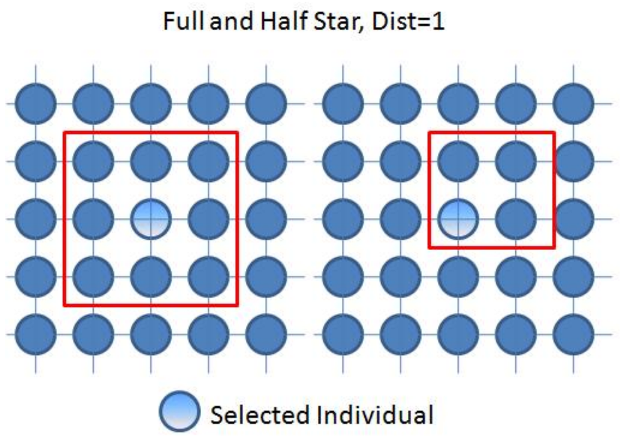

Figure 3.5: Two-dimensional local population neighborhoods 2.

In conclusion, generally speaking, truncation selection is much more likely than either roulette-wheel selection or tournament selection to replace less-fit individuals with more fit individuals. This causes overall a loss of diversity, and a lack of variance in the solution set. Correspondingly, it causes the genetic algorithm to take longer to plateau. With regards to diversity and variance, tournament and roulette-wheel selection perform similarly [16]. While interesting, local selection does not directly contribute to a selection method, although it can influence the overall algorithm positively by aiding in the preservation of diversity in some schemes [45]. Local selection also is very applicable to the parallelization of genetic algorithms.

3.3 Crossover

Crossover, sometimes referred to as recombination, draws inspiration from biological reproduction. Specifically, the mechanism by which offspring are created that contain genetic material from both parents. In this way, the individuals selected by the selection algorithm are combined to produce unique offspring. In this section we discuss some of the methods in which this operation takes place.

With regards to problems with real representations, or where the expected outcome is a number of series of numbers which can be thought of as coordinates on the graph, the term recombination is generally used to describe this process. There are many methods for it in the literature including discrete recombination, intermediate recombination, line and extended line recombination [84, 81]. These methods are discussed in brief because of their pertinence to genetic algorithms, but detail is not used due to the focus of this work being upon problems with binary representations.

Discrete recombination [84] allows each offspring to be randomly assigned a value from one of its parents for each variable. There is no weight given to the various variables, and by definition, both parents are potential offspring of this process, although with a large number of variables this becomes increasingly unlikely. Figure 3.6 illustrates a simple example of discrete recombination consisting of parents with only two variables, and thus graphable on a two dimensional plane. Interestingly, discrete recombination can be applied to both real number representations and binary representations, and theoretically to any problem representation.

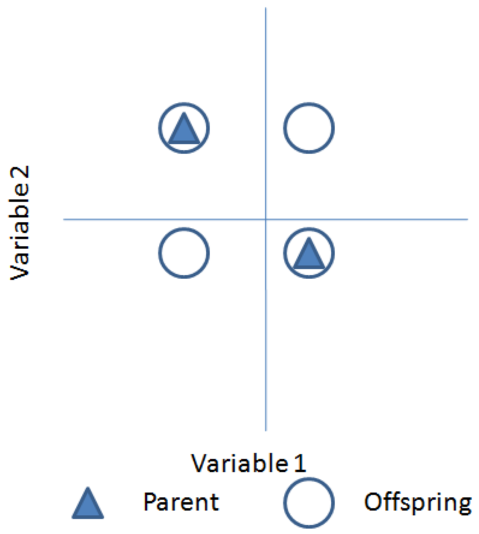

Figure 3.6: Discrete recombination.

Intermediate recombination [84] applies only to problems represented by real values. The basic idea is to define offspring values near or between the values of the respective parents. This can be expressed as the offspring's value falls on the line between the values for the respective parents. The algorithm allows the addition of a small constant to each parent value to extend the line slightly. intermediate recombination can create an offspring anywhere within the square defined by the four potential offspring in figure 3.6 and slightly outside of it.

Line recombination [84] picks offspring on the line between the two parents. Once again, consider figure 3.6 and the line between the two parents. If you extend this line a small length in both directions, all the points on this line are valid positions for offspring. This method as well only applies to problems with real number representations. There is an extension of line recombination [81] which does not restrict offspring to the line between the parents, but only to the domain of the variables. That is, the offspring can fall at any point on the line upon which the parents lie, provided the value is not restricted by the variable domain. Additionally, offspring are created based on a variable probability that allows offspring to be more often chosen near their parents. If they fitness of the parents is known, then the offspring are chosen in the direction of the fitter parent. Again, this applies to real values, and not binary representations.

If a problem has a binary representation, the process of recombination is generally referred to as crossover. In crossover, one or more slices of the parents is created and exchanged and recombined to create new offspring. Crossover methods are differentiated by the number of points used to define the slices.

In the simplest form of crossover, a single crossover point is picked. Two offspring are created, one from the genetic material of parent a prior to the crossover point, and one from the genetic material of parent b from the crossover point to the end of parent b. Figure 3.7 illustrates single point crossover. If the parents are of the same length, then the offspring will always be the same length. An interesting variation on single point crossover exists, known as Cut and Splice crossover, which produces offspring of variable length even if the parents are of uniform length. It functions by picking two crossover points, one in each parent, and then splicing the resulting substrings together. Figure 3.8 illustrates this method. There is an additional variation of single point crossover where the bits are shuffled before crossover occurs, and unshuffled in reverse afterwards, thus removing any positional bias [24].

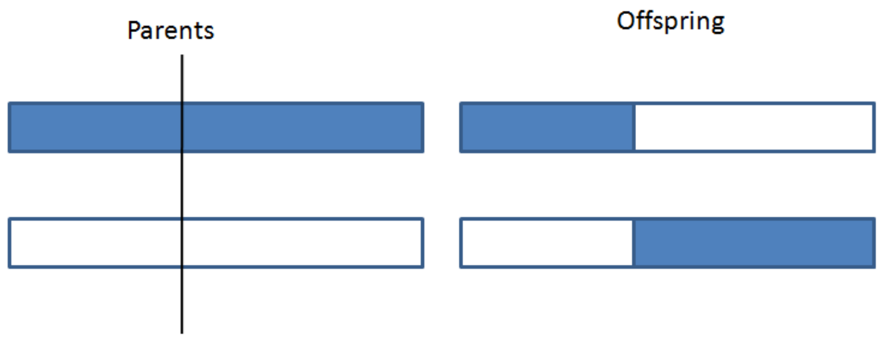

Figure 3.7: Single-point crossover.

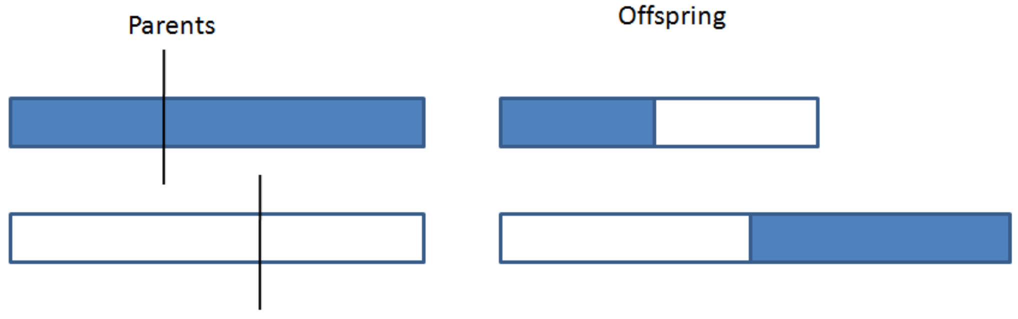

Figure 3.8: Cut-and-splice crossover.

Similarly, two crossover points can be picked, in stead of one. In this case, offspring are formed by swapping the slice that lies between the two crossover points between each organism, yielding two offspring. This is depicted in figure 3.9. Single and two-point crossover are actually special cases of a more generic multi-point crossover algorithm. In a variable crossover algorithm, N crossover points are selected, and two offspring are produced by selecting first from one parent and then from the second parent, starting with the second defined slice. It is stated in the literature that single and two-point crossover are often the best choices, but for some problems multi-point crossover will yield better results, possibly due to the fact that substrings with the key parameters for the solution of the problem do not necessarily lie adjacent to each other [18]. Additionally, the disruption caused by multi-point crossover yields a more robust search due to decreasing the likelihood of early convergence [118].

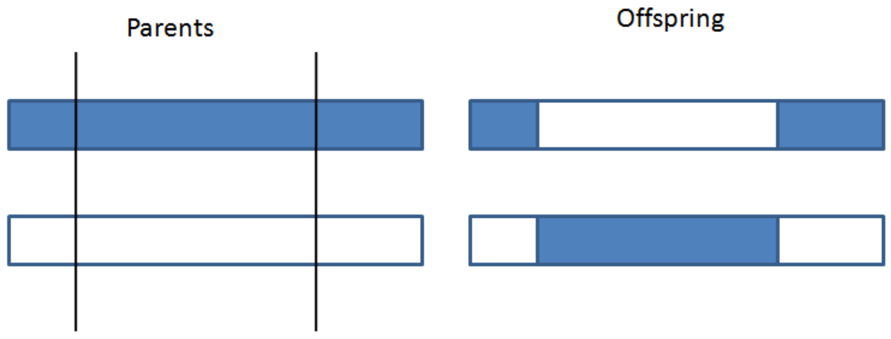

Figure 3.9: Two-point crossover.

There is an additional type of crossover known as uniform crossover [125], in which each gene can individually be swapped. This can be done by generating a bit mask of the same length as the parents chromosomal length, where each bit indicate which parent contributes genetic material to the offspring. This concept is illustrated by figure 3.10.

Two offspring can be generated from a single bit mask by inverting the bit mask for the second offspring. Note that in generating the bit mask, equal probability is given for each parent to be selected. That is to say, there is an equal probability that each bit will be 1 or 0, with no weight given to any bit. Theoretically, this reduces the bias towards short substrings inherent in single point crossover. A parameterized version of uniform crossover that applies a probability to determine which parents bits are selected has been proposed. This parameter controls the amount of disruption caused without introducing a bias towards short substrings [119].

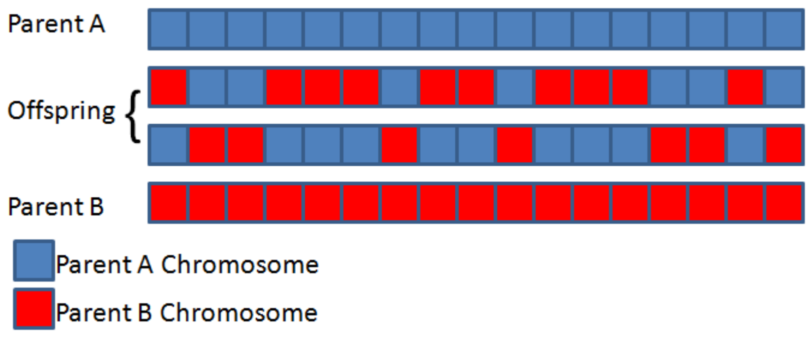

Figure 3.10: Uniform Crossover

3.4 Mutation

Mutation refers to the process in which individuals in the population are randomly altered. These changes are generally small, and occur with a low probability after the offspring has been created via crossover. Generally, two parameters exist, mutation rate, and mutation step size. Mutation rate refers to the amount of mutations that should occur, and mutation step size refers to the amount of change that should occur upon mutation. These two parameters are either constant over the course of all generations, or vary depending on the previous generation. The following section discusses methods of defining these parameters, as well as methods for dynamically selecting them.

The probability of mutating a variable is dependent upon the number of variables that exist. That is, the more variables, the less likely it is that mutation should occur. There is some debate as to what exact relationship this should be, but it has been reported that $1/N$ where N is the number of variables provides good results in a wide variety of cases [84, 8, 9]. While this is not necessarily a good mutation rate for every problem, it is a useful starting point.

Choosing a good mutation step size is more problematic, as it is dependent upon the problem and can vary during the optimization process. When near optimal, small step sizes are best. Conversely, at the beginning, large step sizes can, if successful, lead to much quicker optimization. It has been proposed that a good mutation operator should produce more small step sizes than large step sizes [81, 84].

It is also possible to adapt the values dynamically over the course of the algorithm. This has been applied to evolutionary strategies[113, 103], evolutionary programming [43] and other areas [86, 87, 54]. Allowing the mutation step and or rate to vary increases the storage requirements of the algorithm, due to the fact that the variables must be stored for each individual.

3.5 Termination

The final step of a genetic algorithm is termination. Termination is whenever the processes of selection, mutation, and crossover cease. There are several reasons this can take place, and generally the definition of when to terminate execution is somewhat dependent upon the application domain. One possible termination condition is a satisfactory solution being reached, that is, a solution that fits certain minimum criteria. Another, common condition, is to terminate automatically after a certain number of generations. Additionally, a termination condition in scheduling application often involves the resources to be allocated or money to be spent being reached, or approached to within a specified minimum value. Another possible termination point would be the difference between successive generations becoming very small, or possibility negative. In other words, the algorithm plateaus, and further executions do not appear to improve the answer, and might actually harm it. Lastly, a human operator might be employed to observe the results, and decide manually to end the simulation.

3.6 Other Types of Genetic Algorithms

Virus Evolutionary Genetic Algorithms (VEGA) was first proposed by Kubota in 1996 [72]. VEGA are a subset of genetic algorithms that consists of an additional population known as the virus population. The standard population inherent to genetic algorithms is known as a host population, and functions identically to the standard genetic algorithm. The virus population interacts with the host population via the operations transduction and reverse transcription. Transduction allows the absorption of material into the virus similar to the way in which crossover works, by selecting a substring consisting of both viral code and the original genetic material of the host, and copying it onto the virus thus replacing the virus. The purpose and effect of these operations is to allow the virus population to copy effective subsets of the genetic algorithm from the host population, and preserve them against accidental destruction by crossover and mutation, thus causing the whole simulation to converge on an optimal solution more quickly. Figure 3.11 demonstrates these operations. These operations take place in a step that follows the mutation of the host population. Additionally, an improvement of it has been proposed that includes the theory of static multiplication as well as optimizing the virus generation process leading to increased performance [78]. Local selection has also been used to improve the results yielded by this algorithm [45].

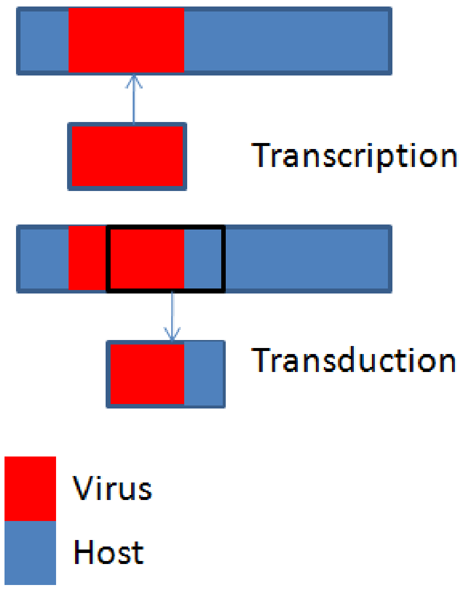

Figure 3.11: Virus Evolutionary Genetic Algorithms operations, adapted from [45].

There exists a subset of genetic algorithms called genetic programming. There are two primary differences between genetic algorithms and genetic programming. In genetic algorithms, the intent is to derive a solution, an answer, to a problem. Genetic algorithms explore a search space, attempting to find a number. In genetic programming, the goal is a program, or set of code, that can solve a problem. In a sense, genetic programming attempts to find the function, instead of the solution. The other primary difference is that in genetic algorithms, potential solutions are generally represented as variables and or numbers. Sometimes these numbers are represented in binary, and a single potential solution might have many numbers or variables associated with it. In genetic programming, however, the solutions are represented by a set of code. There are two primary subsets of genetic programming, the divide having to do with the representation of the solution. In the majority of applications, the program is represented as a tree. In this tree, the nodes are operands, and the leaves are operators. This lends itself very well to representing mathematical equations, and implementation within language such as LISP that support tree's.

Figure 3.12 represents the function $(4+5)*(10/13)$ as a tree. Additionally, because not every programming language or application lends itself well to a tree structure, genetic programming can also be represented linearly. Most, but not all, methods applicable to genetic algorithms can be equally applied to genetic programming, with some modification as necessary. In particular, the methods that are applicable to binary data representations can be easily applied, in most cases, to genetic programming. In the case of tree based genetic algorithms, the nodes in the tree can be considered chromosomes for the purpose of applying mutation and crossover operators, while in the linear representation, individual instructions can be considered chromosomes.

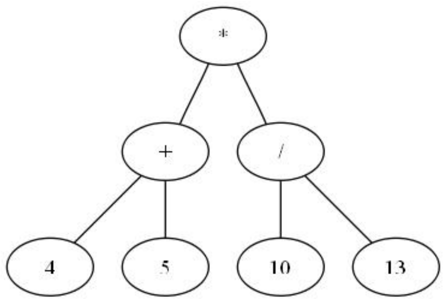

Figure 3.12: Genetic Programming Tree Structure for $(4+5)*(10/13)$

An additional modification to genetic algorithms is evolutionary programming. The primary difference between evolutionary programming and genetic algorithms is that evolutionary programming does not use crossover [52].

Davidor introduced the Ecological genetic algorithm model [34]. In this model, selection is local among individuals that exist in a grid. An individual interacts with their immediate nieghbors and new individuals are inserted randomly into this grid. Upon insertion, the difference in fitness between the existing algorithm at a cell and the new algorithm is computed, with the fitter algorithm being the one that 'wins' and survives [34]. Additionally, this model has been successfully parallelized, in a coarse way, refer to [22] for a discussion.

3.7 Decentralized Genetic Algorithms

Due to limited computing resources, and the abundance of cheap machines, it is often beneficial to execute processes in parallel. Genetic algorithms have been modified in such a way as to be decentralized and thus executable in parallel on an array of machines or processors. This can be done in several ways, and these ways are distinguishable by their granularity. They can be either coarse, or fine grained. This granularity refers, at least partially, to the type of topology the approach is suited to execute upon: coarse-grained architectures with a few, powerful processors, or fine grained populations with many weaker processors [111]. Additionally, some effort has been made to combine the two approaches in a hierarchical model with the lower level being a fine grained GA and the upper level being a corase-grained GA [22].

In coarse-grained decentralized genetic algorithms, the population is segregated into non-overlapping populations. Each subgroup is executed in a centralized fashion usually indistinguishable from a traditional genetic algorithm. A migration operation that takes place after the process of crossover and mutation is introduced to allow interaction between the entire population. If no migration operation is defined, this is equivalent of running several parallel, but independent genetic algorithms [111].

A fine grained genetic algorithm is defined in such a way that the local neighborhoods overlap. Each individual in the simulation is responsible for interacting with their neighbors and choosing who to reproduce with. The overlapping neighborhoods allow implicit migration of information across the whole simulation [111].

Local selection, discussed earlier, describes methods of defining these populations and their neighborhoods. There is also research into the idea of segregating the population due to the fact that segregating populations allows them to maintain diversity far longer than a traditional genetic algorithm with random, universal reproduction [111, 114]. That is to say, the segregated populations generally tend to be similar to each other, but dissimilar to other populations. There are other methods to accomplish nonrandom reproduction this besides segregation and migration, such as tagging subpopulations and restricting reproduction to be only between these defined subpopulations, or by introducing a distance metric that restricts mating to being between similar individuals [111].

3.8 Applications of Genetic Algorithms

Genetic algorithms have been applied to many many problems successfully. Examples from computing include microcode compaction, traveling salesman [48], and job scheduling [11]. Commercially speaking, they have been applied to the problem of determining proper placement for cell phone towers weighing coverage against costs and other considerations [98]. The field of education has benefited from the application of genetic algorithms to the generation of time tables for schools successfully [96]. Genetic algorithms have also been applied to Lego Mindstorm robots [112] and robotics in general [40], which could impact industry. Genetic algorithms have even been applied to network coding [66]. Genetic algorithms have even been applied to military applications, such as the generation of plans for the joint suppression of enemy air defense [105]. Medically, they have been applied to drug scheduling for chemotherapy [75].

A VEGA based algorithm has been developed to aid in the resolution of the problem of scheduling N tasks with different due dates and ready states [114]. Their results show that their VEGA based algorithm outperforms the GA based algorithm, due to the exchange of genetic information within a generation provided by the viral properties of the VEGA.

Genetic programming has been applied to the detection of munitions in an Air force shooting range [44], reducing the number of false alarms required to clean the site of munitions. They have also been applied to creating variations of buffer overflow attacks for the purpose of improving detection and defense methods with the data generated by their 'white hat' attacker. The buffer attacks developed by them obfuscated the purpose of the code, and were able to defeat the popular 'Snort' Intrusion Detection System. They use a linear genetic programming method, and define an instruction set of operands and opcodes which consist of bits, similar to assembly code [63].

Chapter 4

ARTIFICIAL LIFE: A DIFFERENT APPROACH TO EVOLUTION

4.1 Introduction to Artificial Life

Artificial life refers to the study of the logic of living systems in artificial environments. Chris Langdon defines it as "the study of man-made systems that exhibit behaviors characteristic of natural living systems [74]." The phrase "locating life-as-we-know-it within the larger picture of life-as-it-could-be" is also often quoted [74]. In some sense, evolutionary algorithms can be seen as a subset of, or at least a field related to artificial life. Artificial life involves the study of these principles, while evolutionary algorithms attempts to apply the mechanisms of life to optimization problems.

There are two main connotations associated with the word artificial life. They are, respectively, weak or strong artificial life. Proponents of weak artificial life see their creations as simulations of life, perhaps useful for deriving knowledge about living things, but not living themselves [17, 117]. Conversely, proponents of strong artificial life see their creations as being as deserving of the title living as living things in the world, at least in theory, if not currently in practice [74, 101]. There is additionally a hybrid school of thought that, while it thinks life must be embedded in physical matter, simulated physics might eventually be sufficient for this, if treated in a scientific manner [89, 124]. Most applications involving artificial life that have a practical payoff are of a weak nature; in the sense that they use artificial life for the purpose of achieving some payoff in the real world. This actually forms the bulk of research into this area, although there is research into strong artificial life. The focus of the remainder of this section focuses on weak artificial life.

There are two main areas of artificial life algorithms, learning based and evolution based. Learning based artificial life algorithms generally revolve around neural networks. Evolutionary based neural networks typically revolve around genetic algorithms [52]. Additionally, it is possible to combine the two areas, and some work has been done to this effect. such as attempting to capture the "Baldwin" effect. The Baldwin effect refers to how learning, particularly capability to learn, can influence evolution, possibly leading eventually to encoded behavior [83]. The focus of this work is on the evolutionary models of artificial life, rather than models involving learning.

Most work in artificial life can be traced back, in some sense, to Von Neumann's description of a self replicating automation. Von Neumann proposed a machine that could, given resources, produce a copy of itself. Mutations could be passed down to its offspring, thus allowing evolution [126]. Conway's Game of Life is also interesting, and somewhat influential. In this way, many approaches to artificial life revolve around cellular automata, although this is not necessarily an algorithm, but more of a platform upon which many simulators are built [52]. Again, the focus of this work is not on cellular automata, but on agent based systems, and thus this aspect of artificial life is not discussed in detail. For this reason, the next section discusses notable work that has been done with respect to evolving code or gene based entities, such as Tierra. Table 4.1 lists the simulators discussed in this work.

4.2 Common Features of Alife Simulations

This section attempts to summarize some of the common features of artificial life simulations. All artificial life simulations begin with the concept of a world, which might be defined as:

- A grid-like structure of cells, similar in some sense to Conway's "Game of Life." This is not always a two dimensional grid, and occasionally one-dimensional. The organisms may or may not be able to move between cells. This is the most common representation.
- A two-dimensional plane. This is distinct from the 2D grid in that this plane is not subdivided. Of the simulators surveyed, Darwinbots is the only one exhibiting this sort of world.
- A fully three-dimensional world. This world may or may not simulate aspects of physics such as gravity.

All simulations that have evolution as an aspect, and thus all that are discussed in this text, additionally have the idea of DNA or Genotype. Genotype is distinct from phenotype in that all organisms of the same species will share a genotype, but their specific state in the world, or phenotype, differs. Some common features of genotypes are

- Usually, but not always, is some sort of specialized programming language.
- Usually represented as a string, often of numbers, to ease the application of mutation and crossover to it.

- The execution mechanism for this language varies greatly, some examples include
  - Stack-based
  - Event-driven
  - Jumps and or breakpoints
- Similarly, the data storage mechanism for this language varies; data stacks and variables are common, but some implementations include registers.
- The language defined is extremely robust; literally no instruction can cause a crash, and any program is valid in the strictest sense.

In a few sims, the concept of a fitness function, or guided evolution, is used, but in most there is no explicitly defined fitness function. The focus is on attempting to simulate natural evolution, which is assumed to be unguided. Thus, the rules of the simulation are set, and organisms are left to find novel ways of exploiting these rules through mutation and or experimentation. There is a goal of emergent complexity, which is taken to the extreme in simulations like Amoeba and Tierra. Usually, but not always, the system is seeded with a hand-crafted replicator which is then allowed to reproduce.

While the concept of mutation is virtually universal in Artificial Life simulations, the concept of crossover, or sexual reproduction, is rarely explicitly defined. The reason for this is related to the goal of emergent complexity; there is hope that sexual reproduction will spontaneously emerge.

In most artificial life simulations, organisms are capable of some interaction with their environment and each other. This allows mechanisms such as parasitism to evolve, whereby organisms exploit each other in order to reproduce. Organisms can often kill each other, and do as a means of gaining resources, but this is less common in the extremely bottoms-up approaches such as Tierra. The higher-level simulations generally allow communication and cooperation between entities, although usually this is reserved for multi-cellular entities. Multi-cellular entities themselves are composed of more than one "cell" but operate cooperatively with each other. Generally they are connected and reliant on each other. In some simulations death of key linking cells leads to fission, in other simulations it leads to the death of the smaller piece; or of the organism as a whole. Some of the very high level simulations implement senses, such as vision, touch, or smell. This is more common in the 3D or 2D implementations that allow movement; many of the grid-based implementations do not allow movement.

Lastly, almost all simulations provide a mechanism to control population. How this is accomplished depends on structure. The grid-based algorithms generally restrict cycles in the sense that each organism is allowed a certain number for each global cycle. There might be reward or punishment mechanisms to reduce or increase this number, and there is implicit reward in that organisms that operate more efficiently in terms of instructions will naturally out-compete more wasteful organisms. When these simulations, which are rigidly defined in terms of space, run out of space, culling is implemented. Sometimes, culling is implemented on creation, such that the very act of creating offspring replaces some neighbor. In the less structured simulations, the concept of energy is often used. There might be a constant inflow and outflow of energy in the system, or a static amount of energy that is recycled continuously through the population. One implementation even includes the concept of waste, which is created as a byproduct of energy use, and will eventually lead to death.

4.3 Notable Alife Simulations

Now that artificial life has been defined, and its general characteristics defined, it is appropriate to spend some time discussing some of the notable simulations in detail. Thusly, this section is divided into a number of subsections discussing notable artificial life simulations in detail.

Table 4.1: Artificial Life Simulations

Simulation	**Year**
Bugs	1989
Coreworld	1990
Tierra	1991
Avida	1993
Amoeba	1996
Evolve 4.0	1996-2007
Darwinbots	2003-2008
Framsticks	1996-2009
Evita	1999
Physis	2003
Breve	2006+
Evogrid	2009

4.3.1 CoreWorld

Coreworld was developed by Ramussen in the early 1990s. It was influenced by the computer game "Core Wars." In this approach a one-dimensional address space is initialized with assembly-level instructions. A number of execution points are distributed through the address space, controlling the order in which instructions are executed. Each address has a resource value associated with it, with execution being dependent upon local resources being sufficient. Noise is introduced by the introduction of new instruction pointers to the system with a random, but low probability. The purpose of this work was to attempt to observe the emergence of self-organizing and cooperative structures [128, 124].

4.3.2 Tierra

A program called "Tierra" was created by Tom Ray. Ray's view was that, essentially, systems in which artificial selection is introduced with predefined fitness functions are inherently limited and dead ended, mostly due to natural selection producing much more innovative and complex fitness functions than humans ever could [101]. Tierra defined an instruction set that was robust to mutations, and thus evolvable. The instruction set was made robust by providing template-driven addressing for instructions, thus increasing the number of functional programs in any population. Evolution begins by the introduction of a hand-written self replicating program ancestor into the memory space. During each iteration, each program in the address space is allowed to execute a set number of instructions, thus avoiding the problem of halting due to poor code. Instructions are not necessarily concretely defined, there is a probability of error. For example, a copy operation might mutate a byte of the data it is copying instead of performing a perfect copy. Programs are also subject to random mutations at a low rate. These two features cause variations of the original program to appear as execution occurs, and if they maintain the ability to self-replicate, can continue to appear in the address space. Programs are easily mutable because the instruction set consists of operations represented by 5 bit numbers. This implies that an organism or program is simply a string of numbers, and is thus easily modifiable by genetic operators [52]. The simulation defines a reaper operation which periodically kills off some members of the population to avoid completely filling memory. The order of the queue for death is determined primarily by a merit system, and secondly by age. There is a mechanism for giving a program a demerit whenever it generates an error code in an instruction; thus encouraging tighter, less erroneous programs due to the merits increasing the likelihood of death. Additionally, Tierra allows for the possibility of rewarding organisms for executing hard instructions [52]. That is, programs which reproduce quickly with relatively few errors tend to dominate the population [52]. Interesting results have been observed, such as the evolution of parasite programs that run another programs copying procedure to copy themselves [101, 52, 128].

In Tierra there is no explicitly defined fitness function [52]. This is achieved due to the fact that in Tierra there are two essential resources, cpu-time and space. Programs which allocate more of these two resources are by definition "fitter" than the rest of the population, thus the fitness function is implied, not explicit [52]. Tierra approaches the problem of artificial life evolution from the bottom up. This makes high level abilities such as learning difficult to obtain, and even low level abilities such as multicellularity have proven difficult [52]. It has been reported in studies that the evolvability of the system is closely tied to the underlying instruction set [128]. Ray also proposed a distributed version of Tierra, known as NetTierra. It was to be distributed among several Universities, with the idea that the increased space and number of organisms and interaction between them would might lead to its own self-sustained evolutionary dynamic [104].

4.3.3 Amoeba

In 1996, Andrew Pargellis created a program called Amoeba. Amoeba is very reminiscent of Tierra, but there is a few important differences. Instead of being seeded with self-reproducing organisms, Amoeba is seeded with random pieces of instructions, and over time the semi-successful execution of these instructions leads to the spontaneous appearance of self reproducing organism. This is accomplished due to the simulation executing any executable instruction it finds, and over the course of many iterations, this leads to code capable of copying itself. Additionally, the copy instruction introduces mutations, as in Tierra. Lastly, if the address space becomes full, a percentage of the cells are cleared, and some are filled with new random instructions [133]. Amoeba is an attempt to simulate the theoretical primordial soup from which life first arose on Earth. Pargellis went on to propose Amoeba-II; which allows the system to partially define its own genetic code [104].

4.3.4 EvoGrid

A more recent attempt to model the pre-biotic chemical soup that lead to the evolution of life is the EvoGrid [32]. The EvoGrid is a recent proposal to create a worldwide grid of computers to simulate the conditions prior to the emergence of life on earth. A second set of computers will search through the grid for life-like emergence and report back on them. This work is in its early stages, but theoretically could have applicability to both the modeling of chemical evolution to aid attempts to create artificial life in a lab, and in studying emergent self-organization [32].

4.3.5 Avida

"Avida" is an artificial life system that was introduced by Adamai and Brown in a paper in 1994 [2]. It was inspired by Tierra and is similar in some respects; although it is designed to be more parallelizable. It combines some aspects of cell automata with the code based aspects of Tierra. Each organism, or string as they refer to it in their paper, occupies a cell in a two dimensional grid in the shape of a torus. The string itself is composed of a series of numbers representing instructions in an instruction set. The definition of this instruction set can vary, the only requirements is that it provide a mechanism for self replication.

Death occurs in Avida by the replacement of the oldest cell in the immediate neighborhood upon each creation of a child string. That is, in order for a cell to replicate itself, it must replace one of its neighbors, and the oldest is replaced. Additionally, during the copy procedure strings are subject to modification. In other words, copies are not guaranteed to be exact, and are changed probabilistically. This is the primary mechanism for evolution in the system, as it is in Tierra. Adamai and Brown indicate that, in both Tierra and Avida, more complicated mechanisms of modification emerge, such as insertion or deletion of instructions, and doubling of the genome, emerge from this simple modification scheme. For this reason, they chose not to include an explicit crossover operation or sexual reproduction scheme [2].

Adamai and Brown state that they abstain from the flawed execution of instructions; unlike in Tierra where instructions can execute imperfectly. The cited reason is that they do not feel it is a crucial feature in their simulation. Parallel execution is obtained by allotting each cell of the grid a time slice. Once its time slice is completed, each cell is in some defined state. Once every cell has had its time-slice, and computed its next state, the entire grid is updated at once to the next state. This is akin to the way in which cell automata update. If the time slice is kept small enough, a string is unable to affect its local neighborhood unduly in the allotted time, such as by reproducing twice. There is a conflict in that Adamai and Brown define reward for the correct execution of user-defined tasks, in the form of bonus time slices. This conflict is resolved by a mechanism that keeps the average time slice constant while allowing individual's to have more or less than the average dependent upon their individual bonus [2].

4.3.6 Physis

A somewhat radical departure from the mold established by Tierra was proposed by Egri-Nagy and Nehaniv in 2003 [39]. In their Physis simulator they propose the incorporation of the processor definition and instruction set to the genome, and allowing these definitions to be modified along with the actual executable code. Simulation is initialized, as in Tierra, by the insertion of a hand-crafted replicator. This replicator consists of a processor definition, an instruction set definition, and executable code.

Processor modification is accomplished by allowing for universal processors with variable numbers of register, stacks, and queues. This is defined in the genome by a series of R's, S's, and Q's. Stacks and queue's can be separated from each other by a blank, or B. In their paper, they give an example processor definition as the string "RRSSSBSS", which translates to a processor with 2 registers, and 2 stacks of size 3 and 2 [39].

Allowing for a modifiable instruction set is accomplished by first defining a set of primitive, RISC like instructions. These operations provide functions like load, store, basic math and comparison operations, etc. A full list is available in [39]. Instructions themselves are represented by integers, and operations, if required, are integers as well, though they are modulus'd by the number of components, e.g. the number of registers, to ensure a valid address. From these primitives new instructions are defined. These defined instructions are the actual instructions that the organism can execute.

The reported results were somewhat mixed. They reported modification of the processor definition in their experiments, but the modified definition did not achieve dominance. Additionally, while it should be possible for the number of defined instructions to change, this behavior was not observed. The instructions themselves were modified, however. Chiefly, they demonstrated the evolutionary potential of universal processors to be equal to that of fixed processors [39].

4.3.7 Cosmos

Cosmos is another artificial life simulator. It is again a cellular based arrangement of organisms. Cells in cosmos can only read code in its own cell, unlike some other simulators such as Tierra. Communication between organisms is achieved by message passing. This message passing takes the form of two stores that respectively hold messages being composed, and received messages. Once composed, messages can be broadcast to the local environment. Additionally, messages can be received with a command as well. Reproduction is achieved by writing genetic information into a special store, and then executing a divide instruction. This divide instruction causes the information in this store, called the nuclear working memory, to a nearby grid location. In order for an organism to execute instructions, cells must pay a fee in energy tokens. Cell's gain energy tokens from the environment, tokens being distributed to each cell in numbers defined by various distribution schemes, and this is also used to determine cell death. Energy tokens are also removed from the environment periodically, although it is not necessary that the same amount will be removed as added; this can lead to a sparseness or abundance of tokens. Additionally, in the event of a full memory space, cells with the lowest number of tokens are culled. Thus, a cell must balance action with hoarding of its energy stores. Upon death, unused tokens are returned to the environment. Execution itself is controlled by bit strings called regressors and progressors. Interestingly, these can be included in messages, and if a cell does not protect itself against this, execution will begin in the message in a form of parasitism or attack [127].

Organisms in Cosmos can consist of more than one cell. The logical restrictions of adjacency apply to this. Multi-cellular programs can share energy between its cells, and due to the fact that cells can die at different times, and individual cells can chose to die via a kill command, implicit fission is possible whereby a single organism becomes two. There is a cost in energy tokens associated with being part of a multi-cellular organism. Organisms, whether single-celled or multi-cellular, can move about the grid, although in the case of a multi-cellular organism this is a vote. Additionally, cells trying to move against each other incur a friction parameter, which will limit the amount of movement possible [127].

The programming language for Cosmos is similar to that used by Tierra, with additional operations defined to account for the additional features of Cosmos. Jumps are replaced by Cosmos concept of promoters and repressors. Promoters are strings that exist on a special store. The promoter at the top of the store is the currently active promoter, and execution begins at the point that the promoter maps to in the genome. If non-existent, or upon execution termination due to a repressor or end of the genome, that promoter is moved to the bottom of the promoter store. Repressors work similarly, although all are active at any time. Essentially, if execution reaches a position that a repressor maps to, execution stops. In a way, repressors are akin to break points, and promoters are akin to jumps. As mentioned above, both may be communicated to other cells, allowing for parasitism or attacks as in viruses or bacteria, as well as specialization in a multi-cellular organism. Mutation occurs at a low rate throughout Cosmos, and can affect any cell structure. Additionally, as in other artificial life simulators, instructions can randomly execute in a flawed manner and thus produce abnormal results [127].

4.3.8 Evita

Evita is an artificial simulator akin to Avida and Tierra, but no code parasitism is allowed, and interaction is restricted to adjacent cells in a two dimensional grid. The simulation begins with the seeding of a self-reproducer, which then reproduces into nearby cells. If no unoccupied cell exists, the system selects one of the oldest neighbors at random and replaces it. Mutation chance is based on each instruction, thus longer programs are more likely to mutate [13].

4.3.9 Bugs

Bugs is an agent-based model wherein the aforementioned agents move about their world, sensing it, ingesting resources, reproducing and dying dependent upon internal resource levels. Resources are distributed about the environment, and are replenished upon consumption. Movement is governed by a "Sensorimotor" map; a table of behavioral rules of the form If X sensed then Y. Agents receive sensory input about resources only. Agents that try to move onto an occupied position randomly walk to another position. Agents must consume resources to exist or move, or to reproduce asexually. The Sensorimotor map is passed along to offspring, although mutations can occur. There is no defined fitness function, fitness being implicit through the fact that poor amps will lead to agents being out competed and dying off. Population size is indicative of the relative fitness of the dominant maps [13].

4.3.10 Framsticks

Framsticks [70, 69, 71] is a 3D artificial life simulation. The organisms in this case are "stick" figures. Framsticks incorporates neural networks into its framework, in addition to genetic algorithms. Unlike some other simulators, such as Tierra, the evolutionary techniques in framsticks are much closer to traditional genetic algorithms, with crossover, selection strategies, and fitness functions defined. In theory, the creator's hope to move towards an open-ended evolutionary model, one possible way of doing so is to define "life span" as the fitness function, which implies ability to survive and reproduce are the selection pressures [69, 71].

The world of framsticks is a 3D simulation, although physical interactions are sometimes limited, such as the ability of entities to collide with themselves. There is a necessity to simplify things for the purpose of speed [70]. The system also limits the number of simultaneous individuals being simulated, for purposes of speed and interactivity [70].

The organisms themselves, as mentioned, are comprised of "sticks." These sticks are connected by musculature. More formally, the organisms are composed of sticks, neurons, receptors, and effectors. Sticks correspond to body parts, having attributes such as health or strength [71]. Effectors correspond to muscles, and again have attributes such as strength, or range of motion [71]. Receptors correspond to senses, with examples being equilibrium, smell, and touch [70]. Neurons are neural networks, unrestricted with size. This allows for intelligence, although the degree is uncertain [70].

Evolution in framsticks is handled similarly to genetic algorithms, with crossover and mutation being defined [70]. They propose the use of similarity to encourage population diversity; with similarity to competing organisms lowering the organisms corresponding fitness. While similarity can be computed in several ways, the system they use is a heuristic algorithm that treats both individuals as graphs [70]. The type of selection model can vary, from the traditional roulette-wheel selection [69] to one at a time selection. Tournament selection, or the combination of the various schemes for use in separate populations is also mentioned [70]. The sim records several parameters over the life of the organism, such as age, distance traveled, or average velocity, and these can make up the fitness function for the purpose of directed evolution [69].

Significant time is devoted to the genotype representation in framsticks. They include the neural networks in their genotype, although it is unclear if the 'state' of these networks is also included, or simply their configuration. Multiple encodings are defined, with a basic encoding that corresponds to simply a list of all the components of an organism, and all their attributes. Two other encodings are discussed at length, a tree-like representation that allows for connections to be moved around without being broken. Most interestingly, they discuss a developmental encoding, whereby a phenotype is created by the differentiation of cells as they divide [70, 69]. In testing, the higher level encodings performed better, possibly because of the fact that they restrict the problem space [70]. All encodings define their own mutation, crossover, and optional repair methods. The repair method is more necessary in higher-level encodings whereby invalid creatures might be created, and attempts to fix these simple errors [69, 70]. Other encodings were mentioned, but not defined [69].

Results with framsticks were reported for the evolution of many walking/swimming organisms [69, 71]. In these simulations, the evolution of concepts of movement was observed. Good ideas evolved once, and then were propagated widely by crossover until being replaced by a better method [69]. Additionally, a height fitness function was used to compare the different encoding methods [70]. It is noted that evolution is an extremely effective way of finding errors in the simulation, due to the organisms discovering and ruthlessly exploiting them [71].

4.3.11 Darwinbots

Darwinbots [29] is an artificial life simulator currently under development. In it, the world is comprised of a flat plane, upon which the 'bots' move and interact. Unlike many artificial life simulations, it is not subdivided into cells.

The bots themselves are two dimensional, with their behavior being defined by their dna. They can see omni directionally, and detect collisions with and attacks by other bots. Actions require energy to execute, and the execution of actions produces waste. This leads to a balancing act between energy and waste. This is an act that the bot will eventually lose, due to all mechanisms for waste management being imperfect by design. This waste accumulation will eventually lead to a condition defined as "Alzheimer's." This condition causes the insertion of random numbers into random places, causing the bot increasingly to malfunction. This eventually leads to death. Bots acquire energy either from the environment in the case of vegetables, or from eating other bots as animals, and can store it as body. Bots can interact with each other through various "shots" which are projectiles and can serve as attacks, attempts to feed, communication, or infection. Interaction is also possible through ties, which can also be used for feeding. Ties can also be allowed to "harden" and become permanent, leading to the formation of multi-cellular organisms, or multibots. Multibots can share information, energy, and protection. Bots can die due to waste accumulation, or through losing more than 50% of its current energy in a single cycle [29].

Reproduction in Darwinbots is handled by commands, and occurs whenever a certain energy threshold is reached. This threshold is defined by the organism. Once reproduction occurs, children are temporarily tied to their parent for 15 cycles by a birth tie. They inherit some memory from their parent immediately, and through the birth tie. Children also inherit a portion of the parents body and energy, and the whole of the parents DNA. This DNA is subject to mutation, and this is the main agent of change in Darwinbots. Darwinbots, as with most artificial life sims, defines no artificial fitness function, selection and fitness being an emergent phenomena [29].

The structure of the DNA in the current version of darwinbots (2.0) is designed around the concept of daemons. These daemons are triggers that wait for some action to occur, then execute. These daemons are represented by genes in darwinbots, with the triggering action appearing at the beginning of the gene. The instructions within the gene themselves are restricted from having jumps or cycles. Execution is stack based, and memory is also provided for the preservation of information between cycles. The creators propose a more complicated encoding for the next version, but that is not discussed here, primarily for lack of detail. Information can be found on the website [29].

Several interesting behaviors are listed, but it is unclear if these behaviors emerged naturally, or were designed. They include swarming of groups of bots being aware of each other and choosing to move in the same direction, and bots creating a permanent tie to a vegetable and using it as a portable food source, while keeping it alive. Many commonly evolved behaviors are also listed, some interesting ones including Big Berthas, Cancerous Bots, Cannibots and Lame Bots. Big Berthas are bots that lose the capability to reproduce, and increase in size until they become semi-immortal and kill off the rest of the population until only they remain. At this point, they die due to waste. Cancerous bots are bots that continually reproduce, leading to spikes in population size, although this is beneficial for a vegetable. Lame bots are bots that lose the ability to move, and again this is not necessarily a bad thing if there is a stable nearby food source [29].

4.3.12 Evolve

Ken Stauffer wrote a cell-based artificial life simulation called Evolve in 1996. Various updates continued through Evolve 4.0 in 2007 [123]. Evolve 4.0 is somewhat similar to Tierra and Avida in that it is based on a cellular automata, and the cells have a programming language that they executed. Evolve allows multi-cellular organisms. Organisms are always composed of adjacent cells, and in the event of cell death causing an invalid organism, the simulation partitions the organism, and kills the smallest partition. Cells make the choice to 'grow' into multi-cellular organisms through a command. Additionally, cells within an organism have the ability to move around. Organisms themselves move as a whole, assuming that the destination is empty. Unlike in Cosmos, there is no concept of friction, the organism simply does not move. Within an organism, cells can communicate through setting a 'mood' value that other cells can read, and by sending a message to adjacent cells or broadcasting it to all cells in an organism. This is similar to the mechanism described in Cosmos, although in this case it cannot include executable code, or occur between organisms [123].

Evolve allows organisms to use a data stack to receive data. To regulate the use of this stack, the concept of energy is introduced. There is a finite, constant amount of energy in the system. Organisms must maintain at least one energy to exist, and it costs energy to maintain more than 50 items on the data stack. Additionally, energy is required to create offspring. Almost no other operations require energy, however. Thus, its primary purpose is to encourage responsible use of the stack [123]. If organisms do not have at least 1 energy, they die, and any energy expended towards having more than 50 items on the data stack becomes organic material, a resource. Organisms in Evolve can gain energy by eating organic material or other cells.

The genotype for evolve consists of a programming language known as KFORTH, which is based upon the FORTH programming language. This language is tabular, with elements representing numbers or instructions, and rows representing code blocks. Execution starts at row 0, element 0. During execution, numbers are pushed onto the data stack, while instructions are executed. At the end of the row, the previous 'start' location is removed from the call stack. There are operations, such as call, that push a new code block onto the call stack. the if and ifelse instructions call a new code block if a condition is met. This structure makes mutation and crossover simple and fast to perform, while also ensuring executable code [123].

Reproduction in Evolve is handled through spores. Spores are created with energy from the creator, and if eaten give that energy to the eating organism. Additionally, they can be fertilized, creating a new organism. During fertilization, the organism might choose to provide additional energy, but this is optional. If fertilized, the new organism's energy is the sum of the initial energy and any energy provided during fertilization. If fertilized asexually, then genetic code is inherited from parent, though possibly with mutations. Mutations may affect single instructions or whole code blocks, and include duplication, deletion, insertion, transposition, and modification. If fertilized sexually, then uniform crossover is performed between the two parents, with respect to code blocks. That is, uniform crossover determines which parent contributes which code block to the offspring. If lengths differ, then the organism automatically receives code blocks from the longer parent [123].

Evolve's language provides primitives for looking around, with one of the goals of the simulation being the evolution of organisms that look around a lot. Another stated goal was the evolution of interesting, possibly intelligent behavior. Stauffer planned to run the simulation continuously for a whole year, but it is unclear what results were gleaned from this process, or if it was even completed. The last available snapshot is on day 298 [123].

Although not strictly an artificial life simulator, Hicks and Driscol showed that the 3D virtual environments of games can be used as a viable simulation environment for the evolution of AI software agents [56].

4.3.13 Breve

Another simulation environment of note is the Breve simulation environment [68]. This is a tool set allowing for the relatively rapid implementation of artificial life and distributed systems simulations, particularly by users from a non-programming background. It provides modules for the 3D rendering of scenes, physical simulation of the environment, collision detection, event detection and scheduling, and other things. It is worth mentioning here for completeness, as well as the 3D aspects of the environment, which have proven useful to aid in the interpretation of behaviors. This is illustrated in a paper by Spector and Klein in which the 3D aspects of Breve allowed them to notice multi-cellular organisms in their evolutionary swarm algorithm [121]. There were further experiments where they added musical notes to this simulation, generated from the interactions between the beings in the swarm algorithm [120].

Chapter 5

DNAGENTS 1.0: GENETIC MOBILE AGENTS

5.1 Mobile Agent Concerns

It is clear that mobile agents are a promising avenue of investigation. There are many advantages, as shown in section 2.3. These include reduced network traffic, their inherent parallel nature, adaptiveness, tolerance of intermittent connections, reduced maintenance, and portability. There are also, unfortunately, disadvantages which were discussed in section 2.4. These include security, robustness, efficiency, complexity of agents and of the framework, and authentication. Unfortunately, the relative scarcity of mobile agent implementations in the real world would indicate that the advantages do not overcome the disadvantages. Thus, we must set ourselves to the task of addressing these concerns.

One could spend a great deal of time discussing approaches to the various disadvantages of mobile agents. The focus of this work is on addressing security, robustness, efficiency, and the complexity; thus discussion of authentication approaches is minimized, except where it connects with the other disadvantages.

Mobile agent security is one of the largest areas of concerns, with a great deal of work. This work can largely be classified into work on the security of the host environment, and security of the agent. The approaches to securing each, while related, are distinct, so can be discussed separately. Host security can be further subidivdied into the concepts of Identification and authentication, and the Sandbox/Safe Language concept. The related, but distinct area of the security of the actual agents is generally accepted to be much harder. There are several areas attempting to address this such as detecting and managing compromise [99, 27, 61, 58], trusted nodes [58], secure hardware [27, 26], and protecting the code [110].

Another major disadvantage of mobile agents is the insurance of robustness. This area is also known as fault tolerance; and is a major field of inquiry in and of itself, beyond the application of fault tolerance to mobile agents [115, 116, 94]. Replication is one major way of ensuring fault tolerance [88], but it becomes difficult to ensure the exactly-once principle [107], which is very important to mobile agents.

The last major disadvantage of mobile agents that is of interest is the question of their efficiency and their complexity. The major approach to combating the complexity of mobile agents and improving their efficiency is the development of modeling systems such as the API calculus [97], and the API-S security extension to it [49].

5.2 Analysis of Possible Solutions

Having identified again the major disadvantages of mobile agents, it is now possible to discuss possible solutions to these. Approaches to host security are discued in sections 5.2.1 and 5.2.2. The other aspect of security, agent security, is covered in sections 5.2.2 and 5.2.4. Next robustness is addressed in section 5.2.5. Lastly, approaches to minimizing compromise are presented in section 5.2.6.

5.2.1 Identification and Authentication

The concept behind identification and authentication as a method of increasing host security stems from the idea of "trust." Trust in this context refers not to specific implementations to ensure trust, but to the concept. If an individual and his associations are known, then a judgment about whether or not he should be allowed into a server can be made. As an analogy, consider a uniformed officer. The average person would not invite a stranger into their home, but the average person would be much more likely to allow a representative of authority, such as a uniformed police officer, into their home. Now, it is possible for anyone to pass themselves off as a police officer, with varying degrees of success. Authentication and Identification try to ensure that this does not happen. Generally, these approaches revolve around uniquely and reliably identifying an agent, possibly including its point of origin, and making a judgment about the trustworthiness of that agent [136]. For example, by applying a signature to the code that if tampered with, no longer validates. To put it differently, if the signature is intact, then one can be assured that the agent was in fact written by a certain entity and the trustworthiness of that entity is transferred to the agent. Similarly the entity is accountable in some sense, for the agents actions. This functions reasonably well assuming that there are no mechanism by which the identity of an agent can be copied. A related mechanism is to in addition to identifying an agent or type of agent, is to try to ascertain its trustworthiness by surveying other "peers". Rather, the agent's reputation is an aggregate of the reported experience with that agent by many nodes [36]. Similarly, this functions well if there is no means to change the identity of an agent. There is another mechanism for disruption with a reputation based system, in that it is difficult to ensure that an agents reputation cannot be tampered with by false or malicious reporting, for instance.

5.2.2 Safe Languages and Sandboxes

The other major mechanism for ensuring agency security stems from the idea of making it impossible for an agent to do any real or lasting harm to the agency. One example of this type of approach is a "safe language." A safe language is a language designed such that no actions can be taken which damage the host; its a restricted language in which the agent has limited power. This is difficult to design, perhaps, and can limit the usefulness of the agent, but is effective. A related and more common method, is the concept of a sandbox. The sandbox refers to a virtualized, limited, encapsulated environment in which the agent has limited access to local resources, so any disruption the agent can inflict is localized and minimized [7, 67].

5.2.3 Detecting and Managing Compromise

To move on to the concept of protecting the agent, the first and biggest category is the detect and manage category. These approaches generally assume that it will happen, and seek to minimize the damage when it does occur. One method of doing so is the use of multiple agents, as in [99]. Multiple agents can address security in several ways; by distributing the algorithm to many agents and compartmentalizing each agents involvement, one can protect the overall algorithm from compromise[27]. By parallelizing the search process through multiple identical or redundant agents, one can help ensure the task completes successfully in spite of compromise or bad information(bad information can be identified through mechanisms such as voting [93]). These two approaches to multiple agents are by no means mutually exclusive.

Two other mechanisms for managing compromise are the encrpyion of partial results, such as in [61]. In this sort of approach, at each step the partial results are encrypted in such a way that the home system can decrypt it, but the agent itself lacks the mechanism to do so. In this way, if the agent is compromised at the next step in the algorithm, at most only the current step the agent is working on is compromised, previous data and steps are still encrypted, and possessing the agent does not help the malicious entity in decrypting the data since the agent only has the mechanism for encrypting the data. A similar approach involves the agent returning home frequently, and offloading any partial results or sensitive data. This type of approach is described in [58]. While these approaches have focused on protecting the data, the code might also be important as well. Two major approaches to doing so are obfuscation and encryption. Obfuscation is a process by which all meaningful whitespace and variable names are removed from the code in order to make it as unreadable to humans as possible. This attempts to make it more difficult to determine the contents of the code. Encryption of the code ensures that the code cannot even be executed unless either the encrpyion protocol is compromised, or the host is authorized to do so. Descriptions of measures to protect the code are described in [110]. Generally, encrpyion is only as strong as the encryption algorithm, but encrpyion can be quite strong and works well in practice.

5.2.4 Avoiding Compromise

The other major approaches to agent security involve means in which to avoid compromise. It was proposed in [58] to compile a list of trusted nodes; nodes that were trusted not to be compromised. Presumably this list would be periodically re-evaluated. This is a good idea, although the trustworthiness of nodes might change rapidly, as compromise is by nature an unanticipated, and generally a temporary condition. Secure or tamper-proof hardware has been proposed as a workaround to this problem in [27] and [26]. This would solve the issue, if the list of visitable nodes only contained nodes with secure, tamper-proof hardware, then nearly all forms of compromise would be eliminated. There is a much higher cost associated with this approach than all others, unfortunately, which makes it unlikely to be implemented widely.

5.2.5 Fault Tolerance

Somewhat ironically, one of the selling points of mobile agents is their increased fault tolerance compared to traditional client-server architectures and protocols. It is also one of their weaknesses, in that although yes, they are less susceptible to flaky Internet connections, they do exist in the network and are thus subject to link outages and node failures. Ensuring that agents are not blocked, or prevented from completing their task by a node or link failure, agent redundancy is usually used. While redundancy in agents makes it easy to recover from an individual link or node failure, it becomes correspondingly more difficult to ensure that each action is executed exactly once. This is partly due to the uncertaintity the Internet adds to the theoretically simple action of detecting a crash: in an environment such as the Internet it is difficult to accurately and reliably detect a crash. The uncertainty of when or if to relaunch or restart execution from a redundant agent leads to uncertainty about whether or not an agent will execute exactly once. The exactly-once property is not significantly important for all applications; search queries and any kind of data gathering agent does not suffer from this. On the other hand, it is of paramount importance for an agent that takes an action on behalf of a user, such as a financial transaction.

One type of approach to providing mobile agent fault tolerance in the use of multiple agents for redundancy, but requiring the group of agents to vote on whether or not an action was successfully completed. This is comparable to the multi-agent approach to security where agents vote on whether or not data has been compromised. One approach, proposed in [108] proposes that an agent at a node pick a primary destination node, and send an active copy of itself to that node. It also, simultaneously sends observer copies of itself to N alternative execution points. In the event of no failure, the observer copies do nothing; but in the event of failure, they allow the agent to resume execution from that node instead. To further ensure that actions happen only once, it requires a majority vote of all N copies and the active agent before a commit can occur, or the decision to resume execution. This type of mechanism of sending copies to alternative points of execution, or reserving copies, is known as replication. The approach in [108] makes the assumption that their are sufficient available redundant execution points. Indeed, as the agent gets closer to finishing its tour of steps, called stages in their algorithm, the potential restart points begins to shrink, and the algorithm is more prone to blocking. Most approaches to fault tolerance ensure that the agent will "survive" fault through replication, and go about ensuring the exactly-once property by introducing some sort of protocol for inter-agent communication that attempts to ensure the agents as a whole act to a single purpose, such as [94] which models agent execution as a series of agreement problems. Alternatively, one can attempt to modify the platform itself, the agency, in order to attempt to ensure this property[51] through the accurate detection of fault or crashing. There has been quite a bit of work in this area, but the problem has not been completely solved.

5.2.6 Combating Complexity

The last major disadvantage of mobile agents has to do with their complexity. Simply put, mobile agents are more complicated than client-server approaches and protocols. They violate most security assumption, and many other assumptions that other similar protocols are build upon. This can make understanding them difficult. Indeed, correctly implementing both the platform or framework itself can be difficult. On top of this, implementing an effective mobile agent is also difficult. There isn't much being done to combat this complexity. On the understanding approach, modeling tools such as an extension of the Pi Calculus for mobile agents, API, have been created [97]. Similarly, others have since extended API to handle some security specific situations and concerns [49]. While these models are effective tools for studying the implications of various effects to mobile agents, they do not directly combat the underlying problem that mobile agents are complicated to implement and use effectively. Indeed, it is uncertain how to go about providing this, and this is truthfully possibly one of the greatest obstacles to widespread use of mobile agents, after security. Of those that understand them, many rightly fear them for the security implications, and to those that fail to fully understand mobile agents the advantages are not apparent.

5.3 Other Approaches to Genetic Mobile Agents

There is another possibility for addressing the concerns with mobile agents: applying genetic algorithms to mobile agents. A full review of genetic algorithms is provided in chapter 3.1 Genetic algorithms have several aspects that might naturally compliment mobile agent concerns such as security, robustness, efficiency, and complexity.

The natural obfuscation of candidate solutions inherent to genetic algorithms is of benefit to security concerns; it is often difficult to understand exactly how a particular solution arrived at by a genetic algorithm works, much less reverse engineer it. The code of the solution is seldom encoded in a way that makes it easily human readable, and seldom follows any logical patterns that a human can pick up on easily. Rather, genetic algorithms find ways to do things, they are not always the way a human would do the same thing. All this makes it more difficult to compromise code. Additionally, the importance of a genetic algorithm lies in the methodology used to generate the candidate solution, not on an individual candidate solution. This alleviates some concern about preventing the algorithm of the agent itself being compromised.

Genetic algorithms are based on evolutionary theory. This seems a natural fit for improving somethings robustness and efficiency. That is, evolutionary theory states that the fittest candidate is more likely to survive to reproduce, thus over time all candidates improve. While genetic algorithms artificially define a specific fitness function, it is difficult to evolve a solution that isn't capable of surviving. While this might require tweaking for optimal results, it would seem that genetic algorithms would simply tend towards providing a solution more capable of persisting through disruption, and thus, more robust. Similarly, genetic algorithms are primarily used in optimization problems. Again, this naturally seems to fit the desire to make mobile agents more efficient.

Lastly, genetic algorithms would provide a degree of abstraction. This could potentially simplify the creation of mobile agents; the user only needs define the goal, not all the steps. Genetic algorithms are generally used to find solutions to difficult problems. If it were simple to find an optimal solution to the problem, why use a genetic algorithm? Indeed, it is largely the purpose of genetic algorithms to solve difficult problems.

It should not be terribly surprising at this point that others have tried to apply genetic algorithms to mobile agents. It is somewhat surprising that there has not been a great deal of work in this area. In a cursory look at the material, it might sound like some attempts at artificial life (for a discussion of artificial life in general see chapter 4.1) use the term Agent, and also use aspect of genetic algorithms, they are distinct from genetic algorithms, and their concept of agent refers to the original definition of agent, and is distinctive from mobile agents. A notable example of this is the Bugs simulator, discussed in section 4.3.9. Artificial life as it relates to the evolution of mobile agents is discussed later in section 5.9. The actual work combining genetic algorithms and mobile agents has revolved around evolving neural networks to control agents [19], using layered learning to evolve teams of agents for multi-agent systems [57], evolving behavior graphs for agents[62], and evolving communication behaviors for multi-agent systems [79]. This might not be a completely exhaustive list of attempts to apply genetic algorithms to mobile agents, but it does comprise the prominent and easily discovered attempts at the time of this writing.

5.3.1 Evolving Neural Networks for Agent Control

Braun et al [19] drew an association between intelligence and control mechanism. That is, intelligence for an agent is defined as a problem of optimizing its control scheme; ergo the application of a genetic algorithm to aid the search for the optimal parameters for the control scheme of mobile agents. The author defends the selection of neural networks as being the control scheme most commonly associated with intelligence, but admits this selection is somewhat arbitrary. There are two types of optimization to consider; the optimization of connectivity and the optimization of weights. The connectivity can be considered a design problem; what is the optimal design for the neural network. The weights problem corresponds to a learning problem. Thus, this approach uses genetic programming to optimize the design of the neural network, and as a second level each individual is trained by a local heuristic. This is not an entirely unique approach, in that genetic algorithms have been applied to the optimization of neural networks in prior work. It is unique in the application to mobile agents. While this work is interesting, it does not necessarily demonstrate a capability to overcome any of the other issues inherent to mobile agents; indeed it adds complexity. This is not to say that it is a flawed approach, simply that this work demonstrates genetic algorithms and neural networks can be evolved to effectively control agents; not that these agents are more secure or resistant to fault.

5.3.2 Using Layered Learning to Evolve Teams of Agents

The second approach that will be examined is an application of layered learning to multi-agent systems described in a work by Hsu et al [57]. The "goal" in this work is to use layered learning to evolve agents capable of working together to play soccer. While this involves agents, and not mobile agents, it is an example of using genetic programming to evolve team-based behavior. Layered learning using genetic programming involves evolving solutions using a population of solutions to a simpler problem. Specifically, in layered learning a problem is divided into a hierarchy of subproblems. Then, starting at the leaves of this hierarchy, solutions to each subproblem are evolved. These solutions are then used as seed population for the parent nodes. The authors were successful in evolving intelligent agents for a cooperative Multi-agent-system task, namely soccer. Partly this is due to the ease with which a cooperative task such as soccer can be decomposed into subtasks. This is akin to the way in which groups of humans learn to play soccer. This is not a weakness of the approach per se, but it does not necessarily follow that the approach is applicable to mobile agents.

5.3.3 Evolving Behavior Graphs Using Genetic Programming

It was proposed by Katagiri [62] to evolve "intelligent" behavior in agents by representing various simple actions as nodes in a graph, and intelligent behavior as a transition order across that graph. Rather, the nodes represent actions, and the links represent the order of the actions. Nodes are divided into judge nodes and process nodes. These nodes correspond to judgment or process actions that are predefined by the user as possible actions. Execution is taken by visiting the nodes, although unlike regular genetic programming execution does not resume from the root node for the next action, the argument in the paper is that this allows previous states to affect future states. The initial connectivity between nodes is randomly selected, then subsequently the optimal connectivity is evolved through basic behaviors such as crossover and mutation. Their results indicate that their method, genetic network programming, has better generalization flexibility than regular genetic programming. For the purposes of this work, it is an interesting approach, but does not deal with mobile agents directly. While the idea of predefining possible actions and then selecting the optimal set and sequence of actions would be an interesting methodology to attempt to choose between competeting survival parameters, it is at this point pure speculation and indeed outside the scope of the work in question; [62]. It is also somewhat outside the scope of this work, in that the goal of this work is to evolve behaviors themselves, and not select between existing behaviors.

5.3.4 Evolving Communication Behaviors for Multi-Agent Systems

The focus in Mackin's[79] work is on the evolution of communications behaviors for multi-agent systems. To review, a multi-agent system is a system in which multiple independent agents cooperate with each other to complete related tasks. This is an important and worthwhile endeavor, chiefly because effective communication systems for multi-agent systems can be complex and difficult to design, yet are essential to the success of nearly all multi-agent systems. Specifically, this work applies a specific type of genetic programming to this problem, primarily to increase the ability of the mutli-agent system to react to a changing environment. That is, predefined communication protocols, while effective, limit the systems flexibility and hamper its autonomy. The form of genetic programming they use, Automatically Defined Function genetic programming is somewhat similar to the layered learning method described previously, indeed that paper makes reference to it in contrast to itself. It revolves around decomposing the problem into individually solved subproblems. The results of this work were promising, the work demonstrating that it is possible to evolve an efficient communication mechanism without prior knowledge of the domain. This work is applicable to the question of efficiency and complexity in mobile agents, but does not neceseisarily indicate any advantages for security. Indeed it does however show that genetic algorithms is a promising avenue of investigation; if its possible to have agents dynamically evolve an effective communication protocol, why not a security protocol?

5.3.5 In Summary

While there has been some work applying genetic algorithms to concepts such as autonomous agents, this work is mostly focused on agents, and much if it is targeted at a robotics domain. While some of this might be applicable to the problem of mobile agents, it has not been applied to it, to date. While in some sense mobile agents are descended from agents, their domain is very different, and many of the challenges are different, ergo the approaches to solve complexity issues with agents do not solve these problems with mobile agents. Similar, agent security and mobile agent security are drastically different, as is agent and mobile agent robustness, both due to the Internet's inherent uncertainty. All this leads to the conclusion that this is a promising area, and that further investigation is warranted.

Various methods for addressing mobile agent concerns were discussed previously. This conversation lead to the discussion of various attempts to combine agents and genetic algorithms, and the promising nature of combining genetic algorithms with mobile agents. Towards that goal, in this chapter two things are presented. First, the initial attempt to apply canonical genetic algorithms to mobile agents is presented. This attempt was successful if not completely ideally optimal, and more importantly it highlighted some concerns with applying genetic algorithms to the domain of mobile agents. This is followed by a discussion of various modifications to genetic algorithms that might address these shortcomings, and ultimately the second, more major part of this work, the description of a variation of genetic algorithms, called DNAgents, that is more well suited to the domain of mobile agents.

5.4 Evolving Genetic Mobile Agents

In order to set about evolving genetic mobile agents, one must first have a mechanism to simulate a network. Mobile Agent Simulator (MAS) was developed for the purpose of simulating agents, due to the lack of a pre-existing mobile agent simulator [60]. Figure 5.1 depicts an early version of the visualization component of Mobile Agent Simulator. The simulator is discussed in more detail in [60]; although any modifications to it necessary for various experiments, and the aspects of it that directly impact those experiments, will be discussed as necessary. It should be noted that the simulator includes an execution environment for code, similar in structure to assembly or machine code. Discussion of the simulator is limited primarily to discussing the agent language, as the language that comprises the genotypes has a great impact upon the ease of evolving solutions.

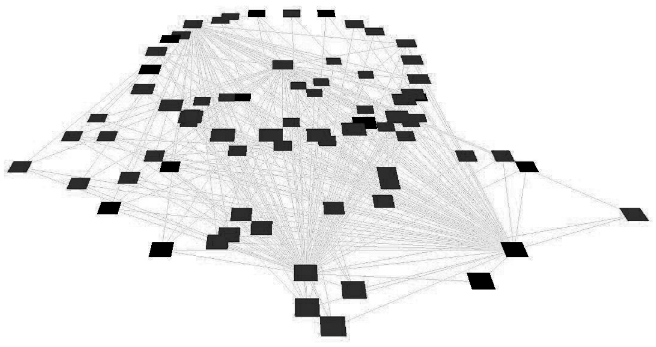

Figure 5.1: Early screenshot of Mobile Agent Simulator

With a simulator in place, the task of evolving agents could commence. First, what would be an appropriate early task to attempt to evolve agents for? While things such as survival were considered, ultimately it was decided that for a starting point the evolution of an agent capable of exploring the network would be a good starting point. This algorithm could in the future be of benefit to search algorithms, so this is a worthy starting point.

Towards that end a mutation operator was implemented along with Elitist selection, and an initial population of 100 agents comprised of 25 lines of random code each were created. Specifically, in the selection algorithm the lower ranking half of the population is replaced with new individuals. The agents were created by selecting first a random number between 0 and the number of defined instructions, then two random numbers between 0 and 100. The 100 was arbitrarily chosen as the upper limit for identifiers. A random network was generated, and the agents were executed for a generation consisting of 15 seconds of simulation time. Movement within the simulation is not free; transit time is computed and an explanation of the algorithm can be found in [60]. The fitness function was defined as the number of unique nodes visited, which was tracked within the simulation, and not by the agents themselves. It is worth noting that the previously described implementation differs not at all from the canonical, traditional genetic algorithm described in chapter 3.1.

These random starter algorithms did not perform very well; although due to the number of instructions possible, and the number of lines that comprised them, they would often visit one or two nodes simply through luck, and because undeclared variables return a zero, so calling send with an invalid variable attempts to visit node zero. After several generations, the interaction between mutation and elitist ranking caused agents with several sends to be more favored. Very little looping was evidenced, but agents continued to visit more nodes, reaching about 8 unique visits in the 15 second generations. Eventually, code appeared that mirrored very closely the hand-written code, except with no looping. Looping was the last thing to evolve, and ironically the code that was first observed looping was essentially the hand-written code described above, but with a few extra operations that were 'jumped' over.

In order to describe the evolution of agents, a necessary discussion of the agent language is required. The initial language was very simple, consisting of only a few operations, described in table 5.1. These were the bare minimum instructions required to have an agent visit nodes in a network, and included no conditional operations. With respect to the simulator, variables are currently referred to by a numerical ID; for example variable 1. At the time of this experiment, there were no logical limits on the number of variables an agent could have. Additionally, it should be noted that all commands are valid, and none are capable of causing a crash. For instance, trying to jump before or after the agents code is ignored. The first agent, handwritten for testing purposes and depicted in table 5.2, serves to illustrate the genotype. For clarity, the actual genotype of this agent is represented by the string 1104101207123222220500.

Table 5.1: Initial Instruction Set

Instruction	Description
NoOp	Does nothing.
Alloc X	Allocate variable referred to by id X.
Store X,Y	Stores integer X in address Y
Jump Line	Jumps to line specified by the number; not a variable
GetNumNeighbors X	Store in X the number of neighbors of the current node.
GetNeighbor X,Y	Get neighbor X and store it in Y
Random X,Y	Selects a random number between 0 and X, and stores it in Y.
Send X	Transfer agent to node X.

Table 5.2: Genotype of a Simple Agent

Instruction	Operand 1	Operand 2	Translation
1	1	0	allocate memory location 1
4	1	0	1=mynode.getnumneighbors()
1	2	0	allocate memory location 2
7	1	2	2=rand(0,1)
3	2	2	2=mynode.getneighbor(2)
2	2	0	send agent to node 2
5	0	0	jump to line 0

These results, while interesting, are not groundbreaking. The very narrow scope of the agent language led naturally to the result that was observed. Put differently, the search space was very narrow. Thus more complexity was added to the language in the hopes of evolving an agent even better at this 'hopping' behavior. Table 5.3 lists the math operations, table 5.4 lists the conditional operations and utility operations, and table 5.5 lists the array operations that were added to the language at this point. These additions greatly increase the size of the language, and the performance the agents are capable of. It is notable that while the conditional and math operations continue to perform in a way similar to most assembly languages, the array operations begin to add complex behavior not often found in an assembly language. As before, an agent was written by hand for testing the functionality and to provide a performance baseline. It, on average, visited between 40 and 50 nodes.

Table 5.3: Initial Math Operations

Instruction	Description
Add X,Y	Adds integers X to Y and leaves the result on the adder
Sub X,Y	Same as add, but for subtraction
Mul X,Y	Multiplication Operator.
Div X,Y	Division operator.
Cmp X,Y	Performs comparison, setting flags on adder indicating relationship

Table 5.4: Initial Conditional and Utility Operations

Instruction	Description
JGT Line	Jump to line specified by number if adder greater than flag set.
JLT Line	Jump to line specified by number if adder less than flag set.
JEQ Line	Jump to line specified by number if adder equal flag set.
JNEQ Line	Jump to line specified by number if adder less than flag set.
Load X	Takes the value in X and pushes it onto the adder.
Unload X	Takes the value from the adder and puts it in X.
Incr X	Takes value stored in X and increases it by one.
Decr X	Takes value stored in X and decreases it by one.

Table 5.5: Initial Array Operations

Instruction	Description
New X	Creates new array at ID X; separate from variable memory.
AddVal X,Y	Adds value X to array Y
GetVal X,Y	Gets value at index X from array Y and stores on adder
Pop X	Removes last value added to array X, and puts it on adder
RemoveAt X,Y	Removes value at position X from array y
RemoveVal X,Y	Removes value X from array Y
Contains X,Y	If array Y contains value X, adder EQ condition set
Empty X	removes all values from array X.
Copy X,Y	Copies all values from array X into array Y.

Crossover was implemented, another initial population created, and evolution begun again. The method of crossover was taking the top 10%, creating 40% of the population from them, and then creating the other 50% of the population by mutation. After 5,000 generations of crossover and mutation, code was producd that perofrmed approximately as well as the handcrafted code. This evolution was done with random insertion points and randomly generated networks, and again the simulation time was 15 seconds. Note that peak perfromance was gained within approximately 200 generations, and it did not change much for the remaining iterations.

5.5 Results of Evolving Mobile Agents

At this point it was speculated that totally random networks and totally random insertion points were unfair, so experiments were performed. Each generation of this simulation consisted of 25 seconds of simulation time, or 25,000 tics, and populations consisted of 1,000 individuals. This time value is the value used for all remaining graphs in this section. Figure 5.2 illustrates the overall performance of the top 5 agents in each generation for the entire simulation in a nonrandom network and nonrandom insertion point. Figure 5.3 illustrates the performance of a random network with nonrandom insertion. The top 10% refers to the 100 individuals with the highest number of unique visits. Surprisingly, despite the assumption that nonrandom networks would perform better; this was not observed in the trials. At any rate, logically, random networks with nonrandom networks are more in line with the goals of this experiment; the evolution of an agent that can explore and learn the network dynamically, not the evolution of an agent that racially remembers a network.

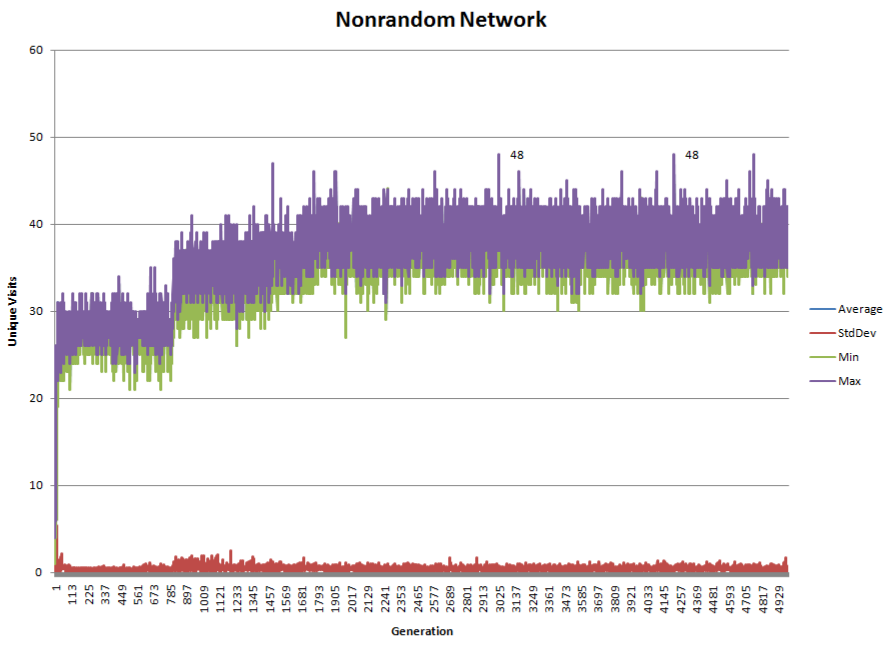

Figure 5.2: Results of evolution in a static network with static insertion points.

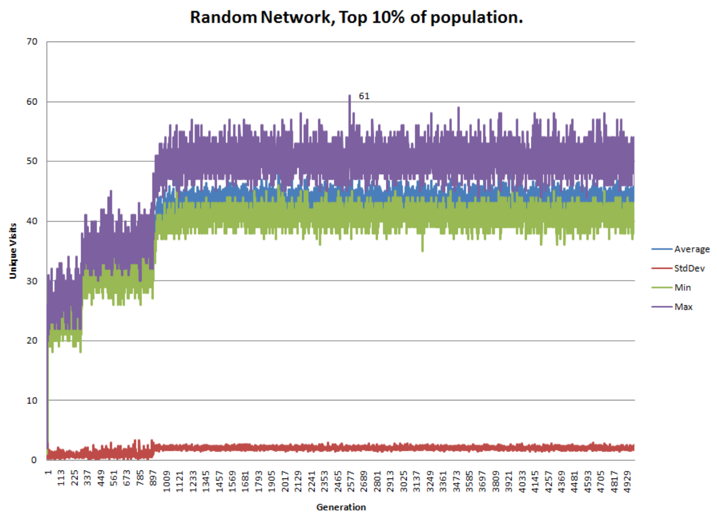

Figure 5.3: Results of evolution in a random network with static insertion points.

Figure 5.4 illustrates the overall results of an experiment with reinserting the handcrafted agent generation 0. It is evident from the graph that the resultant agent outperforms both by a significant margin. Figure 5.5 illustrates the result of the first 100 generations after insertion, and the rapid improvement. While it was evident from these results that the evolved agent outperformed IDA, averages were computed by running each agent through 2 separate trials of 25 second explorations of 10,000 random networks. Figure 5.6 shows the result of these trials, which were applied to the result of the simulations. On average; the agent evolved from reinserting IDA, known as IDA-ReInsert, outperformed IDA by about 7 values. IDARI's minimums were significantly higher, however. More disturbing, however, the result of purely random code is not competitive with IDA, this indicates that evolving the proper use of arrays is perhaps too difficult, or selection pressure is insufficiently high. Better results might be obtainable through the use of better selection methods, or with lower mutation percentages, as well.

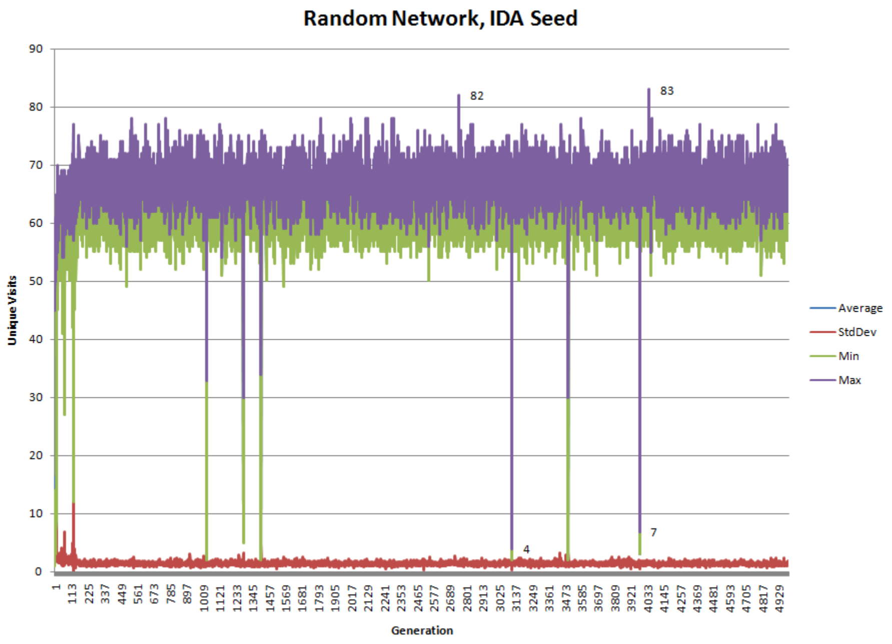

Figure 5.4: Overall results of repeated reinsertion of handcrafted agent into generation 0.

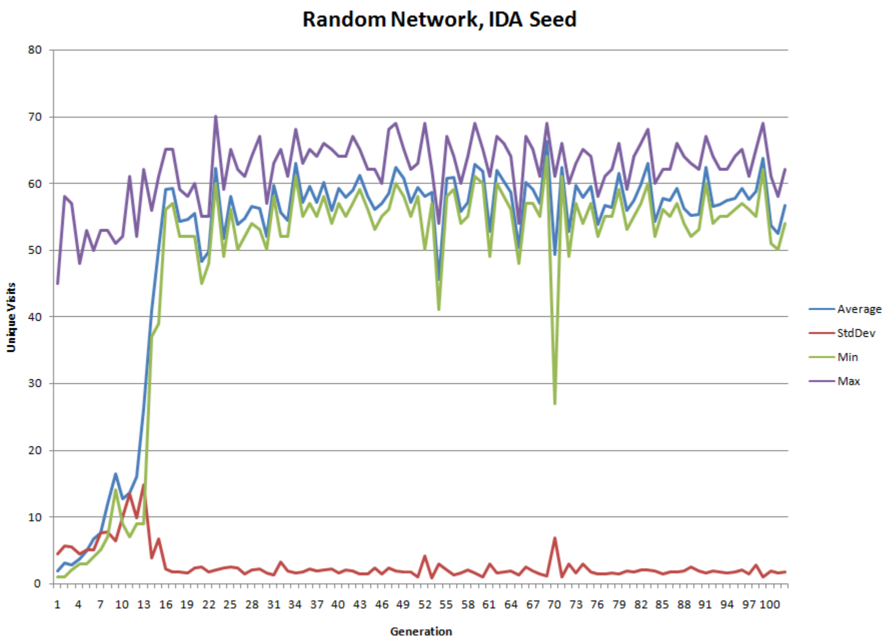

Figure 5.5: Graph illustrating very rapid improvment due to the reinsertion of a handcrafted solution.

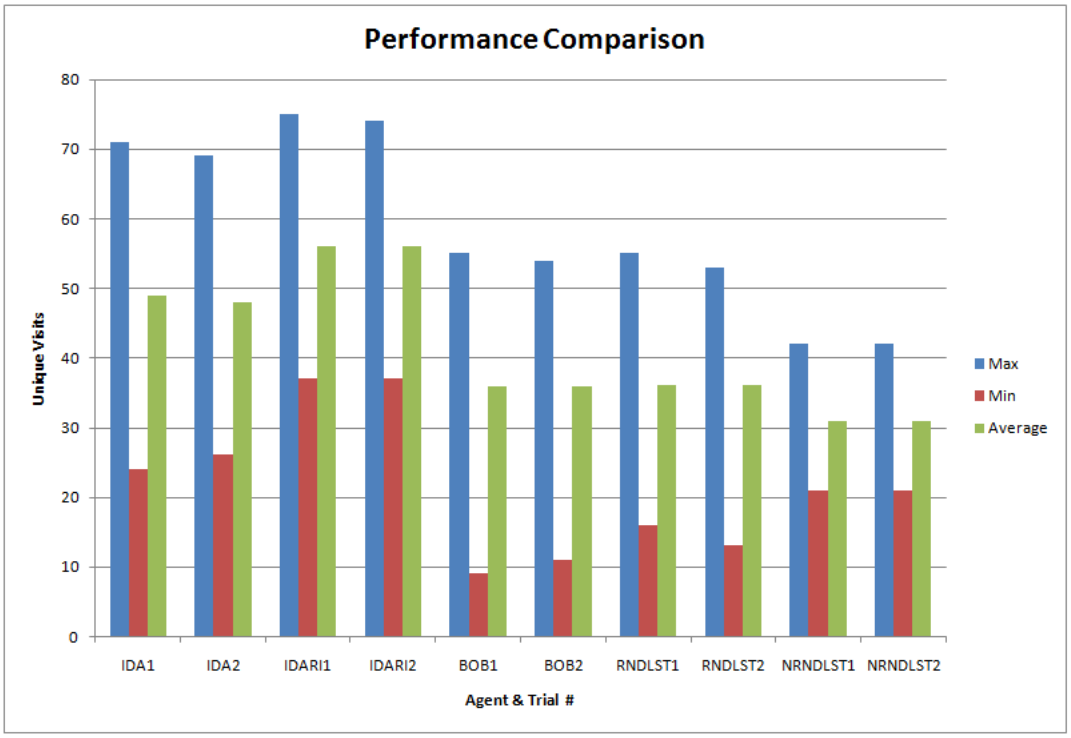

Figure 5.6: Performance comparison of all agents.

5.6 Genetic Mobile Agent Concerns

The results of this work are interesting, but do raise some concerns. While combining Genetic algorithms is clearly feasible, and shows promise, the fitness function is extremely difficult to define. For instance; how does one rank the various actions that are important; for virtually every agent, searching the network and dynamically learning it is an important behavior; but it is not the only behavior. It might be difficult to craft a fitness function to balance "hopping" with other behaviors. Additionally, some behaviors exhibit an all-or-nothing type metric; for instance, placing an order. How to distinguish between one agent and another, accurately, for ranking purposes?

Another, more serious concern, is not all agents are designed such that their performance can be measured in a timely manner. This is an issue due to the generational nature of genetic algorithms. One can easily envision agents that never return. For instance:

- A crawler that occasionally sends back updates, but continuously attempts to discover new resources.

- Autonomous Intrusion Detection Systems implemented as agents; only return or communicate on a success; the absence of which does not necessarily indicate poor performance on the agent's part.

- Autonomous agents in simulations that potentially run continuously, such as bot detection in online games or even opponents in online games or simulations.

Even for tasks that do not end, the network is by nature malleable. Thus, any attempt at evolving mobile agents is aiming towards a moving target. The moving target in this case is the optimal way to react to a network, an environment that changes. Thus, evolution must, in some sense, be continuous. In addition, it must be distributed; while for early experimentation a simulator is sufficient, ultimately for the end product, a distributed version that runs in real networks would be preferable; ergo it is safe to assume non-global knowledge as a constraint. This leads to a discussion of some variations on genetic algorithms as well as distributed genetic algorithms.

5.7 Considering Variations on Canonical Genetic Algorithms

The traditional approach to genetic algorithms, or the Canonical Genetic Algorithm, is discussed extensively in chapter 3.1. Indeed, the preceding section described in detail an implementation of a traditional genetic algorithm for the purposes of evolving mobile agents. It was found lacking. There are modifications to the traditional scheme of genetic algorithms that are worth considering for mobile agents: steady state genetic algorithms, continuous genetic algorithms, and self adapting mutation rates.

5.7.1 Steady State Genetic Algorithms

Steady state genetic algorithms generally have a fixed population size. Periodically an individual is selected by some criteria to be replaced by a new individual created by crossover of the surviving population. The criteria for selecting this individual varies, but often includes worst solution or fitness rating, or the oldest solution. The selection step, in this algorithm, is performed on crossover: several solutions are created by crossover, cloned, and then mutated, with the best being selected. This means that steady state genetic algorithms require the evaluation of many potential solutions at each instance of crossover. This is workable for solutions with a fixed or small evaluation time; in the case of mobile agents, they need to be continually re-evaluated, and cannot be evaluated in this way. That is to say, theoretically in simulation a prediction of their behavior could be performed in order to select the new individual to join the population. In practice, this would be somewhat impractical to truly evaluate the agents in a real-world environment, due to the nature of the delays involving with evaluating agent performance meaningful. Mobile agents simply do not have a fixed execution time. Put differently, it takes time to determine if a potential solution is viable with mobile agents.

5.7.2 Continuous Genetic Algorithms

A more promising modification of genetic algorithms are continuous genetic algorithms. Continuous genetic algorithms are designed to explore a 'moving' search space. That is, the 'goal' or fitness value is influenced by environmental factors, that are changing. These algorithms propose new crossover or mutation operators in order to in some sense avoid convergence. Convergence is the tendency of search algorithms such as genetic algorithms to converge towards an optimal solution by eliminating less optimal solution sets as it begins to fully search the problem space. That is to say, the nearer to convergence the algorithm is the more uniform the set of solutions become, and the less variation occurs. Thus, convergence can be negative when the fitness requirement can change radically; too specialzied a solution will be unable to quickly adapt to a changing environment. Some of these algorithms instead cause the algorithm to enter a state of extremely high mutation called hypermutation periodically to partially reset the solution set. These algorithms seem fairly appropriate for mobile agents, although they do not contain any special logic for distributing the process appropriately; and are not necessarily designed to be run forever; they are designed to produce an widely applicable solution instead of an optimal one. Additionally, the proposed algorithms do not address the need for decentralization.

5.7.3 Self-Adapting Mutation Rates

Lastly, some genetic algorithms have self-adapting mutation rates. This is not necessarily a great departure from canonical genetic algorithms, which sometimes have variable mutation rates. It can be, as some algorithms have proposed encoding the mutation rate as part of the solution. That is, the solution set contains a set of potential solutions, and each solution has its own individual mutation rate. Thus, the algorithm convergence both on an optimal mutation rate, and an optimal solution, simultaneously. This is an interesting idea, that might be applicable to genetic mobile agents. This process does not necessarily produce a solution that is adaptive; the focus is more on automatically selecting the appropriate mutation rate in order to speed up convergence. That is, the goal is the improvement of the performance of the algorithm; not necessarily of the solution. Although, solution quality is not compromised, it is simply not the goal of these algorithms to produce highly adaptive solutions like mobile agents calls for. Similarly to continuous genetic algorithms, they do not address the issues with the network environment working against centralized selection. This leads us to the discussion of distributed or decentralized genetic algorithms.

5.8 In Consideration of Distributed Genetic Algorithms

As in Canonical Genetic Algorithms, in Distributed Genetic Algorithms a population of potential solutions exists. In Distributed Genetic Algorithms, however, the population is distributed according to the system architecture, with migration between populations occurring periodically. Generally, one can think of each island as an independent canonical genetic algorithm. Obviously, knowledge of the system architecture is required to implement this. There is another approach for extremely fine grained architectures that consists of a series of overlapping neighborhoods that interact with their neighbors for selection, without migrating solutions. The fine grained approach is more of a depature from traditional genetic algorithm than the coarse-grained approach. These two approaches can be hybridized, in some sense, by having a coarse-grained collection on fine grained algorithms. There is also another type of distributed genetic algorithm, in which only the evaluation process is distributed, selection still being global. That is, candidate solutions are distributed for evaluation, and the fitness value is returned.

Coarse-grained genetic algorithms seem somewhat poorly suited due to a difficulty of deciding how to implement them on a network. While each node could be considered a system in the architecture, mobile agents would have such a high degree of mobility that the individual systems would be unable to observe much of their behavior; ergo they would be reliant upon the individuals themselves to tell them their behavior. It seems perhaps more logical to offload this behavior to the individuals, themselves, and allow them to self-rank.

Similarly, fine grained and hybrid approaches would require a great deal more network communication, though it would be possible through applying a topology to the network, and allowing each node to communicate with its neighbors when performing selection and ranking. Again, while the algorithm would have access to more observations of the agents, to truly have an idea of how they are performing would require relying upon them to report actions. So, again, we return to the idea of agents being self-ranking.

Lastly, distribution of the evaluation method and centralization of the selection method works counter to the whole idea of mobile agents; the removal of a central authority or client-server type mechanism. Additionally, since the agents are naturally already distributed in the medium (network) simply allowing themselves to self-rank makes a great deal of sense.

This leads naturally to a discussion of self-ranking agents. If agents were allowed to rank themselves, and thus decide what combinations of locally present potential solutions should be combined; then they could be said to be organisms. If treating them as organisms, we should consider the area of artificial life, which deals with the creation and evolution of artificial organisms in an electronic medium.

5.9 Considering Artificial Life for Integration

There are a wide variety of artificial life simulators, and an overview of the individual simulators and their common features is given in a previous chapter. Still it seems pertinent to consider their common features. They universally have some concept of a world, generally two dimensional. While many of these representations are grid-based, some are planar. In comparison, a network can be represented as a two dimensional graph, so it is not too much of as stretch to go from a two dimensional planar world or grid-based world to a graph based world.

All artificial life simulations have the concept of DNA. This DNA takes the form of a programming language, and is generally similar to programming languages defined for genetic programming; or indeed to that defined previously for the initial attempt at combining genetic algorithms to mobile agents. Thus again, this is an extremely painless matching.

In a departure from genetic algorithms, very few artificial life simulations actually include a fitness function. This is due to the focus in artificial life generally being to mimic natural evolution, which is assumed to be unguided. Or rather, guided only through environmental pressures, which by nature change. Ergo, artificial life can generally be said to ’set up rules’ and then allow genetic-algorithm like organisms to find novel ways of exploiting the rules in order to thrive. This sounds promising; given the nebulous nature of trying to define a fitness function for ’survival,’ and the general difficulty of defining a fitness function for mobile agents, period.

While mutation is practically universally present in artificial life, crossover is seldom defined. This goes back to trying to model natural evolution; it is thought that reproduction is a result of evolutionary pressure, not a mechanism of it. Ergo; an organism capable of reproducing itself sexually or otherwise is more fit than an organism incapable of it. This is the first situation in which artificial life might seem to be poorly suited to the evolution of mobile agents; although the first iteration of genetic mobile agents did not include crossover and was successful. Indeed, from the section of genetic algorithms early, crossover is not strictly necessary for evolution, it simply makes convergence quicker. Indeed, it would be easy enough to add crossover back to artificial life.

Lastly, artificial life simulations provide their "organisms" with various mechanisms for interacting with their environment and each other. This allows mechanisms such as parasitism and predation to evolve as the organisms exploit their fellows and their environment. Communication and cooperation are also possible. This is again a natural fit for mobile agents in the network; they often communicate with each other, interact with servers, and cooperate to perform a task more efficiently. Ergo, these are easily translatable. Additionally, in artificial life sometimes certain behaviors (such as killing each other) is encouraged by reward mechanisms such as "energy." Energy is a resource, and is generally required for some other behavior such as reproduction. This encourages the evolution of some sort of life-cycle or behavior cycle, by which the organisms performs some action to gain energy, then once it gains enough, reproduces. It is also possible to invert the logic, and 'remove' a resource, such as energy as punishment to discourage bad behavior, such as in one notable simulation in which organisms are encouraged to be 'efficient' in using their storage space due to extra space costing energy. This is again naturally translatable to mobile agents. operation are also possible. This is again a natural fit for mobile agents in the network; they often communicate with each other, interact with servers, and cooperate to perform a task more efficiently. Ergo, these are easily translatable. Additionally, in artificial life sometimes certain behaviors (such as killing each other) is encouraged by reward mechanisms such as energy. Energy is a resource, and is generally required for some other behavior such as reproduction. This encourages the evolution of some sort of life-cycle or behavior cycle, by which the organisms performs some action to gain energy, then once it gains enough, reproduces. It is also possible to invert the logic, and 'remove' a resource, such as energy as punishment to discourage bad behavior, such as in one notable simulation in which organisms are encouraged to be 'efficient' in using their storage space due to extra space costing energy. This is again naturally translatable to mobile agents.

In conclusion, while pure genetic algorithms seem to cause issues when combined with mobile agents; mobile agents as organisms in artificial life simulations seems like a natural match. Unfortunately, while in the strictest sense artificial life can be considered a type of evolutionary algorithm; generally they are rather aimless. It is this that necessitates the combination with some of the 'goal oriented' nature of genetic algorithms with the different approach of artificial life in order to more efficiently evolve agents. This brings us to the purpose of this document: DNAgents.

Chapter 6

DNAGENTS 2.0: AGENTS AS ORGANISMS

The previous chapter discussed the motivation for and benefit of combining genetic algorithms. It also discussed at length DNAgents 1.0, in particular the difficulties that arose in implementing it. The chapter went on to discuss various modifications of genetic algorithms that were considered for inclusion into DNAgents, but ultimately found lacking. It ended with a discussion of artificial life, which again, while promising, was not quite sufficient in itself. This chapter begins with a further elaboration of the motivation behind DNAgents 2.0 in section 6.1. While the motivation for combining genetic algorithms and mobile agents has been discussed at length, this is not the only motivation for this direction. In some sense, DNAgents 2.0 is motivated by "What may be" instead of purely a reaction to "what is." It is this potential that section 6.1 attempts to show. This is followed by an overview of DNAgents 2.0 in section 6.2. The various new mechanisms introduced by this new framework are discussed in sections 6.3 and 6.4. Experiments are proposed in section 6.5, with an actual experiments implementation and its impact upon the algorithm being discussed in section 6.6. Finally, the results of this experiment are discussed in section 6.7.

6.1 Algorithm Motivation

It is becoming increasingly difficult to safeguard the network against intrusion and attack. It is possible that a sort of pseudo-lifeform might be created to reside in the network, and protect its environment. This perhaps sounds a bit like science fiction, but this pseudo-organism might take the form of a mobile agent capable of evolution. Specifically:

- A Strain of agents exist within the network.
- This strain is capable of reproduction, and evolution.
  - Attempts to discover new exploits and viruses through evolution.
  - Disseminates this information through both communication and genetics to its fellows.

    - Attempts to attack and destroy any examples of the virus or exploit it discovers, continuously.

- The most important part of this potential strain of agents is that it is omnipresent, and continuous.

It might be instinctual to disregard this as a 'neat' idea but otherwise fantastic and unbelievable, and furthermore unnecessary. This is perhaps true, at the current time. On the other hand, if the potential for the creation of this sort of artificial immune system is remotely possible, then certainly its dark counterpart is also possible. What if, through maliciousness or greed or some misplaced emotion, someone were to create a strain of entities that existed solely to discover and exploit these hacks; and to avoid detection. The chief means of detecting viruses and exploits currently is the identification of non variant substrings that comprise whatever specific security loophole it relies upon. In the nature of natural selection, these evolving exploits would not necessarily need to hang on to old exploits, and indeed, keeping only the most current exploit would be a survival mechanism against detection. It would be difficult to defend against this threat with conventional methods, if it is indeed possible; thus it is worth investigating the benign counterpart.

For this sort of task traditional genetic algorithms are a bit lacking due to their generational nature and artificially defined fitness function. For this purpose and others, a biologically inspired framework that contains much of genetic algorithms, but not all, is proposed in the following section.

While there has been a great deal of work on evolutionary programming, there has been relatively little work with regards to applying evolutionary programming specifically to the area of mobile agents. The preceding sections have talked about attempts to combine these as well as the underlying issues. This leads to the conclusion that this relative scarcity is due in part to the complexity of defining a fitness function for mobile agents, and partly due to the decentralized nature of mobile agents and their "black hat" counterpart, viruses. In fact, due to the fact that some never report back, it is difficult or possibly impossible for a central authority to gauge their fitness. It can be difficult to even define a fitness metric for these incredibly complex applications.

Consider a mobile agent, and what is desirable for it. Certainly there is a task, which we assume to be defined by the user, and thus outside the realm of our evolutionary algorithm, for the moment. In addition to this, there is the concept of managing an itinerary: which nodes to visit, how to visit them, how to stay alive, what happens if a failure occurs. How do these various sub goals factor into the definition of a fitness function? It certainly defines a huge search space. Assuming that a reliable fitness function is derived for these; they would have to keep a record, in some way, of their fitness. This could be accomplished by computing their fitness function themselves or by carrying a log of information back to the central server. Furthermore, it is necessary for the agents to return home so that selection and crossover can occur. Due to the vagaries of network traffic in the Internet, the obvious application space for a mobile agent, it is difficult to ensure that all agents will return. Successful agents might not even return, dying after their task is accomplished. Some tasks that do not require the agent to return at all, such as network monitoring or intrusion detection. Lastly, assuming the adequate definition of a fitness function, which is very non-trivial, and the insurance that enough of a test population returns in a timely enough manner to facilitate useful reproduction, the task itself does have an impact upon the fitness function. This is a very difficult problem; staggering even; although for some 'base cases' it might be more tenantable, such as network exploration and discovery.

6.2 Algorithm Overview

This work proposes a novel method of applying evolutionary programming to mobile code. The proposed method mirrors, in some sense, the broader scope of artificial life simulation, rather than the focused area of genetic algorithms. It seems that due to the mobile aspects of mobile agents, they can be thought of as living beings. These beings live within a world of machines. Thus, it seems logical to allow them to evolve within this world directly, in a completely decentralized manner. Table 6.1 lists the major components of the proposed algorithm.

Table 6.1: Algorithm Components

Component	**Description**
Selection	How and when mating occurs
Senescence	The aging process
Revitalization	Reward mechanism, counters effects of senescence.
Death	Death of individuals

The broad idea is, in summary, to allow individuals to semi-freely interact with their environment, and reproduce. All individuals age naturally, over time. This aging process can be represented as an increasing chance of failure, or death. Death can be postponed by doing beneficial actions, as defined by the user. The definition of 'beneficial' and 'detrimental' actions takes the place of the traditional fitness function, and should be easier to define for complicated problems such as mobile agents. Death generally occurs eventually, although it could be possible to allow for a biologically immortal organism. Additionally, death can occur as a function of the environment; accidental death, as it were. The four stages listed in 6.1 are discussed in sections 6.3 and 6.4.

6.2.1 Possible Applications

This algorithm is extremely applicable to concepts such as mobile agents, especially for applications involving virus or intrusion detection. Currently, when applying evolutionary programming or neural networks to a problem such as intrusion detection, the monitor must be trained to the network. At some point, training ends, and the monitor must be retrained occasionally to maintain security. Theoretically, if successful, an algorithm like this need not end; the population will shift over time to encounter new threats. Individuals could be trained separately, and introduced into the environment to influence the whole population by interbreeding.

6.3 Selection

In genetic algorithms the algorithm itself defines criteria for mate selection, usually universally among a pool of all individuals, or within a predefined local community. This process takes place between generations, all at once. The translation of this stage into a process that occurs continually is proposed, allowing reproduction and mate selection to occur on an ad hoc basis. It can either managed by the individuals themselves, or by the algorithm. There are several potential methods by which the individuals could select potential mates, and a non-exhaustive list is presented in table 6.2

Table 6.2: Potential Selection Criteria

Criterion	**Description**
Difference	How different or similar are the two mates.
Random	Select completely at random, or probabilistically.
Age	How old is the individual
Fitness	How successful has the individual been.
Energy Based	"Energy" is required for reproduction.
Dynamic	A mixture of the above methods.

Selecting based on difference or similarity could be used to encourage or discourage diversity in the population. This can be calculated by checking for similar instructions between the two individuals. Theoretically, the offspring of similar individuals would gain little from mating. Random selection refers to either probabilistic or completely random selection, upon encountering an individual there is a preset probability that mating will occur. Age based selection assumes, to an extent, a hostile environment. Additionally, with the inclusion of the concept of senescence, as defined below, the older an individual is the more fit they are. Fitness implies that the individuals could share their respective history; in the case of viral strains this could be a list of exploits found. The fitness method assumes that some metric for measuring success exists, provided outside the direct context of this algorithm, and is very similar to the fitness function of traditional genetic algorithms. Dynamic selection is included to indicate that these are not necessarily all or nothing selection schemes, and can be freely combined as necessary or desired.

Completely unregulated reproduction could be problematic, so it might be necessary to introduce additional control features. For instance, population could be maintained to a fixed size by restricting each individual to exactly one mating. In the case of restricted mating, however, some mechanism must exist to maintain population equilibrium. It might be more beneficial to define a territoriality metric that restricts population to network size; individuals choose not to interbreed while they are encountering other individuals too often.

Crossover and mutation can and should function similarly to the way in which they function in genetic algorithms. Any crossover or mutation metric can be applied. Technically, this proposes the use of genetic programming, although no necessary distinction is made between linear or tree-based genetic programming. With some modification, any crossover scheme should be applicable; but if genetic programming style representations is used, then of course some methods are excluded. This is covered in the pertinent section on genetic algorithms.

6.4 Senescence, Revitalization, and Death

This section discusses the processes of senescence, revitalization, and death as they relate to DNAgents 2.0. Senescence is the mechanism through which agents decay, while revitalization is the inverse of this process. Ultimately senescence leads to death, although sufficient revitalization might postpone this indefinitely. Each process is discussed in greater detail in its own subsection.

6.4.1 Senescence

The word senescence refers to biological processes that take place in organisms as they age. For the purposes of this algorithm, this takes the form of an increasing probability to die naturally. Optionally, a flat value can be used, and allowed to tick up or down, death occurring when this value reaches zero. The testing of both possibilities could yield interesting results. The probability of death, however, bears a closer parallel to natural biology. The older an organism becomes, generally the greater the chance of some failure occurring. This does not necessarily correspond to a certainty, however. In fact, some research indicates that senescence effectively plateau's for some species, at certain ages.

The minimum and maximum values for senescence could have an interesting impact upon the algorithm, and testing will have to be performed to derive optimal values. If senescence were allowed to decrease to a value that is, or is effectively zero, then an organism could, through success, become biologically immortal. Biologically immortal beings do not age, and unless they encounter trauma, do not die.

6.4.2 Revitalization

Revitalization is the concept that through some positive action, an individual is rewarded. This could be in the form of extended life expectancy, by reducing their effective senescence. While this has no real biological parallel, it is a useful tool for encouraging certain behaviors. Through the process of revitalization, more fit solutions will survive longer, and thus interact more. Their offspring will likewise encompass a larger percentage of the population pool. Through this process interaction, the algorithm will move towards the fittest solution. This concept can be inverted, such that individuals are punished for negative action, and thus age faster.

6.4.3 Death

Death is the cessation of biological function. If essentially uncontrolled reproduction occurs, some mechanism must exist to control this process. Artificial life simulations often rely on the concept of limited food. Since introducing the concept of senescence, individuals will expire eventually. Additionally, the environment itself could be hostile, leading to even more population control. As a last ditch effort to manage population levels, a food type resource could be abstracted as a ratio between network size and population size.

6.5 Proposed Experiments

Previously, the Mobile Agent Simulator was used to evolve, using a traditional genetic algorithm, a hopping behavior. This algorithm is discussed previously in section 5.4, as it was a major part of what lead to this algorithm. Regardless, it performed well enough. An obvious first experiment would be to apply a version of the proposed algorithm to the same problem, with the goal being to at least approach similar performance, to prove the viability of the assumption that senescence and death will lead to a more fit population over time.

Once a base line is established, experiments can be made with the various selection criteria described in table 6.2, as well as with the reproduction control methods discussed above. This will yield comparative results about the performance of the various operators. In a similar fashion, accumulating senescence can be compared to probabilistic senescence, as well as different methods of population control. If done correctly, it should be trivial to turn these various features on and off, so that comparative data can be collected for future experiments.

It might be interesting to define operations relating to reproduction selection, to allow the individuals themselves input into how they reproduce. If practical, it might allow for the further relaxation of the selection mechanic. Population control is actually a bigger problem, and it is difficult to envision a way to encourage individuals to control their own population. Although, it might be possible to define a reward behavior based around minimizing the amount of competition, assuming a mechanism for individuals killing each other.

Once the behaviors have been played with in a known situation, it would be useful to attempt the evolution of a detection behavior. For this experiment, the individuals are searching for suspicious behavior on the network, and this can be represented as a probabilistic chance to detect a compromised platform. Once a compromised platform is detected, it could be represented as variable, such that if that exploit is in place on another machine, any agent with that variable could detect it with much greater certainty. This experiment should also reward the hopping behavior discussed above, since wide dissemination is also a desirable trait for an intrusion detection system. Interesting mechanics that could be introduced in this simulation include the ability to trade information among each other, for the purpose of disseminating detection behavior as far as possible. This should have an impact on reproduction behavior as well. To test the interplay of the detection and propagation behaviors, random exploits could be introduced to random nodes in the network, either permanently, or with short duration to increase the difficulty of detection. The simulation itself could record the number of ’successful’ breaches, while analysis of the agents would yield values for the number of detected breaches. Additionally, a reporting mechanism could be employed, which introduces a third behavior to the simulation: reporting. Thus, three levels of success could be defined: detection, dissemination, and reporting. It might be useful to treat this as three different experiments.

A further possible refinement of the detection experiment might be to introduce hostile agents similar in nature to our agents, but as a separate non-interbreeding population. This second population would attempt to exploit platforms, while the benign agents would attempt to detect this exploitation, and possibly combat them. Operations might be defined for the detection of hostile agents, the spoofing of agents, and attack and defend operations. That is, the hostile agents would attempt to discover and exploit security loopholes while avoiding death, while the benign agents would attempt to detect exploits, the agents that are doing the exploitation, and attack and destroy them. Again, both populations could easily be tracked in the simulation, as well as their interactions. Again, it might be useful to treat this as three separate experiments: a 'white hat' detection scheme with the attacker's abstracted out and not actually existing, a 'black hat' experiment with the detectors being abstract or possibly nonexistent, and lastly a simulation involving two separate strains of genetic beings. Preventing the cross-pollination of these populations, particularly if the attackers can spoof the defenders, might yield interesting results.

A potentially better method of exploring attack procedures might be to allow the agents and or viruses access to each others memory, assuming they have exploit-level privileges on the system. The underlying assumption here is that for a normal program, executing in memory, the operating system will attempt to prevent it from modifying the memory and instructions of other programs. This assumption leads to the requirement of exploit-level privileges, which is simply an abstraction for whatever process is required to gain access to other programs on the stack. At any rate, modification of other programs memory and the gaining of information about that programs memory could be defined as a series of operations. These operations could be designed to fail with some probability; obviously modifying memory is a risky operation and could lead to several outcomes. This can be experimented with both ways, without any possible failure of the attack mechanism, and with failure. Failure could take the form of modifying one's own memory by mistake, causing the operating system to kill your process, or causing the operating system itself to fail.

Future research could attempt to encourage the formation of communities of individuals, although this is somewhat outside the scope of this work.

6.6 Experimental Implementation: Evolving a Hopping Behavior

As stated above, Mobile Agent Simulator, MAS, provided the simulation environment. On top of this environment an implementation of the proposed algorithm was created. Currently, every 1000 ms of simulation time, each agent pair currently on each node is considered for reproduction by any active reproduction mechanisms. An additional mechanism, called cloning, was introduced to prevent premature die offs. Additionally, there is no mechanism in the algorithm for controlling population at a fixed rate; the death mechanic does serve to provide selection pressure and some means of population control; it does not restrict the number of individuals alive at one time to a fixed value. For this reason, a population capping mechanism was added. Speaking of death, both tic-based and probability based death were implemented and tested; although the focus was on probability based. Mutation is not limited to crossover behaviors, and can happen periodically to existing agents. Also experimentations were performed with allowing mutation to decrease over time, as agents are rewarded, although there is a min and max chance of mutation. If varied, offspring and clones inherit the average of their parents mutation chance; ergo in the later stages there is a low rate of mutation. The rest of this section is organized as follows: First a discussion of the additions of the cloning and population control mechanisms, which were unforeseen but necessary. This is followed by a discussion of the actual implementation of the Reproduction, death, and reward mechanics.

It is beneficial at this point to introduce some basic terminology at this point. Generally constants or parameters in formula are represented by characters from the latin alphabet. Variables that are measured by the simulation are represented by lower case greek letters. Upper case greek letters are reserved for the formula defining the major componets of the algorithm, and are referred to in the glossary as functions. Additionally, often variables and functions are defined in terms of an agent or agency, which are represented by the subscripts x or n, respectively. More formally $x \in \{1 \ldots X\}$ where X denotes the total number of agents. Similarly $n \in \{1 \ldots N\}$ where N denotes the total number of agencies. The various constants, variables, and functions are defined as encounted in the text.

6.6.1 Cloning

Initial experimentation lead frequently to unexpected die offs. This is partly due to the fitness/reward mechanic used; agents were rewarded for discovering new nodes. Once an agent evolved code capable of migration, generally they ended up isolated, and died alone in the middle of nowhere. Rarely, an agent came back to civilization through chance, and spread his "optimal" subprogram, leading to a mass exodus. This exodus was usually followed by a golden age of evolution. Rarely, this exodus lead to a mass extinction because the majority of good agents ended up alone and ran out of life due to senescence. Once this occurrence was observed; a cloning mechanism was added to the algorithm to preserve optimal subprograms that get isolated from the "swarm." More formally: If an agent finds itself alone, with no compatible agents to cross-breed with, it can clone itself.

Cloning was initially implemented as a carte-blanch right; each agent is always capable of self-cloning if it finds itself alone. This lead to many many copies of the first successful agent, and greatly decreased diversity in the early simulation. The mechanic was later changed to a one time action: each agent is capable of reproducing itself by cloning once and only once. This mechanism lead to the desired effect; the clone is capable of copying itself also, ergo even in isolation, a successful 'strain' of agents can persist indefinitely in isolation without succumbing to the death mechanism. Note that this does not prevent agents from becoming subject to foul play, and at this state of the simulation failure was not turned on; ergo in networks with high failure rates, a greater degree of cloning might perform better. Indeed, similar to mutation, cloning might eventually be varied according to an agents obseverations of the network; in great isolation an increased degree of cloning might be desirable; which would lead to recombination of the solution that lead to the isolationism, and perhaps improve the solution. This needs to be studied further.

Another conceived attempt to fix the die off problem was setting 'successful' agents as immortal until they had reproduced. This was experimented with as both a toggle-based system where a reward toggled immortality until reproduction occurred, and a increment/decrement system where multiple rewards could stack immortality counters and agents could not die until they ran out of counters by reproducing. Cloning performed better, although this does not necessarily mean these approaches are invalid and could not be fruitful with further investigation; early implementation simply did not indicate that they were. Lastly, it is worth noting that although cloning was introduced to solve the problem of mass-extinctions, sometimes the average-performance goes from better to worse. That is to say, it fluctuates, and this could be because of isolated die offs of promising strains. It could also be caused by the nature of the changing fitness landscape. It is not even necessarily a bad thing; a adaptive algorithm is prized over an optimal one.

6.6.2 Population Control

A population control mechanism was overlooked in designing the algorithm. While in simulation, it is easy to restrict the number of agents to a given number. In the field, this will be impossible because no one node is going to be privy to the number of agents in existence. Two obvious mechanisms for controlling population in a distributed manner are capping the number of agents on each agency at a given time and preventing reproduction if the average number of agents seen is too high. In both cases, the theoretical 'desired' agents on a per-agency basis is given by

$$v_n = \frac{d}{N}$$

where d represents the desired number of agents in the population.

Initially had decent results with capping the agents in existence on each agency by preventing agents from reproducing if there are v_n or more agents on a given agency. This works fairly well for the early to middle simulation.

Eventually this mechanism breaks down and population grows prohibitively large. One possible reason for this is that the "edges" of the graph, i.e. poorly connected nodes, my act as spawning areas, the results of which migrate inward and spend most of their time in well-connected nodes. That is, agents are continually created in sparse areas, and then move to more populated areas, causing a population imbalance.

Alternatively; it could be because the agents have no way of knowing if agents are inbound to a node when they decide to reproduce. That is to say, if the majority of agents are in transit, agents that are not in transit might decide to reproduce when they really should not. This is complicated by the selection pressure of this experiment, which encourages exploration, and the fact that agents cannot choose to suicide while in transit. That is, agents are encouraged to develop algorithms that spend the majority of their time in transit, which complicates culling in a way that normally would not exist.

The other method of capping reproduction is agent based. Each agent keeps track the average number of agents seen, and only reproduces if the average falls below the threshold, which is calculated currently in the same way as v_n. The average agents seen is calculated by

$$\varsigma_x = \frac{\sigma_x}{\phi_x}$$

where σ_x represents the total number of agents encountered by x, and ϕ_x represents the total number of agencies visited by x. This mechanism performs better than the other mechanism; since agents are not as likely to spend enough time visiting the sparse nodes for their average agents seen value to fall below v_n. It does not eliminate this phenomena, nor does it address the issue of agents in transit. More work is required to find a satisfactory way of managing the total number of agents. This method does perform better, on average, than the first mechanism.

Both population control mechanisms generally stabilize at some point. Figure 6.1 illustrates a fairly typical execution for the Average Agents Seen population control mechanism. Figure 6.2 illustrates the major problem with the Agency-based population cap mechanism. These are fairly typical examples of their performance; while the Agency based mechanism stabilizes for awhile, becoming unstable in the later simulation. The Average Agents Seen mechanism occasionally explodes, but generally re-stabilizes. By contrast, the agency based population cap has not been observed to re-stabilize after a population explosion. As said above, more experimentation into the reasons behind this, and investigation into better population control mechanisms is needed.

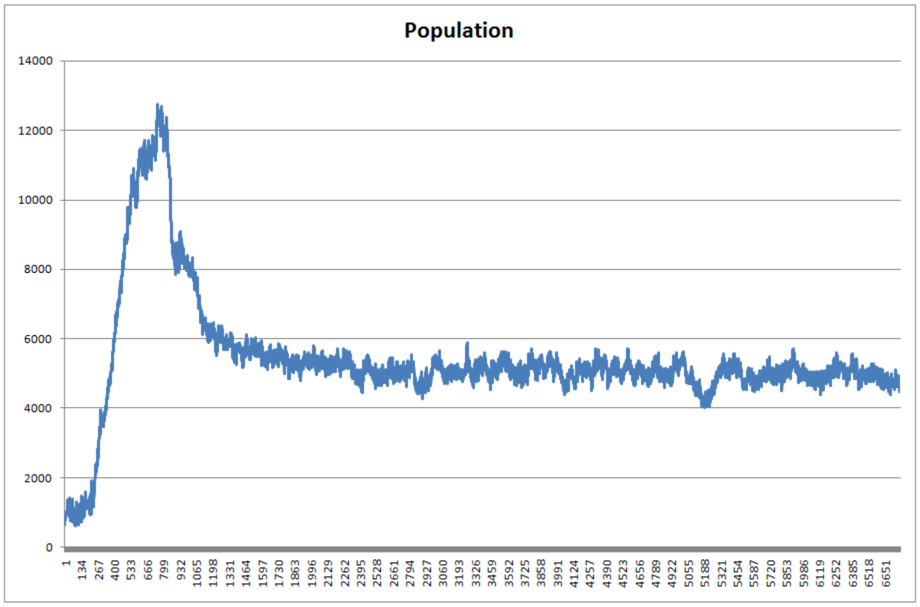

Figure 6.1: Performance of Average Agents Seen (AAS) population control mechanism.

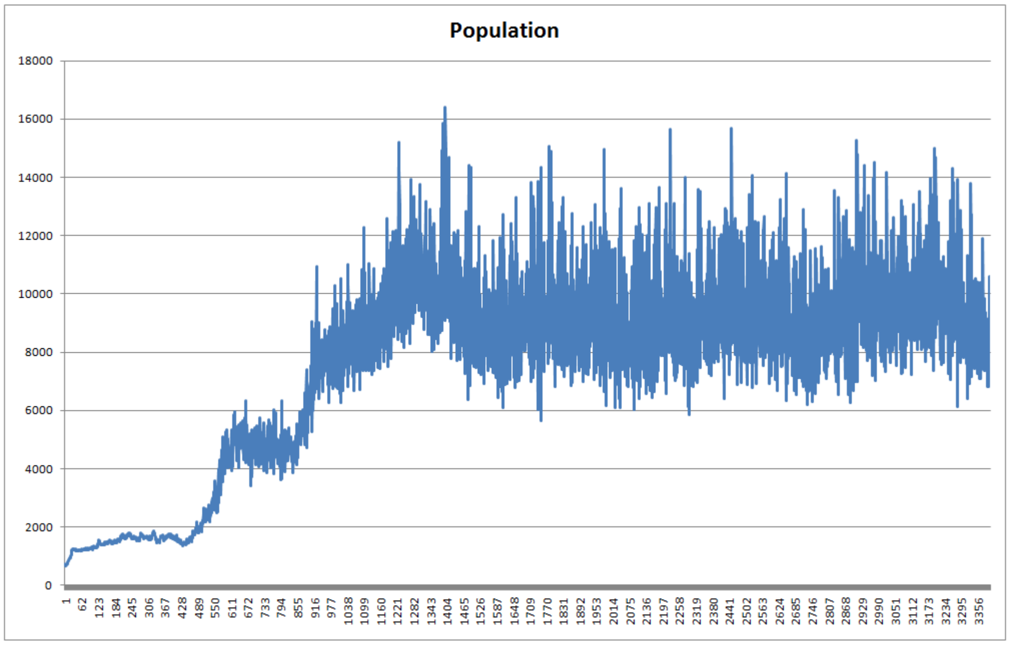

Figure 6.2: Performance of Agents Per Agency (APA) population control mechanism.

6.6.3 Reproduction

While in section 6.2 it was speculated that the agents could evolve the reproduction choices; at this point the choice-by-algorithm aspect has been investigated. Similarly, many potential mechanics for reproduction were proposed, but the full comparison of all these mechanics is beyond the scope of this work. Random reproduction was implemented to provide a baseline for determining the actual influence of reproduction on the simulation. Age-based reproduction was picked, somewhat arbitrarily, as another mechanic to implement and experiment. Lastly, while this section is primarily concerned with the reproduction selection process, it is worth mentioning that in this experiment crossover was implemented as single-point crossover. Also, the population control mechanism supersedes reproduction currently; the decision making process for reproduction is only entered in to if the population control mechanism authorizes it. Both types of reproduction occur every 1,000 tics of simulation time.

Random based reproduction, perhaps more accurately called pseudo-random reproduction, is performed based on a flat chance. That is, each agent, if eligible for reproduction based on its environment, has a percent chance to reproduce. This percent chance is input as a parameter by the user. As expected, the combination of the death and reward mechanism proved sufficient for evolution to occur even without guided reproductive selection. Also expected, the performance wasn't exemplary.

Age-based reproduction was something of a challenge to implement due to the restriction against global knowledge. First, a given agents age is given by α_x, while the oldest local agent is given by γ_x. The actual formula to determine if they reproduce is defined in terms of an agent, x and its potential partner, y, and is given by:

$$\Delta_{x,y} = (\alpha_x/\gamma_x > t) \wedge (\alpha_y/\gamma_x > t)$$

The constant t is a constant threshold value. It is trivially observable that using the oldest global age instead of γ_x would produce slightly better results; this implementation still produces workable results without violating the distributed requirement. Alternatively, using the oldest observed age was considered, but disregarded due to poor initial performance. The superiority of inferiority of this approach was not quantitatively proven one way or the other, this the selection of this formula to represent age is somewhat arbitrary.

6.6.4 Death

The primary mechanic investigated so far is probability based death. While tic-based death was implemented and limited experimentation carried out, initial results leaned towards probabilistic based death. This is in keeping with the assumption that probabilistic based death would increase the churn of the problem space, ultimately aiding evolution. Tic-based death was implemented pretty much as described, with a flat, parameter-based number of tics being gifted to an agent upon birth, and this number can be changed due to rewards and punishments. Probabilistic based death is controlled by a parameter that is the minimum chance for death. This probability increases each time the agent is considered for death, and survives. Formally, the probability of death is given by:

$$\Omega_x = m + s\alpha_x - \sum \lambda_x$$

The constant m is the minimum chance for death, while the constant s is the relationship between age and senesence. The variable α_x is the agents age, and λ_x represents a given reward for an agent x. The formula for λ_x is given in the following section. The number of tics between death evaluations is the same as the number of tics between chances for reproduction, 1,000. Currently the constant s is equivalent to $\frac{1}{10{,}000}$, which indicates how often the chance of death is incremented; in effect $\frac{1}{10}$ of a percent each 1,000 tics. Death occurs after reproduction and growth so it would be possible to reproduce and expire in the same 'tic.'

6.6.5 Rewards

Due to the goal of the experiment: evolving a hopping behavior, the reward mechanism revolves around unique-visits. This is the same goal as the initial experimentation with traditional genetic algorithms. The attempt was made to emphasize time by making the reward a percent based on performance instead of a flat reduction, although per the design specifications the chance cannot go lower than the minimum chance to die. The formula for agent rewards is given by:

$$\lambda_x = r\frac{\upsilon_x}{\alpha_x}$$

The constant r refers to the user parameter reward increment, while υ_x is the number of inque visits attained by the agent. α_x is as always the age of the agent. The constant r is a user parameter, and indicates a ballpark figure for the size of the reduction in death chance. Thus, agents are rewarded based on the ratio of success's to their age; older agents are rewarded less for the same work, while younger agents are rewarded more.

6.7 Results of Hopping Experiment

With a functional, parameterized implementation in place, the next step is to find baseline values for those parameters that perform well. Table 6.3 lists and defines the parameters tested. Mutation chance for all the tests started at 0.1%. Age Threshold (t) was allowed to vary between 0.5 and 0.9. Min Senescence (*m*) was allowed to vary between 10 and -1000. Reward Size (*r*) was allowed to vary between 5 and 1000. Each test was ran for 1000 12,000 tic intervals. Values were reported and saved for each interval. The main performance evaluation for performance were Average Unique Visits and Average Unique Visits per Tic.

Table 6.3: Algorithm Parameters

Parameter	**Description**
Age Treshold	*t* as defined in section 6.6.3.
Min Senescence	Minimum and Starting Senescence (*m*).
Reward Size	*r* as defined in section 6.6.5.
Mutation Variation	If mutation is allowed to vary.

6.7.1 Best Average Unique Visits

The parameters which yielded the best Average Unique Visits were an age threshold of 0.7, 0 Min Senescence, a Reward Size of 1000, and non variable mutation. Figure 6.3 displays the population changes for this test. After a period of initial churn, most agents were living to fruition, and thus there were not a great deal of deaths or births. Figure 6.4 shows number of unique visits at each interval. The fact that the max was always 100 is not terribly significant. Unlike the genetic algorithm test, some agent is always around who has eventually managed to explore the whole network. The average and min visits are much more telling. Also, the standard deviation would seem to indicate that it was still moving towards an optimal solution when stopped. Figure 6.5 shows similar values for the unique visits per tic. In this graph, all the values stay fairly stable, which indicates that although they were visiting many nodes, they were not improving the speed at which they did it. When taken together with the previous graph, it would indicate an increasingly thorough algorithm; as evidenced by the climb of the min visits. Or, perhaps, an algorithm increasingly resistant to churn. Finally, figure 6.6 shows the average, min, and max ages for agents at each interval. As you can see, the average age climbs for awhile, and eventually stabilizes. The max age is quite high, which again would indicate that these agents are ’slow and steady’ as opposed to quick.

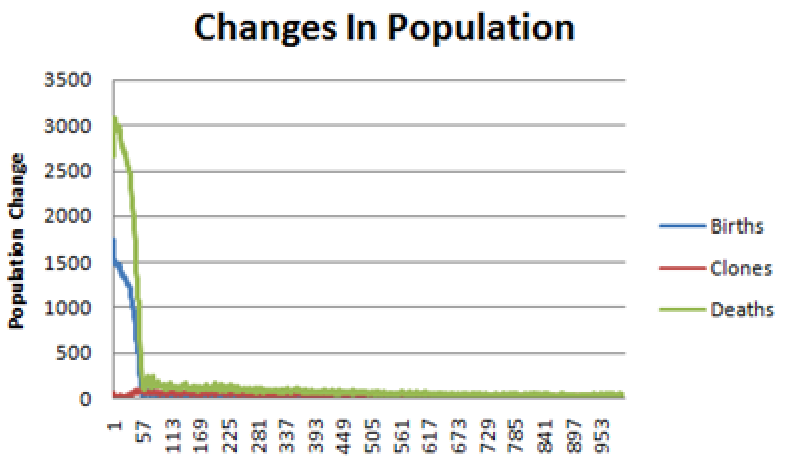

Figure 6.3: Population changes for best average unique visit test.

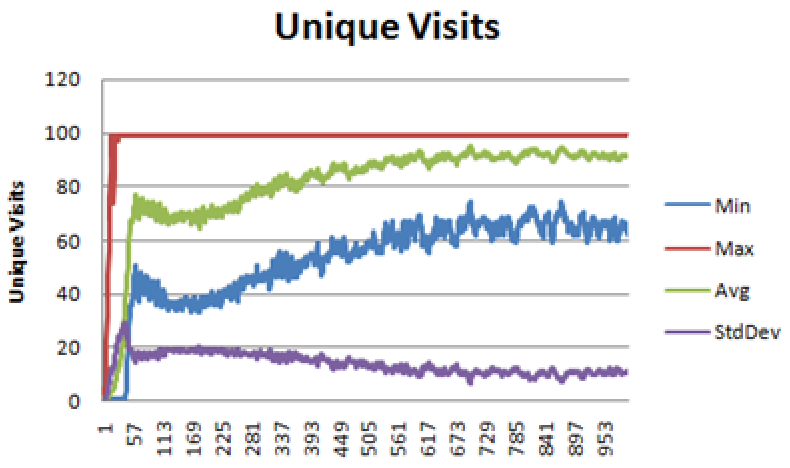

Figure 6.4: Unique visit values for best average unique visit test.

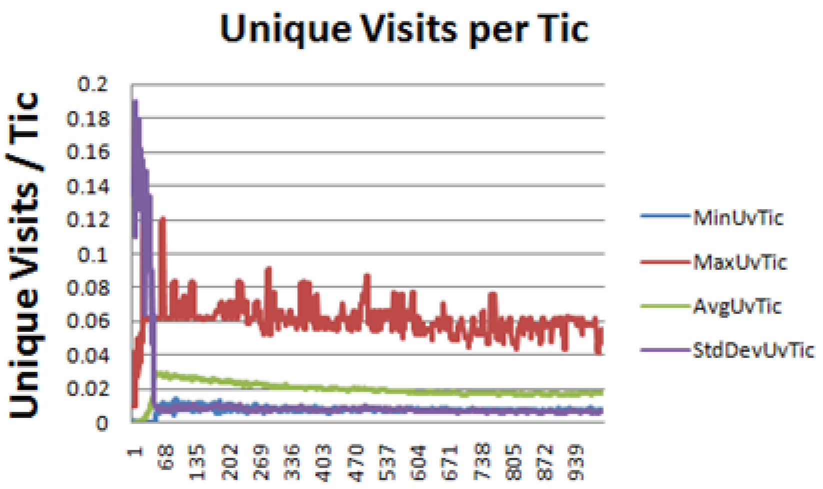

Figure 6.5: Unique visits/tic values for best average unique visit test.

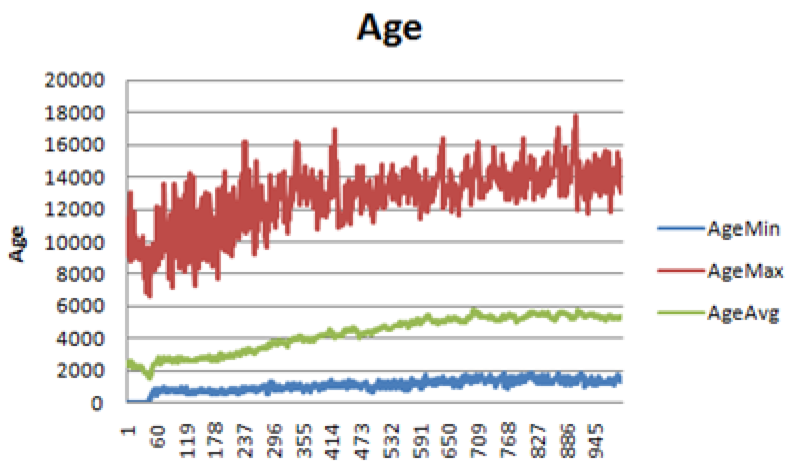

Figure 6.6: Agent age values for best average unique visit test.

6.7.2 Best Average Unique Visits per Tic

The parameters which yielded the best Unique Visits per Tic were an age threshold of 0.5, 0 minimum chance for death, and a reward size of 15. This test also allowed for variable mutation. Figure 6.7 shows the population changes over the course of this test, and again, after an initial period of churn things stabilized fairly nicely; with a fairly constant rate of cloning throughout the simulation, births and deaths dropped off to very low levels. This makes sense, again, as it implies that the agents in the simulation are leading fairly 'full' lives, ergo, there is a low amount of churn from death. The primary difference between this simulation and the previous one, is that there is slightly more churn in this one.

This is probably due to the smaller reward percent; it is harder for agents to stick around. Figure 6.8 shows the unique visits statistic from this simulation. Again, the max UV stays at 100, because at least one agent is always capable of lingering long enough to fully visit the network. Figure 6.9 shows the Unique Visits per tic performance of this test; and this is the metric for which this test was the highest rated. The Max UV tic varies wildly, since this value is a ratio involving time, and is generally a very small number, these variations look huge but are probably not statistically significant. The average value is much more useful; it indicates a period of steady improvement over the first half of the simulation where it then appears to plateau. While this seems like a very small number, it should be noted that the network size was 100 nodes; ergo the denominators max value is 100, while the divisor is the age in tics of the agent. Given that each mark on this graph represents a jump in 12,0000 tics, the tiny numbers become more understandable. Lastly, figure 6.10 shows the age statistics for the simulation.

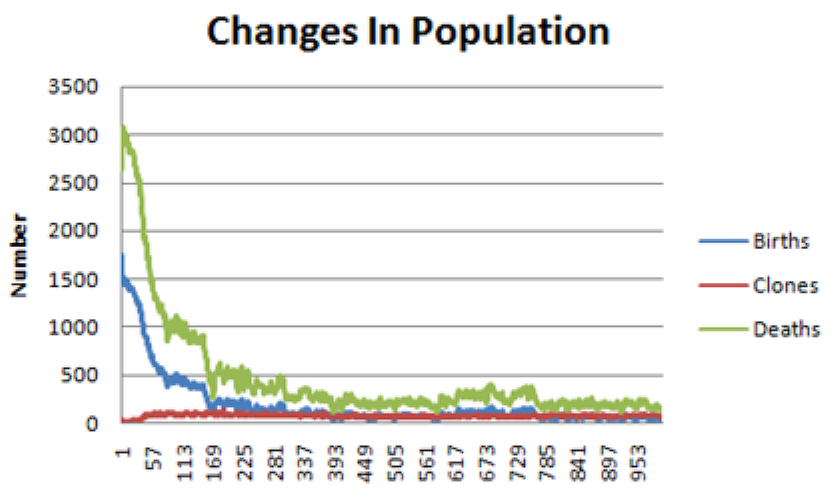

Figure 6.7: Population changes for best unique visit/tic test.

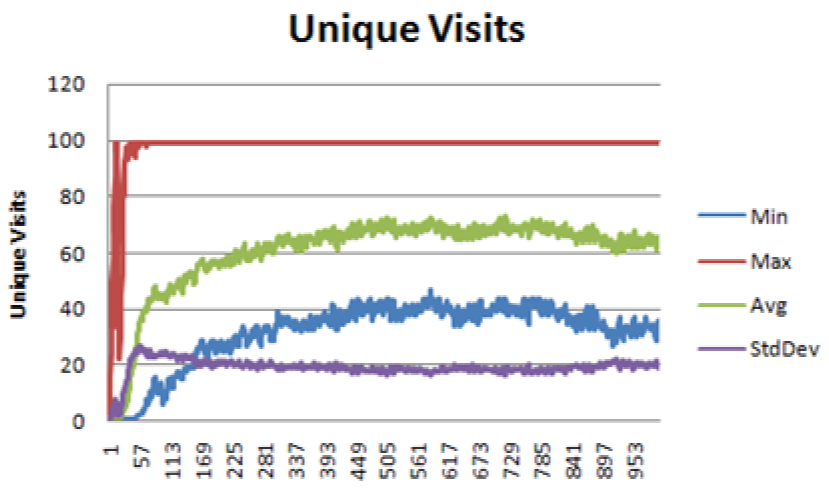

Figure 6.8: Unique visit values for best unique visit/tic test.

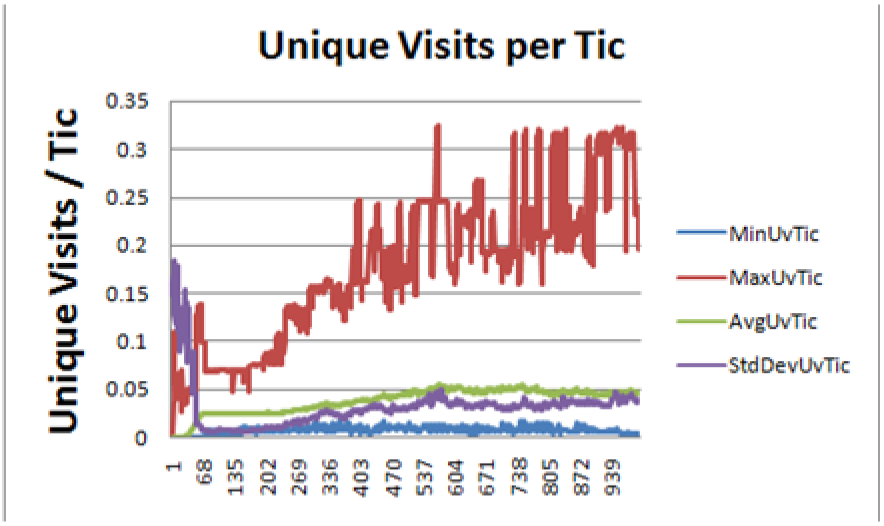

Figure 6.9: Unique visits/tic values for best unique visit/tic test.

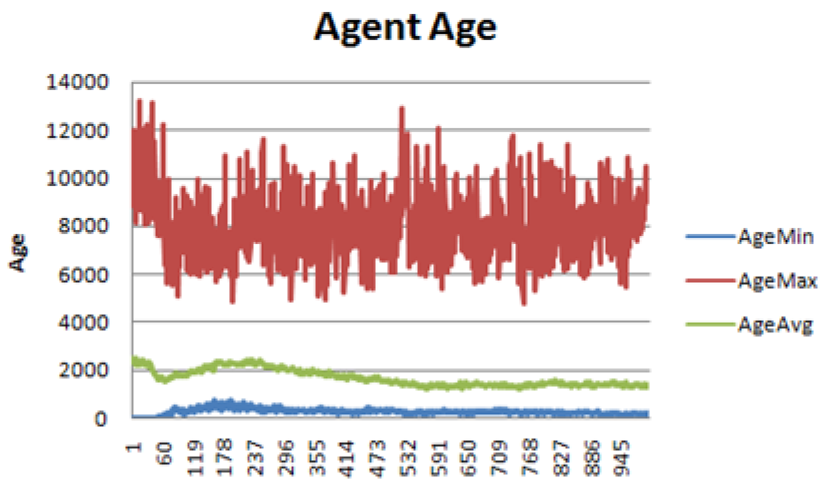

Figure 6.10: Agent age values for best unique visit/tic test.

6.7.3 Considering the Implications of the Test Results.

After comparing the simulation results, it is fairly apparent that the optimal value for minimum death (*m*) is 0. The effect of this value is that if agents continue to perform, they are effectively immortal, although they cannot 'store' performance. This has an interesting effect in the first simulation, of apparently encouraging agents to linger; that is, slow and steady definitely won the race, and produced better average Unique Visits values. This was further compounded by the large reward values for successes; an occasional reward was sufficient to reset their chance of death. This was not the case in the second discussed simulation, and predictably the average UV values were not as high, although the rate of Unique visits as measured by unique visits per tic was higher. Tests were ran with negative minimum death, and minimum death above 0, but they did not perform competitively.

With regards to the age threshold (*t*, the results are less conclusive. No values were tested below 0.5. This value measures the 'closeness' of an agent to the local oldest agent before it decides to breed. Put differently, 0.5 means that both breeding pairs must be at least half as old as the oldest local agent in order to reproduce. One conclusion that can be drawn from the lack of performance evidenced by high threshold values of 0.8 and 0.9 is that they were too exclusive, and poor at exploring the problem domain. The threshold value of 0.7 probably performed well because of other factors such as the high reward setting, but this does not conclusively prove that it is a poor value. It is evident that lower age thresholds would logically increase churn, which might explain why the second test evidenced the best rate of unique visits, although the max rates shown in figure 6.9 were not very stable in that simulation, unlike in the first simulation, shown in figure 6.5.

The results would seem to indicate that smaller reward values (r) increase the selection pressure to perform quickly; the average rate of unique visits in figure 6.9 is equal to the maximum value from the first test shown in figure 6.5. The large rewards in the first simulation made it less vital to immediately discover a new node, while in the second simulation agents would have died from senescence if they had not kept a fairly high rate of discovery. Unfortunately, the results did not conclusively prove one way or another whether variable mutation is superior to non variable mutation.

Chapter 7
CONCLUSION

This work has visited many subjects and discussed many things. It is necessary now to draw to a close, and it seems appropriate to revisit the major points. A brief review of the background material and justification for this work is provided in section 7.1. A summary of the contributions of this work and their logical implications is provided in section 7.2. Lastly, we take a look at the possible future directions for this work in section 7.3.

7.1 Prelude

This work looked at mobile agents fairly extensively. They provide many advantages over the classic client-server architecture. The chief benefit is the potential to be much more efficient in the consumption of resources, and more resistant to fault. Mobile agents are unfortunately not without faults such as security concerns, fault concerns, and complexity issues.

Many approaches to ensuring security, fault tolerance, and reducing complexity were naturally examined as part of this work. In some sense, all are somewhat lacking, at least in the sense of being comprehensiveness. Indeed, the very fact that mobile agents are still not widely adopted would indicate that the problems have not been adequately solved.

Genetic algorithms were the next major topic explored in this work. Genetic algorithms are a rich optimization technique with a great depth of work. At the core of things, though, is an analogy to natural evolution. The survival of the fittest. This is modified through the mechanisms of selection, which in the case of most genetic algorithms is much more goal-based than natural evolution. Regardless, an optimization strategy designed at its core around survival is naturally attractive to a field that suffers from security and fault concerns. Genetic algorithms also remove the programmer from the complexity of the solution, in some sense. Lastly, genetic algorithms have been successfully applied to some agent based problems. This work considered these implications, and together with the failings of other approaches, a worthy case for applying genetic algorithms to mobile agents was made.

7.2 Summary

The next major aspect of this work was an actual implementation of genetic mobile agents. This was called DNAgents 1.0, and was successful. This endeavor did highlight some difficulties, unfortunately. The chief problems resolve around fitness and selection. Fitness proved to be tricky to define adequately for mobile agents. Additionally, even when adequately defined, it is even more difficult to measure. This is primarily due to the fact that mobile agent tasks are often unbounded with respect to time. Put differently, it is hard to predict, due to the network, how long a task will take, and thus how successful an agent is. Indeed, it is sometimes even difficult to rank agents other than in a black and white manner: succeed or fail. This complicates selection, which is reliant upon ranking agents respective of each other. DNAgents 1.0 served to nicely illustrate these issues, and lead to the considerations of various modifications to genetic algorithms such as distributed genetic algorithms, continuous genetic algorithms, and steady state genetic algorithms. These variations were ultimately found to be as problematic as traditional genetic algorithms and were thus disregarded. Artificial life was also considered, as it takes genetic algorithms back to their origins in some sense, with a stronger focus on natural evolution and less emphasis on ranking.

Unfortunately, artificial life itself is somewhat lacking, at least as a strategy to evolve goal-based agents, but it lead to DNAgents 2.0. That is, what was needed was neither genetic algorithms nor artificial life. A new framework that combined the best aspects of both seemed appropriate, and well suited to mobile agents. The key part of this is the analogy of considering mobile agents as organisms, and the network as their world. This allows for a much more ad-hoc selection and culling mechanism, with much less emphasis on ranking and selection. Fitness is still encouraged, and here the goal-oriented nature of genetic algorithms and mobile agents is telling, by a revitalization mechanism. This mechanism interacts with the culling mechanism of senescence to provide a direction for the evolution of agents beyond survival. By far the greatest benefit of this framework is that evolution need not ever end; it could continually occur within an active network. With a framework formulated, an experiment was implemented, which lead to a few new mechanisms that were not quite anticipated being created, such as cloning and population control. Ultimately, the experiment was successful and provided a basis for analyzing the parameters of the algorithm. The results of this were presented, and discussed, and are promising.

7.3 Future Work

While DNAgents 2.0 were successful, the entirety of the vision was not explored in the context of this work. Firstly, many potential mechanisms for selection behavior in DNAgents 2.0 were proposed, including random, age-based, energy-based, difference-based, fitness-based, and dynamic. These behaviors were all defined and discussed, but only random and age-based behaviors were investigated in the experimentation conducted. This was partially due to the fact that age-based selection functioned, but also partially due to scope and scale. This work is not insurmountable, but time constrained prevented it. Additionally, even if all possible selection behaviors were experimented with, it would be incorrect to assume that this is an exhaustive list of possible selection behaviors.

Similarly, there were two proposed death mechanisms: senescence-based and tic-based. While both seemed to work in the experimentation, early on the decision was made to focus on senescence-based because early results seemed to favor it. This does not exclude tic-based death from being considered at all, and more experimentation might be worthwhile. Again, it is not necessarily the only two mechanisms for measuring death that might be viable, and attempting to define new ones might provide interesting results.

The implementation of DNAgents 2.0 only experimented with fairly standard mutation and crossover mechanisms borrowed from genetic algorithms. There are many variations on crossover that might produce different results, and some variations on mutation such as allowing self-selection of mutation rates, which might prove beneficial, but were not quantitatively explored. Another obvious future task would be the application of this framework to the other proposed experiments that were not undertaken, or new experiments. That is, apply it to the evolution of agents for other tasks besides exploration.

Early in the experimentation on DNAgents 2.0, problems with extinction and then overpopulation were encountered. Quick brainstorming lead to fairly basic mechanisms to counter this. In the case of extinction, a cloning behavior was introduced, and proved successful. Another possibility would be allowing for temporary immortality of successful agents that had never reproduced, thus preserving useful genes that temporarily separate from the main population. This was not experimented with because allowing each agent to produce one clone if alone on an agency proved so successful. This of course lead ultimately to overpopulation, which was caused by an oversight. If agents exist, and are not limited by some artificial mechanism, they can quickly overpopulate. Some selection mechanisms, such as energy, would avoid this problem entirely by artificially limiting the number of existent agents. In this case, two simple behaviors were experimented with: limiting agent reproduction based on the number of agents locally present on the node, and based on the average number of agents seen by the agent. The latter method performed better. Again, a major potential area for improving the framework would be improving these admittedly primitive population control mechanisms, or simply replacing them.

Lastly it would be useful to revisit the ultimate motivation for a mechanism such as

DNAgents 2.0. This motivation at its most simple is that a population of agents continuously self-adapting to a changing network are theoretically much more flexible than human beings ever could be. That is to say, especially with respect to security, that if the agents were continuously attempting to detect and ultimately solve security vulnerabilities in a platform, the rate of response could be much greater than the turn around required for human operators to notice a problem, investigate it, and then devise a solution. Certainly the dark reflection of this is absolutely terrifying: malicious agents that continuously try to exploit. While none of this is currently in existence, a framework such as DNAgents 2.0 that allows for the continuous optimization of mobile agents might be the first step towards such a state, and is thus worthwhile.

BIBLIOGRAPHY

[1] Adel M. Abunawass. Biologically based machine learning paradigms: an introductory course. *SIGCSE Bull.*, 24(1):87–91, 1992.

[2] Chris Adami and C. Titus Brown. Evolutionary learning in the 2d artificial life "avida"'. *Artificial Life IV: Proceedings of the Fourth International Workshop on the Synthesis and Simulation of Living Systems*, pages 377–382, September 1994.

[3] Christos Ampatzis, Elio Tuci, Vito Trianni, and Marco Dorigo. Evolution of signaling in a multi-robot system: Categorization and communication. *Adaptive Behavior - Animals, Animats, Software Agents, Robots, Adaptive Systems*, 16(1):5–26, 2008.

[4] James E. Baker. Reducing bias and inefficiency in the selection algorithm. In *Proceedings of the Second International Conference on Genetic Algorithms and their Application*, pages 14–21, New York, NY, USA, 1987. ACM.

[5] M. Baldi and G.P. Picco. Evaluating the tradeoffs of mobile code design paradigms in network management applications. In *Proceedings of the 20th international conference on Software engineering*, pages 146–155. IEEE Computer Society Washington, DC, USA, 1998.

[6] Edwin Roger Banks, Paul Agarwal, Marshall McBride, and Claudette Owens. A comparison of selection, recombination, and mutation parameter importance over a set of fifteen optimization tasks. In *GECCO '09: Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference*, pages 1971–1976, New York, NY, USA, 2009. ACM.

[7] J. Baumann, F. Hohl, K. Rothermel, and M. Straßer. Mole–Concepts of a mobile agent system. *World Wide Web*, 1(3):123–137, 1998.

[8] T Bäck. Optimal mutation rates in genetic search. In *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 2–8. Morgan Kaufmann Publishers, 1993.

[9] T Bäck. *Evolutionary Algorithms in Theory and Practice - Evolution Strategies, Evolutionary Programming, Genetic Algorithms.* Oxford University Press, 1996.

[10] Thomas Bäck. Selective pressure in evolutionary algorithms: A characterization of selection mechanisms. In *Proceedings of the First IEEE Conference on Evolutionary Computation. IEEE World Congress on Computational Intelligence (ICEC94).*, pages 57–62. IEEE, 1994.

[11] Steven Beaty, Darrell Whitley, and Gearold Johnson. Motivation and framework for using genetic algorithms for microcode compaction. *SIGMICRO Newsl.*, 22(1):20–27, 1991.

[12] Padmanabha V. Bedarhally, Rafael A. Perez, and Weon S. Chung. A family elitist approach in genetic algorithms. In *SAC '96: Proceedings of the 1996 ACM symposium on Applied Computing*, pages 238–244, New York, NY, USA, 1996. ACM.

[13] M.A. Bedau, E. Snyder, C.T. Brown, and N.H. Packard. A comparison of evolutionary activity in artificial evolving systems and in the biosphere. In *Proceedings of the Fourth European Conference on Artificial Life*, pages 125–134. MIT Press, 1997.

[14] Mark A. Bedau, John S. McCaskill, Norman H. Packard, Steen Rasmussen, Chris Adami, David G. Green, Takashi Ikegami, Kunihiko Kaneko, and Thomas S. Ray. Open problems in artificial life. *Artif. Life*, 6(4):363–376, 2000.

[15] A. Bieszczad, B. Pagurek, and T. White. Mobile agents for network management. *IEEE Communications Surveys*, 1(1):2–9, 1998.

[16] T. Blickle and L. Thiele. A comparison of selection schemes used in genetic algorithms. Technical report, Computer Engineering and Communication Networks Lab (TIK), Swiss Federal Institute of Technology (ETH), Zürich, Switzerland, 1995.

[17] Margaret A. Boden. Is metabolism necessary? *British Journal for the Philosophy of Science*, 50, 1999.

[18] L. Booker. Improving search in genetic algorithms. In *Genetic Algorithms and Simulated Annealing.*, pages 61–73. Morgan Kaufmann Publishers, San Mateo, California, USA, 1987.

[19] H. Braun. Evolution| a Paradigm for Constructing Intelligent Agents. *Prerational Intelligence: Adaptive Behavior and Intelligent Systems Without Symbols and Logic*, page 279, 2001.

[20] B. Brewington, R. Gray, K. Moizumi, D. Kotz, G. Cybenko, and D. Rus. Mobile agents in distributed information retrieval. *Intelligent Information Agents*, pages 355–395, 1999.

[21] M.G. Bulmer. *The Mathematical Theory of Quantitative Genetics.* Clarendon Press, Oxford, 1980.

[22] Erick Cantú-Paz. A survey of parallel genetic algorithms. *Calculateurs Pralleles, Reseaux Et Systems Repartis*, 10, 1998.

[23] Erick Cantu-Paz and David E. Goldberg. Efficient parallel genetic algorithms: Theory and practice. In *Computer Methods in Applied Mechanics and Engineering*. press, 2000.

[24] Rich Caruana, Larry J. Eshelman, and J. David Schaffer. Representation and hidden bias ii: Eliminating defining length bias in genetic search via shuffle crossover. In *IJCAI*, pages 750–755, 1989.

[25] D. Chess, C. Harrison, and A. Kershenbaum. Mobile agents: Are they a good idea? *Lecture Notes in Computer Science*, 1222:25–45, 1997.

[26] D.M. Chess. Security issues in mobile code systems. *Lecture Notes in Computer Science*, 1419:1–14, 1998.

[27] J. Claessens, B. Preneel, and J. Vandewalle. (How) can mobile agents do secure electronic transactions on untrusted hosts? A survey of the security issues and the current solutions. *ACM Transactions on Internet Technology*, 3(1):28–48, 2003.

[28] Robert J. Collins and David R. Jefferson. Selection in massively parallel genetic algorithms. In *Fourth International Conference on Genetic Algorithms*, 1991.

[29] Carlo Comis. Darwinbots, July 2008. `http://www.darwinbots.com/`.

[30] J.F. Crow and M. Kimura. *An Introduction to Population Genetics Theory*. Harper and Row, New York, 1970.

[31] Jonathan Dale and David C. Deroure. A mobile agent architecture for distributed information management. Technical report, In Proceedings of the International Workshop on the Virtual Multicomputer, 1997.

[32] B. Damer and R. El. The EvoGrid. *Technoetic Arts: a Journal of Speculative Research*, 7(2):175–190, 2009.

[33] P. Dasgupta. Improving peer-to-peer resource discovery using mobile agent based referrals. In *Proc. of the Int. Workshop on Agents and Peer-to-Peer Computing, Springer-Verlag, Lecture Notes on Computer Science*, volume 2872, pages 186–197. Springer, 2004.

[34] Y. Davidor, T. Yamada, and R. Nakano. The ECOlogical framework II: Improving GA performance at virtually zero cost. In *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 171–176, 1993.

[35] A. Demers, K. Petersen, M. Spreitzer, D. Terry, M. Theimer, and B. Welch. The Bayou architecture: Support for data sharing among mobile users. In *Proc. of IEEE Workshop on Mobile Computing Systems & Applications*, pages 2–7. Citeseer, 1994.

[36] G. Derbas, A. Kayssi, H. Artail, and A. Chehab. Trummar-a trust model for mobile agent systems based on reputation. In *Proceedings of the IEEE/ACS International Conference on Pervasive Services (ICPS Ś04), Beirut, Lebanon*. Citeseer, 2004.

[37] T.C. Du, E.Y. Li, and A.P. Chang. Mobile agents in distributed network management. *Communications of the ACM*, 46(7):132, 2003.

[38] C.R. Dunne. Using mobile agents for network resource discovery in peer-to-peer networks. *ACM SIGecom Exchanges*, 2(3):1–9, 2001.

[39] A. Egri-Nagy and C.L. Nehaniv. Evolvability of the genotype-phenotype relation in populations of self-replicating digital organisms in a tierra-like system. *Advances in Artificial Life*, pages 238–247, 2003.

[40] Khaled El-Sawi. *Guided Genetic Evolution: A Framework for the Evolution of Autonomous Robotic Controllers*. PhD thesis, The University of Southern Mississippi, August 2006.

[41] K. Fall. Network emulation in the Vint/NS simulator. In *IEEE International Symposium on Computers and Communications, 1999. Proceedings*, pages 244–250, 1999.

[42] S. Fischmeister. Mobile code paradigms, 2002. `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.95.9290&rep=rep1&type=pdf`.

[43] D. B. Fogel. *Evolutionary computation: toward a new philosophy of machine intelligence.* IEEE Press, New York, 1995.

[44] Frank D. Francone, Larry M. Deschaine, and Jeffrey J. Warren. Discrimination of munitions and explosives of concern at f.e. warren afb using linear genetic programming. In *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 1999–2006, New York, NY, USA, 2007. ACM.

[45] Ping Fu, Jia qing Qiao, and Hong tao Yin. A virus evolutionary genetic algorithm using local selection. *Innovative Computing ,Information and Control, International Conference on*, 0:582, 2007.

[46] M. Fukuda, Y. Tanaka, N. Suzuki, L.F. Bic, and S. Kobayashi. A mobile-agent-based pc grid. In *Autonomic Computing Workshop*, pages 142–150. Citeseer, 2003.

[47] D. E. Goldberg and K. Deb. A comparative analysis of selection schemes used in genetic algorithms. In G. J. E. Rawlins, editor, *Foundations of Genetic Algorithms*, pages 69–93. Morgan Kaufmann Publishers, San Mateo, California, 1991. Lectures in Applied Mathematics, Vol. 22, Part 1.

[48] Jonatan Gómez, Roberto Poveda, and Elizabeth León. Grisland: a parallel genetic algorithm for finding near optimal solutions to the traveling salesman problem. In *GECCO '09: Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference*, pages 2035–2040, New York, NY, USA, 2009. ACM.

[49] Jean Gourd. *API-S Calculus: Formal Modeling for Secure Mobile intelligent Agent Systems*. PhD thesis, The University of Southern Mississippi, August 2007.

[50] R.S. Gray, D. Kotz, R.A. Peterson, J. Barton, D. Chacon, P. Gerken, M. Hofmann, J. Bradshaw, M. Breedy, R. Jeffers, et al. Mobile-agent versus client/server performance: Scalability in an information-retrieval task. *Lecture notes in computer science*, pages 229–243, 2001.

[51] Z. Guessoum, N. Faci, and J.P. Briot. Adaptive replication of large-scale multi-agent systems–towards a fault-tolerant multi-agent platform. *Software Engineering for Multi-Agent Systems IV*, pages 238–253, 2006.

[52] Howard Gutowitz. Artificial-life simulators and their applications, 1995.

[53] R.S. Hall, D. Heimbigner, and A.L. Wolf. A cooperative approach to support software deployment using the software dock. In *Proc. IntŠl Conf. Software Eng.,(ICSEŠ99)*, pages 174 – 183, 1999.

[54] N. Hansen, A. Ostermeier, and A. Gawelczyk. On the adaptation of arbitrary mutation distributions in evolution strategies: The generating set adaptation. In *Proceedings of the Sixth International Conference on Genetic Algorithms*, pages 57–64. Morgan Kaufmann Publishers, 1995.

[55] Francisco Herrera and Manuel Lozano. Adaptation of genetic algorithm parameters based on fuzzy logic controllers. In *Genetic Algorithms and Soft Computing*, pages 95–125. Physica-Verlag, 1996.

[56] Jonathan R. Hicks and Joseph A. Driscoll. Evolution of artificial agents in a realistic virtual environment. In *ACM-SE 43: Proceedings of the 43rd annual Southeast regional conference*, pages 365–369, New York, NY, USA, 2005. ACM.

[57] W.H. Hsu and S.M. Gustafson. Genetic programming and multi-agent layered learning by reinforcements. In *Genetic and Evolutionary Computation Conference*, pages 764–771. Citeseer, 2002.

[58] W. Jansen and T. Karygiannis. NIST special publication 800-19–mobile agent security. *Gaithersburg, MD: National Institute of Standards and Technology*, 1999.

[59] K. Jun, L. Boloni, K. Palacz, and D.C. Marinescu. Agent-based resource discovery. In *9th Heterogeneous Computing Workshop*, pages 43–52. Press, 2000.

[60] J. Kackley, P. Wahjudi, and Ali D. A mobile agent simulator. In *2009 International Conference on Industry, Engineering, and Management Systems*, 2009.

[61] G. Karjoth, N. Asokan, and C. Gülcü. Protecting the computation results of free-roaming agents. *Personal and Ubiquitous Computing*, 2(2):92–99, 1998.

[62] H. Katagiri, K. Hirasama, and J. Hu. Genetic network programming-application to intelligent agents. In *2000 IEEE International Conference on Systems, Man, and Cybernetics*, volume 5, pages 3829–3834, 2000.

[63] Hilmi Güneş Kayacik, Malcolm Heywood, and Nur Zincir-Heywood. On evolving buffer overflow attacks using genetic programming. In *GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 1667–1674, New York, NY, USA, 2006. ACM.

[64] Robert E. Keller and Wolfgang Banzhaf. Genetic programming using genotype-phenotype mapping from linear genomes into linear phenotypes. In *GECCO '96: Proceedings of the First Annual Conference on Genetic Programming*, pages 116–122, Cambridge, MA, USA, 1996. MIT Press.

[65] Jin-Lee Kim. Permutation-based elitist genetic algorithm using serial scheme for large-sized resource-constrained project scheduling. In *WSC '07: Proceedings of the 39th conference on Winter simulation*, pages 2112–2118, Piscataway, NJ, USA, 2007. IEEE Press.

[66] Minkyu Kim, Varun Aggarwal, Una-May O'Reilly, and Muriel Medard. A doubly distributed genetic algorithm for network coding. In *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 1272–1279, New York, NY, USA, 2007. ACM.

[67] J. Kiniry and D. Zimmerman. A hands-on look at Java mobile agents. *IEEE Internet Computing*, 1(4):21–30, 1997.

[68] J. Klein. breve: a 3d simulation environment for the simulation of decentralized systems and artificial life. In *Proceedings of Artificial Life VIII, the 8th International Conference on the Simulation and Synthesis of Living Systems*, pages 329 – 334. The MIT Press, 2002.

[69] Maciej Komosinski. The world of framsticks: Simulation, evolution, interaction. In *VW '00: Proceedings of the Second International Conference on Virtual Worlds*, pages 214–224, London, UK, 2000. Springer-Verlag.

[70] Maciej Komosinski. The framsticks system: versatile simulator of 3d agents and their evolution, 2003.

[71] Maciej Komosinski and Szymon Ulatowski. Framsticks: Towards a simulation of a nature-like world, creatures and evolution. In *ECAL '99: Proceedings of the 5th European Conference on Advances in Artificial Life*, pages 261–265, London, UK, 1999. Springer-Verlag.

[72] Naoyuki Kubota and Koji Shimojima. Virus-evolutionary genetic algorithm- ecological model on planar grid. In *Fuzzy Information Processing Society, Biennial Conference of the North American proceeding*, pages 505–509, 1996.

[73] Christopher G. Langton. Studying artifical life with cellular automata. *Physica D*, pages 120–149, 1986.

[74] Christopher G. Langton. Artifical life. *Santa Fe Institute Studies in the Sciences of Complexity*, 6:1–47, 1988.

[75] Yong Liang, Kwong-Sak Lueng, and Tony Shu Kam Mok. Automating the drug scheduling with different toxicity clearance in cancer chemotherapy via evolutionary computation. In *GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 1705–1712, New York, NY, USA, 2006. ACM.

[76] Cláudio F. Lima, Kumara Sastry, David E. Goldberg, and Fernando G. Lobo. Combining competent crossover and mutation operators: a probabilistic model building approach. In *GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 735–742, New York, NY, USA, 2005. ACM.

[77] J. Liu and V. Issarny. Enhanced reputation mechanism for mobile ad hoc networks. *Lecture Notes in Computer Science*, 2995:48–62, 2004.

[78] JingJing Liu. Research on an improved virus evolutionary genetic algorithm. *Information Engineering, International Conference on*, 2:114–116, 2009.

[79] KJ Mackin and E. Tazaki. Unsupervised training of multiobjective agent communication usinggenetic programming. In *Knowledge-Based Intelligent Engineering Systems and Allied Technologies, 2000. Proceedings. Fourth International Conference on*, volume 2, pages 738 – 741, 2000.

[80] Silvano Martello and Paolo Toth. *Knapsack Problems: Algorithms and Computer Implementations.* John Wiley and Sons., Boston, MA 02116, 1990.

[81] H Mühlenbein. The breeder genetic algorithm - a provable optimal search algorithm and its application. In *Colloquium on Applications of Genetic Algorithms*, London, 1994.

[82] Heinz Mühlenbein and Dirk Schlierkamp-Voosen. Analysis of selection, mutation and recombination in genetic algorithms. *Neural Network World*, 3:907–933, 1993.

[83] Melanie Mitchell and Stephanie Forrest. Genetic algorithms and artificial life. *Artificial Life*, 1993.

[84] H. Muhlenbein and D. Schlierkamp-Vosen. Predictive models for the breeder genetic algorithm. In *Evolutionary Computation*, pages 25–49, 1993.

[85] E. Olougouna and S. Pierre. Mobile agents and their use for information retrieval: A brief overview and an elaborate case study. *IEEE network*, page 34, 2002.

[86] A. Ostermeier, A. Gawelczyk, and N. Hansen. A derandomized approach to self adaptation of evolution strategies. Technical report, TU, Berlin, 1993.

[87] A. Ostermeier, A. Gawelczyk, and N. Hansen. Step-size adaptation based on non-local use of selection information. In *Parallel Problem Solving from Nature - PPSN III: International Conference on Evolutionary Computation. Vol. 866 of Lecture Notes in Computer Science*, pages 189–198. Springer-Verlag, 1994.

[88] T. Park, I. Byun, H. Kim, and H. Yeom. The performance of checkpointing and replication schemes for fault tolerant mobile agent systems. In *PROCEEDINGS OF THE SYMPOSIUM ON RELIABLE DISTRIBUTED SYSTEMS*, pages 256–261, 2002.

[89] H.H. Pattee. Simulations,realizations, and theories of life. *Santa Fe Institute Studies in the Sciences of Complexity*, 6:63–77, 1988.

[90] V.A. Pham and A. Karmouch. Mobile software agents: An overview. *IEEE Communications magazine*, 36(7):26–37, 1998.

[91] Alan Piszcz and Terence Soule. A survey of mutation techniques in genetic programming. In *GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 951–952, New York, NY, USA, 2006. ACM.

[92] Alan Piszcz and Terence Soule. A survey of mutation techniques in genetic programming. In *GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 951–952, New York, NY, USA, 2006. ACM.

[93] J. Pitt, L. Kamara, M. Sergot, and A. Artikis. Voting in multi-agent systems. *The Computer Journal*, 49(2):156, 2006.

[94] S. Pleisch and A. Schiper. Fatomas-a fault-tolerant mobile agent system based on the agent-dependent approach. In *Proc. of Int. Conference on Dependable Systems and Networks (DSNŠ01)*, pages 215–224, 2001.

[95] A. Puliafito and O. Tomarchio. Using mobile agents to implement flexible network management strategies. *Computer Communications*, 23(8):708–719, 2000.

[96] Rushil Raghavjee and Nelishia Pillay. An application of genetic algorithms to the school timetabling problem. In *SAICSIT '08: Proceedings of the 2008 annual research conference of the South African Institute of Computer Scientists and Information Technologists on IT research in developing countries*, pages 193–199, New York, NY, USA, 2008. ACM.

[97] S. Rahimi. *API-Calculus for Intelligent-Agent Formal Modeling and its Application in Distributed Geospatial Data Conflation*. PhD thesis, University of Southern Mississippi, 2002.

[98] Larry Raisanen and Roger M. Whitaker. Comparison and evaluation of multiple objective genetic algorithms for the antenna placement problem. *Mob. Netw. Appl.*, 10(1-2):79–88, 2005.

[99] S.D. Ramchurn, D. Huynh, and N.R. Jennings. Trust in multi-agent systems. *The Knowledge Engineering Review*, 19(01):1–25, 2005.

[100] Steen Rasmussen, Michael J. Raven, Gordon N. Keating, and Mark A. Bedau. Collective intelligence of the artificial life community on its own successes, failures, and future. *Artif. Life*, 9(2):207–235, 2003.

[101] Thomas S. Ray. An approach to the synthesis of life. *Santa Fe Institute Studies in the Sciences of Complexity*, 10:371–408, 1991.

[102] Thomas S. Ray. An evolutionary approach to synthetic biology: zen and the art of creating life. *Artificial Life 1*, pages 195–226, 1994.

[103] I. Rechenberg. *Evolutionsstrategie.* Frommann-Holzboog, Stuttgart, 1994.

[104] Jean-Philippe Rennard. *Perspectives for Strong Artificial Life*. 2004.

[105] Jeffrey P. Ridder and Jason C. HandUber. Mission planning for joint suppression of enemy air defenses using a genetic algorithm. In *GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 1929–1936, New York, NY, USA, 2005. ACM.

[106] J. Riely and M. Hennessy. Trust and partial typing in open systems of mobile agents. *Journal of Automated Reasoning*, 31(3):335–370, 2003.

[107] K. Rothermel, F. Hohl, and Informatik Gesellschaft fur. *Mobile agents*. Citeseer, 1998.

[108] K. Rothermel and M. Straßer. A fault-tolerant protocol for providing the exactly-once property of mobile agents. In *Proceedings of the Symposium on Reliable Distributed Systems*, pages 100–108. IEEE Computer Society Press, 1998.

[109] Franz Rothlauf. Representations for evolutionary algorithms. In *GECCO '09: Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference*, pages 3131–3156, New York, NY, USA, 2009. ACM.

[110] T. Sander and C.F. Tschudin. Protecting mobile agents against malicious hosts. *Lecture Notes in Computer Science*, 1419:44–60, 1998.

[111] Jayshree A. Sarma. *An Analysis of Decentralized and Spatially Distributd Genetic Algorithms*. PhD thesis, George Mason University, 1998.

[112] Jacob B. Schrum. Study and comparison of genetic algorithms when applied to lego mindstorms robots. *J. Comput. Small Coll.*, 19(4):353–353, 2004.

[113] H.P. Schwefel. *Numerical optimization of computer models.* Wiley and Sons, Chichester, 1981.

[114] Hu Shicheng, Chu Dianhui, and Xu Xiaofei. A virus evolution genetic algorithm for scheduling problem with penalties of independent tasks on a single machine. *Intelligent Systems, WRI Global Congress on*, 1:574–578, 2009.

[115] F.M.A. Silva and R. Popescu-Zeletin. An approach for providing mobile agent fault tolerance. In *Second International Workshop on Mobile Agents*, volume 1477, pages 14–25. Springer, 1998.

[116] L. Silva, V. Batista, and J. Silva. Fault-tolerant execution of mobile agents. In *Proc. of the International Conference on Dependable Systems and Networks*, pages 135–143, 2000.

[117] Elliott Sober. Learning from functionalism|prospects for strong artificial life. *Santa Fe Institute Studies in the Sciences of Complexity*, 10, 1991.

[118] W.M. Spears and K. A De Jong. An analysis of multi-point crossover. In *Foundations of Genetic Algorithms.*, pages 301–315. Morgan Kaufmann Publishers, San Mateo, California, USA, 1991.

[119] W.M. Spears and K.A. De Jong. On the virtues of parameterized uniform crossover. In R. Belew and L. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 230–237, 1991.

[120] L. Spector and J. Klein. Complex adaptive music systems in the breve simulation environment. In *Proc. ALife VIII Workshops*, pages 17–24. Citeseer, 2002.

[121] Lee Spector and Jon Klein. Evolutionary dynamics discovered via visualization in the breve simulation environment. In *The 8th International Conference on the Simulation and Synthesis of Living Systems, Artificial Life VIII*, 2002.

[122] M. Srinivas and Lalit M. Patnaik. Genetic algorithms: A survey. *Computer*, 27(6):17–26, 1994.

[123] Ken Stauffer. Evolve 4.0, July 2007. `http://www.stauffercom.com/evolve4/index.html`.

[124] Carsten Knudsen Steen Rasmussen and Rasmus Feldberg. Dynamics of programmable matter. 10:211–254, 1991.

[125] Gilbert Syswerda. Uniform crossover in genetic algorithms. In *Proceedings of the 3rd International Conference on Genetic Algorithms*, pages 2–9, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

[126] T. Taylor. 1 Creativity in Evolution: Individuals, Interactions, and Environments. *Creative evolutionary systems*, 2001.

[127] Tim Taylor. The cosmos artificial life system. *Mind*, 75:527–541, 1997.

[128] Timothy John Taylor. *From Artificial Evolution to Artificial Life*. PhD thesis, University of Edinburgh, 1999.

[129] O. Tomarchio, L. Vita, and A. Puliafito. Active monitoring in grid environments using mobile agent technology. In *Active middleware services: from the proceedings of the 2nd annual Workshop on Active Middleware Services*, page 57. Kluwer Academic Pub, 2000.

[130] David A. Van Veldhuizen and Gary B. Lamont. Multiobjective evolutionary algorithms: Analyzing the state-of-the-art, 2000.

[131] Sébastien Verel. Fitness landscapes and graphs: multimodularity, ruggedness and neutrality. In *GECCO '09: Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference*, pages 3593–3656, New York, NY, USA, 2009. ACM.

[132] John F. Walker and James H. Oliver. A survey of artificial life and evolutionary robotics, 1997.

[133] Mark Ward. *Virtual Organisms: The Startling World of Artificial Life*. Thomas Dunne Books, 2000.

[134] David C. Wedge and Douglas B. Kell. Rapid prediction of optimum population size in genetic programming using a novel genotype -: fitness correlation. In *GECCO '08: Proceedings of the 10th annual conference on Genetic and evolutionary computation*, pages 1315–1322, New York, NY, USA, 2008. ACM.

[135] G. Weiss. *Multiagent systems: a modern approach to distributed artificial intelligence*. The MIT Press, 2000.

[136] U.G. Wilhelm, S. Staamann, and L. Buttyan. On the problem of trust in mobile agent systems. In *Symposium on Network and Distributed System Security*, pages 114–124. Citeseer, 1998.

[137] H.C. Wong and K. Sycara. Adding security and trust to multiagent systems. *Applied Artificial Intelligence*, 14(9):927–941, 2000.

[138] Carl Zimmer. Testing darwin. *Discover Magazine*, Februrary 2005. `http://discovermagazine.com/2005/feb/cover`.

INDEX