Computer Science Department Technical Report
University of California
Los Angeles, CA 90024-1596

# STUDIES IN ARTIFICIAL EVOLUTION

R. J. Collins

July 1992
CSD-920037

# Studies in Artificial Evolution[1]

**Robert James Collins**[2]
Artificial Life Laboratory
Department of Computer Science
University of California, Los Angeles
Los Angeles, CA 90024

[1]A dissertation submitted in partial satisfaction of the requirements for the degree of Doctor of Philosophy in Computer Science, University of California, Los Angeles, 1992.

[2]Electronic mail address: rjc@cs.ucla.edu

## Abstract

We define *artificial evolution* as a particular class of genetic algorithms. While the design of traditional genetic algorithms is driven by the goal of optimization, the intent of artificial evolution is biological realism. Artificial evolution genetic algorithms require a clear separation between genotype and the information encoded in the genotype. The genotype is represented as a linear string, and the genetic operators of recombination and mutation operate randomly at the lowest level of organization of the string, without reference to any syntactic nor semantic structure that may be encoded there. We view the genotype as encoding a program; the fitness of the genotype is determined by decoding and executing the program, perhaps in an environment that is shared by the other members of the population. Often, the selection and mating process of the artificial evolution genetic algorithm will include spatial structure. To achieve realistic population dynamics, we simulate large populations (at least tens of thousands of individuals in each generation).

We apply artificial evolution to three classes of problems: the study of natural evolution, the evolution of complex behavior in artificial organisms, and function optimization. We simulate elaborations of Kirkpatrick's analytic model of sexual selection, exploring the effects of relaxing the simplifying assumptions (required for the analytic solution). We demonstrate that both the equilibrium and non–equilibrium dynamics are strongly affected by the details of the model. We also simulate the effect of host–parasite coevolution on the evolution of a recombination rate modifier gene, which is a test of the parasite hypothesis for the maintenance of sexual reproduction. Our results empirically demonstrate a strong correlation between the rate of parasite evolution and the equilibrium recombination rate in the host species.

We also use artificial evolution to evolve foraging behavior in colonies of artificial ants. This study attacks the problem of representing a computer program both as a function that produces complex behavior and as a bitstring that is subject to a genetic algorithm. We introduce the *connection descriptor* artificial neural network (ANN) encoding scheme that places both the connection strengths and the connectivity pattern under genetic control, and use this encoding to evolve ant–like behavior.

Our study concludes with the application of artificial evolution to function optimization. We perform a head–to–head comparison of conventional selection and mating schemes to those that involve spatial structure. Our studies involve populations ranging in size from 8,192 to 524,288 individuals applied to graph partitioning problems. We have found that spatial structure leads to much faster and robust discovery of optimal partitions.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

## 1.1 The Field of Artificial Life

The field of artificial life is the study of systems that exhibit life–like behavior. Artificial life researchers have varying goals, including increasing our understanding of the nature of life, creating new life forms (or at least near–life forms), and exploiting life–like processes in engineering domains. For hundreds of years, biologists have pursued an understanding of natural life *as we find it here on Earth*, but the field of artificial life is directed toward the study of *life as it could be* (Langton, 1989a). Biologists have focused their attention on DNA–based living systems. Those who study "life as it could be" seek to understand the properties and processes that will be characteristic of any living systems that might exist.

The scientific methodologies underlying biological and artificial life research are quite different. Biologists have traditionally used a top–down approach, *analyzing* complex living systems by breaking them into subsystems, and further analyzing those subsystems, deducing the form and function of each component. In sharp contrast, the basic artificial life approach is bottom–up, attempting to *synthesize* complex, life–like systems. The analytic and synthetic approaches are complementary, and together form a set of very powerful scientific tools.

The dichotomy between the top–down analysis in biology and the bottom–up synthesis in artificial life studies is very important. Biological analysis involves the deduction of the mechanisms underlying known living systems in great detail, leading to inferences about general principles underlying the types of life with which we are familiar. It is difficult to separate the features of natural living systems that are general properties of life from those features that are simply quirks of the way life evolved on Earth. In contrast, the synthetic approach allows us to explore systems composed of the simplest underlying mechanisms that result in the emergence of life–like behavior.

Biology and artificial life are complementary fields. Most artificial life studies are based on knowledge gained from the study of natural life, and the synthetic approach of artificial life can be applied to many questions about natural life; see (Langton, 1989b; Langton et al., 1991; Meyer and Wilson, 1991) for a number of examples. To date, most of the information flow has been in direction from biology to artificial

life. Hopefully the synthetic approach to the study of life will become more useful to biologists as the field of artificial life matures.

## 1.2 Contributions

This dissertation focuses specifically on bottom-up, artificial life simulations of evolving populations. We focus on evolution, because it is the most important and widespread property of natural life, and is the driving force behind all of the diversity, adaptation, and ecological complexity that we see today and in the fossil record.

The format of the dissertation is a series of simulation studies involving the evolution of large populations. In this dissertation, we make the following contributions:

- Demonstrate the broad applicability of artificial evolution.
- Test Kirkpatrick's (1982) analytic model of sexual selection for sensitivity of the equilibrium and non-equilibrium dynamics to variations in the details of the model (such as spatial structure and diploidy).
- Test the parasite hypothesis for the maintenance of sexual reproduction.
- Evolve foraging behavior in ant-like artificial organisms.
- Introduce an artificial neural network (ANN) encoding method that supports the programming of ANNs via evolution.
- Demonstrate that spatial structure leads to faster and more robust optimization than panmixia in genetic algorithms.
- Introduce various metrics for quantifying characteristics of a population during optimization via a genetic algorithm.
- Introduce a scalable graph partition optimization problem.

## 1.3 Overview of the Simulated Models and Results

We report the results of studies in three areas:

- The study of natural evolution (the **Peacock** and **Parasite** simulations of Chapters 3 and 4).
- The evolution of complex behaviors in artificial organisms (the **AntFarm** simulations of Chapters 5 and 6).
- The application of evolution to an optimization problem in an engineering domain (the **Partition** simulation of Chapter 7).

These studies illustrate the range of applications of artificial evolution. The common theme the binds them is low–level, biologically motivated artificial evolution.

We are particularly interested in *macro*evolution, which occurs in large populations over the course of many generations. In our studies, each experiment typically lasts for thousands of generations, with tens of thousands of individuals in each generation. The simulations are low–level in the sense that each individual organism is separately represented, as are each individual's genes. The transmission of these genes from one generation to the next is modeled after natural genetic systems, and includes sexual reproduction with recombination and point mutations.

In Chapters 3 and 4, we present simulations based on population genetics or evolutionary biology models of evolving populations. In these studies, we use low–level simulations to generate realistic population dynamics, allowing us to use the simulations as a laboratory for studying natural evolution. We present two simulations motivated by biological models: **Peacock** and **Parasite**.

**Peacock** (Chapter 3) extends an important analytic model of sexual selection and female choice (Kirkpatrick, 1982), where the females may evolve preferences for males that possess maladaptive traits. In natural life, a classic example of sexual selection is the peacock. Female peacocks possess very little in the way of ornamental plumage, while the males carry enormous, brightly colored tail feathers. The male's tail is so large and eye–catching that it reduces his mobility and makes him easily detected by predators; male peacocks are much easier prey than the females of the species. Although there is some selection against long tails, the females provide a balancing selective force by preferring to mate with males having long tails: a short–tailed male is more likely to survive, but less likely to find a mate; a long–tailed male is less likely to survive, but if he does, he is more likely to mate successfully. The paradox of the evolution of female preferences for otherwise maladaptive traits in the males is an old problem (Darwin, 1871; Fisher, 1958), but it was not until 1982 that a simple genetic model was devised (Kirkpatrick, 1982). To make the analysis tractable, the genetic model assumes a very idealized population. We use **Peacock** to test the robustness of the predictions of that analytic model with respect to variations of the simplifying assumptions. We find that replacing the haploid genetics (one copy of each gene) of the model with diploid genetics (two copies of each gene) does not affect the equilibrium, although it strongly affects the non–equilibrium dynamics. Replacing the panmictic population (global competition and mating) with a spatially structured population (competition and mating only with individuals that are nearby) affects both the equilibrium and non–equilibrium dynamics. We also find interactions between spatial structure and ploidy.

The origin, maintenance, and prevalence of sexual reproduction are paradoxical, because sexual reproduction costs much more than asexual reproduction. The problem of sex (why natural selection does not remove the more costly method of reproduction) is perhaps the most significant outstanding problem in evolutionary biology (Michod and Levin, 1987). **Parasite** (Chapter 4) is designed to test the hypothesis that host–parasite coevolution can result in selection for higher recombination rates in the host species. This is called the ***parasite hypothesis***, and has been proposed as one possible explanation for the prevalence of sexual (versus asexual)

reproduction in the extant species. Our empirical simulation studies support the parasite hypothesis to the extent that we find strong selection for non–zero recombination rates in the presence of parasitism. In addition, we find a strong correlation between the rate of evolution in the parasite species and the equilibrium recombination rate in the host species.

The **AntFarm** simulations (Chapters 5 and 6) illustrate the synthesis, via evolution, of complex behavior in artificial organisms. These simulations evolve (potentially cooperative) foraging behavior in colonies of ant–like artificial organisms. All ants from all colonies forage in a common environment. The ants can communicate through the use of pheromones (volatile chemicals used for communication) deposited in the environment. The central issue in **AntFarm** is the representation of the behavior–producing computer program at two levels: (1) as an executable program, and (2) as a bitstring chromosome on which evolution can operate effectively. We have succeeded in developing a representation for the behavior program that results in the evolution of ant–like foraging behavior. Our successful representation is an artificial neural network (ANN), with the new *connection descriptor* scheme for encoding both the connectivity pattern and the connection strengths in the genome.

**Partition** (Chapter 7) is aimed at applying the power of artificial evolution to the discovery of optimal solutions to high–dimensional functions, focusing on the discovery of optimal partitions of graphs. This study incorporates the idea of spatial structure (locality in competition and mating) from evolution theory into genetic algorithms. We perform a head–to–head comparison between conventional panmictic selection algorithms and spatially local selection algorithms using populations ranging in size from 8,192 to 524,288. The main result is that local mating results in significantly faster and more robust optimization. We also quantify some of the dynamic differences in the population during optimization between panmictic and spatially structured selection algorithms. Spatial structure enables the genetic algorithm to maintain greater diversity and examine more unique genotypes each generation. We also introduce a scalable graph partitioning problem, and use it to explore the effect of the neighborhood size used in local mating on the robustness of the genetic algorithm. We find that smaller neighborhoods result in more robust optimization. Also, for a given neighborhood size, increasing the population size makes the genetic algorithm more robust.

## 1.4 Comparison Among Evolution Mechanisms

In this section, we present an overview of three distinct evolution mechanisms that play central roles in this thesis: *natural evolution*, *genetic algorithms*, and *artificial evolution*. In general, genetic algorithms are an optimization technique inspired by natural evolution, but in most genetic algorithms there has been little or no effort to model natural evolution in detail. We define *artificial evolution* as a kind of genetic algorithm designed with the explicit intent of biological realism.

### 1.4.1 Natural Evolution

Biological evolution is the progressive change in genetic composition of a population over many generations. This change need not be adaptive. In fact, many population geneticists believe that most evolution may be adaptively neutral (Kimura, 1968; Hartl and Clark, 1989). The two principal components of the evolutionary process are *natural selection* and *random genetic drift* (Wright, 1931; Mayr, 1983). Natural selection is the process by which more "fit" individuals leave more offspring (on average) in succeeding generations than do the less "fit" individuals in the population (Darwin, 1859). (We discuss fitness in detail below.) Random genetic drift results from random events in the lives of the individuals making up the population. Such events include random mutations in the genetic material, the accidental death of an apparently highly fit individual before it gets an opportunity to reproduce, etc. Apparently highly fit individuals can get unlucky and leave few offspring in the next generation, and that low fitness individuals can get lucky and leave many offspring. The stochastic effects of genetic drift are most noticeable in small populations, where they can swamp the effects of even strong selection. In large populations the stochastic effects tend to "even out," so drift is less noticeable.

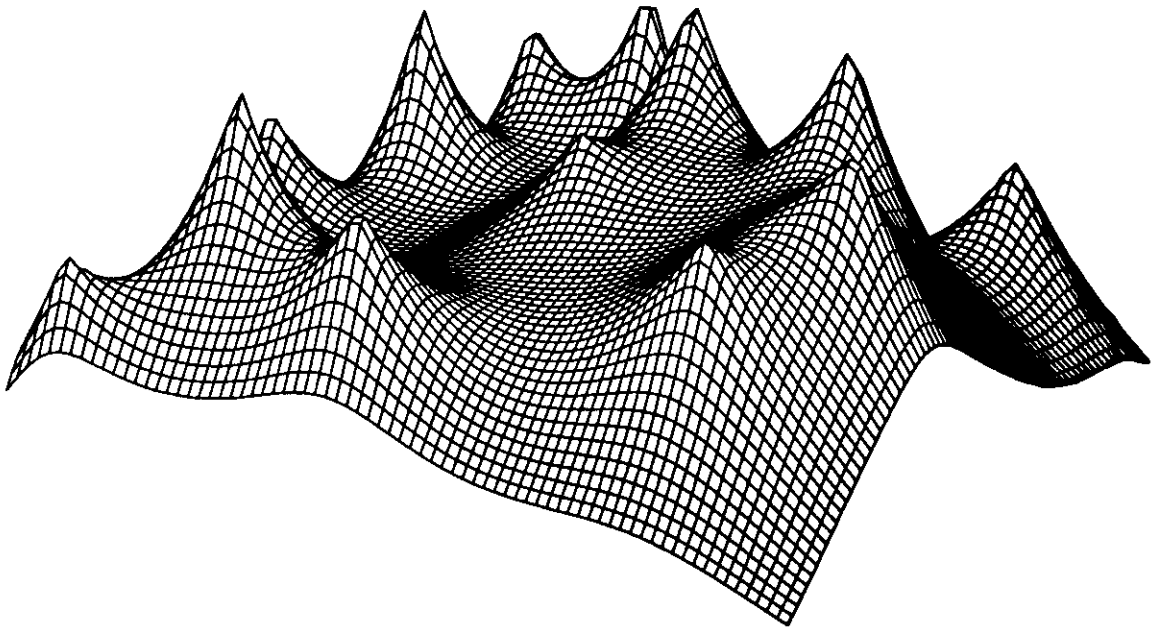

Figure 1.1: A 2-dimensional adaptive landscape. The height of the surface is the fitness for each allele ("gene") combination in a particular environment.

The idea of an *adaptive landscape* or *fitness surface* is a useful way of visualizing how selection acts on an evolving population (Wright, 1932; Kauffman and Levin, 1987). The basic idea is to plot fitness as a function over the space of possible genetic combinations for one species in a particular environment (Figure 1.1). The resulting graph forms the fitness "surface." The space of possible organisms may have thousands of dimensions, and the fitness surfaces are thought to typically have

a huge number of peaks and valleys, varying in height, because some allele ("gene") combinations are more adaptive than others.

A population is represented as a cloud of points on the fitness surface, one for each organism in the population; the more variation in the population, the more scattered it is on the adaptive landscape. The offspring of the more fit individuals in the population will tend to be both more numerous and more fit than the offspring of the less fit individuals in the population. In this way, over many generations natural selection tends to move populations uphill (higher fitness) in the adaptive landscape. Chance, the other component of the evolutionary process, moves the population randomly upon the fitness surface. If selection dominates the evolution of a population, it will tend to move up gradients in the adaptive landscape; if chance dominates, evolution tends to proceed in random directions.

In biology, *fitness* is defined as the relative ability to survive and reproduce in the context of a particular environment and gene pool. In nature, the fitness of an organism is often very difficult to determine, and the fitness contribution of a particular trait is even more difficult to calculate. Accurate fitness measurements are necessary because even very small ($< 1\%$) differences in fitness can have significant evolutionary consequences (Hartl and Clark, 1989).

Fitness should be regarded as an attribute of an entire genotype, rather than of any particular gene, trait, or phenotypic phase of an organism's life cycle. Although it is not unusual to refer to the fitness of a particular allele without regard to the rest of the genotype or the environment, this is technically an ill-defined concept. As Dobzhansky put it (1956, p. 340),

> It cannot be stressed too often that natural selection does not operate with separate 'traits.' Selection favors genotypes. The reproductive success of a genotype is determined by the totality of the traits and qualities which it produces in a given environment.

We might attempt to make sense of the notion of fitness of a single gene by determining the mean fitness of the genotypes in the population that contain a particular allele, and this might help us understand how the frequency of the allele might change. For example, if the genotypes containing the allele are of above average fitness, the allele might become more abundant due to natural selection. Unfortunately, this "mean fitness of an allele" calculation may actually tell us very little about the actual *effect* of the allele on fitness in general. In essence, this is simply a calculation of the *correlation* between fitness and the presence of the allele, not the causal contribution of that allele toward the organism's overall fitness. The allele may have no influence on the fitness of the genotypes in which it is found, and simply be there by chance; it may have effects that are strongly dependent on the presence or absence of certain other alleles in the genotype; or it may be in *linkage disequilibrium* (Hartl and Clark, 1989) with other alleles that have a much stronger influence on fitness. The term "linkage" usually refers to the association of genes on the same chromosome, but "linkage disequilibrium" refers to *any* non-random association of alleles, even if they reside on different chromosomes. Such associations can result from the interacting effects of the alleles (or interacting effects of alleles that are located nearby

on the chromosome). These interactions can take a number of forms such as mating preferences (see Chapter 3), or lethal allele combinations.

The fitness of a particular genotype is determined by events at all stages of the organism's life cycle. The most visible part of fitness is *viability*, which is the ability of an organism to develop and survive from a zygote (fertilized egg) to an adult. The culling of less viable organisms by natural selection is called *viability selection*. Another component of fitness is the ability to form mating pairs, known as *sexual selection*. Sexual selection can take the form of male–male competitions (e.g. head butting in bighorn sheep), male choice, female choice, or mutual choice via male-female interactions. Another component of fitness is called *meiotic drive*, which is due to non–random differences in the production of the various genetic combinations of gametes (unfertilized sperm and eggs) during meiosis (gamete formation). Another form of selection is *gametic selection*, which results from differential fertilization success among the gametes that have been produced. The *fecundity* of an individual, the number of zygotes (fertilized gametes) produced, is still another component of the overall fitness of a genotype. The combined effects of all of these components determine the fitness of a given genotype. For instance, an individual may be very viable (well adapted to its environment) and healthy as an adult, but fail to find a compatible mate due to sexual selection. Such an individual has zero fitness, despite its apparent health and vitality.

### 1.4.2 Genetic Algorithms

Genetic algorithms (Holland, 1975; Goldberg, 1989a) are a class of optimization algorithms loosely based on natural evolution. They are typically used to search for good or optimal solutions to complex optimization problems (Goldberg, 1989a). A solution is represented as a string of arguments that (more or less) maximizes the particular function to be optimized. This function is called the *objective* or *fitness function*. The string of arguments is analogous to the genome of a natural organism. The location of each parameter in the string is analogous to a locus on a chromosome, and the value of the parameter is analogous to the allele at that locus. The function is analogous to the environment faced by a natural organism, which determines its relative fitness.

A genetic algorithm evolves a population by assigning a numeric score to each string (genome) based on how well the genetically encoded arguments maximize the function. This is roughly analogous to viability fitness (the ability to survive to adulthood) in a natural organism. New strings are then created through a process of sexual reproduction between relatively high scoring strings that are currently in the population. The likelihood that a particular string will be chosen for mating is a function of its own score and those of the rest of the population. Sexual reproduction provides the offspring with a mixture of the genetic material from its two parents. In addition, small random changes in the genetically encoded arguments (analogous to mutations) are made to the offspring's string. The new string usually displaces a low–scoring individual from the population, or the whole population is simultaneously replaced by newly created strings.

The way biologists think of "fitness" is very different from the way the term is used in the context of genetic algorithms for optimization. The notion of fitness used in genetic algorithms is a numeric score used to determine the number of offspring that should be produced. To biologists, however, fitness is an emergent property that can only be measured *after* reproduction has already occurred, and is based on the number of viable, fertile offspring that actually were produced.

### 1.4.3 Artificial Evolution

We define *artificial evolution* to be a particular class of genetic algorithms. While the design of traditional genetic algorithms is driven by the goal of optimization, the intent of artificial evolution is biological realism. Artificial evolution genetic algorithms require a clear separation between genotype and the information encoded in the genotype. The genotype is represented as a linear string, and the genetic operators of recombination and mutation operate randomly at the lowest level of organization of the string, without reference to any syntactic nor semantic structure that may be encoded there. We view the genotype as encoding a program; the fitness of the genotype is determined by decoding and executing the program, perhaps in an environment that is shared by the other members of the population. Often, the selection and mating process of the artificial evolution genetic algorithm will include spatial structure.

#### Methodology

Simulations based on the observation that the execution of a computer program is very similar to the life of an organism have emerged in recent years (Taylor et al., 1989a; Taylor et al., 1989b; Fry et al., 1989; Coulson et al., 1987; Werner and Dyer, 1991; Jefferson et al., 1991). In these "life–as–process" simulations, each organism is represented by a program, as are the various environmental factors. Only the local interactions between the individual organisms and environmental factors are modeled directly, but based on them, the complex global behavior of the population emerges.

These "life–as–process" simulations separately represent each individual organism and environmental effect, and the biologically significant events in an organism's life are all separately simulated in detail. Each organism in the population is represented as a program, and its life is represented by a process (i.e. the execution of a program), a detailed sequence of events, including its birth, its interactions with a dynamic environment (potentially including many other organisms and environmental factors), its mating and reproduction (if any), and its death.

An *artificial evolution* simulation is a genetic algorithm that applies the low–level, detailed "life–as–process" model of each organism to a large, evolving population. The simulation consists of three main parts:

1. the genetic algorithm, which drives the evolution;

2. the organisms, each represented as a computer program of some kind; and

3. the environment in which the organisms live, represented by zero or more additional programs.

All of the thousands of programs that make up the simulation execute in parallel. The organism programs are heritable because they are encoded in the organism's chromosome. A genetic algorithm is typically used to drive the evolutionary process, but the behavioral repertoire of the organisms may include death, differential reproduction, mate choice, etc., in which case many of the components of the genetic algorithm are omitted. The environment of a particular organism may consist not only of one or more environmental processes, but also all of the organism programs from its own (and possibly other) species. In other cases the environment might be extraneous to the hypothesis that is being tested, and thus might not be explicitly simulated at all.

This dissertation describes a family of artificial evolution simulations. By executing an organism's program, the fitness of that individual is determined either implicitly (fitness is emergent) or explicitly (fitness is assigned by an external formula). Sometimes the processes execute in a shared environment, allowing for interactions with the other organism processes. In **Peacock** (Chapter 3), fitness evaluation is largely implicit, since we are modeling the evolution of mate choice. In **Parasite** (Chapter 4), fitness is evaluated explicitly based on the interactions between an individual in the host species and its infecting parasite. **AntFarm** (Chapters 5 and 6) performs explicit fitness evaluation of the ant colonies based primarily on the amount of food deposited in the nest by the ants behaving in a shared environment. **Partition** (Chapter 7) explicitly determines fitness based on the quality of the graph partition encoded in the genome. With the exception of **Partition**, the fitness of an individual depends on the genetic makeup of other individuals in the population.

### Advantages and Limitations

The major tools that are available for the study of natural evolution are the fossil record, molecular analysis, observational and experimental studies, and mathematical analysis. These approaches are inherently limited in some ways that computer simulations are not (Taylor et al., 1989a). The fossil record is notoriously biased, incomplete, and difficult to interpret. Also, while molecular studies can determine the underlying genetic similarities and difference of various species, these studies are time–consuming, and the results are often as difficult to interpret as the fossil record. Evolutionary experiments in the laboratory or field are usually limited to small populations and at most a few dozen generations because natural organisms (other than microbes) grow and reproduce slowly compared to the time scale of human experiments. In addition, such experiments are difficult to control and repeat, because of the complexity of the interactions between organisms and their environments. Taylor (1983) points out that many of the experiments you might want to perform have the potential to be very damaging to the ecology. And finally, mathematical analysis can completely describe only the simplest genetic systems.

In contrast, artificial evolution makes it possible to study nontrivial models of evolving systems over thousands of generations (macroevolution). Although these

models are inevitably idealized in some ways, they are much more complex and realistic than those that can be attacked mathematically. Computer simulations are also easily repeated and varied, with all relevant parameters under the full control of the experimenter. Some biologists saw the potential for computer simulations even 30 years ago (Crosby, 1963, p. 415):

> For some biologists, experiments with living organisms are hardly practicable. For example, many problems of evolution would obviously need too much time. As an alternative, experiments with realistic models of evolutionary systems would go far towards overcoming this difficulty, if a sufficient speed of operation could be achieved. This is where the electronic computer can become a valuable tool for population genetics.

Of course, a weakness of computer simulation is the inability to attain the full complexity of natural life and environments, although the degree of complexity that is feasible scales with improvements in available technology.

Most computer simulations in biology (including most previous attempts at simulated evolution) are based on solving differential equations from mathematical models (Swartzman and Kaluzny, 1987; Taylor, 1983), where the equations specify the dynamics of the system. Due to the difficulty of mathematical analysis, models of evolving systems are usually simple and unrealistic. Complex models that incorporate a large number of both intrinsic factors (e.g. the life history of the organisms) and extrinsic factors (e.g. weather, competitors, etc.) are more accurate and useful. But unfortunately, such complex evolutionary models are difficult or impossible to solve analytically. Differential equation–based models that are tractable are usually *linear* and of *low order*, and thus do not properly capture the discrete nature of the dynamics of real populations.

The distinctions between equation–based and the artificial evolutionary simulation paradigm are not new. They were described well by Crosby (1963, p. 415)

> The computer can simulate, in mathematical terms, complex genetical and evolutionary systems [equation–based simulations], or mathematics can largely be eliminated by forming, within a computer, electronic "organisms" possessing electronic "genes" [artificial evolution simulations]. These organisms can reproduce themselves in any way we choose, obeying the ordinary laws of genetics or any other desired pattern of heredity; while the natural variability of real biological systems can be imitated with fair realism by the use of computer–produced numbers (pseudo–random numbers) in sequences which effectively imitate true randomness.

Schull and Levin have noted the virtues of artificial evolution (Schull and Levin, 1964, p. 180):

> Digital computers and Monte Carlo solutions now afford us the opportunity to seek numeric answers to some of the problems which have thus far proven to be mathematically intractable. Simulation methods admittedly lack the appeal of explicit mathematical statements, but if one is

> pragmatic, these methods hold great promise for an early insight into a variety of interesting and important problems. In fact, at this juncture, it may well be that numeric analysis is more rewarding than an analytic, mathematical approach.

The typical macroevolution simulation that we describe in this dissertation contains at least tens of thousands of organisms and environmental processes, and lasts for thousands of generations. Before the introduction of massively parallel computers such as the Connection Machine-2 (Hillis, 1985; Hillis and Steele, 1986; Hillis and Barnes, 1987), such a detailed simulation was computationally infeasible. Where previous studies that resemble the artificial evolution style of simulation presented in this dissertation have been applied to evolving systems, the simulations have been simple and operate on small populations and for a relatively small number of generations (Crosby, 1963; Schull and Levin, 1964; Ohta, 1987; Taylor et al., 1989a; Ohta, 1989; Keightley and Hill, 1989; Ohta and Tachida, 1990). While these simulations produced interesting results, the computational costs have been generally the limiting factor on the complexity of the simulation models that have been attempted. The costs apparently were too great to attempt artificial *macro*evolution.

## 1.5 Artificial Evolution Applications

### 1.5.1 Biological Applications

While we cannot expect artificial evolution to reconstruct an actual scenario in the history of natural life, we can explore particular hypotheses about evolution, eliminating some and giving credence to others. Simulations provide an artificial world in which to perform evolutionary experiments that can be fully controlled and repeated, and can span thousands of generations.

There are a large number of important macroevolutionary problems that traditional biological techniques cannot readily address. Artificial evolution experiments might be used to shed light on a number of open evolutionary problems, including

- modes of speciation (which of many hypotheses are more likely, and in which sexual systems and ecological situations);
- evolution of mutation and recombination rates (under what conditions is there not strong selection for reduced rates);
- evolution of information processing behavior (e.g. sensory–motor integration, communication, etc.);
- evolution of sexual reproduction (under what conditions is it maintained in competition with asexual reproduction?);
- punctuated equilibria (i.e. is it true that most evolution occurs at speciation events, and not within species?);

- dynamics of the evolution of predator–prey relationships and competitive "arms races;"
- influence of host–parasite interaction on evolution;
- stability of ecosystems;
- evolution of evolutionarily stable strategies (ESS);
- sexual selection and the evolution of maladaptive characteristics; and
- evolution of cooperation (especially among kin).

In this dissertation, we present simulation results for a subset of these problems. By simulating these phenomena, we demonstrate that artificial evolution can usefully augment analytic modeling in population genetics.

Population geneticists are constantly stymied by the inherent limitations of mathematical analysis of genetic systems. In formulating an analytic model, it is nearly always the case that a large number of simplifying assumptions about the genetics and life history of the organism, details of the genetic and population dynamics, etc. must be made in order to obtain a solvable model. Even when such an analytic model is derived, it can generally only be solved for the equilibrium state. In fact, the simplifying assumptions often make the model inapplicable to any known population. This leaves us with two questions:

1. do the conclusions drawn from the model still hold when applied to populations with some of the simplifying assumptions relaxed (i.e. is the model robust)?, and
2. what are the evolutionary dynamics away from the equilibrium?

Large–scale artificial evolution can be used to evaluate both the robustness and non–equilibrium dynamics of a much–simplified analytical model (e.g. see Chapter 3). Variations of these simple models are generally simple to add to the simulation, while they are essentially impossible to handle analytically. For example, an infinite population can be replaced by a finite population, haploid genetics (one copy of each gene) can be replaced by diploid (two copies of each gene) genetics, two possible alleles per locus by multiple alleles, single locus traits (traits determined by one gene) by polygenic traits (traits determined by many interacting genes), panmixia (mating with any other individual in the population) by spatial structure (mating only between nearby individuals), ecologically independent organisms by ecologically interacting ones, etc., and any combination of these extensions. Traditional population genetics techniques cannot reasonably handle these variations.

In addition, the non–equilibrium dynamics (e.g. the dynamics of the invasion of a new allele into a population) can easily be explored. Again, this is possible using traditional simulation techniques in the cases where the appropriate equations can be derived, but there is nothing special about exploration of the non–equilibrium dynamics as far as simulation is concerned. Even if the equilibrium predictions of

the analytic model are found to be robust under a large number of variations, these variations may dramatically affect the evolutionary dynamics away from equilibrium in very different ways.

Consider the invasion of a new allele into a population with a two–allele, haploid genetic system. Let $p$ be the frequency of the invading allele, and $p'$ be the frequency of the phenotypic trait produced by the new allele. Because there is only one copy of the allele in each individual (haploid genetics), the new trait is expressed phenotypically at a rate equal to the allele frequency ($p' = p$), and thus will have an opportunity to begin affecting the population dynamics immediately. Now, consider the same situation, but with a diploid genetic system and a recessive allele. In diploid genetics, there are two copies of each chromosome, with potentially different alleles at each locus. When an allele is recessive, its effects are masked by the other allele when one copy of each is present in an individual. Because the two alleles are different, such an individual is referred to as heterozygous. The recessive allele is only expressed in individuals carrying two copies of the allele (homozygous recessive). Therefore, in a diploid panmictic population, $p' = p^2$. As the allele invades, it is unlikely to affect the evolutionary dynamics until genetic drift causes $Np'$ (where $N$ is the population size) to approach 1, in which case it becomes likely that individuals expressing the new trait will begin to appear in the population each generation. While the equilibrium phenotype frequencies for the haploid and diploid cases may be identical, the dynamics of the evolution away from the equilibrium are clearly different.

In this haploid vs. diploid case, it is clear that varying this component of a model will lead to different non–equilibrium dynamics, even without doing a simulation. However, aspects of evolution such as mutation rates, linkage, migration rates, spatial structure, sex–linked traits, etc. can have much more subtle effects on the non–equilibrium dynamics, even if they do not affect the equilibrium state. These variations can interact, making analysis very difficult. However, all of these variations are relatively easy to implement simultaneously in our simulations.

### 1.5.2 Evolving Artificial Organisms

We also use artificial evolution simulations to evolve artificial organisms that live and reproduce in relatively complex environments and that possess many sensors (both internal and external) and effectors. Our **AntFarm** simulations fall into this category. The organisms may possess some amount of internal memory, allowing their behavior to be history sensitive. In the course of its life, each organism is born, makes thousands of decisions (eat, move, mate, etc.), and eventually dies. As in natural life, the reproductive success of an artificial organism is affected by its behavior throughout its lifetime. Although we focus our attention towards the evolution of the behavior of the organisms in this dissertation, artificial evolution also allows for heritable morphological features, which might be sensed by nearby organisms or affect the behavior of the organism.

## Representing Artificial Organisms

A major issue in evolving artificial organisms is the organism representation. In the artificial evolution paradigm, we view the life of an organism as a process (an executing program). We need to represent the program at two different levels (Figure 1.2): as a genotype and as a *behavior function* (an executable program). We store the genotype as a linear bitstring, so that biologically realistic recombination and mutation can be performed. But what is an appropriate representation for the behavior function? How can we encode this program in a bitstring such that adaptive evolution of the behavior function is possible?

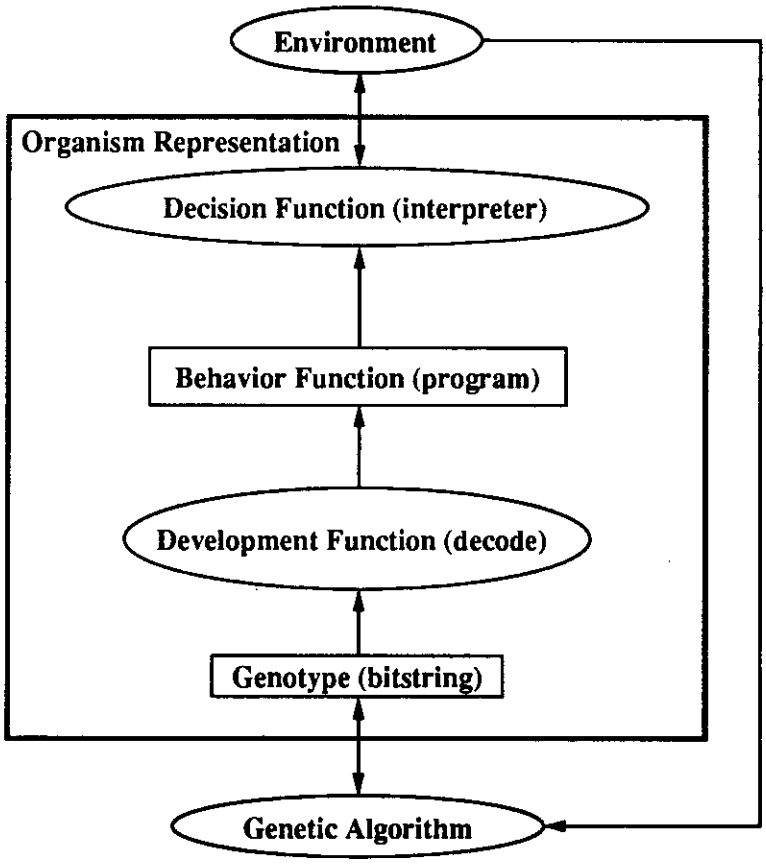


Figure 1.2: The organization of a simple organism representation. The passive data structures are represented as rectangles, and the active processes as ovals. The arrows indicate the flow of data. The genotype is decoded by the development function into the behavior function, which is interpreted in the context of the environment. The environment provides the genetic algorithm with a fitness score for the genotype.

For this kind of simulation, we need a behavior function representation that scales well to relatively complex sensory/memory/motor capabilities and highly varied behavior. At the same time, the representation must not require us to explicitly encode any of our knowledge about the artificial world or possible adaptive behaviors into the representation before the evolution begins. If we fine-tune the low-level details

of the behavior function for a particular artificial world, we will almost certainly bias the direction of the evolution, which will invalidate our evolutionary experiments.

Once we decide on the representation for the behavior function, we must provide each organism with an *interpreter* that will execute its behavior function, and a *development function* that decodes the genotype to produce the behavior function (Figure 1.2). The development function may perform a simple mapping, or it may be a complex process that interacts with the environment, the genotype, the partially elaborated behavior program, and its own internal state. In the latter case, the development function will also require an interpreter to execute the development program. Much of the development function may itself be encoded in the genotype. The genotype may also encode genes modifying the expression of various properties and morphological features of the organism. The behavior function of an organism, determined initially by the genotype and execution of the development function, can in principle change during an organism's lifetime if there is some provision for learning, growth, or other dynamic update of the behavior function. The issue of the representation of artificial organisms is discussed in detail in Chapters 5 and 6.

Of the potential representations that we survey in Chapter 5, the artificial neural network (ANN) representation appears to be most appropriate as the basis for evolving the behavior of complex artificial organisms. However, our experience is that the ANN encoding methods that have been used successfully in previous work do not permit evolution to find efficient foraging behavior in the **AntFarm** world. We present the connection descriptor ANN encoding scheme, a radically different encoding that empirically works with **AntFarm**, resulting in the evolution of efficient foraging behavior.

### The Artificial World

An artificial world consists primarily of a set of programs that simulate each component of the environment. In addition, it may have to maintain artificial physical invariants (the "laws of nature"). In the examples presented in this dissertation, the environment programs tend to be simple in comparison to the organism programs, and may not even be present if the environment is not very active.

Our environment data structure consists of a large toroidal grid of locations. The environment programs usually perform tasks such as updating the local food supply, diffusion of pheromones (chemicals used for communication), etc.

The maintenance of the physical invariants of an artificial world is an important and difficult problem, complicated by the fact that our simulations are implemented in parallel on the Connection Machine. This problem arises because we embed one simulation, the artificial organism programs, within another simulation, the artificial world. If we were hand-coding interacting programs (the organisms) that share a data structure (the artificial world), we would write them in such a way that they would cooperate and synchronize their access and update of the shared data structure. Unfortunately, the organism programs that evolve are not so well-behaved, and frequently attempt to violate the constraints that the world places on their behavior. Thus, instead of allowing the organism to actually do whatever the behavior func-

tion specifies, we view the output of the behavior functions of the artificial organisms as *potential* actions, which may or may not take place. A potential action actually occurs only if it is feasible within the artificial physics. When the potential action of multiple organisms conflict or interact, we must arbitrate all of the conflicts and determine the effects of the interactions in parallel.

For example, in **AntFarm**, an environment grid square can contain a pile of indivisible food particles, and multiple artificial ants can simultaneously occupy the square. Conflicts occur when several organisms attempt to grab the same piece of food. In this case, we must arbitrate (in parallel) among the contending individuals to determine which ones actually succeed in obtaining food.

An example of interacting (rather than conflicting) potential actions occurs when several organisms simultaneously drop pheromones (chemicals used by some animals for communication) at the same location. Here the actions do not exclude one another; they combine. We must combine the effect of these actions (in this case, by adding them) in parallel.

### 1.5.3 Engineering Applications

Engineers have used genetic algorithms for many years to search for optimal solutions to a variety of functions. By making use of large, spatially structured (local mating) rather than small, unstructured (panmictic) populations the genetic algorithms that are used in this dissertation are more closely modeled after biologically realistic evolution than more conventional genetic algorithms. Chapter 7 demonstrates that local mating leads to a more robust genetic algorithm, even on function optimization problems. In addition, multiple solutions can be simultaneously supported by the spatially structured population and optimal solutions are discovered faster, both in terms of number of generations to an optimal solution and computation time in a particular massively parallel implementation on the Connection Machine-2.

Much of the computational technology that we develop in this dissertation is directly applicable to engineering problems. In Chapters 5 and 6, we discuss several techniques for encoding computer programs in a form to which evolution can successfully be applied. Although several researchers have successfully evolved computer programs (including ANNs) using a variety of representations, all of those that we have examined either do not scale to complex functions or require the incorporation of problem– or solution–specific information. The use of solution–specific information might be acceptable for some engineering applications, but in any case, we introduce an ANN program encoding scheme that suffers from neither of these problems.

# Chapter 2

# Design and Implementation

As we described in Chapter 1, our abstraction of an evolving population of artificial organisms is a set of concurrently executing computer processes, with a separate process implementing each individual organism and each environmental factor. We explicitly represent each individual and each one of its genes, and we simulate in detail all portions of the organism's life history that are important to the experiment. At the very least, this includes birth, mating, death, and genetic events such as recombination and mutation; but it may also include numerous interactions with a complex environment, including other individuals in the simulation.

To simulate evolution realistically, we must deal with large populations—on the order of tens of thousands to millions of individuals in each generation. Also, because we are interested in long–term, macroevolutionary time–scales, we usually run each simulation for hundreds or thousands of generations. Since these simulations are computationally enormous in several dimensions, we have implemented all of our simulations on the Connection Machine-2 (Hillis, 1985; Hillis and Steele, 1986; Hillis and Barnes, 1987), which is the fastest computer that is readily available to us.

The UCLA Connection Machine-2 has 16,384 1–bit processing elements (one quarter of a full configuration) running at 7 Mhz, each of which has 65,536 bits of bit–addressable local memory (small memory model). In addition, for every 32 processors there is a single–precision (32–bit) floating point accelerator, for a total of 512 on our machine. Both random access global interprocessor communication and regular local communication are available. Local communication is consistently fast, while the performance of global communication is data–dependent and typically one to two orders of magnitude slower than local communication. The Connection Machine-2 is a single instruction multiple data (SIMD) architecture, meaning that there is a single, global instruction stream for all of the processors, and all processors are synchronized between instructions. UCLA is running version 6 of the Connection Machine software, and drives the Connection Machine with a Sun 4/330 front end running SunOS version 4.

While the simulations could have been run on any machine, some of our design decisions were influenced by the data–parallel, synchronous execution model presented by the Connection Machine-2. For instance, we typically map one individual in the population onto each of the Connection Machine processors. When more processors

are needed than are physically present, the Connection Machine software transparently operates in *virtual processing mode*, with each processor emulating multiple virtual processors. Due to the synchronous nature of the Connection Machine-2, we keep the generations synchronized; the basic steps of the simulation are each applied to every member of the population simultaneously, and at the end of each generation we simultaneously replace all individuals in the population with the next generation of offspring. In addition, we nearly always maintain a constant population size, because there are significant computational costs associated with simulating population sizes that are not a power of two on the Connection Machine-2.

C++/CM++ (Collins, 1990) is the implementation language for all of the simulations that we describe in this dissertation, and the simulations are compiled with version 1 of the GNU C++ compiler. The low-level genetic algorithms code, such as for handling chromosomes and the genetic operators of recombination and mutation, instrumentation, etc. is shared among the various simulations. This basic library consists of 5700 lines of code, 37% of which is devoted to instrumentation and run-time data analysis.

## 2.1 Overview of Genetic Algorithms

In this section, we give a brief overview of genetic algorithms. We are looking toward the design of a genetic algorithm that will operate on large populations (tens to hundreds of thousands of individuals) and will provide dynamics that are similar to those of populations subject to natural evolution. In addition, the genetic algorithm must be able to sustain variation for thousands of generations, so that we can simulate and study macroevolutionary phenomena.

Each of our simulations evolves one or more populations via a genetic algorithm (Holland, 1975). Genetic algorithms are loosely modeled after natural evolution, and have been used by computer scientists and engineers as an optimization technique for more than 25 years. A genetic algorithm evolves a population of strings of function arguments (chromosomes) by assigning each a "fitness" score, which is based on the value of the function when evaluated with those arguments. The likelihood that a particular string will be chosen for reproduction is a function of its own score and the scores attained by the others in the population. The genetic search works by biasing reproduction towards higher scoring strings, and reproduce strings with variation.

The mechanics of one generation of a genetic algorithm are relatively simple, consisting of four phases:

1. Evaluation: assign a fitness score to each string;

2. Selection: select strings that will reproduce (biased toward high scores) and pair them for sexual mating;

3. Recombination: generate an offspring string from each mated pair by recombining the parent's strings; and

4. Mutation: mutate each offspring string.

New populations are created by the repetition of these steps. The assignment of fitness scores is wholly dependent on the particular application. Because selection is biased towards strings with higher scores, the populations typically achieve higher and higher scores as generations pass. Recombination and mutation provide for reproduction with variation. Recombination of the parent strings provides the offspring with a mixture of the genetic material from its parents, and mutation then makes small, random changes to the new offspring.

The literature of genetic algorithms has been of little help in designing a genetic algorithm for sustained, realistic evolution of large populations. Most of the empirical experience in the literature is based on small populations (30–200 individuals per generation), sequential execution, static fitness functions, and restarting with a new, random population whenever premature *convergence* (loss of genetic variation) occurs (Goldberg, 1989a; Belew and Booker, 1991; Schaffer, 1989; Grefenstette, 1987). In contrast, we use large populations ($10^4$–$10^6$ individuals per generation), massively parallel execution, dynamically changing fitness functions (if we have an explicit fitness function at all), and we must be able to sustain evolution for thousands of generations (no restarting on convergence).

Traditional genetic algorithms appear to be modeled on Fisher's (1930; 1958) view of evolution, employing *panmictic* selection and mating schemes, meaning that each individual competes globally with all others in the population during the selection process, and a selected individual can potentially be mated with any other selected individual in the population. One of Fisher's basic assumptions is that populations are large enough that the stochastic effects of random genetic drift can safely be ignored. In practice, most of the empirical genetic algorithm studies that are found in the literature violate this assumption.

Most genetic algorithms select individuals to become parents probabilistically based on their scores, with the score defined by the *objective* or *fitness function*. Many define the probability that string $i$ is chosen to be a parent of a particular offspring as

$$P(i) = \frac{s_i}{\sum_{i=0}^{N-1} s_i} \tag{2.1}$$

where $N$ is the population size, and $s_i$ is the score of individual $i$. This particular selection method is known as stochastic selection with replacement. Linear scaling (based on the population maximum and mean scores) is often applied to the fitness scores, because the use of absolute scores can lead to a number of problems (Goldberg, 1989a).

Although panmictic selection schemes such as stochastic selection with replacement are widely used in genetic algorithms, panmixia is a poor model of a natural population (Wright, 1931). Panmictic genetic algorithms are ill–equipped to search for successful genotypes in large, multimodal adaptive landscapes (Goldberg and Richardson, 1987; Deb and Goldberg, 1989), because the population is unable to maintain radically different, high–scoring genotypes due to convergence. Random genetic drift and/or strong selection invariably causes the population to converge on

only one of the possibly many peaks in the adaptive landscape, and often this fixation occurs at a suboptimal peak. Once the whole population is located at a single peak, selection is likely to keep it there, preventing further adaptation.

In the small populations that have formed the basis for most of the empirical work with genetic algorithms, very strong selection is necessary to avoid domination of the evolution process by drift. (The strength of drift is inversely proportional to the population size.) If weak selection were used and drift were allowed to dominate, the evolution would proceed in a nearly random direction to fixation, almost certainly on a genotype that is far from optimal. (See (Harvey, 1991) for an example of this phenomenon). Whether selection or drift dominates, fixation (convergence) is the nearly inevitable result in a panmictic population (Goldberg, 1989b). In the absence of epistasis (the interaction of alleles at multiple loci), selection on a haploid locus is always directional and therefore drives the locus to fixation (in an otherwise ideal population, e.g. with no mutation, migration, etc.). In the presence of epistasis, it is possible for the 0 allele at a locus to be favored in the context of one set of alleles, and the 1 allele to be favored in another context. In small populations, random genetic drift will rather quickly choose one of these allele combinations over the other and still drive the locus to fixation.

While the presence of mutation does provide the population with an opportunity to escape from convergence to a suboptimal genotype, a small population is unlikely to do so unless there is strong selection in favor of the mutant allele (in which case it will quickly reconverge to a genotype that contains this new allele). The usual method for dealing with a converged population is to restart with a new, random population. While this is prudent in an engineering domain, it is an option for neither natural evolution nor our artificial evolution (because we must be able to simulate natural evolution over the course of thousands of generations).

Not only are panmictic genetic algorithms susceptible to premature convergence, they also are not well suited for a massively parallel implementation, because the survival and mating success of each individual involves global knowledge of the the population (Equation 2.1). A fully distributed algorithm that requires only local information leads to a faster parallel implementation. The Connection Machine-2 general communication performance is good enough that our implementation of some of the common panmictic selection algorithms only runs a factor of 2–5 slower (per generation) than our local mating algorithms. However, panmixia is less robust and requires many more generations to find optimal solutions (see Chapter 7 for a head–to–head comparison).

## 2.2 Avoiding Premature Convergence in Genetic Algorithms

Premature convergence (fixation on a suboptimal genotype) has been a constant problem in traditional genetic algorithms. Several variations on genetic algorithms intended to deal with the problem of convergence can be found in the literature, most motivated by nature's example. Natural populations avoid convergence because

of population structure (different subpopulations use somewhat different strategies), speciation (different species fill and exploit different environmental niches), and large size. These phenomena keep natural populations spread across (and exploring) a number of different peaks in the adaptive landscape.

De Jong introduced into genetic algorithms a scheme called *crowding* (De Jong, 1975). In his work, overlapping generations are used, and new individuals replace existing strings that are similar to themselves (based on genotypic similarity). Individuals compete for space in the constant-sized population with the others at the same adaptive peak, so one genotype (peak) will not take over the whole population. Even when crowding is used, a gene-pool that has the whole population residing at a single peak in the adaptive landscape is stable. In the long run, even in a large populations, genetic drift will result in the whole population residing at the same fitness peak, so crowding does not fully solve the problem of convergence.

Another approach is to use a *sharing function* (Goldberg and Richardson, 1987), based on the idea that phenotypically similar individuals must share limited resources because they are exploiting the same niche. The absolute fitness score of an individual is reduced by an amount proportional to the number of "similar" individuals in the population (before Equation 2.1 is applied). The degree of similarity is calculated in a domain-dependent way based on the phenotype. Phenotypic sharing is inherently problem-specific, difficult to define, and almost certainly inappropriate for a massively parallel implementation.

In conjunction with sharing, the phenotypic comparison function can be used to enforce *restricted mating*: only similar individuals are allowed to mate. This simulates inbreeding and *positive assortative mating* (Crow, 1986). Restricted mating is used to avoid the production of low-fitness offspring that result from matings between diverse subpopulations (the offspring lies in a valley in the adaptive landscape between the fitness peaks where the parent genotypes are located).

A newer approach is the strategy of *incest prevention* (Eshelman and Schaffer, 1991), which works to prevent premature convergence by preventing genetically similar individuals from mating (forced outbreeding, or negative assortative mating). Good optimization performance (defined in terms of number of fitness function evaluations required to find an optimal solution) on standard genetic algorithm test functions has been achieved with incest prevention, because it uses elitist reproduction (the best individuals discovered are always retained), and multiple copies of a genotype are not kept. These features allow the genetic algorithm to employ very disruptive reproduction algorithms, i.e., high recombination rates, high mutation rates, etc., because the offspring only is added to the population if it has a relatively high fitness. Because no duplicate genotypes are allowed in the population, complete fixation (convergence) is not possible. This genetic algorithm retains individuals on different peaks of the adaptive landscape, and causes offspring to be generated over a large portion of the space of possible genotypes. For this reason, the mean fitness of the individuals examined by this genetic algorithm will tend to be low. Incest prevention is not appropriate for realistic simulated evolution, because it is not motivated by natural evolution, does not have dynamic properties similar to natural evolution,

and is not well suited for a massively parallel implementation (due to the centralized control implicit in the algorithm).

While crowding, sharing, and restrictive mating are all motivated by natural examples, they too are inappropriate for realistically simulating evolution. One problem is that the genetic algorithm must be defined in a domain–independent way, with no built in knowledge of either the organisms nor the environment. This means that phenotypic comparisons cannot be performed, because they are highly domain–dependent. In addition, these techniques are not easily implemented in parallel, because global knowledge of the population is still required (in fact even more detailed knowledge is required).

## 2.3 Spatial Structure in Nature and Genetic Algorithms

The basic methodological problem with convergence avoidance techniques such as crowding, sharing, etc. is the attempt to directly impose high–level system behavior. Nature does not appear to have a widespread convergence problem, even after three billion years of evolution. The need for special methods for avoiding convergence in genetic algorithms suggests that there is are basic problems in the way evolution is being modeled.

For much of this century, there has been discussion and controversy in the population genetics community between the camps supporting the two main theories of evolution (Provine, 1986; Hartl and Clark, 1989). On one side is Fisher with the modern synthesis of the theory of evolution by natural selection (Fisher, 1930; Fisher, 1958), and on the other is Wright with his shifting balance theory of evolution (first described in (Wright, 1931) and fully expressed in (Wright, 1968; Wright, 1969; Wright, 1977; Wright, 1978)). Fisher and Wright differ primarily on the question of the prevalence of panmixia in natural populations (Provine, 1986; Hartl and Clark, 1989). Fisher believed that effectively panmictic populations are the rule, while Wright saw them as the exception. We tend towards Wright's point of view, believing that population structure is prevalent and is an important force in evolution (see Chapter 7).

Spatial structure has the effect of dividing a large population into a number of *demes* (Gilmour and Gregor, 1939). Demes are genetically semi–independent subpopulations that remain loosely coupled to neighboring demes by migrants. The local stochastic effects (drift) associated with spatial structure can have dramatic effects on the composition of the gene pool of a population (Wright, 1943; Kimura and Weiss, 1964; Kimura and Maruyama, 1971). The relatively small size of the demes allows drift to play an important role in the evolution of the population, without driving the whole population toward fixation. Even if drift were to drive every local subpopulation to fixation, each deme would be fixed on a different genotype, maintaining diversity in the population as a whole. The simulations described in this dissertation generally use populations of tens of thousands to hundreds of thousands of individuals, along with some sort of spatial structure. We measure the significant differences

in the evolutionary dynamics of panmictic and spatially structured populations in Chapter 7.

Wright's theory is based on the idea that a "shifting balance" between the forces of selection and drift allows rapid evolution. In order to exploit the shifting balance theory in genetic algorithms, we must structure our population into demes. A number of deme/migration models have been proposed and analyzed by population geneticists (Hartl and Clark, 1989), usually categorized as *island*, *stepping stone*, or *isolation by distance* models. In general, the island models have relatively large demes, with arbitrary migration patterns between the islands. The stepping stone models have demes arranged in a lattice, with migration between nearest neighbors. Isolation by distance is characterized by a nearly continuous population and environment, where overlapping demes are defined only in terms of geographical distance - the probability that a given parent will produce an offspring at a given location is a decreasing function of the distance between the parent and offspring locations.

A number of parallel genetic algorithms on coarse–grained multiprocessors have been designed around the island or stepping stone models (Pettey et al., 1987; Tanese, 1987; Pettey and Leuze, 1989; Tanese, 1989; Cohoon et al., 1991). These implementations appear to be motivated by a desire to parallelize the genetic algorithm, rather than to avoid convergence, and generally use only a small number of demes (one per processor on modestly parallel systems) and relatively small overall population sizes ($\sim 10^3$ individuals). Kimura and Weiss (1964) have analyzed the stepping stone model in detail. Let $N$ be the number of individuals in each subpopulation, and $m$ be the fraction of individuals in a subpopulation that are immigrants in any particular generation. Kimura and Weiss have derived analytically that in the stepping stone migration model, a population is effectively panmictic if $Nm > 4$, and that the subpopulations will undergo significant genetic divergence only if $Nm \ll 1$. In (Pettey et al., 1987; Pettey and Leuze, 1989) $Nm = 4$, so the spatially structured population will actually evolve as if it were undergoing nearly panmictic selection and mating. In (Tanese, 1987; Tanese, 1989) a variety of migration rates are used, but the subpopulation sizes are so small (in some cases only 2 individuals) that random genetic drift almost certainly has overwhelming effects within each deme. It is also important to note that the dimensionality of the lattice in the stepping stone model of migration has a large effect (Kimura and Weiss, 1964, p. 576)

> The tendency toward random local differentiation is very much dependent on the number of dimensions; it is strongest in one dimension and becomes weaker as the number of dimensions increases.

In an environment without geographical boundaries, genetically semi–isolated demes can emerge as a result of slow gene–flow across long distances. This process is known as *isolation by distance* and arises where the probability that a given parent will produce an offspring at a given location is a fast–declining function of the geographical distance between the parent and offspring locations. For isolation by distance, the important parameter is the effective size of the neighborhood from which the parents of an individual are likely to be drawn. Wright has analyzed the isolation by distance model (Wright, 1943, p. 124):

> [For a two dimensional area] there is a great deal of local differentiation if the random breeding unit is as small as 10, even within a territory the diameter of which is only ten times that of the unit. If the unit has an effective size of 100, differentiation becomes important only at much greater relative distances. If the effective size is 1000, there is only slight differentiation at enormous distances. If it is as large as 10,000 the situation is substantially the same as if there were panmixia throughout any conceivable range.

As is the case in the stepping stone model, the dimensionality of the population's range is very important (Wright, 1943, p. 124):

> The situation is very different ... in a species whose range is essentially one dimensional (for example, a shore line). Different alleles may approach fixation in different parts of a range only 100 times the length of the random breeding unit if the effective size of the latter is less than 100. The range must be about 1000 times the length of the unit if the latter has a size of 1000 and about 10,000 times its length if the size of the unit is 10,000 to give this result.

and (Wright, 1943, p. 137)

> With linear continuity, there is enormously more differentiation that with area continuity.

Wright's theory of evolution and the creation of demes in continuous populations via isolation by distance has played a role in the design of several parallel genetic algorithms (Mühlenbein, 1989; Gorges-Schleuter, 1989; Manderick and Spiessens, 1989; Collins and Jefferson, 1991a; Mühlenbein et al., 1991; Spiessens and Manderick, 1991; Davidor, 1991; Collins and Jefferson, 1991c). Mühlenbein and Gorges–Schleuter have each individual mate with one of four neighbors with the offspring replacing the local parent. They employ an effectively one dimensional grid of organisms, so they should (and do) observe spatial differentiation quite clearly, even with the small population sizes they use (64 individuals).

Manderick and Spiessens (Manderick and Spiessens, 1989; Spiessens and Manderick, 1991) place their populations in a two dimensional grid and choose both parents from a neighborhood of size 4 to 49 individuals, with total population sizes ranging up to 4,096. On easy optimization problems (where maintaining variation is not critical) they observe faster optimization with the larger neighborhoods, while on more difficult functions the smaller neighborhoods provide the best results. These results are consistent with Wright's analysis. On easy problems, where even a traditional panmictic genetic algorithm is unlikely to get stuck in a local optimum, it is best to focus the maximum effort of the search in the area of the best individuals found so far, so fast gene flow and thus little spatial differentiation results in successful and fast optimization. On the harder problems, where it is necessary to maintain genetic variation in the population for many generations in order to find the solution, small neighborhoods restrict gene flow sufficiently to allow spatial differentiation. We explore this

effect empirically in Chapter 7. Davidor (1991) introduces a relatively complex two dimensional grid-based genetic algorithm that employs overlapping neighborhoods of 9 individuals, and explores the dynamics of niche formation.

## 2.4 The Artificial Evolution Genetic Algorithm

Most of the genetic algorithms we use for simulating evolution differ in many details from more traditional genetic algorithms. Therefore, we are devoting the rest of this chapter to a detailed discussion of the design and implementation of the genetic algorithms that form the basis for our simulations.

Although panmictic genetic algorithms can, at some computational cost, be used in a massively parallel implementation (see Chapters 3 and 7), most of our example simulations make use of the spatial structure of Wright's shifting balance theory. We have found that spatially structured population results in a genetic algorithm that is much more resistant to convergence, is a close approximation of natural evolution, and requires only local information, allowing a fast massively parallel implementation (Chapter 7).

It is not uncommon in applying genetic algorithms to engineering problems to tailor the genetic operators of recombination and mutation to the particular problem. However in our work, we require a clear separation between the genetic algorithm and the simulated world, in an effort to limit the possibility of introducing biases into our evolutionary experiments, and because it is so in nature. This means that the operations of selection, pairing of mates, recombination, mutation, etc. must be defined without detailed knowledge of the environmental "problem" that is driving the evolution nor knowledge of the details of the genetic encoding (representation) of the "solution."

Our genetic algorithms typically maintain a constant population size with no age structure (i.e. generations are synchronized and do not overlap). The initial generation usually consists of randomly-generated bitstring chromosomes. As the first step in each generation, a fitness score is assigned to each individual in the population (based on its whole life, if it is an artificial organism). The organisms are logically located in a grid, one individual in each square. Spatial structure limits competition and mating to organisms that are nearby in the grid. The parents of each offspring are chosen based on the fitness scores of the individuals in the neighborhood of the location where the offspring will be created.

### 2.4.1 Selection/Mating

In this section, we briefly describe the implementation of the important classes of selection/mating algorithms that we use in our simulations, including isolation by distance, stepping stone models, and island models. In general, we map one individual to each processor of the Connection Machine.

To simulate isolation by distance in the selection and mating process, we place the individuals on a toroidal, 1 or 2 dimensional grid, with one organism per grid

location. For the sake of a concrete discussion, we will assume a 2 dimensional grid. Selection and mating take place locally on this grid, with each individual competing and mating with its neighbors. In his quantitative analysis of isolation by distance, Wright assumes a normal distribution for parent–offspring distances (Wright, 1969, p. 303).

> Normal distributions of parents relative to offspring are to be expected if dispersion occurs by a long succession of random movements.

In one of our isolation by distance selection schemes, the two parents of the offspring at location $l$ in generation $g$ are the highest scoring individuals encountered during two random walks that begin at location $l$ and compare potential parents from generation $(g-1)$, with one parent chosen per walk (Figure 2.1). The random steps that are performed are potentially different for each offspring, and each step occurs in parallel across the whole population. The parents are chosen with replacement, so it possible for the same individual to act as both parents for the offspring. $R$, the length of the random walks, is a measure of the deme size (and thus the rate of gene flow). This is similar to the method used by Hillis (1991).

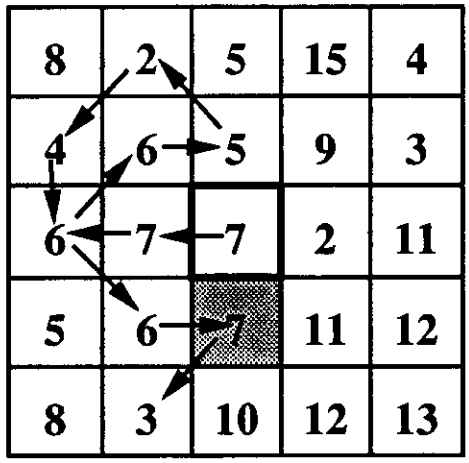


Figure 2.1: Isolation by distance is implemented by choosing parents along random walks from the offspring location. The number in each square is the fitness score of the individual at that location. A random walk of length $R = 10$ is shown. The walk begins in the center square (where the offspring will be created), and each step can be in any one of 8 directions. The steps of this walk are indicated by arrows, and the chosen parent is highlighted. The parent is the highest scoring individual that is encountered on the walk, breaking ties in favor of those discovered later in the walk.

We have examined two policies for breaking fitness score ties during the random walks: (1) take the first encountered and (2) take the last encountered. In the absence of selection, all individuals have the same score. In this case, the former strategy would always choose the individual at the offspring location as both parents of the offspring for the next generation. The latter strategy would always choose the last individuals encountered on the two random walks, producing a normal distribution of distances between parent and offspring locations (which is assumed by Wright).

The last–encountered policy appears to be the better way to break ties, and is the method used in our simulations that use this selection algorithm.

Here is the algorithm in detail. The *get-random-walk-parent(R)* function returns the processor identifier (PID) of the highest scoring individual encountered on a random walk of length $R$. The notation for this algorithm is pseudocode, and the algorithm is performed *synchronously* and *in parallel* at each location.

```
get-random-walk-parent(R):
    x = my-x; y = my-y;
    best-score = score[x][y];
    best-PID = PID[x][y];
    loop R times
        cond (rand-int() mod 8)
            case 0: x + = 1;
            case 1: x - = 1;
            case 2: y + = 1;
            case 3: y - = 1;
            case 4: x + = 1; y + = 1;
            case 5: x + = 1; y - = 1;
            case 6: x - = 1; y + = 1;
            case 7: x - = 1; y - = 1;
        endcond
        toroidalize-x-and-y();
        get score from PID[x][y];
        if (best-score ≤ score) then
            best-score = score;
            best-PID = PID[x][y];
        endif
    endloop
    return best-PID;
```

As the random moves are made, the function *toroidalize-x-and-y()* implements coordinate wrap–around at the edges of the torus. Note that each of the $R$ iterations of the main loop of *get-random-walk-parent()* requires a general communication operation (the **get**). More than half of the computation time in the Connection Machine-2 implementation of *get-random-walk-parent()* is spent performing this communication. We call the *get-random-walk-parent()* function twice to determine the two parents:

```
parent0 = get-random-walk-parent(R);
parent1 = get-random-walk-parent(R);
```

We use a number of variations of this scheme in our simulations. The exact method described above is used in the **Partition** simulations. In **AntFarm**, rather than choosing two parents, the individual at the location where the offspring will be created always act as one of the parents, with the other parent chosen from the local neighborhood via a random walk.

One problem with using *get-random-walk-parent()* to implement isolation by distance competition and mating is that it does not scale well to large parent–offspring distances. An alternative is to generate parent–offspring distances with the desired distribution, and use these distances to randomly choose a sample of potential mates. This is the method that we use in the **Parasite** simulations. A quick and easy way to approximate a normal distribution is to add together several random numbers generated uniformly in the range $(-1, 1)$, and scale the result appropriately.

To simulate the stepping stone model in the selection and mating process, we again place the individuals in a toroidal grid. The grid is then broken into non-overlapping square demes of size $N = d \times d$. In stepping stone models, panmixia is assumed within each island (deme). We implement this local panmixia by randomly sampling a small number of individuals from within the deme. In the stepping stone model, migration occurs between neighboring demes. For this discussion, we again assume a 2 dimensional neighborhood. The migration rate $m$ is the probability that an individual in the subpopulation in a given generation is an immigrant from a neighboring deme. Therefore, if the number of individuals in the deme is $N = d^2$, $Nm$ individuals will migrate in/out of a deme each generation.

We have implemented two migration algorithms: one–way and two–way. Neither is entirely satisfactory. One–way migration is implemented by replacing an individual with a copy of an individual from a neighboring deme. The replacement is necessitated by the mechanical constraint of maintaining a constant population size *in each deme*. Two–way migration is implemented by swapping individuals between neighboring demes, but reciprocal migration is also not a very good model of natural migration.

We implement one–way migration as follows. To migrate an individual from a neighboring deme into a particular location $l$ in deme $D$, we randomly select either the $x$ or $y$ direction, and generate a uniformly distributed random distance along that axis from the range $(-d, d)$ where $d$ is the width of the deme. The chosen individual is either in deme $D$, or one of the 4 neighboring demes. If the chosen individual is in a neighboring deme, it is copied and replaces the individual from grid location $l$. Because half of these attempted migrations will choose another individual within the same deme, this algorithm is applied at each location in each deme with a probability $2m$ each generation. Here is the algorithm in detail. Again, this pseudocode is executed synchronously and in parallel at every location.

*one-way-migration():*

```
if (rand-float() < 2m) then
    x = my-x; y = my-y;
    dist = rand-int() mod d;
    if (rand-float() < 0.5) then
        if (rand-float() < 0.5) then x + = dist;
        else x − = dist;
        endif
    else
```

```
            if (rand-float() < 0.5) then y + = dist;
            else y − = dist;
            endif
        endif
        toroidalize-x-and-y();
        if (different-deme(x, y)) then
            get genome from PID[x][y];
            my-genome = genome;
        endif
    endif
```

The function *different-deme(x, y)* returns `True` if $(x, y)$ maps to a different deme, and `False` if it refers to a location in this deme.

The other migration alternative that we have implemented is two–way migration. The two–way migration algorithm randomly chooses an individual in the same way as in the one–way case. If the chosen individual is in a neighboring deme, it is ***swapped*** with the individual from this grid location. *Arbitration* is required to ensure that an individual is not simultaneously involved in multiple parallel swaps. Like the one–way migration algorithm described above, half of these attempted migrations will choose another individual within the same deme, but two migrations occur for each swap, so this algorithm is applied at each location with a probability $m$ each generation. The two–way migration implementation does not change the genetic composition of the total population, while the one–way implementation duplicates the source of the migration, and deletes the destination.

Here is our standard arbitration algorithm. We make use of the unique processor identifier (PID) that serves as the processor's address for interprocessor communication. This code is specific to the Connection Machine-2.

```
arbitrate():
    send-with-overwrite my-PID to arbitrator at PID[x][y];
    return ((get arbitrator from PID[x][y]) == my-PID)
```

*Arbitrate()* returns `True` in processors that win arbitration, and `False` otherwise. `Send-with-overwrite` is a Connection Machine communication primitive that ensures that exactly one of the (potentially) many contending messages is delivered to each processor. Note that *arbitrate()* consists of two general interprocessor communication primitives, so it should be used as infrequently as possible.

In this simple arbitration algorithm, the outcome depends on the implementation details of the Connection Machine communication primitives. Because communication in the Connection Machine is deterministic, for a given set of contending processors, the same processor will always win arbitration. This means that the arbitration is not fair and random; by virtue of being born in a particular Connection Machine processor, some individuals might always be given priority. A somewhat more complex and expensive method avoids this problem.

The fair arbitration algorithm is conceptually similar to the previous algorithm, but instead of using a unique PID to identify each processor, each processor generates

a random number. When these random numbers are sent to the *arbitrator* variable, the processor with the largest random number wins the arbitration. However, the random numbers are not necessarily unique, so it is possible that two or more contending processors both sent the same maximum random number. If the number that is read back matches the one sent, then the processor *may* have won the arbitration. To check for this case, a processor that potentially won arbitration determines how many others also potentially won at that location. Each "winning" processor sends (combining via addition) the constant **1** to the *arbitrator* variable, and reads back the result. If a processor that potentially won arbitration in the first phase of arbitration reads back a **1**, then it has indeed won. However, if the processor finds that other processors generated the same "winning" random number, they must go through the arbitration algorithm again (with new random numbers). This process is repeated until finally there is only one processor that generated the maximum random number in its location. Note that only those processors that passed all of the earlier iterations of arbitration need to take part in subsequent rounds. Here is the algorithm in detail (again, this algorithm is specific to the Connection Machine-2):

```
fair-arbitrate():
    still-in-the-running = True;
    status = False;
    do
        my-rand = rand-int();
        send-with-max my-rand to arbitrator at PID[x][y];
        if ((get arbitrator from PID[x][y]) == my-rand) then
            arbitrator = 0;
            send-with-add 1 to arbitrator at PID[x][y];
            if ((get arbitrator from PID[x][y]) == 1) then
              status = True;
            endif
        else
            still-in-the-running = FALSE;
        endif
    while (still-in-the-running
        && ((get arbitrator from PID[x][y]) > 1));
    return status;
```

**Send-with-max** is a Connection Machine communication primitive that assigns the maximum of the contending messages to the destination, and **send-with-add** assigns the sum of the contending values to the destination. Both the *max* and integer *add* functions are commutative, so there is no dependence on the details of the Connection Machine message routing algorithm in this arbitration algorithm. The *still-in-the-running* variable is true for all organisms that have yet to lose any round of arbitration. The random numbers should be integers, not floating point numbers in the range $[0, 1)$, because a 32-bit integer has more possible random values than a random 32-bit IEEE format floating point number constrained to fall in a restricted range.

*Fair-arbitrate()* requires a minimum of twice as many general communication operations as *arbitrate()*. We have compared runs with each algorithm, and can discern no significant qualitative difference in our simulation results, so we usually use *arbitrate()* in our production runs, because it is faster.

We empirically tested both in **Peacock**, and found no qualitative differences, so we arbitrarily chose to use the one–way migration algorithm in the **Peacock** production runs that involve the stepping stone model of spatial structure.

### 2.4.2 Recombination

The selection phase of the genetic algorithm produces a pair of strings (chromosomes) for each offspring that is to be included in the next generation. Recombination mixes the genetic information of these parent strings when producing the offspring, so an offspring chromosome contains some of the genetic information from each parent. We consider only *reciprocal recombination*, where equivalent length sub–strings are exchanged (Figure 2.2). The model of recombination that we use in our genetic algorithm begins with an alignment of the pair of chromosomes (Figure 2.2a). At some random point (or points), the chromosomes cross (Figure 2.2b), then the chromosomes are cut and rejoined at the crossover point(s) (Figure 2.2c).

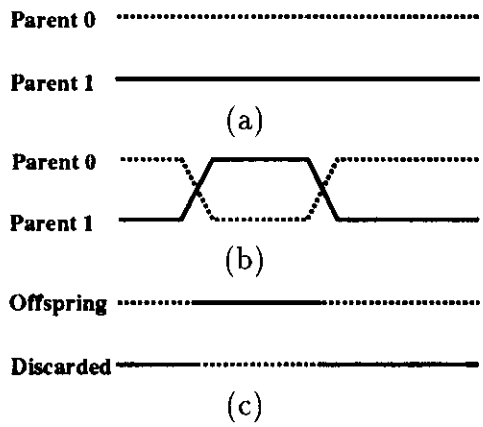


Figure 2.2: A two–point reciprocal recombination. (a) The parent chromosomes are aligned. (b) At a random point(s), the chromosomes cross. (c) The chromosomes are cut and rejoined at the crossover point, resulting in new gene combinations. One of the chromosomes specifies the offspring, and the other is discarded.

As described here, our recombination operation produces two haploid chromosomes. One of these (chosen randomly) is discarded, and the other is retained for use in the offspring. (In practice, we only explicitly generate one of the two chromosomes.) In (primarily) haploid organisms, the two chromosomes used in recombination are the chromosomes of the two haploid parents. In (primarily) diploid organisms, recombination is applied to each of the two diploid parent genomes, creating two haploid chromosomes. These two chromosomes are paired to form the diploid offspring genome. (A diploid example occurs in **Peacock** in Chapter 3).

Traditional genetic algorithms usually use asexual reproduction some fraction of the time, and otherwise use an $n$–point crossover, where $n$ is usually 1 or 2. In contrast, we apply the following recombination algorithm to every new individual:

```
recombine(parent[0], parent[1], ρ) :
    cur-parent = rand-int() mod 2;
    cur-bit = 0;
    while (cur-bit < chromosome-length) do
        if (cur-parent == 0) then
            offspring[cur-bit] = parent[0][cur-bit];
        else
            offspring[cur-bit] = parent[1][cur-bit];
        endif
        if (rand-float() < ρ) then
            cur-parent = (cur-parent + 1) mod 2;
        endif
        cur-bit + = 1;
    endwhile
    return offspring;
```

In this way, each reproduction involves 0 to $l-1$ crossovers, where $l$ is the number of bits in the chromosome. The number of crossover points per chromosome is approximately geometrically distributed if $\rho \ll 1$. The actual code for all of the recombination operators that we use can be found in Appendix A.

This recombination operator differs from those used in many genetic algorithms not only in the use of a variable number of crossover points, but also because it always operates at the level of a bitstring, rather than, for example, a list of parameters. It is defined completely independently of what the genes code for and how the genes are encoded in the chromosome. This is a closer approximation of natural genetic systems. In genetic algorithms that are used for optimization, it is not uncommon to exploit problem specific or representation specific information in the implementation of the genetic operators in an effort to speed the search. Exploiting domain–dependent information is a good approach for engineering applications, but to create unbiased and realistic evolutionary experiments, it is necessary to avoid building the experimenter's preconceptions into the simulation. Therefore, we require a clear separation between the genetic algorithm and the simulated organisms/environment (Collins and Jefferson, 1991b).

Although natural recombination is not completely understood and is not uniform across all taxa, it is clear that our model is at least a crude approximation to nature (Goodenough, 1984). We assume that multiple crossovers can be independently and uniformly distributed along the chromosome. In nature, the physical and chemical features of chromosomes do not really allow this. There is often *interference* generated by a crossover, affecting the likelihood of a second crossover occurring between the same pair of DNA strands. Both positive and negative interference are

observed, which means that a crossover might either increase or decrease the likelihood of another crossover event occurring nearby. Also, the recombination rate can vary depending on the area of the chromosome. More recombinations are observed near the center of the chromosome than at the ends, although recombination is rare near the centromere (the structure that connects chromosome pairs). In addition, certain nucleotide sequences (e.g. 5'GCTGGTG3') have been found to greatly enhance the chances of a nearby crossover (Hartl and Clark, 1989). We do not attempt to model any of these effects.

### 2.4.3 Mutation

After the process of recombination, the new genome is mutated, producing the final version that describes the offspring. Many classes of mutation appear in natural genetic systems (Goodenough, 1984), including base substitution, deletion, frameshift, insertion, inversion, translocation, duplication, etc. Although all of these types of mutation can make sense in the context of artificial evolution, we usually only make use of base substitutions, which are characterized by the substitution of one nucleotide for another. For each bit in the chromosome, we simulate a base substitution by flipping the bit (change 0 to 1, or 1 to 0) with probability $\mu$.

```
mutate(offspring, μ):
    bit = 0;
    while (bit < chromosome-length) do
        if (rand-float() < μ) then
            offspring[bit] = log-not(offspring[bit]);
        endif
        bit + = 1;
    endwhile
```

Similar to our recombination algorithm, each reproduction involves 0 to $l$ mutations, where $l$ is the number of bits in the chromosome. The number of mutations per chromosome falls into an approximately geometric distribution. In Chapter 6, we discuss an insertion/deletion mutation operator that allows us to use variable length chromosomes. The actual code for all of the mutation operators that we use can be found in Appendix A.

In our genetic algorithms, we mutate (flip) each bit of the chromosome with a uniform probability, despite the fact that the distribution of base substitutions is nonuniform at the molecular level in nature (Brown and Clegg, 1983). The nonuniformities probably result from the details of the chemistry of DNA and repair mechanisms. In this dissertation, it is assumed that these details are relatively unimportant and that a uniform distribution of point mutations is a good enough model. Like the recombination operation, this formulation of mutation differs from many genetic algorithm implementations, in that the mutations make small changes in a structureless bit-string, rather than making small changes to a high–level, problem–specific parameter or data structure.

# Chapter 3

# Peacock: The Evolution of Sexual Selection and Female Choice

We, like many other researchers in the field of artificial life, are attempting to increase our understanding of natural life (Langton, 1989a; Langton, 1989b; Langton et al., 1991). In this chapter, we use artificial evolution to augment the study of an analytic population genetics model. We begin with a simple analytical model of sexual selection, and extend it in several directions by relaxing the important simplifying assumptions. While all of the simplifying assumptions are necessary to make the mathematics tractable, they also may prevent the model from applying to real populations. Artificial evolution experiments can be used to supply empirical evidence that an analytical model is robust with respect to variations that cannot be handled analytically. By simulating a more realistic version of the analytic model, we can discover whether the predicted equilibrium is likely to apply to natural populations. Simulated evolution also allows the exploration of the dynamics of a population that is far away from equilibrium.

## 3.1 The Paradox of Sexual Selection and Female Choice in Nature

In nature, it is not unusual for the females of a species to evolve preferences for mates that possess a particular exaggerated secondary sexual characteristic. These preferences become paradoxical when the preferred trait reduces the viability (ability to survive to adulthood) of the males (Kirkpatrick and Ryan, 1991). This phenomenon has been known for more than 120 years. Darwin (1871) describes a number of examples (such as the peacock), and hypothesized that it was due to female mating preferences (sexual selection), but he did not understand the origin of these preferences. In recent years, a number of hypotheses have been proposed (Kirkpatrick and Ryan, 1991; Fisher, 1958; Zahavi, 1975; O'Donald, 1980; Kirkpatrick, 1982; Read, 1988).

This phenomenon is paradoxical in species where a female receives nothing from a mate other than sperm, and where there is no direct relationship between her mate

choice and her own viability or fecundity. Because the expression of the maladaptive trait in a male reduces his likelihood of surviving to reproduce, so the argument goes, females who mate with such males will produce fewer male offspring of mating age in the next generation. At first glance, it appears that females that prefer the *absence* of the trait should be at a selective advantage, therefore eliminating preferences *for* the trait. The fact that this phenomenon appears to be common in nature suggests that females who prefer the absence of the trait are not always at a selective advantage, and/or that the natural situations are actually more complex than they appear.

In the peacock, the females are relatively unremarkable, but the males possess very long and brightly colored tail feathers. The male's tail makes him easier prey, because the tail reduces his mobility and makes him easier to see. Apparently, there is nontrivial component of selection against the males, and the selection grows stronger with increasing tail size (Darwin, 1871). Because we view the peacock as an archetypical illustration of sexual selection, we refer to our sexual selection simulation models as **Peacock**.

## 3.2 Kirkpatrick's Model

Fisher (1958) was the first to provide a possible (qualitative) solution to the problem of the persistence of female preferences for mates possessing a maladaptive trait. More recently, Kirkpatrick (1982) developed a simple analytic model of sexual selection upon which we base our simulations.

Kirkpatrick's model assumes that the males of the species contribute only gametes to the next generation, and that there is no direct relationship between a female's mating preference and her survivorship or her fecundity. The genetic basis for sexual selection in the model consists of two loci, one for the female preference and one for the male trait, that the trait and preference loci are not sex–linked, that they reside on different chromosomes, and that the genetic system is haploid. The preference locus $P$ has two alleles: $P_0$ and $P_1$; and likewise the trait locus $T$ has alleles $T_0$ and $T_1$. Kirkpatrick implicitly assumes an infinite, unstructured population, i.e. panmixia (no spatial structure, so distance is not a factor in mate choice), and nonoverlapping generations (no age structure).

Both males and females have both the $T$ and $P$ loci. The allele at the $T$ locus is expressed only in males, and the allele of the $P$ locus only in females. The $T_0$ allele produces males that do not possess the exaggerated secondary sexual characteristic (i.e. have a short tail), while the $T_1$ allele produces the exaggerated trait (long tail). Allele $T_1$ has the side effect of reducing viability (the probability of surviving to adulthood) to $1 - s$ (where $s > 0$), whereas all $T_0$ males and all females survive to adulthood.

The $P_0$ females prefer to mate with $T_0$ males. Given a two–way choice of a $T_0$ male and a $T_1$ male, a $P_0$ female will choose to mate with the $T_0$ male $a_0$ times more frequently than the $T_1$ male. In the same way, $P_1$ females prefer to mate with $T_1$ males, and the strength of this preference is $a_1$. These preferences are frequency

dependent, so the $P_i$ females will choose to mate with a $T_i$ male with probability

$$P(T_i|P_i) = \frac{a_i t'_i}{t'_j + a_i t'_i} \tag{3.1}$$

We denote the frequency (fraction) of allele $T_i$ *before* viability selection as $t_i$, and *after* viability selection as $t'_i$.

A generation begins with an equal number of males and females. Then, viability selection kills a fraction $s$ of the $T_1$ (long-tailed) males. Mate choice proceeds according to the probabilities given above, recombination between gametes occurs, and the process repeats with the next generation.

Let $p_1$ be the frequency of the $P_1$ allele in the population, and $t_1$ be the frequency of the $T_1$ allele. From this model, Kirkpatrick derives the following equation for the equilibrium allele frequencies:

$$t_1 = \begin{cases} 0 & \text{if } p_1 \leq \frac{a_0+s-1}{(a_0a_1-1)(1-s)} \\ \frac{(a_0a_1-1)(1-s)}{(a_0+s-1)[a_1(1-s)-1]}p_1 - \frac{1}{a_1(1-s)-1} & \text{if } \frac{a_0+s-1}{(a_0a_1-1)(1-s)} < p_1 < \frac{a_1(a_0+s-1)}{(a_0a_1-1)} \\ 1 & \text{if } \frac{a_1(a_0+s-1)}{(a_0a_1-1)} \leq p_1 \end{cases} \tag{3.2}$$

Equation 3.2 is plotted for several sets of parameters in Figure 3.1. Depending on the parameters and initial conditions, the frequency of the trait allele ($t_1$) can take on any value from 0 to 1.

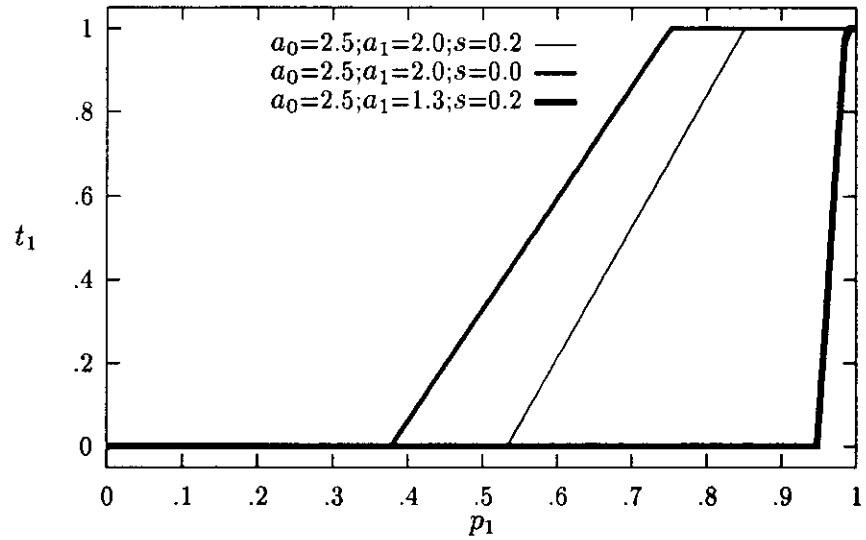


Figure 3.1: Equation 3.2 plotted for various sets of parameters. The axes are the frequency $t_1$ of the trait allele (long tail) in the population, and the frequency $p_1$ of the allele coding for the preference for the trait (preference for a long-tailed mate).

It is interesting that the equilibrium condition describes a curve, rather than a single point. Kirkpatrick provides an intuitive explanation of the various forces at work in this model (Kirkpatrick, 1982, p. 5):

> The finding that there is not a single point of equilibrium can be understood intuitively by separating the effects of natural selection and mating success. At any equilibrium it must be that the viability deficit which trait-bearing [$T_1$] males suffer is exactly offset by the mating advantage they receive so that the two male phenotypes have identical fitness. The mating advantage is determined by both the strength and frequency of the preference allele [$P_1$]. Increasing the frequency of the preference allele increases the mating advantage. This does not necessarily result in fixation of the male trait allele [$T_1$], however, because the strength of the mating advantage decreases as the frequency of trait-bearing males increases.

Due to the preferential mate choices of the females in a population that contains both of the $T$ alleles, a correlation can form between the alleles of the $T$ and $P$ loci. Under strong sexual selection, for both $i = 0$ and $i = 1$, the chances of an individual in the population having the genotype $P_iT_i$ is greater than $p_it_i$, where $p_i$ and $t_i$ are the allele frequencies of $P_i$ and $T_i$ respectively. The strength of the association between $P$ and $T$ alleles is measured by $D$, the linkage disequilibrium (see Section 1.4.1). The result of this association is that when the frequency of $T_i$ changes due to selection, the frequency of $P_i$ tends to change with it. If the *strength* of the preference and trait were under genetic control, rather than just their presence or absence, then we would expect to see selection for more and more dramatic secondary sexual characteristics in males, and the strength of the preferences for such traits should also increase due to this correlated response (Fisher's (1958) "runaway" process).

In this model, there is no direct selection pressure on the $P$ locus; the frequency of the $P$ alleles changes only due to a correlated response ($D \neq 0$) to changes in the frequency of the $T$ alleles. Once the population reaches the curve of equilibrium, there will be no movement along the curve unless some force outside the model perturbs the allele frequencies. Examples of such outside forces that might occur in real populations (but are assumed to not exist for the purposes of Kirkpatrick's analysis) are genetic drift, migration, mutation, selection at other loci, etc.

## 3.3 Simulating the Model

In **Peacock**, our simulation based on Kirkpatrick's model, we place the organisms in a 2 dimensional grid, with one male and one female organism per grid location. With panmictic mating the grid serves no purpose, but we use the grid when we extend the model to use local mating.

A generation begins with an equal number of males and females (65,536 of each, for a total population size of N=131,072). Then, viability selection is applied to the $T_1$ (long-tailed) males, randomly killing them with probability $s$. Each female then chooses one of the surviving males as her mate (according to the probabilities given in Equation 3.1) and together they produce two offspring, one male and one female, which are placed at the mother's grid location. Each of the two offspring is the result of an independent recombination and mutation of the two parent genomes. The $P$ and $T$ loci are implemented as two one-bit genes on different chromosomes, so there

is no direct linkage between them. We add mutation at a low rate of $\mu = 0.00001$ per bit (locus) per generation, in order to prevent permanent fixation for any allele in our simulation experiments.

Kirkpatrick's model does not specify the mechanics of how the females choose their mates; it simply defines the frequency of choices in Equation 3.1. In **Peacock**, each female randomly samples, with replacement, 25 males (some of which may be dead) from the population and counts the number of $T_0$ and living $T_1$ males in her sample to determine approximate values for each $t'_i$ in Equation 3.1. Each female then applies Equation 3.1 to determine with which male phenotype ($T_i$) she will mate, and then randomly selects one of the (living) $T_i$ males from her random sample of males. Note that if only one phenotypic class of males is represented in her sample, Equation 3.1 will cause her to always choose that class. If all the males in her sample are dead, she and her brother survive into the next generation.

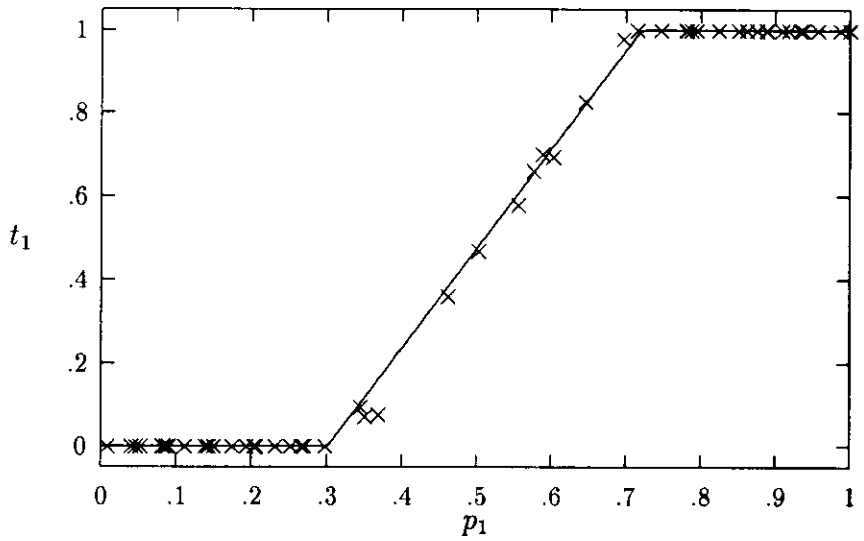


Figure 3.2: Locations of 51 independent populations after 500 generations with panmictic mating, $a_0 = 2.0$, $a_1 = 3.0$, $s = 0.2$, $\mu = 0.00001$, and $N = 131,072$. The line is the equilibrium predicted by Kirkpatrick. The initial conditions for population $i$ were $t_1 = 0.5$ and $p_1 = 0.02i$, where $i$ ranges from 0 to 50.

The **Peacock** model departs from the analytical model in three important ways: (1) finite populations are used; (2) females do not have global knowledge of the population, but rather sample a small number of males in choosing a mate; and (3) a low rate of mutation has been introduced. These modifications make the model more biologically realistic and/or more amenable to massively parallel simulation. Nonetheless, as Figure 3.2 demonstrates, after 500 generations the simulated populations do indeed move to positions near the equilibrium predicted by Kirkpatrick's analysis.

While these initial experiments verify that the **Peacock** model produces the equilibrium predicted by the analytical model, they give us little insight concerning the behavior of populations that begin far from the equilibrium. In particular, we are interested in how a new allele invades a population and how that population subsequently evolves to a point on the equilibrium curve. Figure 3.3 shows the path

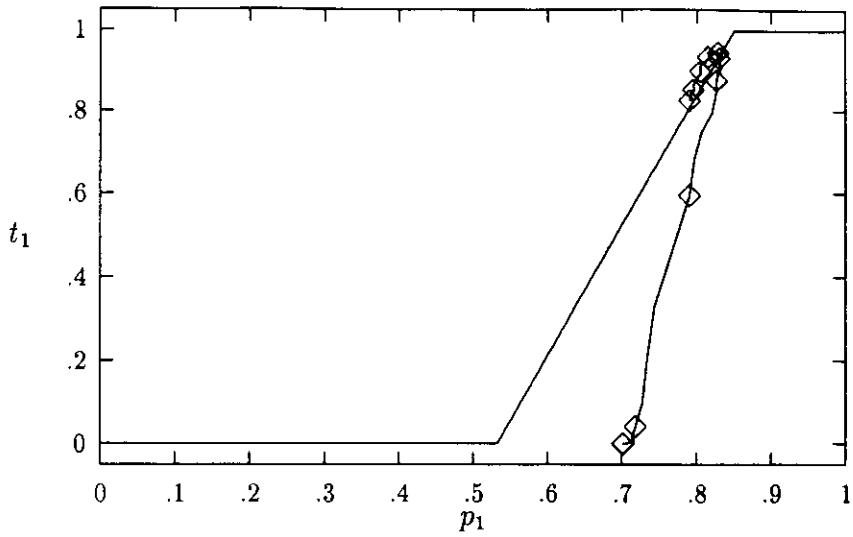


Figure 3.3: The path of a single population through 500 generations with panmictic mating, $a_0 = 2.5$, $a_1 = 2.0$, $s = 0.2$, $\mu = 0.00001$, and $N = 131,072$. The run begins with $p_1 = 0.7$ and $t_1 = 0.0$. The $\diamond$'s are at 50 generation intervals. The equilibrium predicted by Kirkpatrick is indicated by the straight lines.

taken by a population that begins with a gene pool containing an abundance of both preference ($P$) alleles, but no $T_1$ alleles (all the males have short tails). During the experiment, $T_1$ alleles (long tails) are introduced into the population by mutation (which is qualitatively similar to introduction via migration). Although it takes about 100 generations for significant numbers of $T_1$ alleles to build up, once this occurs the population moves quickly to the equilibrium curve.

This experiment dramatically demonstrates the power of sexual selection and female choice. The $P_0$ female's preferences are stronger for the more viable, short-tailed $T_0$ males than is the $P_1$ female's preference for the $T_1$ males ($a_0 > a_1$), yet the less-strongly preferred and less viable $T_1$ males quickly take over the population, because they are preferred by many more females. Figure 3.4 shows how the mean viability of the males decreases as the $T_1$ (long-tailed) males proliferate. This experiment also demonstrates the "runaway" process described by Fisher (1958). Although the $P$ and $T$ loci reside on different chromosomes, a positive association (linkage disequilibrium $D > 0$) forms between the $P_1$ and $T_1$ alleles. As $t_1$ increases due to sexual selection, $p_1$ also increases from 0.7 to at one point more than 0.8.

Kirkpatrick (1982) notes that forces such as random genetic drift may have important effects in real populations. Although the equilibrium is stable (in an infinite, non-mutating population), if the population is pushed off of the equilibrium, it may return to the curve at a different location. Despite the size of the population ($N = 131,072$), random genetic drift is apparent in the experiment in Figure 3.3. The path taken by the population is rather wiggly, and during the latter half of this experiment, it appears that the population has drifted a short distance from the equilibrium, and returns at a point with somewhat lower values for $p_1$ and $t_1$. Although each run is different, this is a typical result.

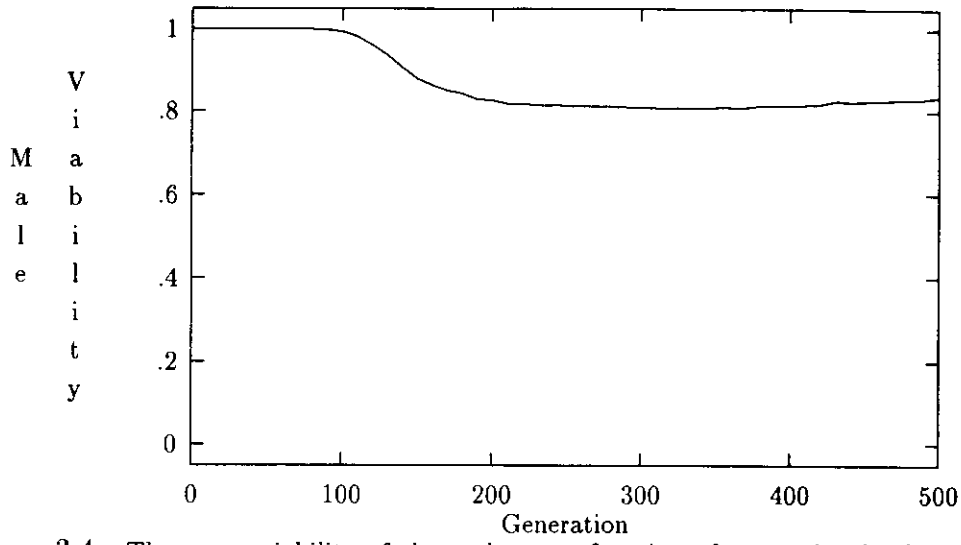

Figure 3.4: The mean viability of the males as a function of generation in the population in Figure 3.3.

We have thus not only verified that the **Peacock** model produces the equilibrium derived by Kirkpatrick, but we have also demonstrated that artificial evolution is a viable way to study the dynamics of populations far from equilibrium. In addition, we are able to observe the effects of finite population size on the system.

## 3.4 Extensions of the Model

In formulating the analytic model, a number of simplifying assumptions were made in order to make the mathematics tractable, including (1) an infinite population; (2) panmixia (global mating); (3) haploid genetics; and (4) specification of the preference and trait phenotypes each by one locus with two alleles. How dependent on these assumptions are the predictions of the model? Unless there is a reasonable expectation that the predictions hold when these conditions are relaxed, the model is not very useful for studying real populations. In the previous section, we replaced an infinite population with a large but finite, mutating population. In this section, we explore the effects of spatial structure (local mating) and diploidy.

### 3.4.1 Sexual Selection in Structured Populations

One of the important assumptions made by Kirkpatrick is panmixia. Although this is important to make the analysis tractable, it may also make the predictions inapplicable to structured populations (which includes most natural populations). In this section, we extend both Kirkpatrick's analytic model and our simulation model to include the *stepping stone* model of population structure (Kimura and Weiss, 1964).

One of the basic assumptions of Wright's shifting balance theory of evolution is that spatial structure exists in large populations and plays a critical role in evolu-

tion (Wright, 1931; Wright, 1932; Wright, 1969; Kimura and Ohta, 1971; Crow, 1986; Hartl and Clark, 1989; Provine, 1986). The structure is in the form of *demes* (Gilmour and Gregor, 1939), or semi-isolated subpopulations, with relatively thorough gene mixing within a deme, but restricted gene flow (migration) between demes. In the stepping stone model of population structure, the demes are assumed to lie in an $n$-dimensional lattice, with migration restricted to neighboring demes in the lattice (Kimura and Weiss, 1964; Crow, 1986; Hartl and Clark, 1989). The migration rate $m$ is the probability that an individual in the subpopulation is a new migrant from a neighboring deme. Therefore, if the number of individuals in the deme is $N$, $Nm$ individuals will migrate into the deme each generation, and the same number (on average) will migrate out. Remember that the subpopulations will become strongly differentiated due to genetic drift if $Nm \ll 1$, but will behave as a single panmictic population if $Nm > 4$ (Kimura and Maruyama, 1971), where $N$ is the population size of each subpopulation (see Section 2.3). Our analytical extensions (see below) to Kirkpatrick's model apply for the case of $Nm \ll 1$, and Kirkpatrick's original panmictic model applies when $Nm > 4$. We would expect some sort of intermediate behavior for the intermediate migration rates, although neither model fully applies in these cases.

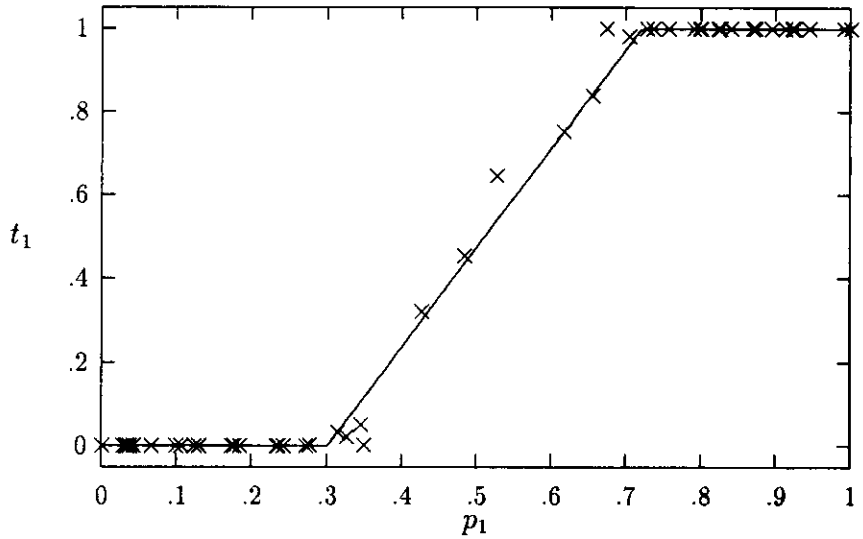


Figure 3.5: Locations of 50 independent, panmictic, haploid populations of size $N = 8192$ after 500 generations with $a_0 = 2.0$, $a_1 = 3.0$, $s = 0.2$, $\mu = 0.00001$. The line is the equilibrium predicted by Kirkpatrick. The initial conditions for population $i$ were $t_1 = 0.5$ and $p_1 = 0.02i$, where $i$ ranges from 0 to 50.

In a stepping-stone structured population with $Nm \leq 4$, we expect the equilibrium allele frequencies to be contained within a region. Consider a structured population consisting of a number of demes. Assume that there is restricted gene flow between demes ($Nm \ll 1$), so the allele frequencies of each deme evolve (roughly) independently to some point on Kirkpatrick's equilibrium. Each of the demes is relatively small, so we expect that drift will have a greater effect. We empirically observe in Figure 3.5 that populations as small as $N = 8,192$ individuals still fall

near to Kirkpatrick's equilibrium. The equilibrium allele frequencies for the whole population are calculated as the mean allele frequencies of the demes, each of which lies on the equilibrium curve defined by Equation 3.2. Therefore, we expect that the equilibrium allele frequencies of the population as a whole will be bounded by

$$t_1 \leq \begin{cases} \frac{1}{\frac{a_1(a_0+s-1)}{a_0a_1-1}} p_1 & \text{if } p_1 \leq \frac{a_1(a_0+s-1)}{a_0a_1-1} \\ 1 & \text{if } p_1 > \frac{a_1(a_0+s-1)}{a_0a_1-1} \end{cases}$$

$$t_1 \geq \begin{cases} 0 & \text{if } p_1 \leq \frac{a_0+s-1}{(a_0a_1-1)(1-s)} \\ 1 + \frac{1}{1-\frac{a_0+s-1}{(a_0a_1-1)(1-s)}}(p_1 - 1) & \text{if } p_1 > \frac{a_0+s-1}{(a_0a_1-1)(1-s)} \end{cases} \tag{3.3}$$

Equation 3.3 forms the boundary within which all possible ensembles of subpopulations residing on Kirkpatrick's equilibrium must lie. This defines the region between the solid lines in Figure 3.6 (where the panmixia equilibrium defined by Equation 3.2 is shown by the dotted line).

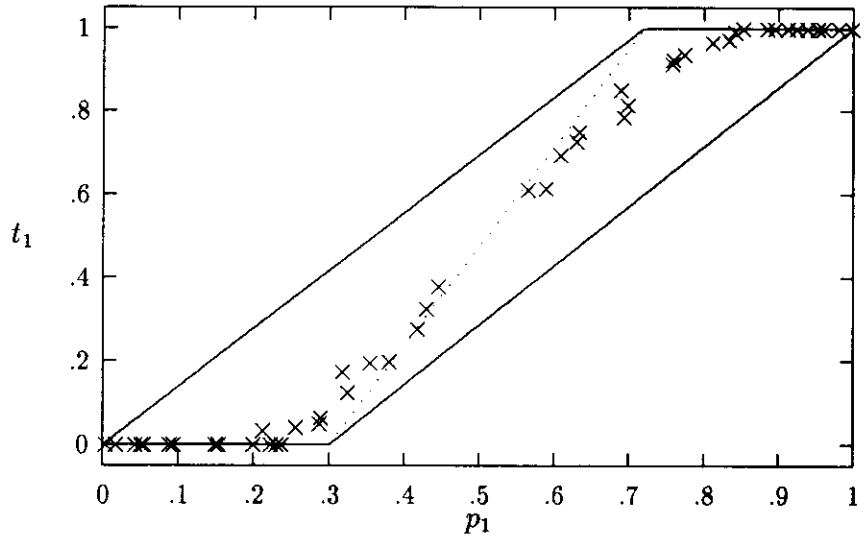


Figure 3.6: Locations of 51 independent populations after 500 generations with local mating (stepping stone model with 16 demes of 8,192 individuals each), $a_0 = 2.0$, $a_1 = 3.0$, $s = 0.2$, $\mu = 0.00001$, $m = 0.00001$, and $N = 131,072$. The solid lines define the predicted region of equilibrium for local mating, and the dotted line is the equilibrium predicted by Kirkpatrick for panmixia. The initial conditions for population $i$ were $t_1 = 0.5$ and $p_1 = 0.02i$, where $i$ ranges from 0 to 50.

Are all points within the region stable equilibrium points? If they were, once a population reaches any point in the region, it will stay at that point (in the absence of drift, etc.). Although the curve of equilibrium under panmixia is stable, this is not the case for the points in the region for structured populations. A point in the graph does not fully describe a structured population the way it does a panmictic one; a population that is globally characterized by a point in the region that bounds the equilibria may not yet have reached equilibrium. It is possible for the allele frequencies for the whole population to fall within the region, yet have many of the subpopulations

still be far from their local equilibria and rapidly evolving. Therefore, we may observe significant change in allele frequencies due to selection pressure induced by sexual selection, even for populations that are within the region of equilibrium.

To simulate the stepping stone model in the selection and mating process, we place the individuals in a toroidal, 2 dimensional grid (again, with one male and one female at each grid location). The 256 x 256 grid is then broken into non–overlapping demes consisting of 64 x 64 = 4096 grid locations, each with a population of 8,192. As with the panmictic simulation, each female randomly samples 25 males (with replacement), but now the samples are chosen only from her deme, rather than from the whole population. Based on this sample, she estimates $t_i'$ for the local males and applies Equation 3.1 to determine the phenotype of her mate. She then randomly chooses one of the (living) males with the appropriate phenotype from her sample of 25, and they produce two offspring (one male and one female) at her location. This provides for random mating (panmixia) within the deme, and no mating between demes. After mating, migration to one of the four neighboring demes occurs with a probability $m$ per individual, with on average $Nm/4$ migrations to each of the neighboring demes. The implementation of migration is described in Section 2.4.1. We use the *one-way-migration()* function in **Peacock**.

The results of 51 simulations are shown in Figure 3.6 (generation 500). Although the allele frequencies might be at equilibrium anywhere within the region, the populations appear to have a greater probability of falling near the long axis of the region of equilibrium, rather than near the edges. As predicted, the populations do not fall on the panmictic equilibrium curve, but they do fall within the region of equilibrium.

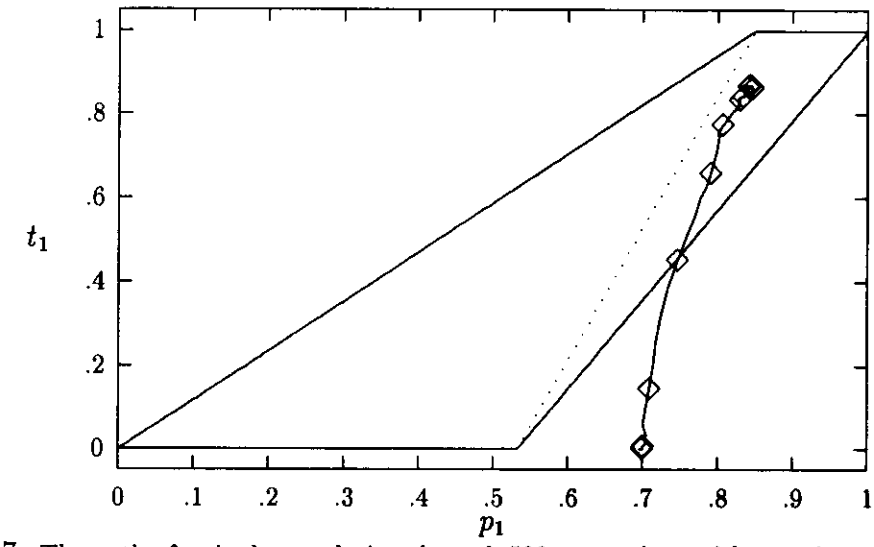


Figure 3.7: The path of a single population through 500 generations with stepping stone structure (16 demes, each consisting of 8,192 individuals), $a_0 = 2.5$, $a_1 = 2.0$, $s = 0.2$, $\mu = 0.00001$, and $N = 131,072$. The run begins with $p_1 = 0.7$ and $t_1 = 0.0$. The solid lines define the predicted region of equilibrium for local mating, and the dotted line is the equilibrium predicted by Kirkpatrick for panmixia.

In Figure 3.7 we again examine the behavior of a population that begins far from the equilibrium, by repeating the experiment where we observed the invasion of a new allele into the gene pool, but this time using the spatially structured population described above. Figure 3.7 plots the path of a population that begins at $p_1 = 0.7$ and $t_1 = 0.0$. Although the evolution again proceeds quickly, the rate of change in $t_1$ appears to be about half that observed in the panmixia experiment. It is important to note that the selection pressure remains strong, even after the population has entered the region bounding the equilibrium states (demonstrating that a population can still be at disequilibrium within the region). Around generation 350 the selection pressure seems to have dropped off significantly, indicating that the population is at or near equilibrium, and from then on it moves by drift. As we saw under panmixia, random genetic drift has noticeable effects: the path under selection is wiggly, and the population wanders once it reaches an approximately stable equilibrium.

### 3.4.2 Sexual Selection in Diploid Organisms

In order to make the mathematics tractable, Kirkpatrick assumes that the organisms are haploid (Kirkpatrick, 1982, p. 10):

> It is apparently impossible to treat comparable two–locus, two–allele diploid models analytically because it requires nine (rather than three) simultaneous nonlinear equations.

In this section, we extend our simulation model to include diploid organisms. Using these simulations, we can determine if a qualitative difference results from moving from haploid to diploid genetics.

The only changes required to add diploidy to **Peacock** are to include a second bitstring for each chromosome, and define a dominance relation between the alleles. Although we (and Kirkpatrick) analyzed the equilibrium in terms of allele frequencies in the previous sections we will now switch to phenotype frequencies. In haploid organisms (which we have worked with up to this point), the genotype is the same as the phenotype. This is because haploid organisms have only one copy of each gene, and it is always expressed phenotypically. In diploid organisms, there are two copies of each gene, possibly of different alleles. When both alleles are the same, that trait is expressed in the phenotype. However, when two different alleles are present (and the individual is said to be *heterozygous* at that locus), they may interact. The simplest interaction is complete dominance, in which case the phenotypic effects of one allele completely dominates (masks) the other allele.

In this chapter, we consider only simple dominance relationships between the alleles, although other systems can be simulated in the same manner. We examine the case where the trait allele $T_1$ and the preference for that trait allele $P_1$ are recessive, as described in Table 3.1, although the other three dominance combinations can be examined just as easily.

Although Kirkpatrick states that the analysis is intractable for diploid organisms, the model can be viewed in terms of phenotypes only, and requires no reference

| Genotype | Male Phenotype | Female Phenotype |
|---|---|---|
| $T_0T_0P_0P_0$ | $T_0$ | $P_0$ |
| $T_0T_1P_0P_0$ | $T_0$ | $P_0$ |
| $T_1T_1P_0P_0$ | $T_1$ | $P_0$ |
| $T_0T_0P_0P_1$ | $T_0$ | $P_0$ |
| $T_0T_1P_0P_1$ | $T_0$ | $P_0$ |
| $T_1T_1P_0P_1$ | $T_1$ | $P_0$ |
| $T_0T_0P_1P_1$ | $T_0$ | $P_1$ |
| $T_0T_1P_1P_1$ | $T_0$ | $P_1$ |
| $T_1T_1P_1P_1$ | $T_1$ | $P_1$ |

Table 3.1: The relationship between the diploid genotypes and phenotypes when both $T_1$ and $P_1$ are recessive.

to either haploidy or diploidy for the underlying genetics. With either panmixia (Figure 3.8) or structured populations (Figure 3.9), we observe no obvious difference in the character of the equilibria due to diploidy.

One area where the evolution of diploid populations differs from haploid populations is the rate of genetic drift. Because a haploid population has half as many alleles as a diploid population with the same number of individuals, the haploid population drifts approximately twice as fast. This is an important difference, especially when we are dealing with relatively small populations over large numbers of generations.

Another way diploidy can have an important effect is the introduction of recessive alleles into the gene pool. In the absence of selection, the frequency of phenotypic expression of a recessive trait is the square of the frequency of the associated allele. For example, if the $T_1$ allele is recessive and is present in the population with a frequency of $t_1 = 0.01$, only $t_1t_1 = 0.0001$ of the males will express the $T_1$ phenotype. Therefore, the invasion of recessive alleles into the gene pool, even if they are strongly favored by selection, may take significantly longer under diploid genetics.

We explore this effect for both panmictic and spatially structured populations by once again repeating the experiment where we observed the invasion of a new allele into the gene pool, with both the $P_0$ and the $T_0$ alleles dominant. The experiment begins with the phenotype frequency of $p_1 = 0.7$ and no $T_1$ alleles in the population of 131,072 organisms. In a population this size, about 800 copies of the $T_1$ allele will be required before it becomes likely that a $T_1$ phenotype male will occur. At the mutation rate of $\mu = 0.00001$, there is a 66% chance of mutating one $T$ allele in the population each generation. Therefore, we would expect that a large number of generations and/or significant random genetic drift is going to be required to get the $T_1$ alleles to a high enough frequency that the sexual selection mechanism can begin to exert its influence.

The simulation results of the invasion of a new, recessive $T$ allele for a panmictic population are plotted in Figure 3.10, and for a structured population in Figure 3.11. As expected, the diploidy results are dramatically different from the results of the corresponding haploid experiments (Figures 3.3 and 3.7). In the case of panmixia, the $T_1$ alleles did not become prominent enough for the selection process to begin,

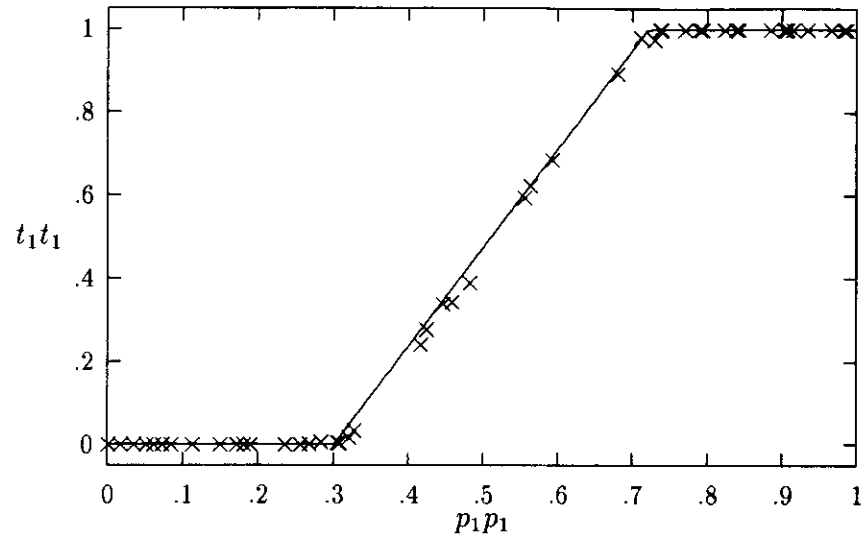


Figure 3.8: Locations of 51 independent populations after 500 generations with panmictic mating and diploid genomes, with $T_0$ and $P_0$ dominant, $a_0 = 2.0$, $a_1 = 3.0$, $s = 0.2$, $\mu = 0.00001$, and $N = 131,072$. The line is the equilibrium predicted by Kirkpatrick. The initial conditions for population $i$ were $t_1 = 0.5$ and $p_1 = 0.02i$, where $i$ ranges from 0 to 50.

even after 1000 generations, although the frequency of the $P$ alleles drifted quite noticeably. In the structured population, the $T_1$ alleles became numerous enough (in at least one deme) to undergo selection due to the female preferences after about 500 generations. It then took about another 400 generations to slowly evolve into the equilibrium region.

We have observed that although extending the haploid models to include diploidy does not alter the expected equilibria, the evolutionary dynamics are significantly different. First, the effective population size is approximately twice that of a haploid population, so genetic drift is a weaker force. Second, the invasion of recessive traits into the population, even when the preference frequencies favor that trait, is a slow process, while in haploid genetic systems (or under diploidy when the trait is dominant) the process is relatively fast. Third, spatial structure can apparently speed the invasion process.

## 3.5 Implementation Notes

**Peacock** requires about 1200 lines of code beyond the core library of routines. About 550 lines implements the model of mate choice, 300 lines for instrumentation, and 350 lines for run–time selection of instrumentation and parameter options. The simulation results presented in this chapter required approximately 10 days on the UCLA 16K processor Connection Machine-2.

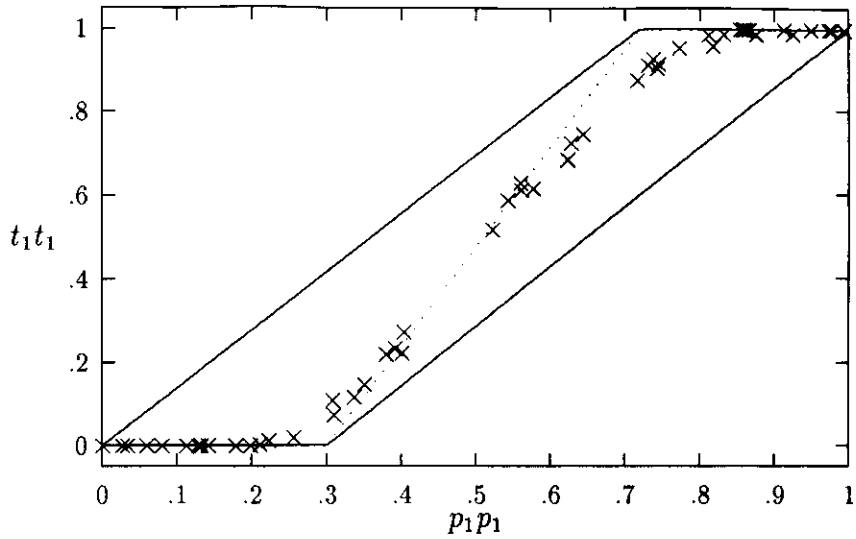


Figure 3.9: Locations of 51 independent populations after 500 generations with local mating (stepping stone model with 16 demes of 8,192 organisms each), diploid genomes with $T_0$ and $P_0$ dominant, $a_0 = 2.0$, $a_1 = 3.0$, $s = 0.2$, $\mu = 0.00001$, $m = 0.00001$, and $N = 131,072$. The solid lines define the predicted region of equilibrium for local mating, and the dotted line is the equilibrium predicted by Kirkpatrick for panmixia. The initial conditions for population $i$ were $t_1 = 0.5$ and $p_1 = 0.02i$, where $i$ ranges from 0 to 50.

## 3.6 Discussion

We have empirically reproduced Kirkpatrick's analytically determined equilibrium curve for his model of sexual selection. The major differences between the **Peacock** model and Kirkpatrick's model are that we simulate a finite (rather than infinite) population and the females choose mates based on a sample of the population (rather than global knowledge). Using this simulation, we are not only able to explore the equilibrium, but also the dynamics of populations on, near to, or far from the equilibrium curve.

Although this chapter is largely an empirical study, we have also analytically extended Kirkpatrick's model of sexual selection to handle populations that are structured into relatively large demes, with relatively low migration rates. Spatial population structure causes the curve of equilibrium to become a region of equilibrium. Again, we have empirically verified the equilibrium (and its transient instability) via artificial evolution (with a stepping stone model of migration), and also studied the evolutionary dynamics of populations away from equilibrium.

Although the mathematics to derive the equilibria under diploid genetics are apparently intractable, we expect the equilibria (in terms of phenotype frequencies) to be same as in the haploid model that Kirkpatrick analyzed. Unlike the mathematical model, the simulation model is easy to extend to handle diploidy. We have empirically verified that the diploid equilibria are at least near, if not identical to, the haploid equilibria, for both panmictic and spatially structured populations. This demonstrates that simulated evolution can supply empirical evidence that the ana-

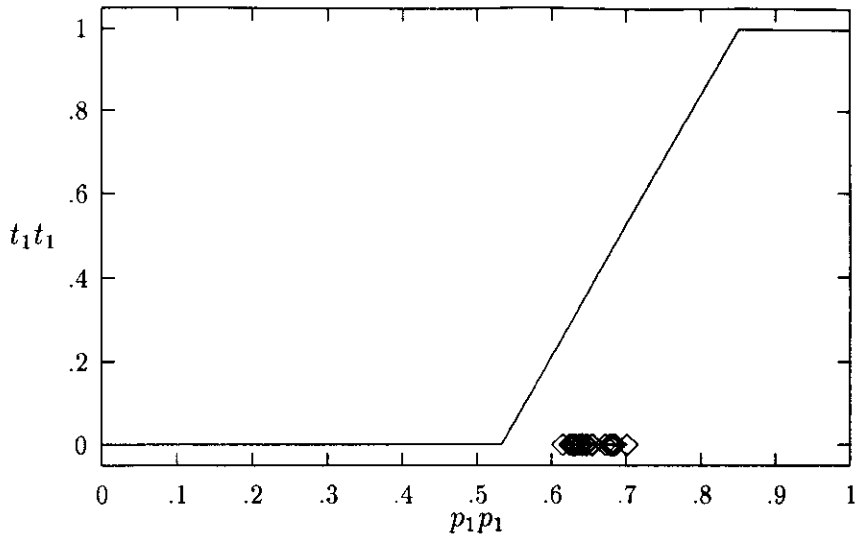


Figure 3.10: The path of a single population through 1000 generations with panmictic mating, $a_0 = 2.5$, $a_1 = 2.0$, $s = 0.2$, $\mu = 0.00001$, and $N = 131,072$. The run begins with $p_1p_1 = 0.7$ and $t_1 = 0.0$. The $\diamond$'s are at 50 generation intervals. The equilibrium predicted by Kirkpatrick is indicated by the straight lines.

lytic model is robust with respect to variations that cannot be handled analytically. Although the equilibria are not affected by diploidy, the dynamics of the evolution can be very different.

We have studied the particular case of the invasion of a preferred but recessive allele into the population. Under haploid genetics, all alleles are expressed in the phenotype, and the invasion of the new allele proceeds rapidly. However, in diploid organisms (under simple dominance), recessive alleles are not expressed in the phenotype unless the allele is present in both copies of the locus. Under random mating, this occurs with a frequency that is the square of the frequency of the recessive allele in the population. This means that forces such as mutation, migration, and random genetic drift are required to get the invading allele to high enough frequency that the associated phenotypic trait begins to appear in the population. This makes it difficult for preferred recessive alleles to invade and take over the population, despite their selective advantage. This problem is most severe for large panmictic populations, due to the relatively slow rate of drift.

It is interesting to note that we observed interesting relationships among the variations that we implemented. In the haploid population, invasion of the $T_1$ allele was slowed down by spatial structure (compared to panmixia). However, in the diploid population, spatial structure had the opposite effect, resulting in the faster invasion of the recessive $T_1$ allele. Spatial structure has the effect of slowing down gene flow within the population. In the case of the very fast haploid invasion, the reduced gene flow resulted in slower evolution. However, in the diploid case, the reduced gene flow and the associated increased rate of drift within demes resulted in much faster invasion of the preferred, but recessive $T_1$ allele. In retrospect, this seems

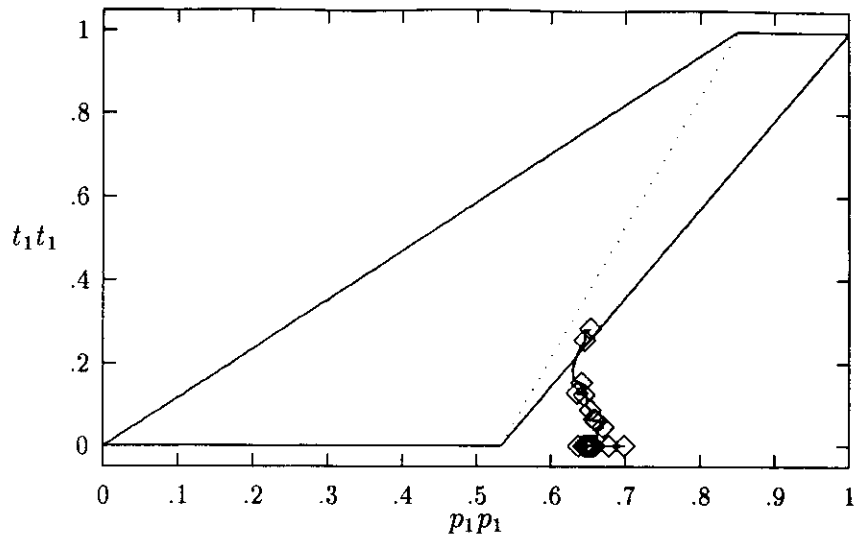


Figure 3.11: The path of a single population through 1000 generations with stepping stone structure (16 demes, each consisting of 8,192 individuals), $a_0 = 2.5$, $a_1 = 2.0$, $s = 0.2$, $\mu = 0.00001$, and $N = 131,072$. The run begins with $p_1p_1 = 0.7$ and $t_1 = 0.0$. The solid lines define the predicted region of equilibrium for local mating, and the dotted line is the equilibrium predicted by Kirkpatrick for panmixia.

obvious, but it is a subtle enough effect that we did not have the foresight to predict it.

# Chapter 4

# Parasite: The Evolution and Maintenance of Sex

As we have seen, artificial evolution can be applied rather easily to models in the style of traditional analytical population genetics, and can provide important insights that cannot be easily revealed via analysis. While these types of models are very important, the potential utility of artificial evolution in biology is not limited to them.

Many of the hypotheses that evolutionary biologists would like to (and do) study are not amenable to the ultra–simplification that is required for an analytical treatment. However, we can construct empirical tests for these sorts of hypotheses. In this chapter, we focus on a hypothesis that offers one possible explanation as to why evolution might favor sexual (versus asexual) reproduction: host/parasite coevolution. We refer to our simulation as **Parasite**.

## 4.1 The Problem of Sexual Reproduction

One of the most significant outstanding problems in the study of evolution is the evolution and maintenance of sex (Bell, 1982; Michod and Levin, 1987). In particular, why did sexual reproduction evolve, and why has it remained so prevalent? This is such a perplexing problem, because sexual reproduction is usually much more energetically and genetically expensive than asexual reproduction. Sex must provide significant adaptive advantages to overcome this cost. We can, to some extent, measure the costs. However, measurable benefits have remained somewhat elusive.

### 4.1.1 The Definition of Sex

In this chapter, the term "sex" refers neither to gender nor the act of mating. Instead, "sex" refers only to the production of offspring that possess a combination of the genetic material of two or more parent organisms.

The main effect of sexual reproduction is *mixis*, the mixing of the genes of the two parents. In the most common sexual systems (Bell, 1982), mixis occurs at two levels: the independent assortment and segregation of the different chromosomes, and the

### 4.1.3 The Benefits of Sex

What is it about mixis that is so beneficial that it can overcome the two–fold genetic and the two–fold ecological cost of sex? Many hypotheses have been proposed to explain why, and under what conditions, sexual reproduction might be advantageous (Bell, 1982; Muller, 1964; Maynard Smith, 1987; Felsenstein and Yokoyama, 1976; Haigh, 1978; Seger and Hamilton, 1987; Hamilton, 1990; Rennie, 1992). Most of these hypotheses are probably right to some degree, with their relative importance varying from species to species and through time for any given taxon.

The hypothesis that we investigate in this chapter is that mixis is beneficial in rapidly changing environments, and in particular in the host species in host–parasite coevolution systems. Mixis can be beneficial in changing environments when the phenotypic traits that are subject to selection are coded for by many interacting genes (i.e. are polygenic), and when at equilibrium, the genes remain unfixed (variation is maintained). Many traits are polygenic, with nonextreme equilibria. The effect of mixis on such a trait is to produce individuals of many different gene combinations, and thus a range of phenotypes. If the environment is constantly changing so that the "optimal" phenotype keeps shifting, the population will be able to evolve relatively quickly to maintain a mix of combinations centered on the "optimal" value. In contrast, an asexual population can change the trait only in small increments due to mutations, and may not be able to track environmental changes quickly enough to avoid extinction.

## 4.2 The Parasite Hypothesis

The hypothesis that mixis allows rapid adaptation is plausible, but for this effect to have a significant influence on selection in favor of sexual reproduction, environments characterized by fairly rapid and sustained change must be common. The physical environment may or may not change quickly relative to the generation time of a species, but the biotic environment often undergoes rapid change. The *parasite hypothesis* suggests that parasites present their host species with a rapidly changing and challenging environment over long periods of time (Seger and Hamilton, 1987; Hamilton, 1990; Rennie, 1992). Furthermore, resistance to parasites in the host species is usually highly polygenic.

For many years, biologists have realized the prevalence and importance of parasitic species (May, 1983), but only recently have they begun to appreciate the dominant role they may play in the adaptation of their hosts. From an evolutionary point of view, the interactions of parasite and host species are ecologically similar to those in predator/prey relationships. However, the the term "parasite" is used when the species has a much shorter generation time than its host, whereas the term "predator" is used when generation time is as long or longer than its prey's. Its comparatively long generation time puts the host species at an evolutionary disadvantage, because the parasites may be able to evolve new methods of attack much faster than the host can evolve new defenses. With a constantly changing set of defenses and attacks, the hosts and parasites each provide the other with a rapidly changing environment,

process of recombination via crossover within individual chromosomes. Independent assortment provides the offspring with one haploid copy of each chromosome from each parent. On average, one fourth of the genetic material is derived from each of the four grandparents, one eighth from each great grandparent, etc. But a chromosome is not necessarily derived as a whole from a distant ancestor. Crossover events swap portions of a chromosome, so within the haploid copy that is passed to an offspring there may be portions of each of the grandparent copies of the chromosome. Assortment provides a mixing at the level of whole DNA molecules, while recombination mixes within DNA molecules.

### 4.1.2 The Costs of Sex

One of the largest costs of sex is the so-called *two-fold genetic cost* (Shields, 1987). The fitness of a genotype is a function of the mean number of viable offspring produced by those individuals carrying that genotype (Section 1.4.1). To be somewhat more precise, the fitness of a genotype is defined in terms of the representation of portions of that genotype in offspring in the next generation. Sexually reproducing individuals contribute only half of their genetic information to each of their offspring. Consider what would happen if a parthenogenic (asexually reproducing) mutant female that produced only parthenogenic female offspring arose in an otherwise sexually reproducing population. Rather than producing unfertilized eggs, she would self-fertilize her eggs and produce offspring without the aid of a male. Assuming that she could produce as many offspring as the wild-type (unmutated, sexually reproducing) females, she would produce twice as many copies of her genome as the wild-type. By mutating to asexuality, the parthenogenic female would double the frequency of her genotype each generation, and her lineage would take over the population in $O(\log N)$ generations. This genetic cost of sex can be mitigated somewhat by inbreeding (mating with genetically related individuals) (Williams, 1980; Shields, 1987); the closer the relation between the mates, the closer the relation between parent and offspring.

Another source of inefficiency in sexual reproduction is the cost of males (Ghiselin, 1987; Seger and Hamilton, 1987), which is sometimes referred to as the *two-fold ecological cost* of sex. In most sexual species that have two genders, the males contribute little or nothing to their offspring other than gametes. The half of the offspring in each generation that are males are essentially a drain on the ecosystem: half of the ecological resources of the species' niche are spent on males that are non-productive (in the sense that they do not invest their resources in their offspring). The advantage of the parthenogenic mutant female here is that she does not waste any of her offspring on males, producing twice as many females as her sexual counterparts. If the males provide some resources to their offspring, then the ecological cost of males will be somewhat lower (Seger and Hamilton, 1987; Shields, 1987). An unequal sex-ratio (in favor of more females) will also mitigate part of this cost.

Other costs of sexual reproduction include the energy needed to find a mate, more complex reproductive systems, avoiding mating with the wrong species, etc.

parasitic population, spatial structure (isolation by distance), a large but finite and drifting population, and 100 loci of host–parasite interaction, each with two alleles.

The evolution in our model is driven by a genetic algorithm. The host genome consists of two chromosomes, one for the host–parasite interaction loci, and one for the recombination modifier locus. These two chromosomes segregate independently. The parasite has only one chromosome, which contains the loci involved with host-parasite interactions. All chromosomes in both species are subject to recombination and mutations (bit flips). See Section 2.4.2 for the recombination algorithm, and Section 2.4.3 for the mutation algorithm.

Some simulation studies suggest that selection for an increased recombination rate is likely to be stronger in models that incorporate more loci that are subject to selection (Martin and Cockerham, 1960; Lewontin, 1964; Franklin and Lewontin, 1970). For this reason, we model the host–parasite interactions with 100 loci in each species (100 loci is an enormous number compared to the 1 or 2 loci that are typical of population genetics models.) There are two possible alleles for each of these 100 loci. As we noted above, all 100 loci are placed on the same chromosome, so the only possibility for mixis among them is by recombination.

| Integer | Binary | Gray |
|---|---|---|
| 0 | 000 | 000 |
| 1 | 001 | 001 |
| 2 | 010 | 011 |
| 3 | 011 | 010 |
| 4 | 100 | 110 |
| 5 | 101 | 111 |
| 6 | 110 | 101 |
| 7 | 111 | 100 |

Table 4.1: Comparison of binary–coded and Gray–coded integers.

The host's recombination modifier locus is a 10–bit gray–coded unsigned integer, with 1024 alleles (possible values). Gray–coding is a way of encoding integers in bit–strings so that all consecutive integers differ by at most one bit position (see Table 4.1). This coding scheme allows a smooth progression from one value to another via mutation, although some mutations can still cause large changes in the value of the encoded number. This integer modifier allele is scaled linearly to the range $[0.0, 0.1]$ to specify the probability per locus of a crossover. The modifier locus is on a different chromosome from the host–parasite interaction loci; the two chromosomes segregate independently.

Both populations (the hosts and parasites) are placed in a two–dimensional, toroidal grid, with exactly one host and one parasite in each grid location. The grid structure both matches a parasite to each host, and defines a neighborhood for local mating. Both populations are the same size, each consisting of 65,536 individuals in each generation.

although the change seen by the host may be more dramatic, due to the shorter generation time of the parasites. It is likely that a good host strategy is to maintain diversity in the population and recombine each generation, challenging the parasites with a different set of defenses each generation. This might offset the speed advantage of the parasites (Levin, 1975; Glesener and Tilman, 1978; Jaenike, 1978; Bremermann, 1980; Bremermann and Pickering, 1983; Hamilton, 1980; Hamilton, 1982; Hamilton, 1986; Hamilton et al., 1981; Anderson and May, 1982; Bell, 1982; Price and Waser, 1982; Tooby, 1982; Rice, 1983).

## 4.3 Testing the Parasite Hypothesis

One of our goals in this chapter is to demonstrate that artificial evolution can be used to study significant biological problems such as the evolution and maintenance of sex. We are not trying to solve the problem of sex once and for all, as there may be no single explanation. Instead, we have designed and implemented **Parasite**, a simple model of host–parasite coevolution, in order to test the plausibility of the parasite hypothesis.

One approach to testing this hypothesis would be to model the host species as having the ability to reproduce either sexually and asexually, under the control of a heritable gene. We could then observe the evolution of this "sex" gene under the influence of fitness based on interactions with a coevolving parasite species. While this model is not too complex to simulate, we have chosen a somewhat simpler model that will get at many of the same issues.

Instead of observing the evolution of a "sex" gene, we use a recombination rate modifier gene in our artificial evolution simulation. While this does not directly attack the problem of the evolution of sex, it measures one of the components of mixis. In our simulation, we place all of the loci involved in the host–parasite interactions on one chromosome, so crossover is the *only* component of mixis in our system. The recombination rate modifier gene specifies the rate of mixis, and thus we can directly measure its evolution on a continuum from no mixis (effectively asexual reproduction) to significant mixis (strong sexual effects).

Placing the recombination rate under genetic control is a biologically realistic thing to do; there are heritable variations in the rate of recombination within many populations (Brooks, 1987). It is a well–known analytic result that in a stable environment (i.e. at equilibrium), there is selection for decreased recombination rates (Fisher, 1930; Feldman et al., 1986; Felsenstein, 1987), and thus reduced mixis. The parasite hypothesis suggests a counterbalancing selective force that favors higher recombination rates, and higher rates of mixis.

Despite the variety of analytic models for the evolution of recombination rates, none of them include realistic population dynamics (Brooks, 1987), and in most cases are limited to two loci, each with two alleles. Our model simultaneously includes mutation, a heritable recombination rate (with the probability of a crossover varying from 0.0 to 0.1 between each pair of loci), selection via competition with a coevolving

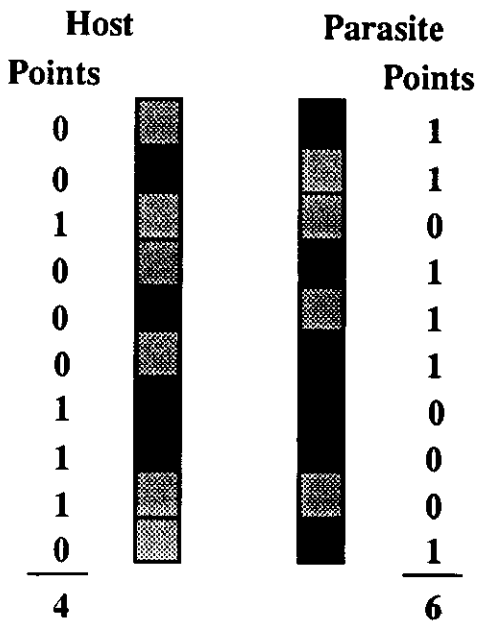


Figure 4.1: Host–parasite fitness calculation. The two chromosomes are aligned; where they match the host gets a point, and where they differ the parasite gets a point. Here, the host has 4 points and the parasite has 6 points. Equation 4.1 is then applied. Here, $l = 10$, so the fitness of the host is $-1$ and the fitness of the parasite is 1.

The host–parasite competition is very simple, consisting of a complementary gene-for–gene system for interaction. Each species has a 100–bit haploid chromosome (the 100 loci). When a parasite infects its host, their chromosomes are aligned (Figure 4.1). Each allele in the host is a "defense" and the parasite's allele at the corresponding loci is its corresponding "attack." If the defense matches the attack, the host scores a point, while if the defense does not match, the parasite scores a point. This calculation is performed for all 100 loci, and the points for each individual are added and normalized, producing a fitness score $s_i$ for individual $i$:

$$s_i = \sum_{j=0}^{l-1} point(j) - \frac{l}{2} \tag{4.1}$$

where $l$ is the number of loci (in this case 100) and $point(j)$ is 1 if the individual scores a point at locus $j$, otherwise it is 0. If the host and its infecting parasite are equally balanced (match on half the loci), both will have a fitness of 0, while if the parasite has the upper hand, it will have a fitness score greater than 0 and its host will have a score less than 0.

Both the hosts and parasites reproduce sexually; the parasites have a constant recombination rate, while the recombination rate in the hosts is heritable. Once the fitness value for each individual is determined, two parents are chosen for each location to produce an offspring organism in that square. The first parent is chosen by randomly sampling two individuals within the 5 x 5 region centered on the offspring location with an approximately normal distribution of parent–offspring distances (see Section 2.4.1). The higher scoring of the two individuals that are examined is selected

as the first parent. The second parent is determined in the same way (with replacement, so the same individual could be chosen as both parents of a new individual). The genetic material of the two parents is copied and the recombination rate of one of the parents (chosen randomly) is used to recombine the two chromosomes to produce the haploid offspring, which is then mutated at a rate of $\mu = 0.0001$ per locus (bit). In the hosts, the same recombination rate is used to perform crossovers between the modifier chromosomes, then mutations are performed at the same rate of $\mu = 0.0001$ per locus. The identical method of selection and mating is used within both the host and parasite populations.

As we noted above, parasites by definition reproduce more quickly than their hosts. To simulate this effect, we allow the parasites to reproduce $p \geq 1$ times during each host generation. The simulation proceeds like this:

1. Initialize the host's recombination modifier genes to 0
2. Initialize all other loci in both the host and parasite to 0 (hosts begin with perfect defenses)
3. Determine fitness in the host population
4. Select and reproduce in host population
5. do $p$ times
   (a) Determine fitness in the parasite population
   (b) Select, and reproduce in parasite population
6. Goto 3

All of the evolution parameters (e.g. mutation rate) are the same for both species, with the exception of generation time and recombination rate. The multiple parasite generations per host generation gives the evolutionary advantage to the parasites. On the other hand, we allow the hosts to compensate by increasing their rate of mixis via the recombination modifier gene.

Figure 4.2 summarizes our simulation results. Each simulation run lasts 2,000 generations, allowing the modifier gene to reach its "equilibrium" value. We vary the speed of the parasite evolution by varying both $p$ (the number of parasite generations per host generation), and the parasite recombination rate ($\rho_p$). In all cases, there is strong selection for non–zero recombination rates. The equilibrium rate generally increases with faster parasite evolution.

The dynamics of these simulations are quite interesting (Figures 4.3–4.7). The runs begin with the arms race biased completely in favor of the hosts, and with no mixis in the host. It turns out that both of these initial conditions are far from their equilibrium values. Because the hosts begin so far ahead in the arms race, there is not strong selection pressure for higher recombination rates in the early generations. In Figure 4.3, by around generation 400 the parasites are outcompeting their hosts, creating selection pressure for higher recombination rates. By generation

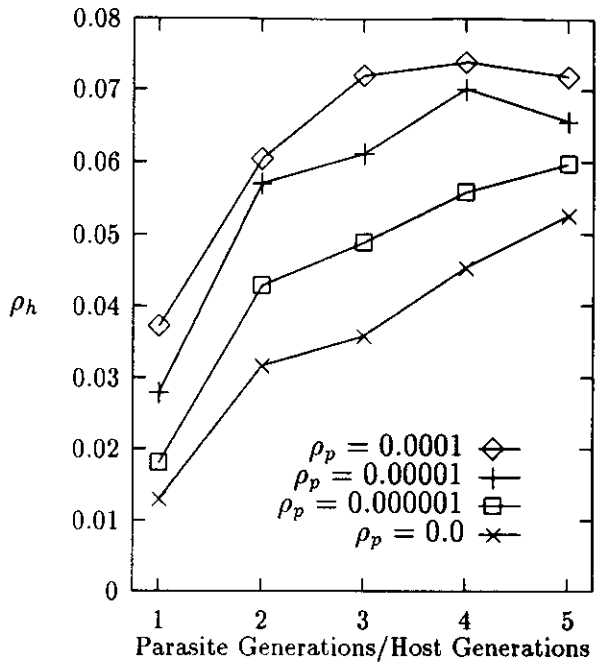


Figure 4.2: The equilibrium recombination rate in the host population ($\rho_h$) as a function of the number $p$ of parasite generations per host generation. Each data point is the mean of 9 runs at generation 2000.

600, the mean host recombination rate has increased significantly, and in fact the hosts begin to get ahead of their parasites in the arms race. The parasite fitness curve exhibits a damped oscillation which appears to settle down by about generation 2,000 (Figure 4.4). The host recombination rate appears to increase slowly throughout the run.

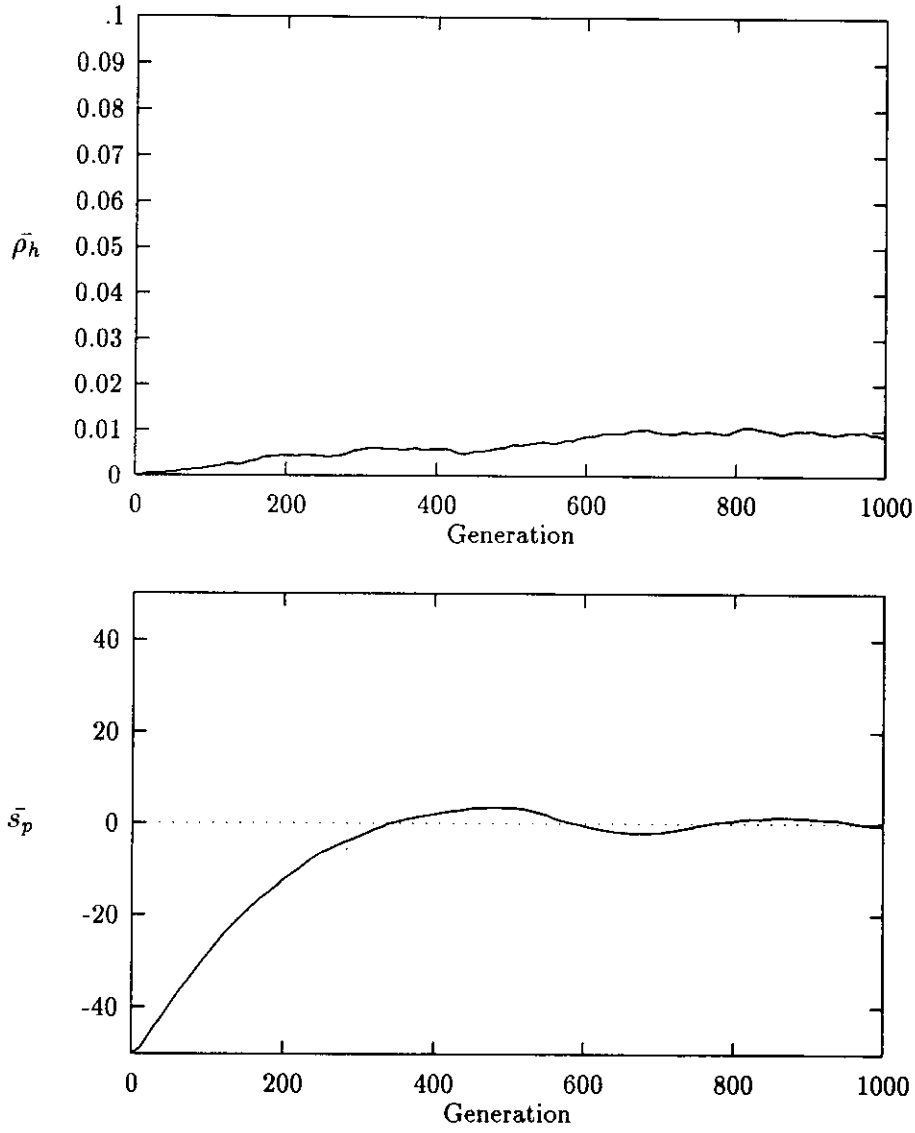


Figure 4.3: The dynamics of the host and parasite evolution with the parasite recombination rate $\rho_p = 0.0001$ and the number of parasite generations each host generation $p = 1$. The run up to host generation 1,000 is shown here (the complete run to generation 10,000 is shown in Figure 4.4). The top graph tracks the evolution of the host recombination rate $\rho_h$. The bottom graph shows the damped oscillation of the mean fitness of the parasite population. Where $\bar{s_p} < 0$, the hosts are more fit on average, where $\bar{s_p} > 0$, the parasites are more fit, and where $\bar{s_p} = 0$, the two populations are equally fit. The size of each population is $N = 65,536$.

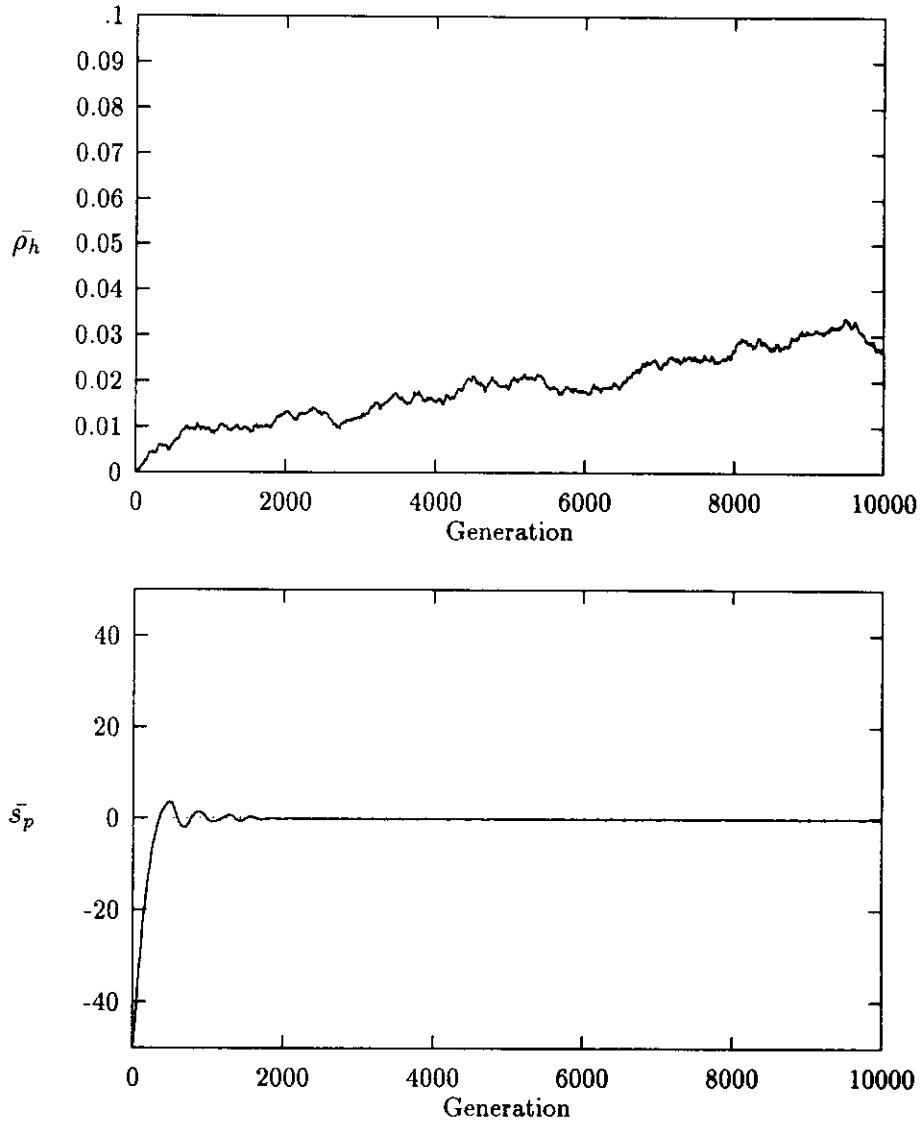


Figure 4.4: The dynamics of the host and parasite evolution with the parasite recombination rate $\rho_p = 0.0001$ and the number of parasite generations each host generation $p = 1$. The run up to host generation 10,000 is shown here (the first 1,000 generations are highlighted in Figure 4.3). The top graph tracks the evolution of the host recombination rate $\rho_h$. The bottom graph shows the damped oscillation of the mean fitness of the parasite population. Where $\bar{s_p} < 0$, the hosts are more fit on average, where $\bar{s_p} > 0$, the parasites are more fit, and where $\bar{s_p} = 0$, the two populations are equally fit. The size of each population is $N = 65,536$.

Figure 4.5 presents another simulation, differing from the run in Figure 4.4 only by virtue of slower parasite evolution due to a parasite recombination rate of $\rho_p = 0$, so the parasites are effectively asexual. Figures 4.6 and 4.7 present additional simulations, differing from the runs in Figure 4.3 and 4.5, respectively, only in that they have faster parasite evolution due to $p = 5$ parasite generations per host generation. The overall behavior is qualitatively similar. The major differences due to faster parasite evolution are wider oscillations in the mean parasite fitness curve, a higher maximum level of host recombination rate ($\rho_h$), and $\rho_h$ seems to peak and then drop to a lower level over many generations.

Again, the system seems to have stabilized by generation 1000, although this is not to say that evolution has stopped. Remember that the model contains spatial structure (via isolation by distance). While the mean allele frequency and fitness for the population as a whole may have stabilized, there could still be strong local fluctuations.

## 4.4 Implementation Notes

**Parasite** requires about 1200 lines of code beyond the core library of routines. About 300 lines implements the fitness function calculations for the two species, 720 lines for instrumentation, and 180 lines for run-time selection of instrumentation and parameter options. The simulation results presented in this chapter required approximately 6 days on the UCLA 16K processor Connection Machine-2.

## 4.5 Discussion

The empirical evidence from the **Parasite** simulations suggests that under fairly realistic evolutionary dynamics, parasites can cause selection for higher recombination rates. Although our model is based on very simple host–parasite interactions, it does have some connection to the real world. The one–to–one interactions between the host and parasite loci appear to be common in some groups of organisms (Flor, 1956; Day, 1974; Barrett, 1983; Barrett, 1985), but in many cases the interactions are more complex (Barrett, 1985). This is really not a problem; more complex genetics are not difficult to implement in our artificial evolution paradigm, so the more complex situations can be simulated.

These are significant results, because they provide empirical evidence that host–parasite coevolution can result in strong selection for higher recombination rates (mixis) in a relatively realistic simulation. This selective advantage due to mixis bears directly on the problem of the maintenance of sexual reproduction, and under what conditions there is selection for higher rates of mixis.

A shortcoming of this study is that we have measured neither the actual strength of the selective advantage for or against higher recombination rates. This information would tell us how great of an influence the parasite population must have on the host's fitness in order to provide selection for higher recombination rates (and thus sexual reproduction).

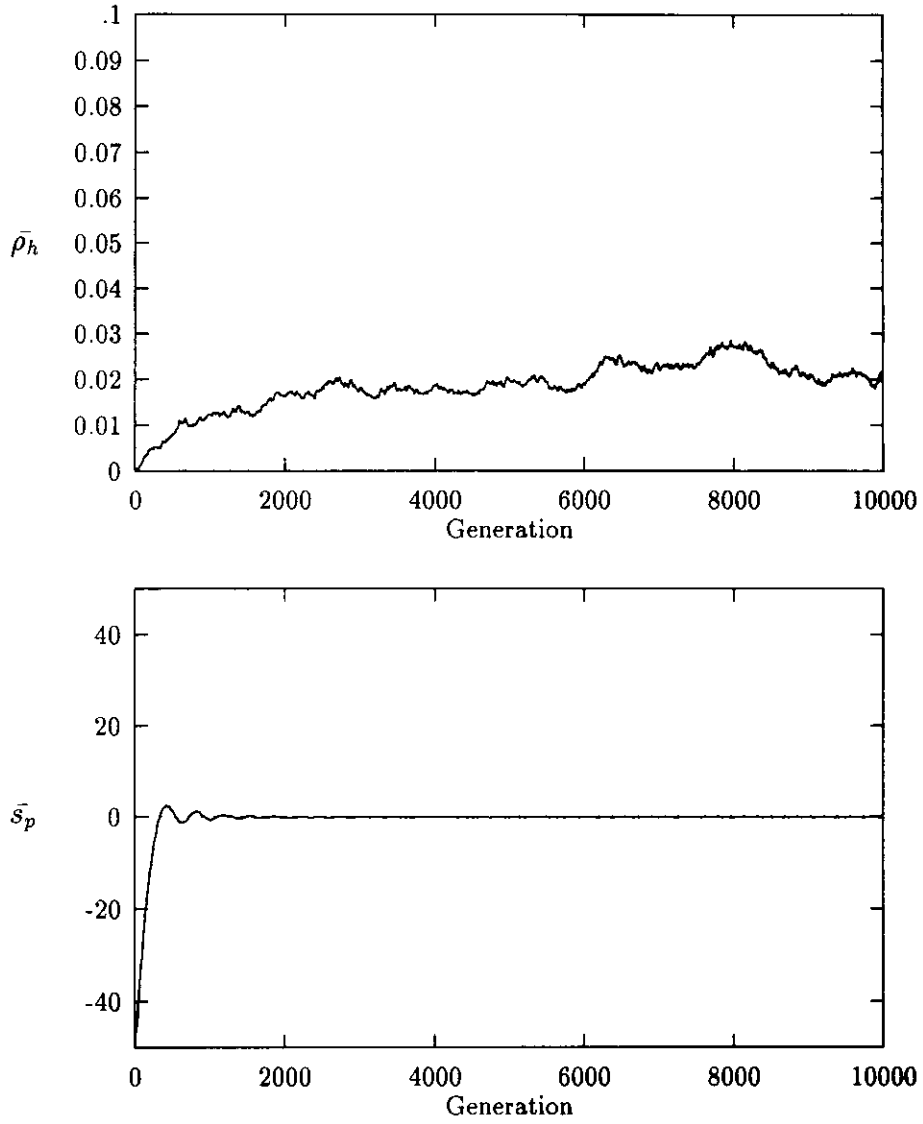


Figure 4.5: The dynamics of the host and parasite evolution with the parasite recombination rate $\rho_p = 0.0$ and the number of parasite generations each host generation $p = 1$. The run up to host generation 10,000 is shown here. The top graph tracks the evolution of the host recombination rate $\rho_h$. The bottom graph shows the damped oscillation of the mean fitness of the parasite population. Where $\bar{s_p} < 0$, the hosts are more fit on average, where $\bar{s_p} > 0$, the parasites are more fit, and where $\bar{s_p} = 0$, the two populations are equally fit. The size of each population is $N = 65,536$.

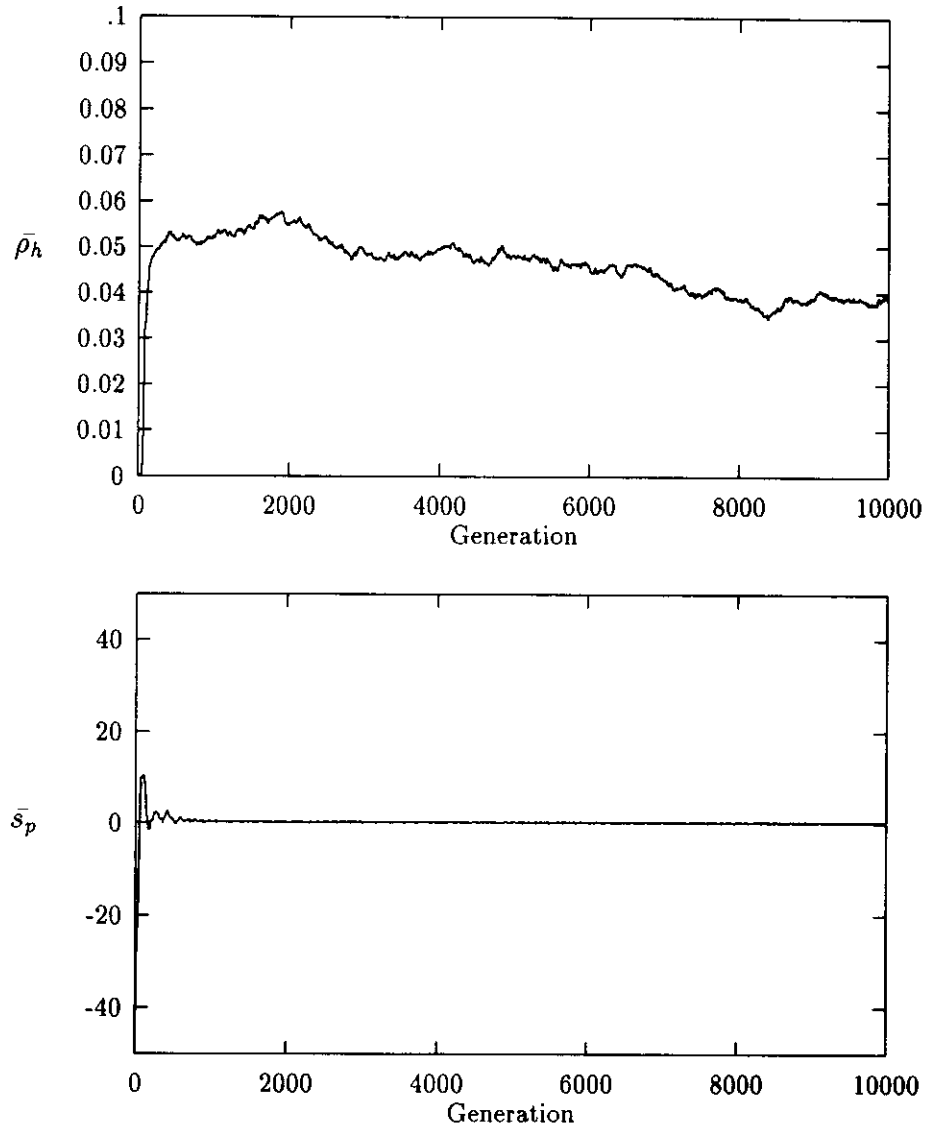


Figure 4.6: The dynamics of the host and parasite evolution with the parasite recombination rate $\rho_p = 0.0001$ and the number of parasite generations each host generation $p = 5$. The run up to host generation 10,000 is shown here. The top graph tracks the evolution of the host recombination rate $\rho_h$. The bottom graph shows the damped oscillation of the mean fitness of the parasite population. Where $\bar{s_p} < 0$, the hosts are more fit on average, where $\bar{s_p} > 0$, the parasites are more fit, and where $\bar{s_p} = 0$, the two populations are equally fit. The size of each population is $N = 65,536$.

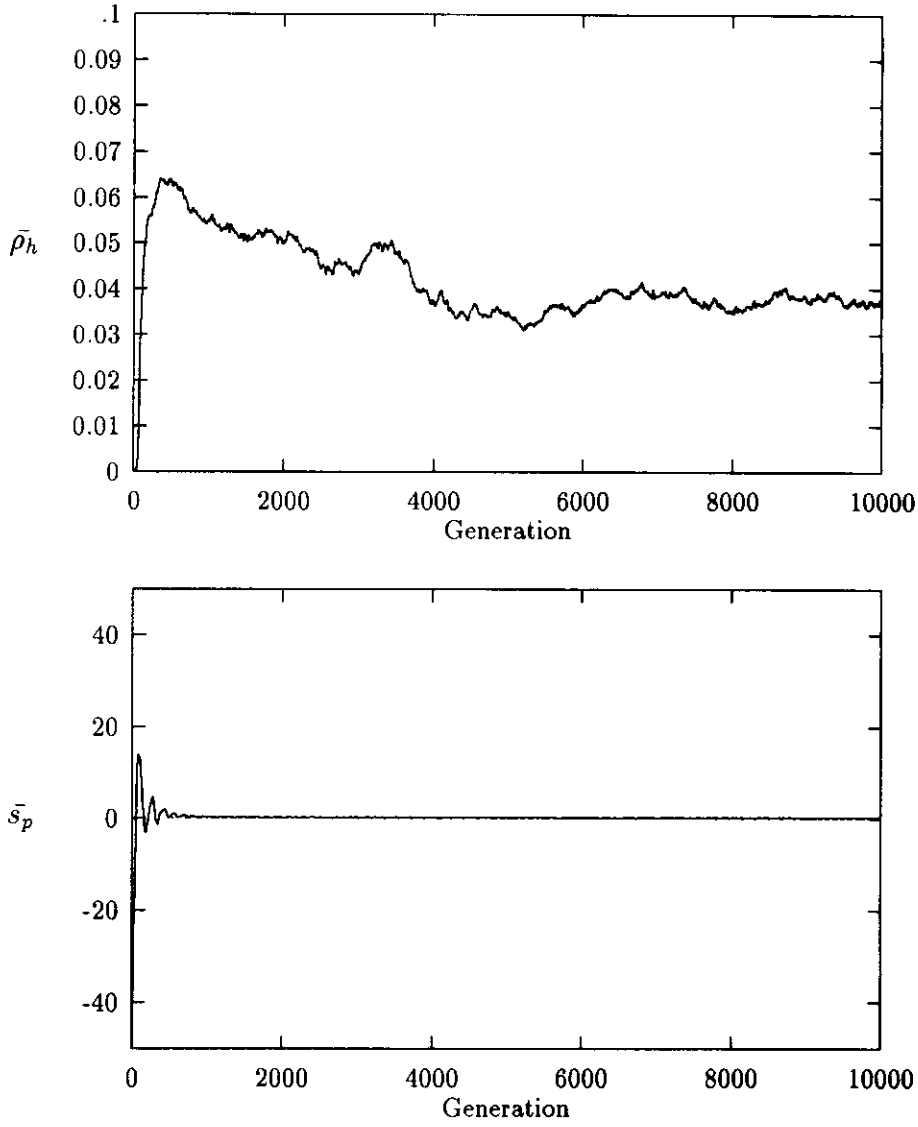


Figure 4.7: The dynamics of the host and parasite evolution with the parasite recombination rate $\rho_p = 0.0$ and the number of parasite generations each host generation $p = 5$. The run up to host generation 10,000 is shown here. The top graph tracks the evolution of the host recombination rate $\rho_h$. The bottom graph shows the damped oscillation of the mean fitness of the parasite population. Where $\bar{s_p} < 0$, the hosts are more fit on average, where $\bar{s_p} > 0$, the parasites are more fit, and where $\bar{s_p} = 0$, the two populations are equally fit. The size of each population is $N = 65,536$.

# Chapter 5

# AntFarm: The Evolution of Cooperative Foraging I

One of our aims in this chapter (and continuing in Chapter 6) is to explore the options and tradeoffs for the representation artificial organisms. The computer program that specifies the heritable behavior or the artificial organism must be represented at two levels: (1) as a bitstring chromosome, so we can perform realistic genetic operations on the program, and (2) as an executable program, to drive the organism's behavior. Also of great importance is the strong effect that the design of the artificial morphology and environment can have on the types of behaviors that are likely to evolve. A major implementation issue that arises from the evolution of behavior in complex environments is the fact that the simulated organisms may attempt to violate the logical constraints of the artificial world. We explore all of these issues in the context of **AntFarm**, which evolves (potentially) cooperative foraging behavior in colonies of ant-like organisms.

In **AntFarm**, we are far more interested in the evolution of the phenotype (defined in terms of behavior, not morphology) than in the details of the underlying allele frequencies. **AntFarm** is our first example of an artificial evolution simulation that requires us to specify and simulate an explicit environment and explicit simulated morphology of the organisms.

## 5.1 Cooperative Foraging in Ants

The dominant insects throughout the world are the ants. All ant species have eusocial societies, characterized by overlapping generations, care of the young by adults, and adults divided into reproductive castes (kings and queens) and nonreproductive castes (workers). Ants live in colonies ranging in size from a few individuals to more than 20 million (Hölldobler and Wilson, 1990). Ant societies have a high degree of organization. Most communication between ants is either tactile, visual, or chemical. Large-scale coordination is achieved through the use of pheromones (chemicals used in communication).

Each individual ant is relatively small and simple, typically able to perform only 20 to 42 distinct behaviors (Hölldobler and Wilson, 1990), yet the emergent behavior of the colony as a whole is amazingly complex. In many contexts, myrmecologists treat the whole colony as a single *superorganism*. The unparalleled success of these superorganisms in all parts of the world, with perhaps as many as 20,000 species (Hölldobler and Wilson, 1990), speaks well for the strength and versatility of the eusocial colony.

In most ant species, much of the life cycle occurs in underground nests. The most easily observed behavior is workers foraging for food. Foraging workers do not immediately eat the food, but carry it back to the nest, where it is processed and consumed by all members of the colony (central place foraging). In many species, a high degree of coordination and cooperation between foragers is observed, often mediated by pheromones (chemical communication).

Central place foraging consists of two phases: (1) the search for food and (2) its recovery to a central location (Sudd and Franks, 1987). Much of the cost of foraging is associated with search (Fewell, 1988; Lighton, 1990), but all of the payoff is from recovery, which consists primarily of transportation of the food to the nest. Foraging strategies that minimize search time will clearly be advantageous.

The Johnson, Hubbell, and Feener (1987) model of central place foraging in eusocial insects is fairly complex and the details are beyond the scope of this dissertation. It predicts the effect of the size and spatial distribution of food "patches" on the number of foraging workers and the style of foraging. In species that feed on small patches of food, a small number of workers, each foraging alone, is optimal. Recruitment of nestmates to help recover the food does not pay off because the food patches are small. In this model, the search for food (in the absence of recruitment) is assumed to be a random walk beginning at the nest so that the area around the nest is searched many times by different foragers. As the number of foragers increases, the amount of additional area searched per forager decreases. The diminishing returns for additional foragers results in the optimality of a small foraging force.

Species that feed on large patches of food should have a large foraging force, with heavy reliance on recruitment. When a patch is too large for the discovering ant to harvest alone, it pays to recruit (rather than rely on rediscovery by other foragers). Recruitment of nestmates to help harvest a known food source can nearly eliminate search costs. With reduced search costs, the diminishing returns for additional workers is not such an important factor, resulting in a large foraging force being optimal.

In real ants, recruitment to harvest food resources has been found to take many different forms (Hölldobler and Wilson, 1990). In the simplest case, a second ant is physically led to the food in a process called *tandem running*. More common is *group* recruitment, which uses a short-lived pheromone trail to bring up to a few dozen workers to the food source. The most impressive form is called *mass* recruitment. In mass recruitment, a relatively fixed, long-lived pheromone trail leads hundreds or thousands of workers to the food source. The trail is reinforced by each successful forager. Mass recruitment is used only in species that forage for food that is found in very large clumps.

We would like to understand more about the evolution of cooperative foraging. This chapter is focused on the artificial evolution of foraging behavior in colonies of

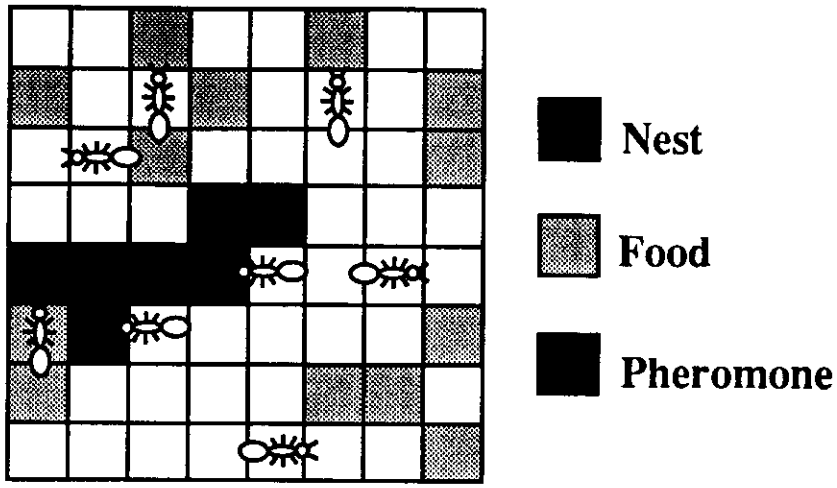


Figure 5.1: The **AntFarm** environment contains a nest, food, pheromone, and ants. Only a small portion of the world and population is shown here. At the beginning of each generation, all the ants are in the nest, food is distributed in the environment, and not pheromones are present.

artificial ants. We refer to the various simulations as **AntFarm**. The **AntFarm** model consists of an evolving population of ant colonies. Each colony is made up of dozens to hundreds of genetically identical ants whose behavior is specified by an artificial neural network (ANN). In addition to the ability to sense and carry food, the ants can sense and drop simulated pheromones. The reproductive success of a colony is a function of the amount of food carried to its nest, producing a selection pressure favoring better foraging strategies.

## 5.2 The AntFarm World

The **AntFarm** evolution is driven by the genetic algorithm described in Section 2.4, operating at the level of colonies (superorganisms) of genetically identical ants, not at the level of individual ants. The behavior and actions of all of the ants in a colony contribute to the colony's fitness score. Each colony has a single chromosome that codes for the behavior function used by all of its ants. All members of a colony are genetically identical, although each ant receives different sensory input, so each may behave differently. Fitness is based primarily on the number of pieces of food carried into the nest, so more efficient foraging means a higher score and greater reproductive success, causing selection pressure for better central place foraging strategies. The initial population consists of randomly generated chromosomes.

**AntFarm** evolves a population of thousands of colonies, with dozens of ants per colony, often with a total of millions of ants. The colonies live in a toroidal grid environment, where each grid location contains some number of ants along with information about the presence or absence of a nest, food, and pheromone ("odor") at that location (Figure 5.1). Any pheromones that are dropped by the ants slowly diffuse and eventually disappear.

At each time step, each ant can sense nest, food, and pheromone in its local neighborhood. The behavior function takes this sensory information as inputs. Based on these inputs and on its internal state, the behavior function outputs the desired motor functions and updates its internal state. The internal state (memory) varies from ant to ant, even within the colony, and changes through time. The possible motor functions that can be performed on any given time step include moving, picking up and dropping food, and dropping pheromones. Note that no ant can directly sense or affect any other ant in the simulation; all interactions occur through changes in the shared environment. This makes it impossible for recruitment strategies that depend on tactile communication, such as tandem running, to evolve. By restricting communication to the use of pheromones, the simulation is greatly simplified. Also, this makes it easier to determine when information about the location of a food patch is actually being communicated.

Inter–colony interactions are possible. All of the ants from all colonies forage in a common environment, so there is direct competition for food. In addition, all of the colonies drop and sense the same pheromone, so they can interact through the use of pheromones.

At the beginning of each generation, the environment is reinitialized so that no pheromone is present and food is placed in a new configuration from a fixed probability distribution. Each generation begins with all ants in their nests and their memory initialized to zero. All ants live throughout the entire generation. A score is calculated for each colony based primarily on the amount of food deposited in the colony's nest in the allowed number of time steps, although the "metabolic" costs of ant movement, pheromone production, etc. are counted against the food that is brought to the nest. We include artificial metabolic costs to better simulate the flow of energy associated with foraging. The inclusion of metabolism in the score results in selection pressure towards more streamlined foraging strategies.

## 5.3 Representing the Ants

The problem of how to represent the artificial organisms is important and is one of the major themes of both this chapter and Chapter 6. The difficulties arise from the fact that we must represent the organism both as a computer program that can run to determine the behavior of the organism, and as a bitstring chromosome on which the genetic algorithm operates. The standard genetic operators of bit–level recombination and mutation must always (or at least usually) yield a legal, runable program. This section specifies the components of the representation and surveys a variety of representations that have been used successfully in the past.

In this chapter, we are considering the simulation of ants that live and reproduce in relatively complex environments, with many sensors (external and internal), and many possible actions at each moment. In addition the organisms often possess some amount of internal memory, allowing their behavior to be history sensitive. In the course of its life each organism is born, makes thousands of decisions (eat, move, mate, etc.), and eventually dies. The reproductive success of a particular organism

is affected by its behavior and by interactions with the environment throughout its lifetime. In **AntFarm**, the "organism" is a whole colony. The behavior of a colony is the aggregate behavior of many copies of the colony's program (and interactions with multiple copies of the programs of neighboring colonies).

The representation of an organism consists of the following parts (Figure 1.2):

- genotype: a bitstring that encodes the behavior function;
- development function: the mapping that decodes the genotype to produce the behavior function;
- behavior function: the program that maps sensory inputs and memories into memories and effector outputs; and
- interpreter: used to execute organism behavior functions (programs).

In all of our **AntFarm** simulations, the development function is fixed for all organisms and for all time; it is not subject to evolution. The genotype, of course, differs from colony to colony, but is static throughout a colony's life. At the time of reproduction, recombination and mutation operators are applied to a pair of parent genotypes to produce an offspring genotype. The behavior function of an organism, determined initially by the genotype and development function, can in principle change during an organism's lifetime if there is some provision for learning; however, our **AntFarm** simulations do not have such a feature, although our ants do have a small "memory" capacity.

In essence, the representation of an organism represents all of the biochemical machinery of natural life. In **AntFarm**, we simplify things somewhat by restricting ourselves to non-genetically coded developmental processes and physical morphology. In addition, we model behavior (mapping of sensory inputs to effector outputs) with a heritable computer program, rather than a more biologically realistic simulation of the physical structures and chemical reactions that underlie behavior in natural organisms. Even with all these simplifications, the representation problem is quite difficult.

In the remainder of this section, we consider a number of possible organism program representations and encodings for use as behavior functions in **AntFarm**: parameterized functions, Lisp S-expressions, finite state automata, rule systems, and artificial neural networks(ANNs). Although all of these representations/encodings have been used successfully in simple evolution simulations, none are entirely satisfactory for **AntFarm** as they have been used in less complex models. We reject some of the possible representations because the programs and genotypes grow too large when applied to organisms as complex as the **AntFarm** ant colonies. Others are rejected because they require us to restrict the types of behaviors that might evolve, or because they must be defined in terms of domain-dependent parameters and components. The ANN representation has acceptable properties, but the existing encoding schemes fail to evolve foraging behavior in **AntFarm**.

### 5.3.1 Parameterized Functions

RAM is a powerful simulator shell that is used for modeling population behavior and evolution (Taylor et al., 1989a). In RAM, the organism representation is a parameterized Lisp function (Steele, 1984) in which the programmer defines the possible behaviors of the organism. The exact behavior that an individual expresses depends on its environment and the parameters to its Lisp function. The parameters are the only portion of the organism that is heritable. These parameters are the behavior function of the organism, and the Lisp function is the organism interpreter.

The main problems with the representation of organisms as parameterized functions is that most of the behavior of the organism is specified in the interpreter, rather than in the behavior function—most of the behavior is not under genetic control. Evolving in a parameter space is not biologically realistic.

### 5.3.2 Lisp S–Expressions

Koza (1990) describes a powerful technique, that he calls genetic programming, for evolving computer programs that are represented genotypically as Lisp symbolic expressions (S–expressions). An S–expression has the form

$$(\text{function operand}_1\ \text{operand}_2 \ldots \text{operand}_N)$$

where *function* is to be applied to the $N$ operands. The set of possible functions is $F$, and each of the operands is either an S–expression or one of the set of terminal symbols $T$. Examples of functions that might be elements of $F$ include IF, AND, OR, NOT, $<$, $>$, $+$, $-$, $*$, etc. The genetic operator is recombination, which is defined such that legal S–expression syntax is maintained (the S–expression is encoded as a parse tree, not a linear bit–string). Mutation is defined in Koza's system, but apparently is never turned on in practice. A given S–expression is a behavior function, and the organism interpreter is constructed to implement the primitive functions in $F$.

Although Koza has successfully applied the evolution of S–expressions to a wide variety of problems, this representation is not appropriate for our intended use in realistic biological simulations. First, the encoding is not in a linear string, so biologically implausible genetic operators must be employed. Second, this technique requires that a set $T$ of terminals and a set $F$ of functions—the basic building blocks of the behavior function—be specified. The only heuristic for choosing these sets that Koza provides is that these sets must be "sufficient" (but not too big) for the particular problem, and the functions must be *closed*, i.e. they are chosen and defined such that any composition of the functions in $F$ is valid for any value that any operand might assume. If $F$ contains extraneous functions, serious performance degradation can result. In order to construct $F$ and $T$, we must decide what sorts of computations the behavior function needs to perform. Apparently, there is no one set $F$ that is appropriate for even large classes of problems. Like the use of parameterized functions (above), the S–expression representation requires us to bias and limit the possible outcomes of the evolutionary process by specifying task–specific information in the organism interpreter. The fact that Koza's technique is not appropriate for

realistic biological simulations is not surprising; genetic programming was designed as an engineering application of genetic algorithms, not for emulating life.

### 5.3.3 Deterministic Finite State Automata

Genesys/Tracker is a Connection Machine system that evolves large populations of simple organisms (Jefferson et al., 1991). Each Tracker organism has its own two dimensional environment containing a noisy, broken trail. The "fitness" of a Tracker animal is based on the amount of the trail that it can follow during its lifetime, so trail-following behavior evolves. The Tracker organism is very simple, receiving one bit of input that indicates the presence or absence of a trail marker in the location ahead of the organism. It can choose one of four options at each time step: move forward, turn left (in place), turn right (in place), or do nothing. The organisms each have five bits of internal memory. The population sizes in the Tracker experiments range from 8,192 to 262,144 organisms. An organism is represented by either a finite state automaton (FSA) encoded in a 453-bit chromosome or by an artificial neural network (ANN) encoded in 448-bits.

The FSA Tracker organisms use a deterministic input/output transducer FSA. Conceptually, the automaton consists of a table with four columns and an initial state. In the FSA Tracker organism representation, the genotype consists of the concatenation of the bit representation of the initial state and the rows of the FSA table. The rows of the table are placed in a canonical order, so only two of the columns are explicitly part of the genotype. The size in bits of the FSA behavior function is

$$(S + O)2^{I+S} + S$$

where $I$ is the number of bits of input and $S$ is the number of bits of state (memory), and $O$ is the number of output bits (encoding possible actions). In Tracker, $I = 1$, $S = 5$, and $O = 2$, for a total of 453 bits. During each time step, the FSA interpreter takes the sensory input and the previous internal state, and produces a motor output and a new internal state.

Although the FSA representation is fine for Tracker-sized problems, the exponential rate of scaling on the number of bits of input and internal memory makes finite state automata unsuitable for **AntFarm**. If the FSA representation were used for **AntFarm I** (described below), $I = 200$, $S = 21$, and $O = 13$, for a total of $\sim 10^{69}$ bits ($\sim 10^{62}$ megabytes). This is clearly not practical.

### 5.3.4 Primitive Rule–Based Organisms

Rule systems can be viewed as variations on finite state automata (e.g. classifier systems (Goldberg, 1989a)). In an FSA representation, each row of the table corresponds to the action and new state based on the inputs and current state. There is a row for every possible combination of inputs and current state (resulting in unfortunate scaling properties). Rule systems are an attempt to avoid the explosion in representation size; they do not represent actions for every possible situation; rather they "factor" the behavioral space so that each rule may represent multiple situations.

A rule–based behavior function consists of $K$ rules, each rule having four parts: inputs, current memory state, actions, and new memory state. The interpreter invokes the rule that has an (input, current state) pair that most closely matches the current situation to determine the action (behavior) and new state. The alphabet of the (input, current state) pair may include a "don't care" symbol, which will match any of the other symbols.

The genotype of the rule system representation consists of the concatenation of the bit representation of the $K$ rules and the initial state. The size of the representation is roughly

$$(I + S + O + 2S)K + S$$

bits, where $I$ is the number of bits of input, $O$ is the number of output bits, $S$ is the number of bits of memory, and $K$ is the number of rules. The main problem for the rule representation seems to be evolvability of appropriate behaviors from low–level sensory inputs. Consider the problem of following a gradient of pheromones in **AntFarm**, where the inputs are the nine pheromone levels in a 3 x 3 neighborhood of locations (as in the **AntFarm I** simulations described below). This simple task will require a large number of rules, because there is no way to express relations among the various inputs, except by enumerating the possibilities. To make the rule system expressive enough, $K$ will have to be made large, making the representation too big to be practical. Or, the rules would have to allow symbolic expressions, and we would have the same problem as with S–expressions, where we must specify what functions may be used in the expressions.

### 5.3.5 Artificial Neural Networks

In the Tracker simulations, the other representation (besides the FSA) is an artificial neural network (ANN). The Tracker ANN consists of three layers: input, hidden, and output (Figure 5.2). The input layer is fully connected to both the hidden and output layers. The hidden layer consists of five units (the five bits of memory) and is fully connected to both itself (recurrent connections) and the output layer. The activations of the input layer are set based on the sensory input from the environment, and the output layer specifies the desired motor action. The heritable portions of the behavior function consist of the weights, thresholds, and initial activations of the network. A generic ANN interpreter is used to execute the behavior function.

In the ANN Tracker organism representation, the genotype consists of the concatenation of the bit representation of the weights, thresholds, and initial activations of the network. The size of the representation is

$$W(IO + IH + H^2 + HO) + (T + A)H$$

bits, where $I$ is the number of inputs, $O$ is the number of outputs, $H$ is the number of hidden units (bits of memory), $W$ is the number of bits per weight, $T$ is the number of bits per threshold, and $A$ is the number of bits per initial hidden unit activation. In Tracker, $I = 2$, $O = 4$, $H = 5$, $W = 6$, and $T = A = 7$, for a total of 448 bits. This rate of scaling seems reasonable for our target organisms.

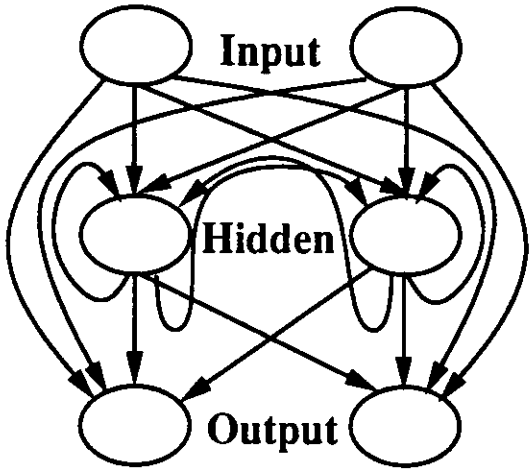


Figure 5.2: The Tracker ANN, showing the architecture of the neural network that controls the behavior of the Tracker organisms. The network consists of three layers. The inputs are fully connected to the hidden layer and the output layer. The hidden layer is fully connected to itself (recurrent connections) and the output layer. The actual Tracker ANN has two inputs, five hidden nodes, and four outputs.

The ANN abstraction is the most attractive of the program representations we have surveyed thus far. Unfortunately, as we demonstrate below, the Tracker-style ANN encoding is unable to evolve the behaviors required for foraging in **AntFarm**. Through the rest of this chapter and Chapter 6, we examine several alternative ANN architectures and encodings, searching for a suitable organism representation. Our final encoding scheme does result in the evolution of ant-like foraging behavior.

## 5.4 Overview of the AntFarm Simulations

We have implemented multiple versions of the general **AntFarm** framework. Each successive version either makes the simulated morphology more ant-like or uses a behavior function that leads to the evolution of more ant-like behavior. When the differences in the implementations are important, we refer to the particular version as **AntFarm I**, **AntFarm II**, etc. **AntFarm I** is our first implementation, and is characterized by a very unrealistic simulated morphology; the behavior that evolves is unnatural and asymmetric. **AntFarm II** makes the morphology more ant-like; the evolved behaviors are somewhat more natural, but still rather bizarre and asymmetric. **AntFarm III** addresses the issue of symmetry in the behavior function; the behavior is **AntFarm II**-like, but symmetric. The strange behaviors evolved by **AntFarm II** and **AntFarm III** are apparently a result of the behavior function's inability to switch between different modes of behaviors. **AntFarm IV** addresses this problem, evolving very realistic ant-like behavior. **AntFarm II** through **AntFarm IV** are described in Chapter 6.

In all four versions of **AntFarm**, the environment is shared among all of the ants of all of the colonies. The nests are placed at regular intervals within the environ-

ment. When real ants choose nest sites, such *overdispersion* usually occurs (Sudd and Franks, 1987). In nature, those colonies that are founded as far as possible from existing colonies suffer less from competition and direct attack. This makes it unlikely that two competing colonies will be found in close proximity in nature.

Also, in all cases the artificial ants are provided with a special sense organ (the "compass") that senses the direction to the nest, although the ants still must evolve behavior that interprets and uses the compass correctly. We chose not to try to evolve both foraging search strategies and strategies for navigating back to the nest in the same simulation. Real ants typically use elaborate techniques for navigation (Hölldobler and Wilson, 1990), often involving memorizing landmarks, calculating average angle of the sun during foraging, etc.

In addition, we always provide each ant with four bits of random sensory input at each time-step. The random input allows each ant in the colony to receive different sensory input, and also allows the ants to behave somewhat randomly. For instance, an ant cannot perform a random walk without using the random inputs.

## 5.5 AntFarm I

**AntFarm** is a descendent of the Tracker task studied on the Genesys system (Jefferson et al., 1991). **AntFarm I** evolves a population of 16,384 colonies, with 128 ants per colony, for a total of more than two million ants. The colony's genetic information is represented by a huge 25,590 bit haploid chromosome. Colony fitness is defined as follows: each unit of food is worth 1000 points, each unit of pheromone dropped by an ant costs 0.1, and each other action (move or pickup/drop food) costs 0.1. During reproduction, both crossovers and mutations occur at a rate of about 0.0001 per bit (about 2.6 mutations and crossovers per colony each generation), which we have found empirically to be satisfactory. Piles of food are scattered in the environment with a uniform probability of a pile being placed at each location.

In each of the 1000 time steps of its life, the ant's sensory inputs (and internal memory) are processed by its ANN-based behavior function, producing a set of actions to perform. An ant has a 3 x 3 sensory array centered on its current location that can sense

- the presence of food,
- the presence of a nest, and
- the amount of pheromone.

In addition, each ant can sense

- whether or not it is carrying food,
- the correct direction to its nest (a *compass* sensor), and
- 4 bits of random input.

| Dimension | **AntFarm I** | Genesys/Tracker |
|---|---|---|
| Population | 16,384 colonies<br>262,144 ants | 65,536 ants |
| Info/Environment Location | 32 bits | 1 bit |
| Level of Selection | Colony | Individual |
| Sensory Input/Time Step | $\sim$ 200 bits | 1 bit |
| Effector Outputs/Time Step | 13 bits | 2 bits |
| Internal Memory (max) | 21 bits | 5 bits |
| Genome Size | 25,590 bits | 450 bits |

Table 5.1: A comparison of **AntFarm I** to Genesys/Tracker. The **AntFarm I** simulation is larger and more complex in many dimensions.

In any time step, an ant can decided to do any or all of the following

- move to any of the eight neighboring locations (or not move at all),
- pick up a unit of food (although it can carry a maximum of one unit of food),
- drop a unit of food, and
- drop from 0 to 64 units of the pheromone.

The ants live on the grid. Their position is described by an integer $(x, y)$ coordinate, and when the ant walks it always moves to one of 8 neighboring grid squares. The ants do not have a changeable orientation; the orientation is absolute and they cannot turn. Their sensor and motor capabilities are (potentially) equally effective in all directions.

The main differences between Genesys/Tracker and **AntFarm I** are the result of the biologically motivated task (central place foraging) of **AntFarm**. Since **AntFarm** is trying to model natural evolution, it is implemented with a more realistic genetic algorithm (local competition and mating, rather than global competition and random mating). In addition, the simulated organisms are more complex in many dimensions (summarized in Table 5.1).

Here are the input/hidden/output details of the **AntFarm I** ANN behavior function. These are features that are available, but particular organisms may actually use many fewer:

- Input Units
  - 9 units for pheromone density
  - 9 binary units for presence of food
  - 9 binary units for presence of a nest
  - 4 binary units for compass (an optimal path to the nest)
  - 4 binary units for random noise
  - 1 binary unit for whether or not it is carrying food

- Hidden Units
  - 21 binary units for memory or internal computation
- Output Units
  - 4 binary units for direction to move
  - 1 binary unit to pick up food
  - 1 binary unit to drop food
  - 1 unit to indicate number of units of pheromone to drop

The next section introduces ANN-based representations in more detail, and compares several different ANN representations. Section 5.7 then chooses the best of these representations and describes the character of the behaviors that evolve in **AntFarm I** using the preferred ANN representation.

## 5.6 Artificial Neural Networks vs. AntFarm I

As we noted above, ANN-based representations look like the most appropriate of the known alternatives for complex, biologically motivated simulations that evolve behaviors. An ANN is a computational abstraction first inspired by the brains and other neural structures of animals, and are characterized by many simple computational *units* ("neurons") with a number of unidirectional *connections* ("axons"). When a unit *fires* or is *activated*, it sends a signal down all of its outgoing connections. The *strength* of a connection is described by a numerical *weight*. The signal that a connection delivers is usually the product of the *activation level* of the unit feeding the connection and the weight of the connection. The computation of a unit can be as simple as calculating the sum of the signals of all of the incoming weights, and determining whether or not the sum exceeds a particular threshold (although it is also common to apply a sigmoid function rather than a simple threshold) to determine the resulting activation level. The units are usually broken into three groups: inputs, outputs, and hidden. The inputs are set by the simulation and the outputs are read as the result of the computation. The hidden units are used for internal computation only.

Although they were inspired by brains, nearly all ANN research to date has used these sorts of highly abstracted networks. We consider the brain metaphor to be quite unfortunate, because when we evolve an ANN-based artificial organism, the reader is bound to make the unwarranted assumption that we are evolving artificial brains. We are *not* evolving brains. We are evolving behavior functions that happen to be represented as a biologically implausible ANN. The decision to use ANNs is based on the fact that the computational model and representation have nice properties, not because they might resemble a real nervous system. We view an ANN as just one of many ways to represent a computer program.

The ANN organism representation that was used in Genesys/Tracker (Jefferson et al., 1991) encodes the network as the concatenation of the binary integer weight

(connection strength) values. The strength of each connection is under genetic control, but not the connectivity pattern itself. The Genesys ANN consists of three fully interconnected layers, with the hidden layer fully connected internally (recurrent, or feedback connections). The connectivity of the network is statically defined, and the weight values are placed in the bit string chromosome in a canonical order.

Unfortunately, our experience has been that this particular representation does not evolve successful foraging behavior in **AntFarm I**. In fact, we have used **AntFarm** extensively as a test–bed for examining a variety of ANN–based organism representations. Based on our previous experience with Tracker, and the apparent ease with which other researchers had been able to find appropriate representations in their artificial life simulations, we believed that this would not be much of a problem. But we were mistaken.

In designing **AntFarm**, we wanted to push the limits of our computational power, and show that we could evolve complex behaviors in complex organisms living in complex environments. Our assumption that an organism representation that works well with simple simulations and that scales (in terms of size and run–time) reasonably will work for larger and more complex simulations was seriously misguided. When the problem arose, we had no theoretical foundation from which to work. We were forced to invent new encodings, implement them, and then empirically test them with **AntFarm**.

This section explores a variety of ANN–based encoding schemes in detail in the context of a somewhat scaled–down version of **AntFarm I**. We begin by introducing a new ANN–based encoding, and then compare it to other ANN architectures.

### 5.6.1 Connection Descriptor ANN Encoding

Our encoding scheme is different from the usual methods in that both the architecture (connectivity pattern) *and* the weights are encoded in the genome, and that the resulting ANNs are *sparse* in terms of connections.

The new ANN encoding is described by $K$ *connection descriptors*. Each connection descriptor consists of three parts: the indices of the units that are to be connected (the *From* unit and the *To* unit), and the weight (strength) of the connection (Figure 5.3). Certain units are designated as inputs and outputs, and the rest are hidden units, which can serve as memory for the organism. The genotype is the concatenation of the bit representation of the $K$ connection descriptors. While the number of units and connections is still fixed, they can be scaled independently and the connectivity pattern is not fixed.

To convert a set of inputs to a set of outputs (behavior), we transmit one signal across each connection in the network. This consists of adding the product of the *From* unit activation and the weight to the *To* unit accumulator. After all $K$ signals have been transmitted, each accumulator is converted to a Boolean value (positive sums to 1; negative or zero sums to 0) and assigned to the corresponding activation. The output unit activations determine the organism's behavior, and the hidden unit activations represent the memory state of the organism.

Genotype

00110111100001100010101100011100001100011001101110101100001100001000100

| From | To | Weight |
|---|---|---|
| 1 (001) | 5 (101) | −2 (1110) |
| 0 (000) | 6 (110) | 2 (0010) |
| 5 (101) | 4 (100) | 7 (0111) |
| 0 (000) | 3 (011) | 1 (0001) |
| 4 (100) | 6 (110) | −2 (1110) |
| 5 (101) | 4 (100) | 3 (0011) |
| 0 (000) | 2 (010) | 4 (0100) |

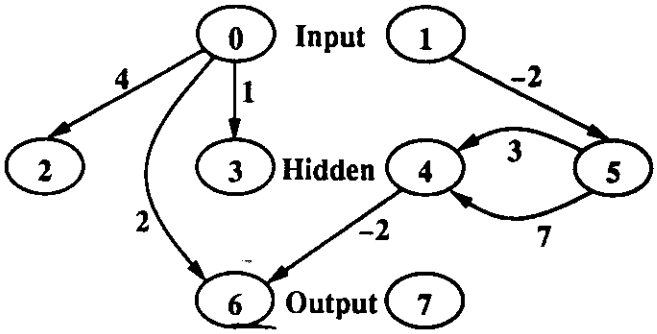


Figure 5.3: The connection descriptors (left), the network (right) and the genotype (bottom) of an ANN encoded with connection descriptors. Each descriptor specifies the pair of units that it connects (*From* and *To* columns), and the strength (*Weight*) of the connection. In this example, the *From* and *To* fields are each 3 bits wide, and the *Weight* field is 4 bits wide. Note that some units have no connections associated with them (e.g. 7), some have no out-going connections (e.g. 2 and 3), some pairs of units are connected by multiple connections (e.g. 4 and 5). Recurrent connections are also allowed.

All possible connection descriptors are legal, including recurrent connections and multiple connections between pairs of units. Connections leading *From* an output unit or *To* an input unit, while legal, have no effect on the output of the ANN.

The use of connection descriptors gives this representation some interesting properties. A mutation can change the value of a particular connection weight, or it can move a connection within the network. A crossover can result in the appearance of a connection that neither parent possesses.

The most important property introduced by this new ANN encoding is the unconstrained and heritable connectivity pattern in the ANN. This freedom is achieved by placing both the location and strength of connections under the control of evolution. Another potentially important property of this representation is the position independence of the connection descriptors, which means that a connection descriptor has the same effect no matter where it lies on the chromosome. This allows linkage patterns between functionally related units to evolve. A feature that is of practical importance is that the number connections has been decoupled from the sizes of the various layers of computational units, permitting us to scale the sizes of the layers of units independently from the number of connections. This allows us to evolve large (in terms of input/hidden/output layers), sparsely connected ANNs encoded in relatively small genomes. It appears to be a feature of **AntFarm** in particular, and perhaps a large class of similar artificial life simulations (Werner, 1991) that successful ANNs do not require dense connectivity patterns. Apparently, in these types of simulations, not all of the hidden or output units are affected by every one of the input units for most successful behavior functions. On these sorts of problems, an ANN with fully connected layers (either with supervised learning techniques such as backpropogation of error, or with evolution) will have many connections with zero weights. In these situations, the ability to easily evolve a sparse connectivity pattern is a very good feature.

The size of the representation is

$$K(2\lceil \log_2 U \rceil + W)$$

where $K$ is the number of connection descriptors, $W$ is the number of bits in each weight, and $U$ is the total number of units in the ANN. This can be refined somewhat by allowing connections only *From* the input and hidden layers and *To* the hidden and output layers. Then, the size of the representation is

$$K(\lceil \log_2(I+H) \rceil + \lceil \log_2(H+O) \rceil + W)$$

where $I$ is the number of input units, $H$ is the number of hidden units, and $O$ is the number of output units. This current implementation is limited in that the total number of units and connections are not under genetic control (i.e. are set by the user and are constant throughout the simulation). We introduce a scheme for variable length chromosomes in Chapter 6, which makes the number of connections heritable.

The development function for the connection descriptor ANN encoding converts the bitstring chromosome into a weight matrix:

```
CD-development(chromosome):
    from-length = 3;
    to-length = 3;
    weight-length = 4;
    matrix[*][*] = 0;
    foreach descriptor in chromosome do
        from = transcribeU(chromosome, from-length);
        to = transcribeU(chromosome, to-length);
        weight = transcribeI(chromosome, weight-length);
        matrix[from][to] += weight;
    endfor
    return matrix;
```

The function *transcribeU(chromosome, B)* returns the next $B$ bits from ***chromosome*** as an unsigned integer. The function *transcribeI(chromosome, B)* returns the next $B$ bits from *chromosome* as a signed integer.

The weight matrix specifies the behavior function representation, and the interpreter for the behavior function runs the ANN specified by the weight matrix:

```
ANN-interpreter(matrix, I, H, O):
    foreach i ∈ I do
        activ[i] = sense(i);
    endfor
    foreach h ∈ H do
        activ[h] = memory(h);
    endfor
    accum[*] = 0;
    foreach f ∈ {I ∪ H} × t ∈ {H ∪ O} do
        accum[t] += activ[f] * matrix[f][t];
    endfor
    foreach h ∈ H do
        if (accum[h] ≥ 0) then memory(h) = 1;
        else memory(h) = 0;
        endif
    endfor
    foreach o ∈ O do
        perform(o, accum[o]);
    endfor
```

The parameters $I$, $H$, $O$ are the number of input, hidden, and output nodes (respectively) in the network. *Activ* is an array of activations, one per node in the ANN. *Accum* is an array of accumulators, one per node in the ANN. The *memory()* function store and retrieves the activation state of the hidden nodes between time steps. The *perform()* function implements the various effector actions, based on the activation of the appropriate accumulator. The actual code for the final version of the connection descriptor encoding can be found in Appendix B.

### 5.6.2 Empirical Comparison of ANNs in AntFarm I

In this section, we empirically compare the performance of both recurrent and feed-forward ANN encodings to that of three foraging benchmarks, a behavior function based on a hand-crafted ANN architecture, and the connection descriptor ANN encoding scheme. The comparisons are performed in a simplified version of the **AntFarm I** simulation.

The simplifications to **AntFarm I** are necessary to ensure the fair evaluation of each colony (so its neighbors do not interfere, etc.). Rather than all colonies foraging in a large, shared environment, each colony is assigned to a separate 16 x 16 environment, which is shared only by the ants within the colony; but the colonies do share an environment for purposes of local mating. In addition, the initial food distribution for each colony in each generation is always the same: one unit of food in each location, except for locations on a straight (horizontal, vertical, or diagonal) line with the nest (Figure 5.4). We chose this food distribution because foraging that only requires walking in a straight line from the nest would involve very few neurons.

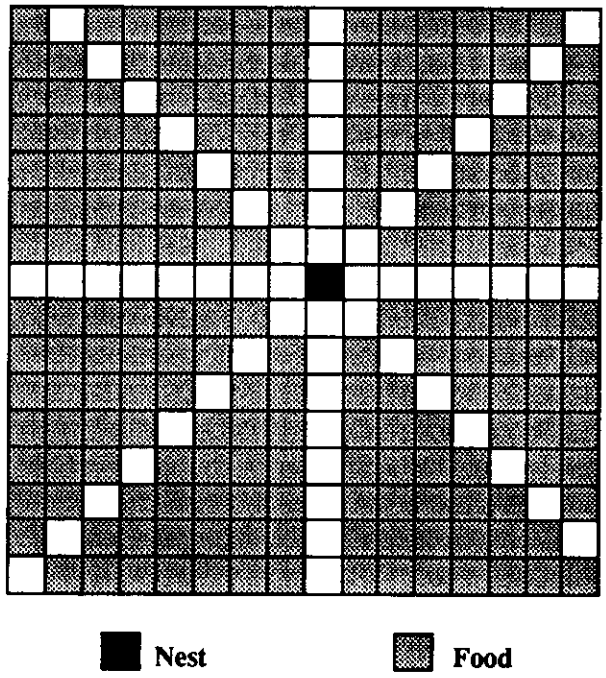


Figure 5.4: The food distribution in the **AntFarm I** ANN experiments. The nest is the black square in the center. One unit of food is placed in each location in the environment, except on the axes and diagonals that are even with the nest.

The population consists of 16,384 colonies, each of which contains four identical ants (initially located in their nest). Each run is 500 generations long, each lasting 50 time steps (i.e. each ant can move 50 times in a generation). The genetic operators are crossover at a rate of 0.0001 per bit and point mutations (bit flips) at a rate of 0.0001 per bit.

How can we tell how well a population is foraging? It is clearly impossible for four ants to find and carry all 196 units of food in the environment to the nest in only 50

time steps, so how much food can we expect them to recover? We have hand-coded three simple behavior functions (in C++) to serve as foraging benchmarks (Table 5.2). BM1 forages using only a random walk. BM2 searches for food with a random walk (ignoring the food sensors) and carries food to the nest by following the compass (the inputs that always point to the nest). BM3 improves on BM2 by using the the food sensors while searching. These benchmarks provide an absolute measure of foraging efficiency.

| Benchmark | Search/Transport | Mean | Max |
|---|---|---|---|
| BM1 | random/random | 1.07 | 6 |
| BM2 | random/compass | 15.07 | 21 |
| BM3 | random+food/compass | 20.82 | 25 |

Table 5.2: Performance of the hand-coded foraging benchmarks. The foraging task is treated as two separate tasks: search for food and transport of the food back to the nest. Random indicates that only the random inputs are used, compass indicates that the inputs pointing the way to the nest are used, and food indicates that the food sensors are used. Mean and Max refer to the amount of food recovered in the population of 16,384 colonies.

We then implemented four different ANN-based behavior functions: a network specified by connection descriptors (**CD**), a Tracker-style three layer recurrent network (**Rec**), a feed-forward network with five hidden layers (**FF**), and an ANN architecture made up of two Tracker-style recurrent networks (**Split**). A successful forager must behave quite differently based on whether or not it is carrying food (in fact, this is the way we constructed the foraging benchmark algorithms), so **Split** invokes one of the networks when the ant is carrying food and the other when it is not. Comparisons of performance between among BM1-BM3 and this dual-network representation will help us determine how successfully the networks have evolved this type of switching behavior.

Each of these behavior functions is encoded in a bitstring of approximately 10,000 bits (Table 5.3). In all four ANN behavior functions, weights are encoded as 3-bit one's-complement integers and the activations of the hidden units are initialized to 0 at the beginning of the generation. The architecture of the **Rec** and **Split** networks is shown in Figure 5.2. **Rec** (Figure 5.5) has a recurrent hidden layer consisting of 32 units (32-bits of memory). **FF** consists of seven layers, each layer fully connected with all forward layers (Figure 5.6). There are five hidden layers, each with eight units. This is a feed-forward network, so it has no memory. The two networks in **Split** each have an 18 unit recurrent hidden layer. Although it has a total of 36 recurrent units, **Split** only has the equivalent of 18 bits of memory because at the beginning of each time step, the activations of the hidden layer of the network that was invoked on the previous time step are copied into the network that will be run this time.

The results are summarized in Figure 5.7. The dual-network architecture (**Split**) out-performs all of the other representations, foraging nearly as effectively as **BM3**, which uses both the food and compass sensors perfectly. It is not surprising that **Split** performs the best, since it has a great deal of information about the foraging task built

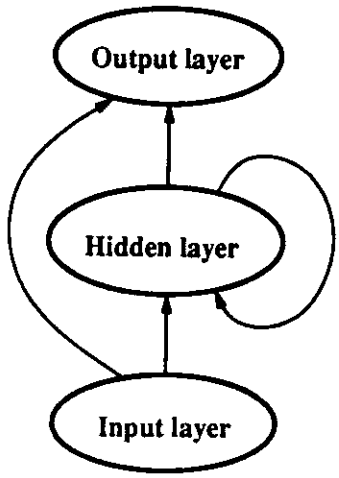


Figure 5.5: The architecture of **Rec**, a recurrent neural network. Each layer is fully connected to all "forward" layers, and the hidden layer is fully connected internally. Each arc in the figure indicates that the layers are fully connected.

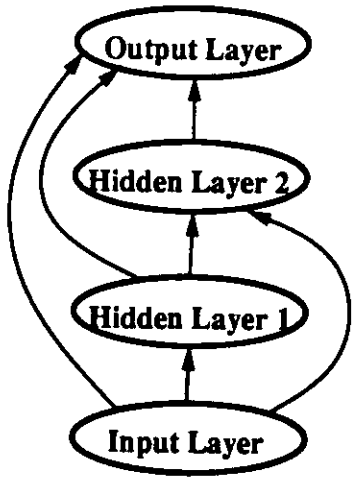


Figure 5.6: The architecture of **FF**, a feed-forward neural network. Each layer is fully connected to all "forward" layers. Each arc in the figure indicates that the layers are fully connected. **FF** has five hidden layers (only 2 are shown here).

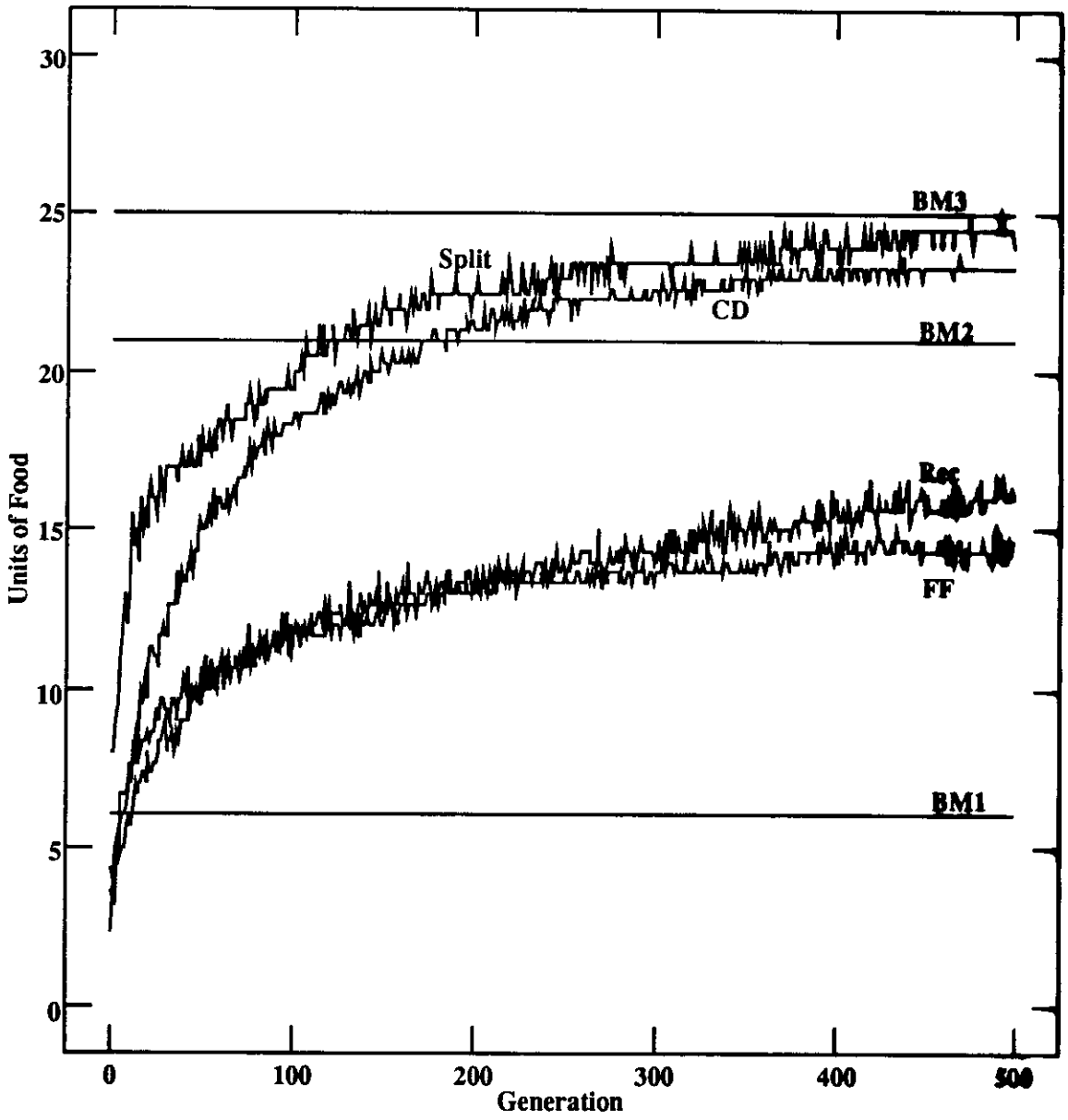


Figure 5.7: The maximum units of food brought back to the nest in a population of 16,384 colonies across 500 generations. Each curve is the average of three runs, differing only in the initial random seed.

| Function | Connections | Hidden Layers | Memory | Chromosome |
|---|---|---|---|---|
| **CD** | 682 | heritable | 21 bits | 10240 bits |
| **Rec** | 2652 | 1x32 nodes | 32 bits | 7956 bits |
| **FF** | 2612 | 5x8 nodes | 0 bits | 7836 bits |
| **Split** | $2 \times 1325 = 2650$ | 2x18 nodes | 18 bits | 7950 bits |

Table 5.3: A summary of the ANN behavior functions, including the number of connections, the arrangement of hidden layers, the number of bits of memory, and the number of bits in the chromosome.

into the representation. The representation based on connection descriptors (**CD**) is nearly as successful, even though it has been provided with no task-specific information. The other two representations with no built-in knowledge perform rather badly—they are not good at changing their behavior based on whether or not they are carrying food, although the recurrent network (**Rec**) does better than the feed forward network (**FF**). Because the connection descriptor representation both does not require problem-specific information and evolves rather successful colonies, we chose it as the basis for the **AntFarm** simulations.

## 5.7 AntFarm I: Evolved Behaviors

**AntFarm I** quite reliably evolves foraging behavior when using the connection descriptor ANN representation. We do not, however, believe that we have evolved cooperative foraging in any of its many runs. The only time we fail to evolve foraging behavior is if the initial environment has no food near the nests. Even the most successful strategy in the initial random population is very inefficient, so the initial environment has to be fairly easy in order to bootstrap the evolution process. During a run, the placement of the food is made more and more sparse in later generations, making foraging more difficult and time-consuming.

The whole connection descriptor ANN consists of 64 neural units and 1706 connections. The connection weights are encoded in 3 bits and the *From* and *To* each in 6 bits, so the network is specified by $(3 + 6 + 6) \times 1706 = 25,590$ bits of genome. We believe that this is at least an order of magnitude larger (in terms of bits in the genome) than any other problem to which genetic algorithms have been applied, and more than two orders of magnitude larger than they are typically applied. When you consider that the size of the search space grows exponentially with the number of bits in the genome, this application is enormously larger than the traditional problems to which genetic algorithms have been applied. We were relieved to find that our (spatially structured) genetic algorithm was able to operate in this enormous search space.

The typical foraging algorithms that evolve have the whole colony searching for food in one particular direction, for instance to the north. Often all the ants in the population will walk north while not carrying food, and to the south (back towards the nest) after they have found food. This is a reasonably successful strategy, and

involves very few of the sensory inputs or motor outputs. This simple strategy could be implemented with only a handful of connections in the ANN. The direction of searching that evolves varies from run to run, often even on the diagonals (e.g. south-west).

More sophisticated search strategies that evolve move the ants slightly off of the straight lines. Wandering slightly allows the ants to cover more area near the nest, decreasing transport time. The straight–line strategies exhaust all of the food that lies on the lines in the whole environment if enough time steps are allowed, so off-the–lines strategies not only find food that is closer to the nest, but discover more total food. The off–the–line strategies require at least partially correct interpretation of the compass inputs in order to reliably return to the nest. A few runs have yielded strategies that search for food in two different directions, such as both to the north and to the west, and off–the–line in both directions.

We have never observed in **AntFarm I** the evolution of ants that make effective use of all of their food sensors. It would be advantageous for an ant that senses food in the location to the south to step to the south and pick it up. While we occasionally observe ants that will consistently make use of one of the eight food sensors associated with neighboring locations, they never seem to use even a majority of them correctly.

We have never observed the evolution of a strategy that involves marking piles of food with pheromones or laying or following pheromone trails leading to piles of food. In a majority of runs, the population evolves to not use pheromones at all, except for the occasional mutant colony. In a few runs, even the most successful colonies spew large volumes of pheromone all over the area surrounding the nest. The coverage is complete, and it is unlikely that any information is being communicated to the nestmates, but it is difficult to know for sure. If it is not helping, why is it that a mutant that does not emit large quantities of pheromone take over the population? This leads to important questions, such as "What is a cooperative strategy?" and "How do you know when you have cooperation?"

Before implementing **AntFarm I**, our research group had a few heated discussions about cooperation and testing for cooperation. We decided that the type of cooperation of interest in **AntFarm** is behavior that conveys information to the other members of the colony such that more food is brought into the nest. A simple test for this sort of cooperation is to run the evolved foraging strategy with an increasing number of ants in the colony. As the colony size increases there should at some point be a per capita increase in food retrieval if cooperation is at work. This procedure should test for the communication of information about the location of food, because cooperation should lead to a decrease in search time and thus increased food retrieval.

We have applied this test for cooperation to the pheromone–spewing colonies, and it indicates a significant amount of cooperation. In addition, if we turn off the ant's pheromone sensors, the ants perform dismally. This result indicates that the cooperation is indeed mediated by the pheromones. Unfortunately, the test for cooperation is apparently somewhat flawed.

While the pheromone–spewing behavior falls within the letter of our cooperation definition, it violates the spirit of the definition. It appears that the pheromones are not actually conveying any information about food, but rather the ANN has a

quirk such that it only forages effectively in the presence of pheromones (when its pheromone sensors are activated). When the colony consists of only a couple of ants, they cannot build up enough pheromone to forage effectively; they have to produce all of their own pheromone, and they are unable to produce enough to cover the whole area near the nest with pheromones. In larger colonies, the pheromones are shared, so all members of the colony can forage effectively. In a sense, this is cooperation: together they can create an environment that is ripe for foraging, but separately they cannot.

It is interesting that this sort of situation occurred in several different runs. In the early generations of **AntFarm I**, many of the random initial colonies liberally drop pheromones everywhere they walk. While the pheromones are not free, they are cheap enough that the do not play a major role in relative fitness until later generations. Apparently the ancestors of the champion pheromone–spewing colonies evolved foraging innovations that happened to rely on the activation of their pheromone sensors. The foraging strategy, even with the cost of large–scale pheromone production, is more efficient than the others that evolved, and soon the whole population is foraging effectively and polluting the environment with large volumes of pheromone. Apparently the reliance on the pheromone inputs to forage effectively is so complete that there is no likely evolutionary path to a pheromone–free foraging strategy. We let some of these runs continue for 5000 generations, and none of them managed to evolve away from this silly reliance on pheromones. This is not the sort of cooperation that is observed in nature, nor did we expect this to evolve.

In nature, foraging ants appear to leave the nest in random directions and perform a somewhat random walk searching for food. When food is discovered, they carry it directly to the nest (perhaps leaving a pheromone trail between the food and the nest to aid in recruitment). In our **AntFarm I** simulations, all the ants leave the nest in the same direction (or occasionally two directions) and walk in a straight (or nearly straight) line until food is encountered, and then carry the food directly back to the nest. Why are the **AntFarm I** behaviors so unnatural?

A primary cause appears to be the fact that the ants lack a *relative* orientation: all sensors and effectors are represented in terms of the absolute global coordinate system and directions. For instance, the food sensor that senses on the north side of the ant *always* senses on the north side of the ant, regardless of how the ant moves. If the behavior function of the colony is wired such that most movement is towards the west, all of the ants will move towards the west. If the ants had a variable orientation (and had the ability to turn), then they could come out of the nest facing random directions and forage in all directions around the nest. This would allow the sensors, effectors, and behavior to no longer be tied to the coordinate system of the environment. The lack of orientation was a design decision aimed at reducing computation time, allowing the use of the Connection Machine-2's fast, local communication to gather together the sensory inputs for the whole 3 x 3 sensor array.

Another contributing factor is probably the enormous number of low–level sensors. To make use of all the sensory information requires a very sophisticated behavior function. An additional difficulty is that each innovation must to be discovered by evolution several times through separate mutations or recombinations. For example,

to follow the compass correctly, each of the compass' four inputs has to be connected to the proper motor output, and circuitry must be in place to suppress the action of the compass while the ant is searching for food. Although the compass sensors and movement effectors are symmetric in all four directions, the behavior function is not necessarily symmetric, and the encoding has no means of replicating a successful structure: evolution must discover the proper circuitry four separate times in order to fully utilize the compass. The same argument holds for the eight food and eight pheromone sensors that provide information from neighboring locations. We address many of these shortcomings in the next chapter.

# Chapter 6

# AntFarm: The Evolution of Cooperative Foraging II

In this chapter we continue to explore the representation of artificial organisms, and the effect of the artificial morphology on the behaviors that evolve. We continue within the **AntFarm** framework, and we modify the ANN encoding and the morphology of the ants in search of the evolution of ant-like behaviors. We do not succeed in this goal until **AntFarm IV**.

## 6.1 AntFarm II

As we explained in Chapter 5, **AntFarm I** did not evolve ant-like foraging behavior (in fact, we had to design a radically new ANN-based encoding to evolve *any* kind of foraging behavior). In order to rectify some of the deficiencies of **AntFarm I**, we have implemented a second version, called **AntFarm II**. The primary goal of the **AntFarm II** design is to evolve foraging behavior that is more comparable to natural ant behavior than produced in **AntFarm I**. Secondary design goals are to increase the speed of the simulation and increase both the number of colonies and the size of the environment. **AntFarm II** differs from **AntFarm I** primarily in the morphology of the ants and the implementation of the environment.

### 6.1.1 Comparison to AntFarm I

To achieve the goals of ant-like behavior, speed, and scalability (in terms of population and environment size), we have simplified both the environment and the sensory/motor morphology of the ants. Table 6.1 compares **AntFarm II** to **AntFarm I** for typical parameter settings. In **AntFarm II**, the 3 x 3 sensor array has been replaced by sensors at the head and ends of two antennae, and the ants turn and move in a real-valued coordinate system, rather than on the grid. These simplifications result in a much smaller genome.

| Dimension | **AntFarm II** | **AntFarm I** |
|---|---|---|
| Colonies | 32,768 | 16,384 |
| Ants/Colony | 128 | 16 |
| Environment Size | 134,217,728 | 4,194,304 |
| Bits/Location | 2 bits | 32 bits |
| Morphology | ant-like | 3 x 3 array |
| Position | real-valued | grid-valued |
| Orientation | real-valued | constant |
| Sensory Input | 30 bits | $\sim$ 200 bits |
| Memory | 8 bits | 21 bits |
| Effector Output | 20 bits | 13 bits |
| ANN Encoding | Connection Descriptor | Connection Descriptor |
| Behavior that evolves | asymmetric circles | asymmetric lines |
| Genome Size | $\sim$ 994 bits | 25,590 bits |

Table 6.1: A comparison of **AntFarm II** to **AntFarm I**. The **AntFarm II** genome, behavior function, and environment data are all smaller and more compact, allowing simulations that are larger and more complex in the other dimensions.

### The Environment

The **AntFarm II** environment maintains only two bits of information per location: the presence or absence of food and pheromone. In contrast, **AntFarm I** maintained the actual number of units of food or pheromone that were present in each location. The location of the nests are no longer explicitly stored in the environment (they were in **AntFarm I**); sensing for the presence/absence of nests is implemented by comparing the current location to the known locations of the nests. In **AntFarm I**, pheromone diffusion was implemented directly, while in **AntFarm II**, pheromone diffusion is approximated by randomly spreading existing areas of pheromone to neighboring locations and randomly deleting pheromone. We approximate finite piles of food as follows: Every time an ant picks up a piece of food from a pile, there is a certain probability that the pile will be removed from the environment, which simulates the case where the ant just grabbed the last piece of food. A high probability of deleting the food pile simulates a small pile of food, while a low probability simulates a large pile of food.

These simplifications (plus a little ingenuity) allow us to simulate worlds with more than 134 million locations even on a 16K processor, 128Mb (small memory model) Connection Machine-2. To handle this large environment data structure, we were forced to abandon the usual method of mapping one environment location onto each virtual processor (data parallel mapping). While the data parallel mapping is simple and very convenient, the overhead that is incurred by a high ratio of virtual to physical processors sometimes can be significant. On the Connection Machine-2, the primary overhead is memory, and to a lesser extent communication time. Each virtual

processor must maintain the state of a physical processor. The physical processors are very simple 1-bit processing elements and maintain only 4 bits of state. While this state information is very small, it adds up, especially on a small memory model machine, which is equipped with only 64K bits per physical processor.

In **AntFarm II** (and later versions of **AntFarm**), we simplify the environment to store only two bits of information per grid square (the presence or absence of food and pheromone). We desired to scale up the size of the world, but quickly ran out of memory. The problem was that we were storing 2 bits of data per virtual processor, but paying twice that in overhead. Not only was memory a limiting factor, but the interprocessor communication is rather slow at high virtual processor (VP) ratios.

Our solution is to run the environment at a VP–ratio of 1, with each environment processor storing the data for a large portion of the grid in a large, packed array. On the Connection Machine-2, constant–time random–access arrays must be stored in a special format (transposed across 32 1-bit processors). There are interprocessor communication primitives that will read and write 32-bit elements, with a limited set of combiner modes (e.g. add, logical or, etc.). Mapping $(x, y)$ world coordinates to this array–based storage scheme requires a simple address translation mechanism. In addition, since only 32-bit elements can be accessed, the correct bit must then be masked to determine the state of the environment. This provides a relatively fast and convenient method for simulating an enormous environment with little overhead.

Unfortunately, this storage scheme suffers some complications. First, while it is easy to drop food or pheromone on a location, it is difficult to remove something from the location. To drop pheromone in the environment, a mask with the appropriate bit set is constructed, and is sent to the environment using the *bitwise logical or* combiner, which will ensure that that bit in the environment gets set (turned on). The logical–or combiner allows the unconditional setting of bits, but there is no combiner for unconditionally clearing bits (the only combiners are bit–wise logical or, unsigned integer add, and overwrite). The obvious solution is to perform a bitwise complement of the environment, *set* the bits we want to be cleared using the logical–or combiner, and then bitwise complement the environment once again. While two bitwise complements of a large portion of the Connection Machine's memory probably sounds expensive, it is not too bad. In fact, the bit–clearing algorithm takes only about twice as long as the bit–setting algorithm; the interprocessor communication time tends to dominate Connection Machine computations.

### Ant Morphology

**AntFarm II** incorporates a major morphological change: the ants are bilaterally symmetric, rather than radially symmetric. The other major change is that we removed the ants from the grid by specifying the ant's location by a real–valued $(x, y)$ coordinate (rather than an integer–valued grid location), and its orientation by a real–valued angle $\theta$ measured relative to the positive $x$–axis (Figure 6.1). The four movement effectors of **AntFarm I** are replaced with effectors for moving forward and turning (both specifying real–valued amounts). In addition, the 3 x 3 sensor array of **AntFarm I** is replaced by three sensory locations, located at the head and the ends

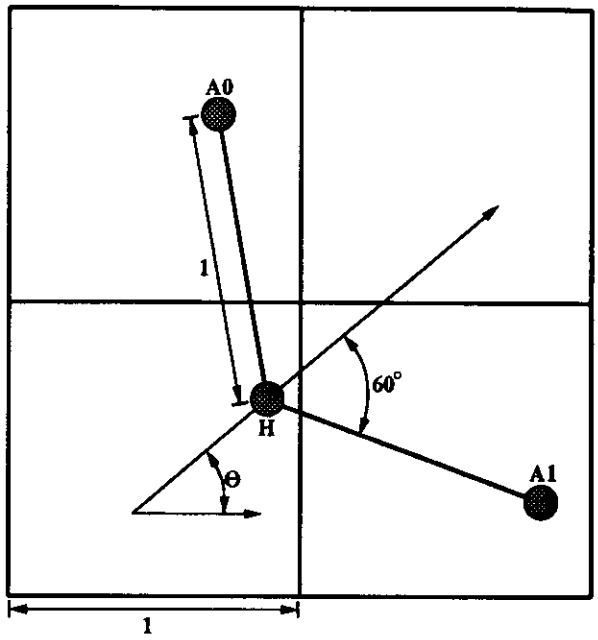


Figure 6.1: The **AntFarm II** ant design. The three sensors (shaded circles) are located at the head (labeled **H**), and at the ends of the two antenna (labeled **A0** and **A1**). The position of the ant is defined by the location of the head $(x_H, y_H)$, and the orientation by the angle $\theta$ from the $x$-axis. The antenna sensor positions $(x_{A0}, y_{A0})$ and $(x_{A1}, y_{A1})$ are defined relative to $(x_H, y_H)$: one grid unit from the head and 60 degrees to the left and right of ant's orientation.

of two "antennae." The relative positions of these sensory locations are shown in Figure 6.1. For each of the sensors, we determine which grid square it is in, and three bits of information concerning the presence or absence of nest, food, and pheromone in that grid location are conveyed to the behavior function. Not only have we reduced the number of sensors, but we have also reduce the number of bits of input per sensor. An obvious consequence of the environmental simplifications is the fact that there is only one bit of pheromone data to be sensed (compared to 12 bits in **AntFarm I**).

In **AntFarm I**, the simulated morphology bore little resemblance to real ants, which we believe contributed to the evolution of behavior that is not ant-like. By making the simulated morphology more ant-like, we expected that **AntFarm II** would evolve more realistic behavior. In addition, by drastically reducing the number of sensors and effectors and using a behavior function that is more compact, the genomes in **AntFarm II** are more than an order of magnitude smaller than those used in **AntFarm I**. The smaller genome and behavior function means that we can simulate larger populations in larger environments, and either many more total ants or many more generations (or some combination of both).

Here are the details of the **AntFarm II** connection descriptor ANN behavior function. These are features that are available, but particular organisms may "use" (have connected) many fewer:

- Input Units

- 3 binary units for presence of pheromone
- 3 binary units for presence of food
- 3 binary units for presence of a nest
- 2 units for compass (direction/amount to turn to point towards nest)
- 4 binary units for random noise
- 2 binary units for whether or not it is carrying food
- 1 constant unit (always on)

- Hidden Units
  - 8 binary units

- Output Units
  - 2 units for the magnitude and direction of turning
  - 1 unit for the amount to move forward
  - 1 binary unit to pick up food
  - 1 binary unit to drop food
  - 1 binary unit to drop pheromone

The whole neural network consists of 32 neural units and a maximum of 142 connection descriptors. The connection weights are encoded in 4 bits and the *From* and *To* each in 5 bits, so the network is specified by a maximum of 1988 bits of genome.

### 6.1.2 Variable Length Genomes

One additional change from **AntFarm I** to **AntFarm II** is the use of a variable-length chromosome, with a maximum length of 2000 bits. We define a length-modifying mutation operator that is applied with probability $\mu_l$ after the recombination and point mutation genetic operators. The length mutation either adds or deletes between 0 and 31 bits to/from the trailing end of the chromosome. The change in length is a one's complement random integer from the uniform distribution $[-31, 31]$ (note that the one's complement representation contains both $\pm 0$). There is no mutation pressure for either longer or shorter chromosomes. Whenever a chromosome is lengthened, the new bits are given random values. We usually begin a run with all of the organisms having a chromosome length of one fourth of the maximum chromosome length. In **AntFarm II**, this makes the initial chromosome lengths 500 bits, encoding 35 connection descriptors.

We have extended the recombination operator to handle variable length chromosomes. As before, when the end of the chromosome is encountered while copying bits from one of the parent chromosomes, copying is terminated. In addition, if a crossover is attempted to a point past the end of the other parent chromosome, copying is terminated. This means that the length of the offspring chromosome after

recombination will always be between the lengths of the parent chromosomes. The offspring chromosome will only be as long as the longer of the two parents if the longer parent was being copied at the point where the shorter parent ends, and no crossovers are generated before the end of the chromosome is reached. The actual code for handling variable length chromosomes can be found in Appendix A.

The development function must also be upgraded to handle transcription of variable length chromosomes. If a partial connection descriptor is found at the end of the chromosome, it is ignored. In **AntFarm II**, the chromosome is a maximum of 2000 bits long, but a maximum of 1988 bits are transcribed, because the last 12 bits make up only a partial connection descriptor.

While these semantics for variable length chromosomes are not very biologically realistic, they are a first step in that direction. We want to begin exploring variable length encodings, because our experience has been that it is difficult or impossible to evolve complex behaviors directly from large, random behavior functions. If we wish to evolve complex behaviors, we are more likely to succeed if the size and complexity of the behavior function itself can be subject to evolutionary pressures.

### 6.1.3 Evolved Behaviors

While we thought that removing the ants from the grid and making them morphologically bilaterally symmetric would suffice to allow ant–like foraging behavior to evolve, this was not the case. The redesigned **AntFarm II** ants certainly behave differently from those evolved by **AntFarm I**, they still do not act like ants.

In most of our **AntFarm II** experiments, the food piles are uniformly distributed throughout the environment, with those within a radius $r$ of a nest being removed before the ants begin foraging. In initial generations, the probability of placing a pile of food in a grid location is $P(food) = 0.5$ with a food–free zone of radius $r = 3$ around each nest. We must make the environment for the initial generation easy enough that at least one of the random colonies is able to forage with some success. During a run, the environment is incrementally made more difficult in later generations, typically to $P(food) = 0.01$ and $r = 20$.

In **AntFarm II**, we invariably observe the evolution of circular foraging patterns. In a given run, the whole population will walk in nearly perfect circles either to the right or to the left as in Figure 6.2. Consider a right circling colony. At each time–step, each ant moves forward a constant distance $f$, and turns to the right at a constant angle $t$. The values of $f$ and $t$ are such that an approximate circle is formed, bringing the ant back to the nest. The colonies vary in their particular values of $f$ and $t$, but the ants within the colony do not vary: each colony has a characteristic circle-size that is hard–coded into each of its ants. Evolution tunes the diameter of the circle to be slightly larger than the radius of the food–free zone $r$, and tracks changes in $r$ through the run. Because the ants within a colony begin with random initial orientations, they all follow different paths, but form circles of the same size. The circle brings every ant in the colony back to the nest at the same time, whether or not the ant has found food. Apparently, the ants are relying on forming a perfect circle to return them to the nest, despite the fact that we provide them with a compass.

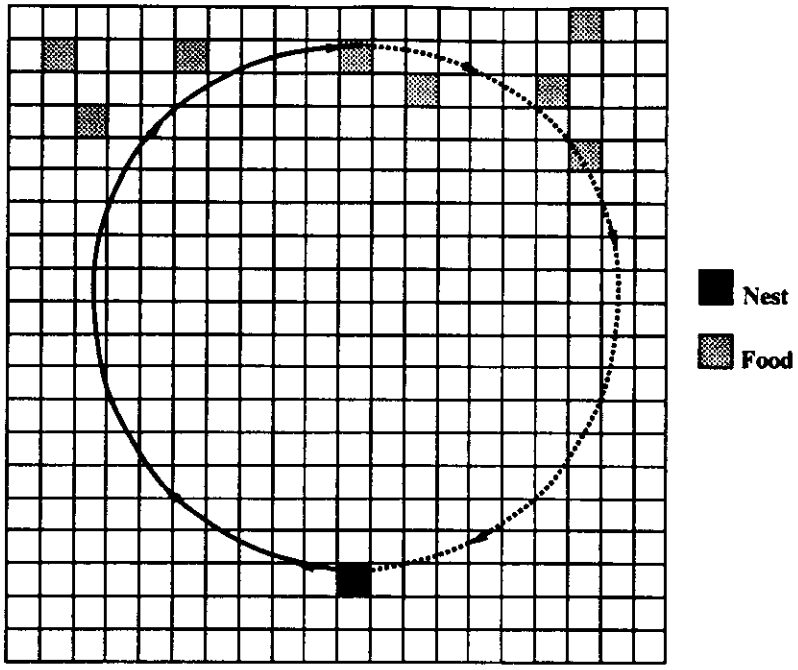


Figure 6.2: The path of an ant using the circular foraging strategy. By moving and turning by constant amounts on each time step, a nearly perfect circle is formed, reliably returning the ant to the nest. If food is encountered, a piece is picked up. The solid line indicates the ant is not carrying food, and the dotted line that it is carrying food.

The simple behavior of walking in circles requires very few connections in the ANN behavior function. Moving forward is controlled by one connection from the **constant** input to the **move** output. Turning is specified by one connection from the **constant** input to the **turn right** output. The strengths of these connections control the values of $f$ and $t$, and thus the size of the circle. Two other connections are required to control dropping and grabbing food. Dropping food is accomplished by a connection from the **nest (head)** input to the **drop food** output. Picking up food can be implemented in several different ways with connections to the **grab food** output, including a connection from **constant**, **food (head)**, **not carrying food**, **compass (turn right)**, etc. All of these inputs are on while the ant is walking in a circle and not carrying food. An ANN that could specify circling behavior is shown in Figure 6.3.

While foraging in circles, the food sensors are not used, and apparently cannot be used. If an ant were to turn off of the circular path to pick up a piece of food that it sensed, it would be unlikely to compensate for the deviation, and thus would not be able to transport the food to the nest. The reliance on a circular path to return the ant to the nest appears to put severe constraints on the possible behavioral improvements.

We had hoped to see the evolution of simple ant-like foraging: pseudo-random walk while searching for food and use of the compass to transport food to the nest. We were curious why the circular foraging behavior always evolved, and never anything more ant-like. We explored two hypotheses.

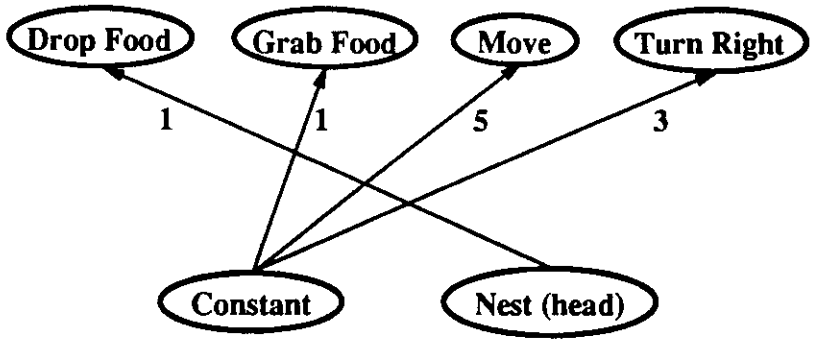


Figure 6.3: An ANN behavior function of the type evolved by **AntFarm II**. The unused connections and units have been omitted.

- More ant–like behaviors require significantly more complex behavior functions, and there is no likely evolutionary path to them from circular foraging. In other words the circular foraging behavior is a local optimum (a peak) in the adaptive landscape, with a very large basin of attraction.

- The circular foraging algorithm is more efficient than the more ant–like behaviors that we had hoped to see evolve.

In order to test these hypotheses, we hand–coded a chromosome that implements simple ant–like foraging. The behavior function moves the ant directly forward when it is not carrying food. If random turns lead to a more efficient search, then we would expect to see them evolve, since this would presumably be a relatively minor modification. When the ant is carrying food, it uses the compass inputs to orient itself towards the nest. The ant grabs/drops food as appropriate.

To see how our hand–coded behavior function would fare in an evolutionary setting and against the circling behaviors, we seeded the initial random population with one colony containing the hand–coded behavior function. In every one of these runs, we end up with a hybrid strategy. The ants still walk in nearly perfect circles when they are not carrying food (and return to the neighborhood of the nest even when food is not discovered), but use the compass to walk directly back to the nest when food is found (Figure 6.4). It is not clear if the linear search of the hand–coded strategy is replaced by the circular search via mutation or via recombination with descendants of the initially random portions of the population which evolve the circle foraging strategy.

Why does the hybrid strategy always evolve in these experiments? While transporting food, the compass–following behavior is clearly more efficient than completing the circle back to the nest. By returning directly to the nest after food is encountered, the foraging efficiency is increased, and each ant can potentially transport more pieces of food to the nest. Searching for food with a circular path apparently is more efficient than searching in a straight line away from the nest. When the piles of food are relatively rare, the linear search may not discover food until the ant is far from the nest, requiring a long time to transport the food back to the nest. On the other hand, the circular search always keeps the ant near the nest, so transport costs once

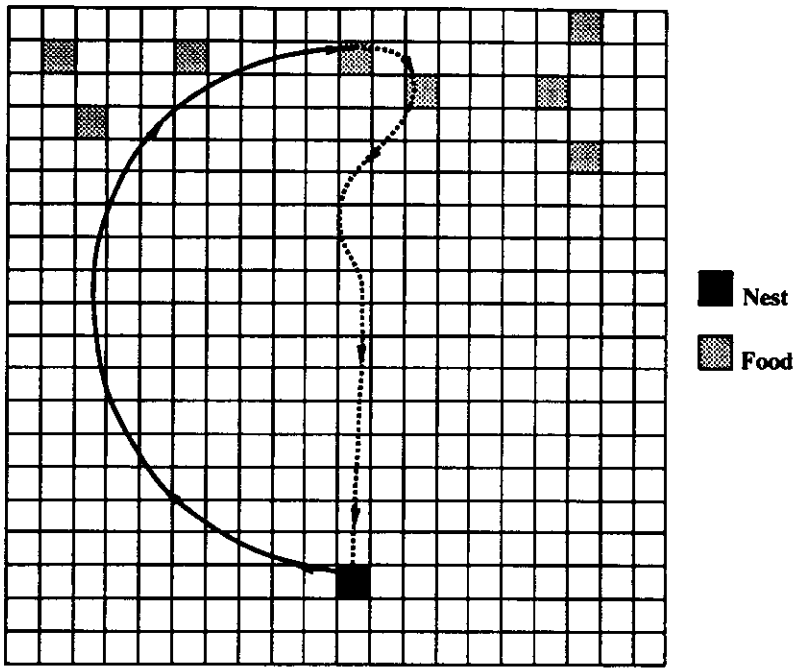


Figure 6.4: The path of an ant using the circular search strategy, but returning directly to the nest when food is discovered. The solid line indicates the ant is not carrying food, and the dotted line that it is carrying food.

food is discovered are kept to a minimum. Apparently, the circular search spends enough time in the area where food is likely to be discovered that it is more efficient than the linear search.

Why use a circular search rather than a random walk? A random walk would actually require fairly complex circuitry (see Section 6.4) in order to operate while searching, but not mess up the following of the compass during food transport. It is not clear whether we do not observe random turns because the circuitry is simply unlikely to evolve, or if it simply is less efficient than a circular search. It is possible that it has evolved many times in our simulations, but it was less efficient than circular search, and thus was selected out of the population.

One reason why the **AntFarm II** behavior is not very ant-like is it's strong asymmetry. All of the ants in a given colony (and usually throughout the population too) form circles by turning in the same direction. No real ant foraging strategy would ever evolve that used only right turns and never left turns (or vice versa). To produce symmetric behaviors, evolution must discover and maintain the same circuitry for both sides of the ANN.

It is interesting to note that we did not observe strong evolutionary pressure for either longer or shorter genomes. In all of our runs, the length of the genome remains relatively constant for thousands of generations. On a couple of occasions, the mean chromosome length increased by 14 bits (the length of one connection descriptor) from 500 to 514 bits. This indicates that a beneficial connection was discovered in this new locus, and it spread through the population.

## 6.2 AntFarm III

In nature, nearly all simple organisms (such as ants) are largely symmetric, both in terms of musculature and neural structure, and thus behavior also. In real ants, symmetric behavior is a result of similar developmental processes occurring on both sides of the ant, resulting in symmetric musculature and neural circuitry. Although the simulated musculature of **AntFarm II** has the bilateral symmetry of real ants, the development of the connection descriptor–based ANN representation is not symmetric, which means that the behavior that evolves is not symmetric.

While we are making a comparison between actual neural structures and our ANN–based behavior functions, we must again stress that we are not evolving artificial "brains." The point is that natural behavior functions ("brains") are usually computationally symmetric, while the **AntFarm I** and **AntFarm II** behavior functions are not. Due to this lack of computational symmetry, the behaviors specified by the evolved behavior functions are not symmetric.

Without a symmetric behavior function, artificial evolution will have to find the proper set of genes to deal with the sensors and effectors twice, once for each side of the organism (assuming bilateral morphological symmetry). We have modified the connection descriptor ANN encoding to result in symmetric networks. Basically, each connection descriptor results in two connections being added to the network. In **AntFarm II**, a descriptor might specify a connection from the **compass (turn left)** input to the **turn left** unit. In **AntFarm III** (using the symmetric connection descriptor encoding), the descriptor would specify a connection from the **compass (turn toward this side)** input to the **turn towards this side** output, which "develops" into two connections: from **compass (turn left)** to **turn left** and from **compass (turn right)** to **turn right**. Not only does this result in symmetric behaviors, but it also compresses and simplifies the genome. Here is the new development function:

```
CD-symmetric-development(chromosome):
    from-length = 3;
    to-length = 3;
    weight-length = 4;
    matrix[*][*] = 0;
    foreach descriptor in chromosome do
        from = transcribeU(chromosome, from-length);
        to = transcribeU(chromosome, to-length);
        weight = transcribeI(chromosome, weight-length);
        matrix[left(from)][left(to)] += weight;
        matrix[right(from)][right(to)] += weight;
    endfor
    return matrix;
```

The functions *left()* and *right()* translate the *From* and *To* portions of the connection descriptor into the descriptor's two symmetric interpretations. The representation of the behavior function and the details of the interpreter remain unchanged.

| Dimension | **AntFarm III** | **AntFarm II** | **AntFarm I** |
|---|---|---|---|
| Colonies | 32,768 | 32,768 | 16,384 |
| Ants/Colony | 128 | 128 | 16 |
| Environment Size | 134,217,728 | 134,217,728 | 4,194,304 |
| Bits/Location | 2 bits | 2 bits | 32 bits |
| Morphology | ant-like | ant-like | 3 x 3 array |
| Position | real-valued | real-valued | grid-valued |
| Orientation | real-valued | real-valued | constant |
| Sensory Input | 30 bits | 30 bits | $\sim$ 200 bits |
| Memory | 8 bits | 8 bits | 21 bits |
| Effector Output | 20 bits | 20 bits | 13 bits |
| ANN Encoding | Symmetric Connection Descriptor | Connection Descriptor | Connection Descriptor |
| Behavior that evolves | symmetric circles | asymmetric circles | asymmetric lines |
| Genome Size | $\sim$ 487 bits | $\sim$ 994 bits | 25,590 bits |

Table 6.2: A comparison of **AntFarm I** through **AntFarm III**.

Table 6.2 compares **AntFarm I** through **AntFarm III**. The number and type of input, hidden, and output units in the **AntFarm III** ANN are exactly as described above for **AntFarm II**. With this new encoding scheme the ANN consists of a maximum of 152 symmetric connection descriptors (coding for up to 304 connections). The connection weights are encoded in 4 bits, *From* in 4 bits, and *To* in 5 bits, so the network is specified by a maximum of 1996 bits of genome.

The behaviors that are evolved by **AntFarm III** are identical to those evolved by **AntFarm II**, except that each ant is capable of making circles in *either* direction (sometimes a constant turn to the left at each time step, and at other times a constant right turn at each time step). Apparently the direction of the turn is specified by the compass input units. Once an ant begins turning to one side, the compass unit on that side will remain on until the circle is complete. At any given time, approximately half of the ants are in the process of making a circle to the left, and the other half a circle to the right. A particular ant often will walk in *both* right-turning and left-turning circles during its lifetime.

It is likely that our failure to evolve ant-like behavior in neither **AntFarm II** nor **AntFarm III** is due to the difficulty of switching between search and transport behaviors during foraging. This switch is difficult to evolve, because it should be triggered by a small change in the sensory inputs: the single bit indicating whether or not the ant is carrying food. We only observed behaviors that switched between search mode and transport mode in the runs where we seeded the population with such a dual-mode behavior.

We characterize behaviors as *continuous* if small changes in the input pattern result in small changes in behavior, and as *discrete* or *conditional* if a small change in

the sensors can result in a large change in behavior. Both the search and nest-finding behaviors can probably be expressed as continuous behaviors, but a discrete change based on whether the ant is carrying food is required to select between them. We empirically demonstrated in Chapter 5 that the connection descriptor ANN encoding is more able to evolve this sort of discrete behavior change than traditional ANN encoding schemes in **AntFarm I**, yet we have never observed its spontaneous evolution in **AntFarm II** or **AntFarm III**.

## 6.3 AntFarm IV

We address the problem of evolving conditional behaviors in a fourth implementation of **AntFarm**. **AntFarm IV** is nearly identical to **AntFarm III**, with changes only in the behavior function interpreter. The primary change in the ANN is in how we define the semantics of a connection from an input unit to another input unit. Previously, such a connection was defined to be a no-op.

Now, the input to input connections are run before the rest of the network. any input unit that receives an overall negative activation from the other input nodes is turned off before the rest of the network is run. This provides an easy (few connections required) way for the network to ignore certain inputs while it computes the behaviors for a given time step. Here is the modified interpreter algorithm (see Section 5.6.1):

*ANN-input-inhibition-interpreter(matrix, I, H, O):*

```
foreach i ∈ I do
    activ[i] = sense(i);
endfor
foreach h ∈ H do
    activ[h] = memory(h);
endfor
accum[*] = 0;
foreach f ∈ I × t ∈ I do
    accum[t] + = activ[f] * matrix[f][t];
endfor
foreach i ∈ I do
    if (accum[i] < 0) then
        activ[i] = 0;
    endif
endfor
foreach f ∈ {I ∪ H} × t ∈ {H ∪ O} do
    accum[t] + = activ[f] * matrix[f][t];
endfor
foreach h ∈ H do
    if (accum[h] ≥ 0) then memory(h) = 1;
    else memory(h) = 0;
    endif
```

| Dimension | **AntFarm IV** | **AntFarm III** | **AntFarm II** | **AntFarm I** |
|---|---|---|---|---|
| Colonies | 32,768 | 32,768 | 32,768 | 16,384 |
| Ants/Colony | 128 | 128 | 128 | 16 |
| Environment Size | 134,217,728 | 134,217,728 | 134,217,728 | 4,194,304 |
| Bits/Location | 2 bits | 2 bits | 2 bits | 32 bits |
| Morphology | ant–like | ant–like | ant–like | 3 x 3 array |
| Position | real–valued | real–valued | real–valued | grid–valued |
| Orientation | real–valued | real–valued | real–valued | constant |
| Sensory Input | 30 bits | 30 bits | 30 bits | $\sim$ 200 bits |
| Memory | 0 bits | 8 bits | 8 bits | 21 bits |
| Effector Output | 20 bits | 20 bits | 20 bits | 13 bits |
| ANN Encoding | Symmetric Connection Descriptor, Input Inhib. | Connection Descriptor Symmetry | Connection Descriptor | Connection Descriptor |
| Behavior that evolves | ant–like | symmetric circles | asymmetric circles | asymmetric lines |
| Genome Size | $\sim$ 487 bits | $\sim$ 487 bits | $\sim$ 994 bits | 25,590 bits |

Table 6.3: A comparison of **AntFarm I** through **AntFarm IV**.

```
endfor
foreach o ∈ O do
    perform(o, accum[o]);
endfor
```

The actual code for this final version of the connection descriptor encoding can be found in Appendix B.

Also, the **AntFarm IV** behavior function does not have any memory; it simply reacts to the current state of the environment. In the **AntFarm I–AntFarm III** simulations described above, when the hidden units were used at all, they were never used for memory, only for computations that produce conditional behavior (see the discussion in Section 6.4). We have not observed any recurrent connections that appeared to have any useful purpose. In addition, the **constant** input unit has been removed. The four **AntFarm** models are summarized in Table 6.3.

Here are the details of the **AntFarm IV** ANN behavior function. These are features that are available, but particular organisms may "use" (have connected) many fewer:

- Input Units
  - 3 binary units for presence of pheromone
  - 3 binary units for presence of food
  - 3 binary units for presence of a nest
  - 2 units for compass (direction/amount to turn to point towards nest)

- 4 binary units for random noise
- 2 binary units for whether or not it is carrying food

- Hidden Units
  - none

- Output Units
  - 2 units for the magnitude and direction of turning
  - 1 unit for the amount to move forward
  - 1 binary unit to pick up food
  - 1 binary unit to drop food
  - 1 binary unit to drop pheromone

The whole neural network consists of 23 neural units and a maximum of 142 symmetric connection descriptors (coding for up to 284 connections). The connection weights are encoded in 5 bits, *From* in 4 bits, and *To* in 5 bits, so the network is specified by a maximum of 1988 bits of genome.

In **AntFarm IV**, colonies from the random initial generation that manage to recover even one piece of food from the environment are quite rare. To begin a run, we are forced to repeatedly restart with random ANNs until a colony is encountered that can recover even one unit of food in an environment completely filled with food. About one in 800,000 random colonies is capable of reliably retrieving at least one element of food from an environment filled with food. For comparison, the frequency of random foraging colonies in **AntFarm I** is about one in 1,000 and in **AntFarm II** and **AntFarm III** approximately one in 10,000.

Why are random ANNs that forage so rare in **AntFarm IV**? In the initial random population, on average 68.8% of the random connection descriptors lead *From* a valid input unit and of those 53.1% lead *To* another input unit and of those 46.9% are inhibitory connections. Therefore, 17.1% of the random connections inhibit an input unit. Only 18.8% of the random connection descriptors lead *To* an output, and of these only 93.8% have a non–zero weight, so only 12.1% of the random connections have the potential to affect the output unit activations. With such a large fraction of the active connection descriptors inhibiting input units, it is not too surprising that working random behavior functions are unlikely.

The evolved behaviors appear to be quite ant–like. Even in early generations, discrete changes in behavior are clearly evident between searching and transporting food to the nest. The typical search strategy that evolves consists of walking mostly straight ahead, with occasional random turns. The random turns keep the ants in the neighborhood of the nest, so severe food transport costs are rarely incurred. We never observed the use of the random inputs in **AntFarm II** or **AntFarm III**, yet every run of **AntFarm IV** utilizes them.

The ants also evolve to use the food sensors on their antennae, which is another innovation that we had not previously observed. We can determine how much the

| Action | Score |
|---|---|
| Food in nest | 1.0 |
| Move | -0.001 |
| Turn | -0.001 |
| Pheromone | -0.001 |
| Grab Food | -0.001 |
| Drop Food | -0.001 |

Table 6.4: The scoring summary for the **AntFarm IV** example run. The *Move* and *Turn* costs are scaled according to the fraction of the maximum permissible distance moved or amount turned (respectively). Costs are assessed even in the event that the action is not feasible (e.g. grabbing food when there is none available).

ants are relying on the antenna food sensors by turning them off and observing the relative performance with and without the sensors. When we do this test in early generations, performance (mean number of units of food returned to the nest) is cut in half when the food sensors are turned off. In later generations, performance only drops about 20%. We interpret this as an indication that the search strategy improves in later generations, so the ant is likely to walk across food without having to turn to grab it.

Once food has been acquired, the ant apparently makes neither random turns nor turns toward food. Instead, the ant follows the compass directly to the nest, ignoring all other inputs. Again, the ability to follow the compass is a behavior that never evolved in **AntFarm II** or **AntFarm III**.

It is quite apparent that input inhibition in the connection descriptor representation has a major effect on the behaviors that evolve. Without inhibition, the ants evolve to make use of only a fraction of their sensory inputs. With inhibition, nearly all of the sensors are used in an effective manner. The result is the evolution of realistic, ant-like foraging behavior.

We will now examine a run of **AntFarm IV** in detail. In this particular case, we evolve a populations of 16,384 colonies in an 8,192 x 8,192 environment. This run was restarted 36 times, because none of the first 36 random populations contained a colony that was capable of recovering even one piece of food to the nest. For the first 900 generations, we have 2 ants per colony, cover the entire world with food, and allow each ant to move 100 times (iterations). At generation 900, we increase the difficulty of the environment by placing food in only 5 percent of the locations, and clearing the food from the environment within a radius of 7 squares of each nest. The score and pheromone usage trace for the first 1000 generations is shown in Figure 6.5. The scoring is summarized in Table 6.4. The maximum score rises rapidly, and reaches a plateau by generation 200. The mean score follows, peaking at about generation 700. The mean number of units of pheromone that are dropped by the ants remains quite low throughout this portion of the run, indicating that the ants are not using pheromones as part of their foraging strategy. These first 1000 generations took about 7 hours to compute.

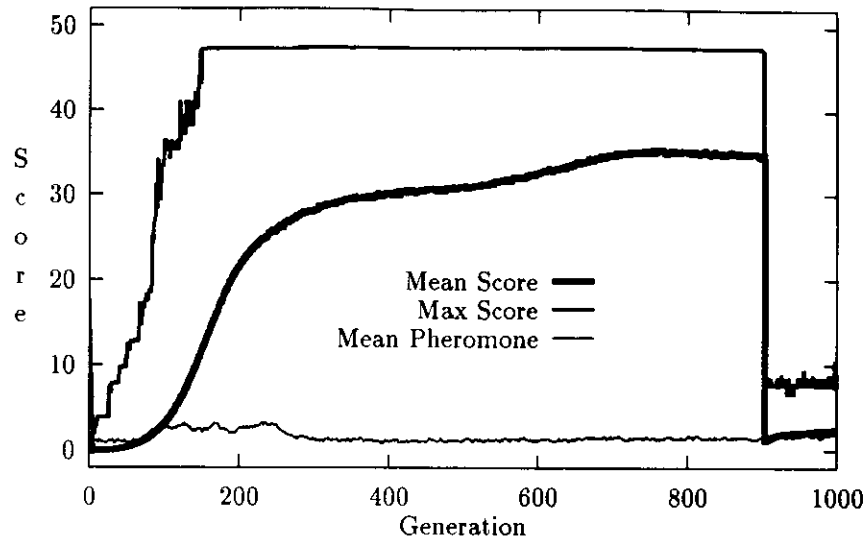


Figure 6.5: The score and pheromone trace for the first 1000 generations of the **AntFarm IV** run.

At generation 1001, we increased the size of the colonies to 10 ants each, and increased the number of time steps per ant per generation to 500. At the same time, we made the environment more difficult by placing food in only 1 percent of the locations, and removing all food that lies within 15 squares of any nest. At generation 1101, the radius of food removal is increased from 15 to 20. At generation 1301, we again increase the difficulty of the environment, by reducing the food frequency to 0.5 percent per grid square. The score and pheromone trace for generations 1000-2000 is shown in Figure 6.6. Although we increased both the foraging force and the foraging time by a factor of 5, this is more than offset by the increased difficulty of the environment. Each time we increase the difficulty of the environment, the mean score drops, but then rebounds to some extent after a couple hundred generations. During this period, there is no significant change in the use of pheromones (although pheromone levels are about 25 times higher than in the initial 1000 generations, due to 25 times as many forager time steps per generation). Generations 1001–2000 took about 175 hours to compute.

In this run, no cooperation has evolved. We can see this both by the small amount of pheromone generated by each colony, but this is also testable. We have re-run various portions of this simulation with the pheromone sensors turned off, with little or no change in behavior or scoring.

There are two likely explanations for not observing the evolution of cooperation: (1) cooperation is not advantageous in this environment, or (2) cooperation is hard to evolve. To test these hypotheses, we have re-run generations 1001–2000, except this time we take control of the pheromone output on each ant: when the ant is carrying food it drops a trail of pheromone, and when it is not carrying food it drops no pheromone. At generation 2001, we put the dropping of pheromones back under the control of the behavior function. At this time, we observe a decrease in the scores,

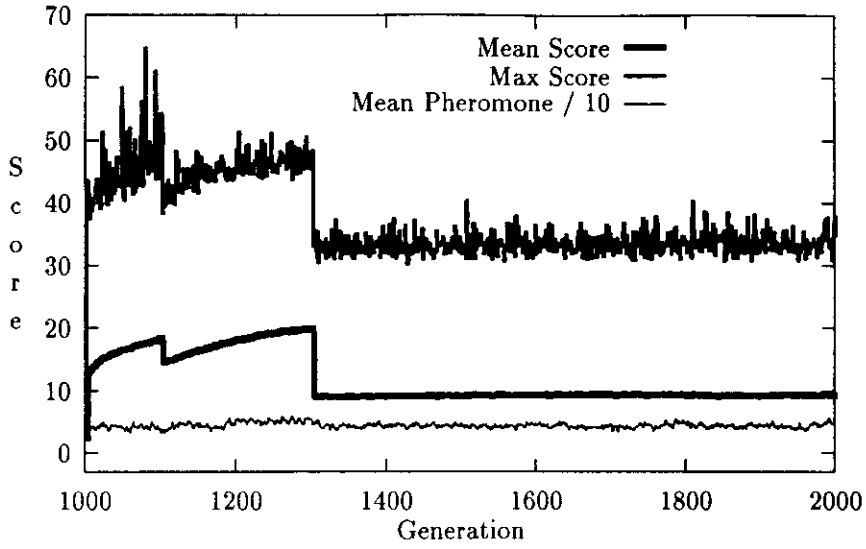


Figure 6.6: The score and pheromone trace for generations 1000–2000 of the **AntFarm IV** run.

along with a dramatic decrease in the amount of pheromone generated. Most of the population has evolve to drop pheromone trails by generation 2100. The score and pheromone trace of generations 1001–2200 are shown in Figure 6.7.

Our interpretation of this result is that when the ants were presented with a meaningful trail between the nest and piles of food, they evolved to follow the trails. When the ants are able to follow the trails, then mutations that cause them to leave trails will be favored by selection, resulting in the whole population dropping and following trails. Figure 6.8 compares the score metrics for the cooperative versus individual foraging strategies that evolved. Cooperative foraging clearly pays off. The mean score in the trail–laying population is about ten percent higher than the population that uses individual foraging, and the maximum score is about twenty percent higher. The scores of the trail–laying colonies include the cost of nearly one full point on average for the pheromones that they use to lay the trails.

This experiment is encouraging, because it demonstrates that **AntFarm IV** can evolve cooperative foraging behavior, and that the environments in which the evolution takes place makes cooperation advantageous. It is just a matter of time before a colony that randomly has the ability to follow a trail "suffers" a mutation that causes the ants to drop a pheromone trail when they are carrying food. When this rare event occurs, trail laying and following behavior will spread through the population, without our intervention.

As before, the length of the variable length chromosomes does not change dramatically, even over thousands of generations. Apparently, the initial length of 500 bits is sufficient to support the evolution of realistic foraging behaviors.

We have been working with **AntFarm** over a period of nearly three years. During this time, we have gained many insights into the evolution of ANNs, the representation of artificial organisms, designing the morphology of artificial organisms, etc. The next two sections summarize our current thinking on these important topics.

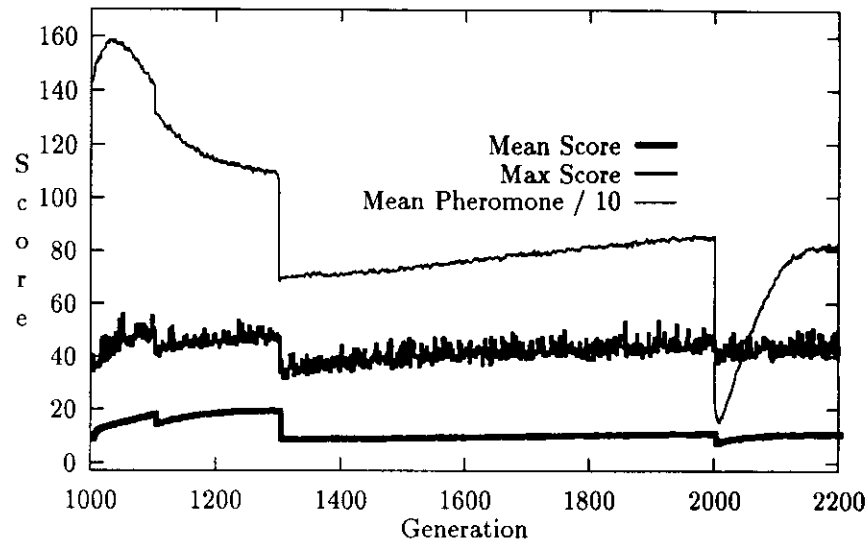


Figure 6.7: The score and pheromone trace for generations 1000–2200 of the **AntFarm IV** run. In this run, in generations 1001 through 2000, the ants are forced to use the pheromones to leave a trail while food is being carried.

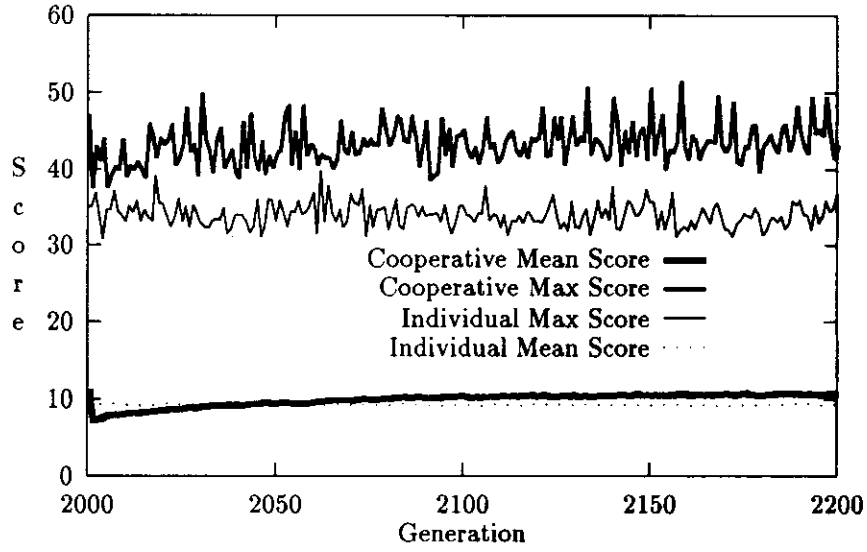


Figure 6.8: The score traces for generations 2000–2200 of the **AntFarm IV** run. Here, we compare a population using cooperative foraging to one using individual foraging.

## 6.4 Discussion: Evolving Artificial Neural Networks

During the past few years, ANNs have emerged as a useful and powerful programming technique for solving a wide range of problems involving pattern classification (Rumelhart and McClelland, 1986; Lippmann, 1987). The ANN programmer creates a network with the desired functionality by defining the size and topology of the network and presenting it with a large training set of sample input/output pairs. Based on the magnitude of the error between the network output and the desired output, the strengths of the internal connections of the network are adjusted by a "learning" rule. This optimization process is repeated until the errors become small. We will refer generically to all the various techniques of supervised learning involving backpropogation of errors as "backprop." Once the ANN has been trained, it can be applied to novel input patterns, and hopefully it will give nearly the correct output.

Although the backprop training process is mostly automatic, it requires the ANN programmer to develop the training set and to set the learning-rate parameters. The programmer must define the exact set of outputs that the network should produce for a representative set of the input patterns that the network is likely to encounter.

Although a great deal of progress has been made towards automating the learning process for many problem domains, little has been made towards automating the design of the network architecture (Miller et al., 1989). When designing the ANN, the researcher must rely on experience and informal heuristics because design principles do not yet exist.

Researchers have begun to apply genetic algorithms to the task of programming ANNs (Weiss, 1990), but they have been used to solve only relatively simple problems (Whitley, 1989; Whitley and Hanson, 1989; Miller et al., 1989; Harp et al., 1989; Jefferson et al., 1991). In each case, a "hybrid" approach has been used, applying the genetic algorithm either to the problem of network *design* or network *training*. In either case, the conventional methods described above are used for the non-evolutionary parts (Weiss, 1990).

When the evolutionary design approach is taken, the number of neurons and layers and/or the connectivity patterns are encoded in the chromosome and thus subject to evolution (Miller et al., 1989; Harp et al., 1989). Usually, conventional training methods such as backprop are applied to the ANN (with the weights initialized with small random weights), and those that learn the fastest or best (least error) are deemed more fit and thus are more likely to reproduce.

When the evolutionary training approach is applied, the training set is usually used to evaluate the performance of each ANN in the population, but no weights are adjusted (Whitley, 1989; Whitley and Hanson, 1989). Instead, the weights are encoded in the chromosome, and are adjusted by the genetic operators of selection (based on performance on the training set), recombination, and mutation. Although a training set is usually used to determine the fitness of a particular ANN, more abstract fitness functions (more appropriate to artificial life applications) are possible (Jefferson et al., 1991).

Genetic algorithms provide a higher-level of abstraction for ANN programming than conventional techniques. The evolutionary design approach frees the programmer from the necessity of designing the network architecture in detail. The evolutionary training approach replaces the learning rules with a genetic algorithm, and not only provides an alternative to conventional training techniques, but also allows ANNs to be applied to a wider range of problems. Evolution can be used to program ANNs in domains in which feedback (error information) is delayed, so it is not necessary to specify the exact output the ANN should produce in each situation. Instead, it is possible to specify the fitness of a network based on the end result of a sequence of many decisions.

The fact that evolution of ANNs makes it possible to deal with delayed feedback allows us to use them to define the behavior of artificial creatures. We gained a great deal of experience with the evolutionary training approach in developing the Genesys/Tracker system (Jefferson et al., 1991). The Genesys system uses a static architecture, three-layer recurrent ANN, with evolution operating only on the connection strengths (weights), threshold values, and initial activations of the various units. In our early work with **AntFarm**, we attempted to simply scale up the size of the Tracker ANN. Unfortunately, all of our attempts at the evolution of foraging behavior failed.

What is it about the **AntFarm** world that makes evolution fail to find weight settings that exhibit foraging behavior for these traditional ANN architectures? Is this only a problem in **AntFarm**, or is **AntFarm** representative of a large class behaviors that we (and nature) might want to evolve? We note that backprop does not have significant difficulty training an ANN to forage efficiently (given a suitable training set). In fact, we are able to train (with backprop) a fully-connected three layer ANN to perform a fairly comprehensive foraging algorithm in about 5 CPU minutes on a Sun4/330, using MITRE's Aspirin/MIGRAINES neural network simulation software. Presumably the difficulties we encountered are a result of using evolution rather than backprop to determine the settings of the weights. Before we discuss this problem in detail, it will be useful to examine the operation of backprop and evolution.

Backprop is a *supervised* learning technique. This means that during training, the network is presented with both the input vector and the corresponding desired outputs. The network is run with the given inputs, and the computed outputs are compared to the desired output. Where there is an error, *all* weights involved in the calculation of incorrect outputs are adjusted so that the output activation will be closer to the desired output. The learning algorithm is presented with detailed error information for a large set of input/output pairs (the training set). The researchers that use backprop hope that, given a relatively small and representative training set, the ANN will correctly generalize the relationships between the inputs and produce the correct (or nearly correct) outputs even for input vectors that do not arise in the training set.

In contrast, evolution provides no supervision. The ANN is run a large number of times, and it either takes part in reproduction, or it does not. Therefore, the only positive feedback that is provided to the ANN is that it has survived. No indication as to which aspects of its behavior were good or bad is provided, and no

error information is used by the genetic operators. All changes to the ANN (through the genetic operators applied to the genome) are random. These changes are as likely to mess up well–adapted portions of the ANN as to result in an improvement. Actually in practice (as in nature), genotypic changes as a result of the genetic operators of recombination and mutation are much more likely to result in a decrease in fitness than an increase in fitness. Those of us who use evolution to design ANNs hope to end up with an ANN that performs well in novel situations on ill–defined problems. In particular, we are interested in attacking problems for which we do not know the the correct output for a given input vector (and therefore a good training set is difficult or impossible to build). We are also interested in problems for which the quality of a particular ANN cannot be judged until it has been run on hundreds or thousands of input vectors, and even then the only possible feedback is "Well, this ANN is a little better than the other alternatives we have discovered." These are the exact constraints that apply to natural evolution.

Given the enormous differences in the way backprop training and evolution operate, it is not very surprising that evolution may fail on ANN architectures that are designed for backprop training. In Section 6.1 on **AntFarm II**, we experimented with a hand–coded behavior function that causes the ant to follow the compass inputs whenever it is carrying food in order to move back to the nest. While this hand–coded behavior appears to be relatively simple, it was rather complex to encode in the **AntFarm II** ANN architecture. The problem is that nearly all of the behaviors are conditional on whether or not the ant is carrying food. This requires all of the signals to flow through the hidden layer to allow inhibition of inappropriate behaviors. Figure 6.9 demonstrates this method: the compass–following behavior is turned on by default, but when the ant is not carrying food the **not carrying food** input inhibits compass–following. Similar connections implement compass–following that requires turning to the right.

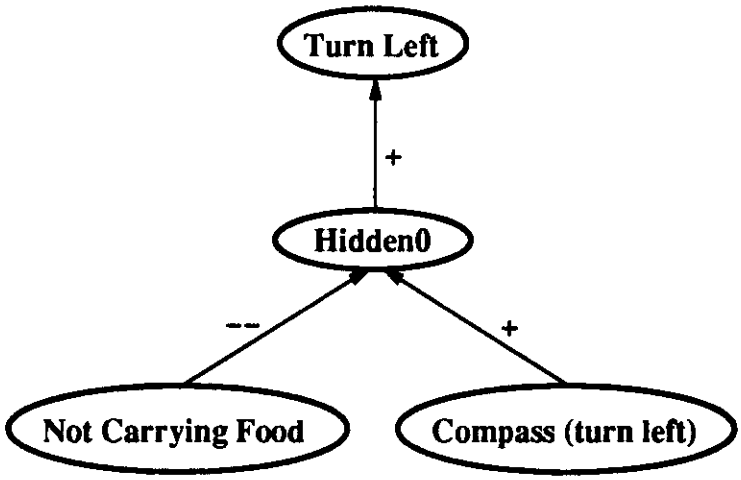


Figure 6.9: Behavior inhibition in the hand–coded ANN. A + indicates a excitatory connection, and -- a strongly inhibitory connection. In this case, the compass–following behavior is on by default, but it is inhibited while the ant is not carrying food.

The need for six connections and two hidden units just to implement compass following might explain why we never saw it evolve in any of our runs. If any of these six connections are missing or have an inappropriate strength or if there are extraneous, active connections to these hidden or output units, the ant is unlikely to ever find the nest. In fact, the incremental accumulation of these six alleles will not increase fitness until *all* are present, and in most cases each individually drastically reduces fitness. In the terminology of adaptive landscapes, there is a fitness valley that must be traversed to reach the compass-following ANN (Figure 6.10).

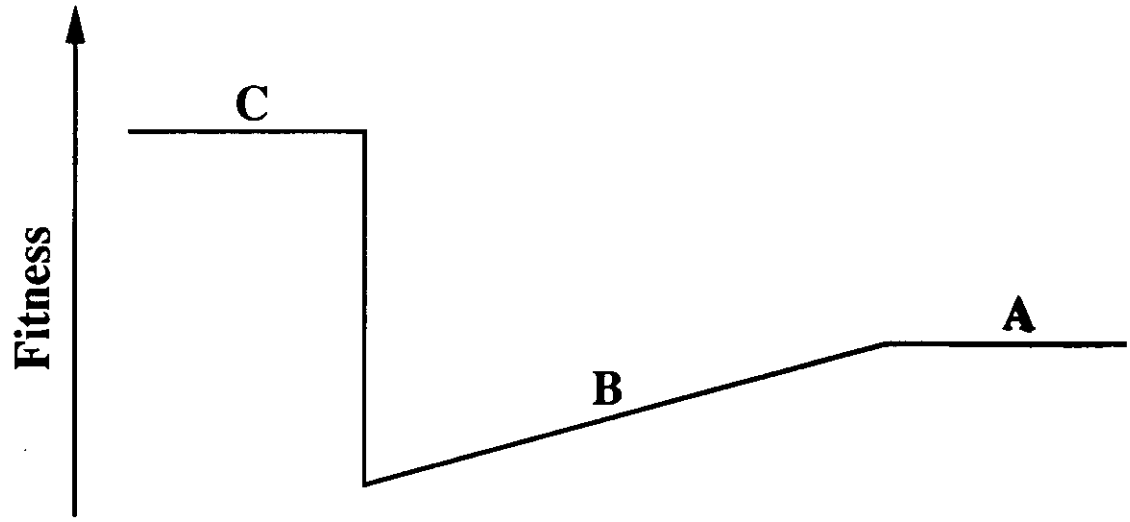


Figure 6.10: The adaptive landscape for the discovery of compass-following behavior in **AntFarm II** by evolution. The plateau **A** represents the portions of the space without any compass-following behavior, and also without any detrimental effects from alleles that are useful for compass following once the complete circuitry has evolved. The plateau **C** represents the portions of the space that successfully perform compass-following behavior. To move from **A** to **C**, it might be necessary to cross the fitness valley **B**, which might represent the effect of lacking an inhibitory connection (always follow the compass). If the alleles are accumulated in the right order, it is possible to find a path from **A** to **C** that does not involve lowered fitness, but it will still be unlikely to find the cliff, because there is no uphill gradient leading to its base.

The discovery of this sort of function is no problem for backprop. It will steadily move towards a correct set of weights by driving the extraneous incoming and outgoing weights to zero, and the relevant weights to appropriate values. Unlike evolution (Figure 6.10), there is no fitness valley to be crossed in order to achieve compass-following. Instead, the *detailed error information* tells it which direction to move the weight values and describes a smooth gradient towards a solution (Figure 6.11). Backprop simply climbs this hill.

The symmetric connection descriptor ANN representation with input inhibition (**AntFarm IV**) avoids these problems by making it very easy to specify and evolve behaviors with discrete modes. This encoding reduces the size of the representation, and minimizes the number of connections that need to be modified to evolve a given behavior. Backprop can update all weights on every training cycle, while evolution can only update a small handful of genes per individual each generation. In general,

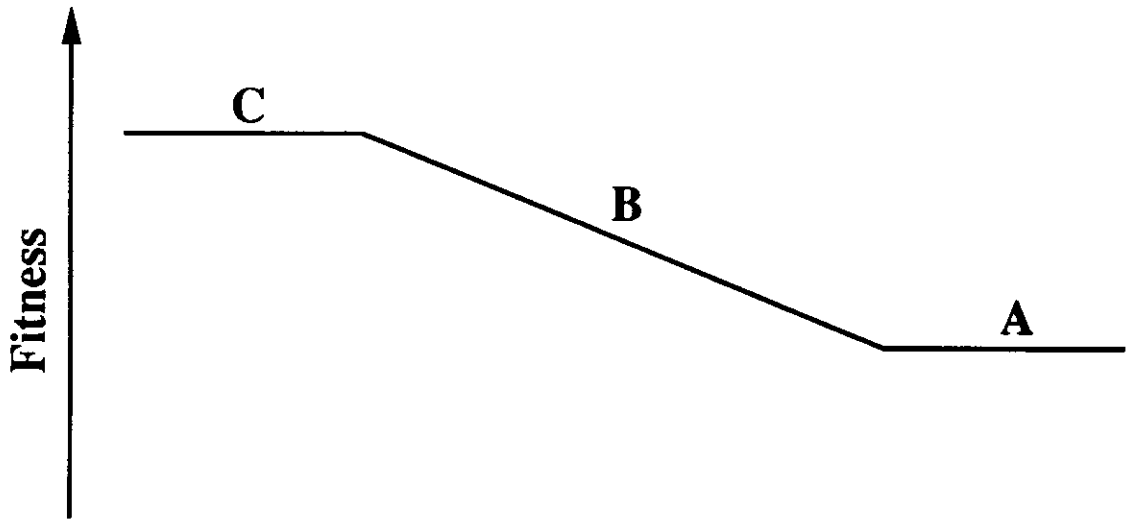


Figure 6.11: The adaptive landscape for the discovery of compass-following behavior in **AntFarm II** by backprop. The plateau **A** represents the portions of the space without any compass-following behavior, and also without any detrimental effects from alleles that are useful for compass following once the complete circuitry has evolved. The plateau **C** represents the portions of the space that successfully perform compass-following behavior. To move from **A** to **C**, there is a relatively smooth gradient **B**, which represents behaviors that are progressively closer to correct.

we conclude that it is unreasonable to expect evolution to be able to operate on *densely connected* ANNs when given a complex task.

The use of input inhibition in connection descriptor encoding is quite important, as we saw in **AntFarm IV** (Section 6.3). Our solution, which allows input to input connections to inhibit the use of inputs when the rest of the network is run, is not the final answer. Probably the right way to solve this problem is to have a separate inhibition network (complete with hidden nodes), in addition to the ANN that specifies the behavior. The inhibition network would be run before the behavior network, to determine the appropriate focus of attention on particular inputs. If the behavior network contains hidden units, the state of these should be inputs to the inhibition network. Another important aspect of the connection descriptor representation is that it combines aspects of both the evolutionary training and evolutionary design paradigms.

In any case, it is clear that there is more work to be done in regards to designing an appropriate organism representation. The next section describes some of the properties that we feel are necessary for a successful artificial organism representation in artificial evolution simulations.

## 6.5 Properties of Representations

In Chapter 5, we considered a number of candidate representations for **AntFarm** organisms. We assessed their strengths and weaknesses, and none of the traditional representations are well-suited for an artificial evolution task such as **AntFarm**. In

their place, we have formulated an acceptable alternative (the connection descriptor ANN). From our successes and failures we abstract a number of properties that we believe are necessary for successful evolution of complex artificial life. We list the properties here, and then discuss them in detail. We conclude that a good representation for organisms in artificial evolution simulations should have the following properties:

1. (approximate) *closure* of the set of legal genotypes under the action genetic operators;

2. *smoothness* of the phenotype under the action genetic operators, i.e. the behavior function should tend to change smoothly as the genotype is changed by mutation and recombination;

3. the ability to *scale* to large behavior functions, i.e. those that can handle large amounts of input and output data;

4. support for *symmetry* in the behavior function that matches the symmetry found in the artificial morphology;

5. the ability to *evolve* phenotypes that exhibit *both continuous and discrete behaviors* as a function of their inputs; and

6. a *uniform computational model*, i.e. the programming paradigm in which the behavior functions are expressed should not contain features that include any kind of explicit or implicit knowledge of the environment, nor bias toward a particular evolutionary trajectory.

Another property that is likely to be important is the ability to grow and shrink the genome size, and thus the complexity of the behavior function. At this point, we do not have enough evidence that it is critical, so we have not yet added it to this list.

Closure (1), smoothness (2), and symmetry (4) are properties of the development function and the genetic operators. Scaling (3), the ability to evolve continuous and discrete behavior (5), and uniformity (6) are all properties of the computational model of the behavior function. Smoothness (2) and the evolvability of continuous/discrete behavior (5) are very different constraints. The smoothness property refers to the effect of changes in the *genotype* on the overall behavior function of the organism, while continuous/discontinuous behavior refers to the effect of changes in the *sensory inputs* on subsequent behavior during the lifetime of the organism.

The property of syntactic closure constrains the development function and the encoding of the phenotype into the genotype. To be syntactically closed (approximately), the genetic operators must always (or almost always) produce syntactically legal genotypes when applied to other syntactically legal genotypes, where the function that maps the genotype to phenotype defines the legal syntax. An evolutionary system will not work if most mutations or recombinations are likely to transform a genotype that encodes a perfectly good behavior function program into one that is syntactically illegal.

The smoothness property requires that most changes to the genotype due to the application of genetic operators result in small changes in the phenotype. For example, a mutation in an ant should usually have a small affect on its foraging algorithm. Of course, it need not *always* cause a small change; some mutations will be fatal, and a few may cause profound but beneficial effects. Still, evolution cannot work if the phenotype space is not relatively smooth as a function of genotype. The encoding and mapping functions should be smooth not only under mutation, but also under recombination, implying that functionally related genes should usually be inherited as a unit (strongly linked). The smoothness property has the effect of requiring the "adaptive landscape" to be correlated with respect to the genetic operators (Kauffman and Levin, 1987). Evolution can successfully search the space of possible organisms only in correlated adaptive landscapes.

Smoothness is an extension of the closure property. Not only does it require a legal program to be (usually) transformed into another legal program by the genetic operators, but it requires it to be (usually) transformed into one that is behaviorally similar to the original.

The scaling properties of a representation are of extreme practical importance, because we must be able to simulate the evolution of large populations (tens of thousands) of organisms for thousands of generations. Scaling refers to the rate at which the size of the representation grows as a function of the number of inputs, outputs, and bits of internal memory. The size of the representation includes

- the number of bits in the genotype;
- the number of bits required to store the behavior function;
- the amount of time to translate from the genotype to the behavior function; and
- the amount of time to run a set of inputs through the behavior function to produce the outputs.

We are interested in organisms with dozens or hundreds of inputs, outputs, and bits of internal memory.

The property of symmetry in the representation is almost certainly critical to the evolution of complex behaviors. Without symmetry, evolution must find each innovative behavior more than once. The development function should use the genotype as a blueprint for the construction of multiple, symmetric copies of the components of the behavior function. In this way, symmetric behaviors are produced, with the fortunate side effect that the size of the search space in which evolution operates can be dramatically reduced. In nature, symmetric structures are usually constructed by multiple copies of the same process of development. In the symmetric connection descriptor ANN representation, the development function builds symmetric ANNs by mapping each connection descriptor into multiple connections.

We characterize behaviors as either continuous or discrete (although many behaviors are intermediate between these two extremes). Roughly speaking, a behavior function produces continuous behavior when a small change in the inputs to the

function results in a small change in the outputs, and discrete behavior when a small change in the inputs results in a large change in the outputs. We believe that in our target class of environments and organisms, a combination of both continuous and discrete behaviors will generally be necessary.

Consider the **AntFarm** task. Foraging ants seem to have two modes of behavior: (1) search and (2) transport. While an ant searches for food, small changes in the local pheromone levels should result in small changes in its behavior (continuous). On the other hand, its behavior should change completely based on whether or not it is carrying food. When an ant is carrying food, it should walk directly home (possibly leaving a pheromone trail if the pile of food is big) and deposit the food in the nest. When an ant is not carrying food, it should search for food, perhaps by doing a pseudo-random walk or following a pheromone trail if one is encountered. The behavior of the ant must change completely based on a single bit of input (a very small part of the sensory inputs). This is an example of discrete behavior. Most or perhaps all complex behaviors will involve the combination of different modes of behavior, and thus require the evolution of discrete behavior. Within each mode, the behavior is continuous, but the transition from one mode to another involves only a small part of the total set of inputs.

The final property is that the computational model of the representation should be uniform. In particular, it must be able to describe all desired behavior functions *without designing features of the problem or possible solutions into the representation.* For example, through all the years of neural network research, the area of network design has remained a "black art" (Miller et al., 1989; Harp et al., 1989; Weiss, 1990)—each new application requires a new design, and no rigorous design principles exist. This is a situation that we must avoid. One way to avoid this problem is to use representations that are based on a programming "language" with few primitives, such as an ANN.

One of our main assumptions about the environment and organisms is that their interactions are too complex for us to completely understand, so we cannot easily construct a good behavior function. For example, in our **AntFarm II** experiments, we verified that searching for food in a circular path was more efficient that walking in a straight line. We were rather surprised by this, and would not have predicted this outcome without having carefully analyzed the environment. If we are trying to shed light on a biologically motivated hypothesis, the results will be invalid if we bias the organisms toward (or away from) some evolutionary path. Even if we are simply interested in engineering an organism for survival in a particular environment, we will almost certainly do a bad job of guiding the representation toward good solutions by designing a special-purpose representation. The computational model must not have knowledge of the problem or solutions embedded in it.

## 6.6 Simulating "Laws of Nature"

**AntFarm** requires the explicit representation of an artificial world. A major issue in simulating the world is the fact that the physical invariants (the "laws of nature")

must be maintained. Difficulties arise because the simulated organisms are embedded in the world simulation, and the organism simulations are not well-behaved. The artificial organisms will often decide to perform actions (such as walking through solid objects) that would violate the physical invariants of the world if we were to allow them to occur. Somehow the computation must be organized so that the creatures may attempt any action, but only actions consistent with the laws of the artificial world are allowed to occur.

This is not a problem that occurs in a more traditional parallel simulation. The programmer would build the physical constraints into each of the interacting processes (analogous to our organisms), which would constrain themselves to not attempt actions that would violate the physical invariants. In a simulation in which all of the code is human-written, the organism programs would never specify an action that cannot be safely executed.

But programs that arise by evolution can, through mutation or recombination, produce code that causes organisms to attempt to walk through each other, pick up food that does not exist, etc. It is not reasonable to try to design an organism representation that will always specify executable actions. The solution that we have adopted is to put a wrapper around the behavior function which basically consists of the logic that the hand-coded simulation would have. This serves to preserve the simulated physical invariants of the artificial world. In effect, we must put a hand-written barrier between the two levels of simulation.

In **AntFarm I**, there are a finite number of food units in each location, and when a piece of food is picked up by an ant, it is removed from the environment data structure. At any moment, there may be many ants that are simultaneously attempting to grab food at a particular location. In order to maintain the proper amount of food in the simulation, we cannot allow more ants to successfully grab food from a location than there are units of food there. One option is to simply detect the conflict, and do not let any of the ants pick up food. This "solution" overly constrains the actions of the ants, and is not very realistic. If they were real ants, some would certainly be successful in picking up food.

Instead, we choose to arbitrate among the contending ants, and allow as many ants as possible to pick up food. We use the *arbitrate()* function that was presented in Section 2.4.1. At each location in the environment, if any ants are trying to grab food there, exactly one will be selected by the arbitration algorithm. Suppose there are ten ants at the same location, and all ten are contending for five units of food. One pass of the arbitration algorithm leaves nine ants contending for four units of food, so four more iterations are needed to satisfy as many ants as possible. Likewise, if there were three ants contending for nine units of food, three iterations would be required to satisfy all of the ants. At each time step in the simulation, the arbitration algorithms described above must be iterated at least the minimum of $a$, the number of ants, and $f$, the number of units of available food:

$$\min(a, f)$$

Because the Connection Machine is fully synchronous, we always handle the worst case of $I$ iterations, which is the maximum number of iterations required by any of

the environment locations:

$$I = \max_{x,y}^{x \times y} (\min(a, f)) \tag{6.1}$$

This method is not very expensive if the organisms or food particles are distributed with low density. However, a high density of both food and ants in even one location can adversely affect the performance of the whole simulation.

Another class of interactions that arises in **AntFarm** occurs when many ants drop pheromones in parallel. This case is much less complex than the problem of grabbing food. Rather than exclude the behavior of other ants, the dropping of pheromones results in a combined effect. When several ants simultaneously drop pheromones in the same environment location, we add them in parallel using the Connection Machine `send-with-add` command.

## 6.7 Implementation Notes

**AntFarm I** predates the core library, so it is by far the largest of the simulations that we have presented in this dissertation, containing about 12,000 lines of code. About 4,300 lines are devoted to the various ANN implementations, and about 1,000 lines of code implement the environment. The organisms themselves (the ants and colonies) are implemented in 2,800 lines of code, and the chromosomes only 400 lines. Checkpointing requires 1,100 lines of code. Most of the remaining 2,400 lines are devoted to instrumentation. During a typical **AntFarm I** run, approximately 70% of the time is spent running the ANNs, and most of the remaining time is consumed by interprocessor communication associated with the ant's interactions with the environment (primarily gathering of sensory information).

**AntFarm II** through **AntFarm IV** are nearly identical, and considerably smaller than **AntFarm I**. The smaller size is due in part to the use of the core library, but also because the only organism representation is the connection descriptor ANN, and because these simulations were designed to be simpler than **AntFarm I**. Each of these simulations is about 4,200 lines (beyond the core library). The ANN requires about 1,000 lines of code. Instrumentation accounts for another 1,000 lines. The colony/ant code is 900 lines long, and checkpointing takes another 400 lines. The remaining 900 lines is split evenly between the environment and run-time selection of parameters. A typical run of any of **AntFarm II** through **AntFarm IV** is divided evenly between running the ANNs and interactions between the ants and the environment. In all of the **AntFarm** implementations, lack of memory has always been a severe problem.

It is difficult to calculate the amount of time we have spent running **AntFarm**. A typical run lasts several days, and we have done many runs. We have run **AntFarm** for at least 9 CPU months on the UCLA 16K processor Connection Machine-2.

## 6.8 Discussion

The **AntFarm** world evolves foraging behavior in colonies of ant-like organisms. The **AntFarm** simulations are quite different from the examples in Chapters 3 and 4. While the previous simulations focused on the evolution of genotypes and phenotypes derived trivially from the genotype, **AntFarm** attempts to evolve the details of high-level, cooperative behavior. The evolution of high-level behaviors leads to the difficult problems of the design of the simulated morphology of the organisms and the design of the behavior function representation.

In this chapter (and Chapter 5), we have described a series of instantiations of the **AntFarm** world and organisms, progressively changing the ant's morphology and the behavior function. These changes have yielded more and more ant-like behavior.

In **AntFarm I**, our decision to use a 3 x 3 sensing organism placed on the environment grid was based primarily on the desire for a fast and simple implementation of the organism. We gave the ants a huge amount of low-level sensory input, a large behavior function, and lots of Connection Machine time. We expected that somehow evolution would take whatever we gave it and provide us with foraging behavior. In fact, the ANN organism representations that had been used successfully in less complex evolutionary situations were useless when scaled up for use in **AntFarm**. We were able to evolve foraging behavior only after inventing the connection descriptor ANN representation. But even then, the foraging behavior did not look very much like what real ants do.

**AntFarm II** took the ants off of the grid, simplified the environment somewhat, and reduced the number of sensors to only the head and two antennae. The implementation details of moving and locating the ants and their sensors is certainly more complex and computationally expensive, but this is offset by the reduced size of the behavior function. The ant's movement was more ant-like, but the foraging behavior consisted of walking in circles (either always to the right or always to the left).

**AntFarm III** modified the connection descriptor ANN representation to always yield symmetric ANNs. The ants still walked in circles, but half the circles were to the right, and half to the left. **AntFarm IV** addressed the problem of discrete changes between two radically different behaviors (in this case searching for food and transporting the food to the nest) based on small changes in the sensory inputs. This change resulted in quite realistic looking foraging behavior.

# Chapter 7

# Partition: Genetic Algorithms and the Importance of Spatially Structured Populations

In this chapter, we introduce several metrics for evolving populations, and use them to characterize the differences between local and panmictic selection and mating schemes. For computational convenience we have performed this study using the optimization of graph partitions as the evolutionary task. We refer to the simulation used in this study as **Partition**. The results of **Partition** indicate that local mating superior to panmictic mating when applied to traditional optimization applications. Local mating seems to

- find genotypes with optimal fitness scores faster;
- find multiple optimal genotypes in the same run; and
- be much more robust, i.e. less susceptible to premature convergence.

These results are encouraging, but we should caution that they are based on a single optimization problem, a single recombination rate, a single mutation rate, one local mating algorithm, and large populations. Further investigation seems both appropriate and necessary.

## 7.1 The Graph Partitioning Problem

The graph partitioning problem (Ackley, 1987) is to find a *partition* of the set $V$ of vertices of a fixed graph $G$ into two subsets $V_0$ and $V_1$ such that

- $V = V_0 \cup V_1$ and
- $V_0 \cap V_1 = \emptyset$. An *optimal partition* as the additional properties that
- $|V_0| = |V_1|$ (which presumes that $|V|$ is even) and

- the cut size (the number of edges with one endpoint in $V_0$ and the other in $V_1$) is minimized.

We represent a partition of $G$ by the string $S$ of $|V|$ bits, such that vertex $i \in V_{S[i]}$ (Figure 7.1).

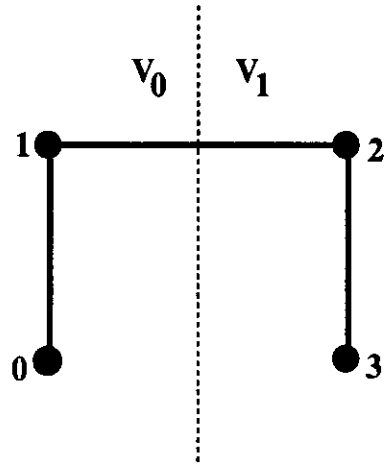


Figure 7.1: An example graph partition. There are four vertices (numbered 0 through 3) connected by three edges (the solid lines). The dotted line indicates the location of the partition. This partition is represented by $S = 0011$ (numbering bit positions in increasing order left to right in $S$), because vertices 0 and 1 are on the 0 side of the partition ($V_0 = \{0, 1\}$), and vertices 2 and 3 are on the 1 side of the partition ($V_1 = \{2, 3\}$). This partition is balanced (two vertices on each side) and has a cut size of 1.

In order to incorporate this problem statement into a genetic algorithm, we use the scoring function defined by Ackley (1987), which has a penalty that is quadratic in the imbalance of a partition:

$$s(x) = -cutsize(x) - 0.1(Z(x) - O(x))^2 \tag{7.1}$$

where $x$ is a string of bits, $Z(x)$ is the number of 0's in $x$, and $O(x)$ is the number of 1's in $x$. The cut size and penalty term are given negative signs to make this a function maximization problem (with maximum value of 0).

## 7.2 Evolution Metrics

In this section, we introduce four metrics that we use to characterize evolving populations. These particular metrics were chosen in order to measure and highlight the differences in the evolutionary dynamics of spatially structured (locally mating) and panmictic (globally mating) populations.

### 7.2.1 Diversity of Alleles

The diversity of alleles in the population is a basic measure of the genetic variation. In this chapter, we consider loci with exactly 2 alleles, so the maximum diversity

occurs when each allele frequency is 0.5, and minimum diversity when one of the alleles is fixed (frequency of 1.0). We measure the allele diversity of a population by comparing the *observed* allele frequencies to those of a maximally diverse population. We define the diversity of alleles of a population at locus (bit position) $i$ as

$$D_i = 1 - 4(0.5 - z_i)^2$$

where $z_i$ is the frequency of the 0 allele at locus $i$ across the whole population. $D_i$ can range from 0, which indicates complete fixation (all strings are identical at locus $i$), to 1 which indicates the maximum possible genetic diversity by this measure at locus $i$. $D$, the genetic diversity of the population for the *entire genome* $S$, is defined as

$$D = \frac{\sum_{i=0}^{|S|-1} D_i}{|S|}$$

$D$ also ranges from 0 (fixation) to 1 (maximal diversity).

### 7.2.2 Diversity of Genotypes

The measure of diversity of genotypes complements the allele diversity measure by observing how the alleles are brought together as complete genotypes. It is possible to have significant diversity of alleles, but low genotypic diversity. For example, a population that is half strings of all zeros and half of strings of all ones has maximal allele diversity, but little diversity of genotypes. The diversity of genotypes is an indicator of the breadth of the genetic search, and measures correlations among loci. We measure genotypic diversity by choosing a random sample (without replacement) of 10 loci and counting the number of unique genotypes with respect to these loci that are represented in the population. A new set of loci is sampled each generation. This provides a measure of the breadth of the genetic search. Since there are two alleles possible at each locus, we will observe between 1 and $2^{10} = 1024$ unique genotypes.

### 7.2.3 Inbreeding Coefficient/Panmictic Index

The inbreeding coefficient and panmictic index are measures of the degree of structure or panmixia (respectively) in a population. Inbreeding is the mating of two individuals who are more similar to each other than would be expected if mates were chosen randomly from the population as a whole. In natural populations of diploid organisms, the primary effect of inbreeding is to decrease the frequency of heterozygous genotypes (Hartl and Clark, 1989). A (diploid) genotype is heterozygous at a locus if the two copies of the chromosome contain different alleles at that locus. The *inbreeding coefficient* $F$ is calculated by comparing the actual proportion of heterozygous genotypes in the population with the proportion that would be expected to occur under random mating. Heterozygosity (and thus the inbreeding coefficient) is defined in terms of diploid organisms, but with few exceptions genetic algorithms use haploid strings. Therefore, we will measure $F$ for a *mating pairs* of haploid organisms.

Let $H$ be the *observed* proportion of heterozygous genotypes, and $H_0$ be the *expected* heterozygosity if the population were randomly mating (i.e. panmictic). The standard form for $F$ is

$$F = \frac{H_0 - H}{H_0}$$

In this chapter, there are two alleles (0 and 1) at each locus. From the Hardy-Weinberg Principle (Hartl and Clark, 1989), if $p$ is the frequency of the 0 allele in the population, under random mating (panmixia), we expect $p^2$ of the population to have genotype 00, $2p(1-p)$ of the population to have either genotype 01 or 10 (which are indistinguishable), and $(1-p)^2$ of the population to have genotype 11. Because $H_0$ is the expected number of heterozygotes, $H_0 = 2p(1-p)$. Falconer (1981) defines the *panmictic index* $P$ to be

$$P = 1 - F$$

We will use $P$ rather than $F$, since we are really interested in measuring the degree of panmixia. $P$ has a value near 1 for well-mixed (panmictic) populations, and low values for subdivided populations.

### 7.2.4 Speed and Robustness

We measure the speed of evolution achieved by a genetic algorithm in two ways.

- Number of generations required to discover an optimal solution. This measure allows implementation-independent comparisons.
- Computational time required to discover an optimal solution. This measure takes into account the varying computational costs of the reproduction portion of the genetic algorithm for a particular implementation.

We define the robustness of a genetic algorithm to be the fraction of runs that find at least one optimal solution. Since we do not always discover an optimal solution, we stop such runs at 1000 generations. We report speed in terms of the median over all runs, including those that do not find optimal partitions.

## 7.3 Selection Schemes

### 7.3.1 Local Selection

To simulate isolation by distance in the selection and mating process, we place the individuals on a toroidal, 1 or 2 dimensional grid, with one organism per grid location. Selection and mating take place locally on this grid, with each individual competing and mating with its nearby neighbors. In our local mating scheme, the two parents of each offspring are chosen by the *get-random-walk-parent(R)* function from Section 2.4.1. Deme size (and thus the rate of gene flow) is a function of $R$, the length (number of steps) of the random walks.

### 7.3.2 Stochastic Selection

The most common panmictic selection strategy used in genetic algorithms is stochastic selection with replacement (or roulette wheel selection) (Goldberg, 1989a). The probability that individual $x$ is chosen as a parent as

$$P(x) = \frac{s_x}{\sum_{i=0}^{N-1} s_i}$$

where $s_j$ is the fitness score of the individual labeled $j$, and $N$ is the size of the population. The stochastic selection algorithm makes the assumption that all fitness values are non-negative. Because $s_x \leq 0$ for the graph partitioning problem, we adjust the fitness scores for the stochastic selection algorithm

$$s'_x = s_x + |\min s_i|$$

where $s_x$ is the score of partition $x$. The details of this final adjustment can have significant effects, because the actual value of $s'_x$ determines the strength of selection that is felt by each individual $x$.

### 7.3.3 Linear Rank Selection

Another panmictic selection algorithm is the linear rank method (Grefenstette and Baker, 1989). The linear rank selection algorithm defines the *target sampling rate* (TSR) of an individual $x$ as

$$TSR(x) = Min + (Max - Min)\frac{rank(x)}{N-1}$$

where $rank(x)$ is the index of $x$ when the population is sorted in increasing order based on score, and $N$ is the population size. Additional constraints are

- $0 \leq TSR(x)$;
- $\sum_{x=0}^{N} TSR(x) = N$;
- $1 \leq Max \leq 2$; and
- $Min + Max = 2$.

The TSR is the number of times an individual should be chosen as a parent for every $N$ sampling operations, where a sampling operation is simply the act of choosing an individual as one of the parents for an offspring.

## 7.4 Comparison of Selection Mechanisms for the Partitioning Multilevel Graphs

We have implemented the graph partition scoring function (Equation 7.1), the evolution metrics, and the four selection algorithms in **Partition**. The next step in this

study is to choose a graph to partition, and the perform the empirical comparison between the selection mechanisms.

There is little structure in small random graphs, so good partitions are relatively easy to find by search methods that operate via simple hill climbing (Ackley, 1987) (e.g. via mutation–like steps on the fitness surface). To make the problem more difficult, a "clumpy" or *multilevel* graph is used. We adopt Ackley's multilevel graphs, where a clump consists of four fully connected vertices. The clumps are connected together to form a hypercube, and two of these hypercubes from the graph. The graph consists of two identical, disconnected pieces. For example, a 64 vertex multilevel graph consists of 16 clumps, as shown in Figure 7.2. Each of the two connected

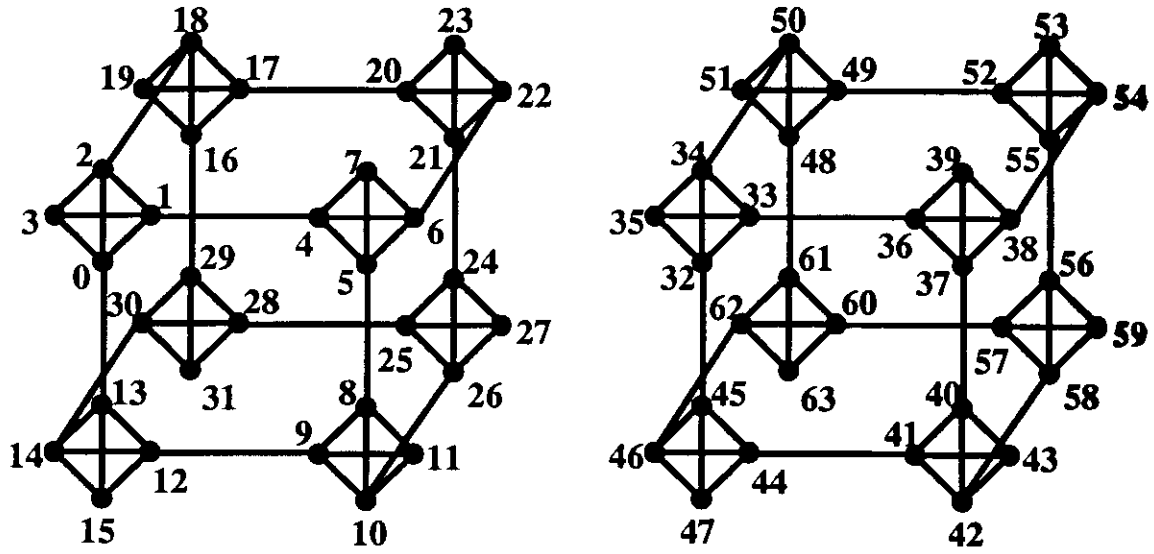


Figure 7.2: The 64–vertex multilevel graph. The number associated with each vertex indicates the locus (bit position) in the chromosome that specifies on which side of the partition it lies.

components is a cube with a clump in each of the 8 corners.

We map the vertices within each clump consecutively on the string $S$, and the clumps of each connected component consecutively on the string $S$ (Figure 7.2). This means that there are two optimal solutions to the multilevel graph partitioning problem: a string with $\frac{|V|}{2}$ 0's followed by $\frac{|V|}{2}$ 1's, and the bitwise complement. Both of these optimal partitions have the maximum fitness score of 0.

Because most of the edges are found within clumps, selection quickly focuses the genetic search on partitions that do not cut clumps (Ackley, 1987). In order to continue the search for an optimal partition, clumps must be moved across the partition via recombination. Recombination is necessary because moving a clump across the cut one vertex at a time (i.e. by point mutations) results in a dramatic decrease in the score. If a given clump is entirely on one side of the partition, a single–bit mutation within the clump can only result in a higher score when the partition is rather unbalanced (and the mutation increases the balance). Such unbalanced partitions will be rare, since they will have lower fitness scores than their balanced counterparts in the populations (of course this is a function of how strongly the

selection algorithm selects against low-scoring individuals). For a vertex with only intraclump connections, the score will increase if (from Equation 7.1)

$$3 < 0.1(Z - O)^2$$

which simplifies to

$$Z - O > \sqrt{30}$$

where $Z$ is the number of zeros and $O$ the number of ones in the string. An imbalance of 6 vertices is required to allow single-bit mutations to result in a partition with a higher score. For a vertex with one interclump connection to a vertex on the other side of the partition, moving this vertex to the other side of the partition will increase the fitness score when the imbalance is as small as 5 vertices. For a vertex with one interclump connection to a vertex on the same side of the partition, moving this vertex to the other side of the partition will increase the score when the imbalance is 7 vertices. The amount of imbalance that is necessary is large relative to the total size of the graph ($|V| = 64$). The result is that the multilevel graph partitioning problem is difficult for genetic algorithms, unless convergence can be avoided, because recombination requires diversity in order to have an impact.

In this section, we compare the various selection algorithms on the 64-vertex multilevel graph partitioning problem. The selection algorithms are local mating in both 1 and 2 dimensional geometries with random walks of length $R = \{1, 5, 10, 20, 30\}$, linear rank selection with $Min = 0.0$ and $Max = 2.0$, and stochastic selection with replacement. We vary the population size $N$ over a range from $2^{13} = 8,192$ to $2^{19} = 524,288$ individuals in each generation.

During recombination, crossovers occur with a constant probability of $\rho = 0.02$ between each pair of consecutive bits, so most matings (64%) will result in zero or one crossovers, but some matings may result in many crossovers. In a similar way, point mutations (bit flips) occur with a constant probability of $\mu = 0.001$ per bit, with only 6% of the individuals in each generation experiencing one or more mutations. See Section 2.4.2 for the recombination algorithm, and Section 2.4.3 for the mutation algorithm.

### 7.4.1 Results of the Diversity of Alleles Experiments

The diversity of alleles that is maintained over time by the four selection algorithms is plotted in Figure 7.3. Both of the panmictic selection algorithms quickly lose diversity even with a population of $N = 65,536$, while both of the local selection schemes maintain a high degree of variation. Although the 1 dimensional local scheme maintains nearly perfect variation, the 2 dimensional algorithm loses a small amount over time. Of the two panmictic selection algorithms, linear ranking loses diversity sooner, and stabilizes at a lower level.

### 7.4.2 Results of the Diversity of Genotypes Experiments

The genotypic diversity for the four selection algorithms is plotted in Figure 7.4. This data is based on a random sample (with a new sample each generation) of 10 of the 64

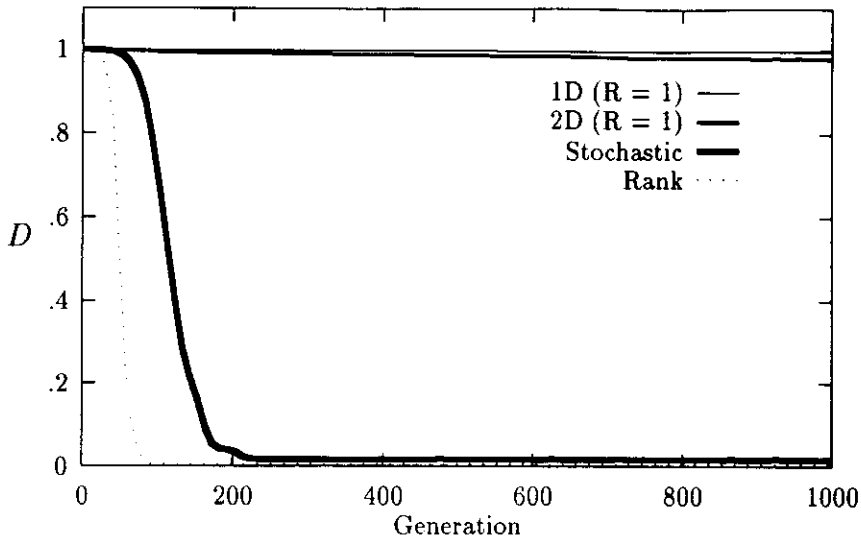


Figure 7.3: The diversity of alleles $D$ maintained by the four selection algorithms. $N = 2^{16} = 65,536$ and the task is the 64 vertex multilevel graph. Each curve is the average of 7 runs.

loci in the genome. In all four cases, the genotypic diversity begins to fall after only a few dozen generations with a population of $N = 65,536$. Both of the local mating schemes maintain a count of about 150 genotypes (out of a possible 1024), while stochastic selection stabilizes around 40, and linear rank selection around 15. With the exception of 1 dimensional local mating, all of the algorithms appear to reach their stable values by generation 250. The 1 dimensional local mating algorithm appears not to stabilize before 1000 generations.

### 7.4.3 Results of the Panmictic Index Experiments

The panmictic index for the four selection algorithms is plotted in Figure 7.5. In the early generations, neither of the panmictic selection algorithms shows any excess homozygosity ($P$ near 1), while both local algorithms immediately show significant and increasing excess homozygosity. A significant degree of excess homozygosity is observed in later generations with both panmictic algorithms, and throughout the experiments for both local selection schemes. In later generations, stochastic selection is characterized by a much higher panmictic index than the others.

### 7.4.4 Results of the Speed and Robustness Experiments

The first speed comparison is based on the number of generations until the first appearance of an optimal partitioning of the 64 vertex graph (Table 7.1). Of the two panmictic selection algorithms, linear rank selection finds solutions nearly twice as fast as stochastic selection. We also note that neither panmictic algorithm reliably finds optimal solutions when applied to the smaller population sizes. For both of

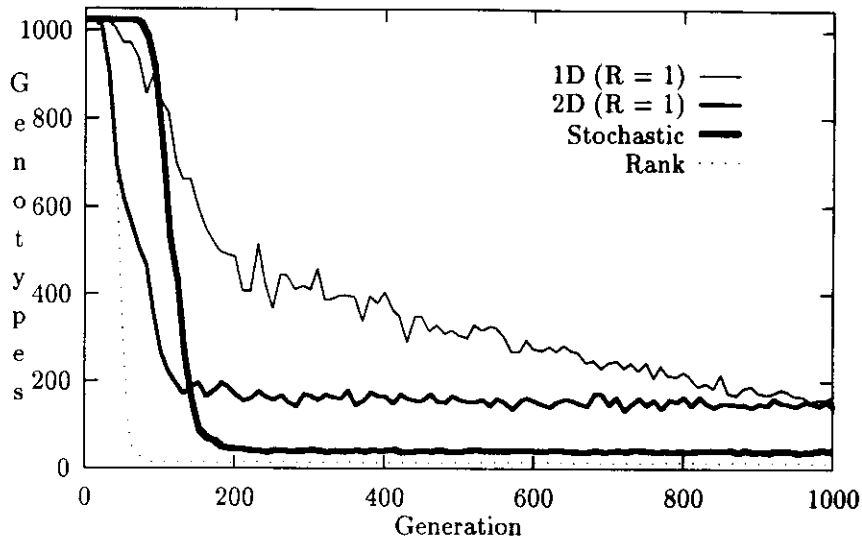


Figure 7.4: The genotypic diversity for the four selection algorithms, based on sampling 10 loci per generation. $N = 2^{16} = 65,536$ and the task is the 64 vertex multilevel graph. Each curve is the average of 7 runs.

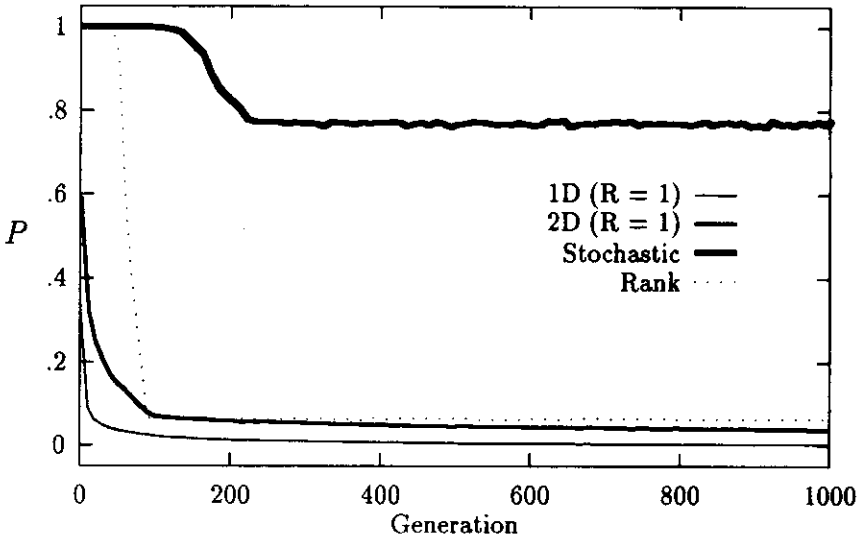


Figure 7.5: The panmictic index $P$ for the four selection algorithms. Each curve is the average of 7 runs. $N = 2^{16} = 65,536$ and the task is the 64 vertex multilevel graph.

the local mating algorithms, longer random walks result in faster evolution, and for the same $R$ value, 2 dimensional local mating is faster than 1 dimensional local mating. With the exception of the most constrained local mating (1 dimensional with $R = 1$), local mating always beats panmictic, and in some cases by about a factor of 7. This suggests that some intermediate amount of spatial structure yields the fastest evolution.

| Algorithm | $\log_2$ Population Size | | | | | | |
|---|---|---|---|---|---|---|---|
| | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
| Stochastic | † | † | 151 | 137 | 119 | † | 136 |
| Linear Rank | † | 57 | 66 | 52 | 35 | 32 | 31 |
| 1D ($R = 1$) | 156 | 148 | 142 | 124 | 126 | 108 | 114 |
| 1D ($R = 5$) | 85 | 79 | 59 | 63 | 57 | 49 | 50 |
| 1D ($R = 10$) | 56 | 50 | 47 | 50 | 43 | 40 | 41 |
| 1D ($R = 20$) | 42 | 40 | 37 | 37 | 34 | 30 | 27 |
| 1D ($R = 30$) | 41 | 39 | 32 | 32 | 29 | 26 | 24 |
| 2D ($R = 1$) | 48 | 43 | 41 | 42 | 40 | 40 | 38 |
| 2D ($R = 5$) | 23 | 20 | 16 | 17 | 16 | 16 | 15 |
| 2D ($R = 10$) | 14 | 13 | 13 | 12 | 12 | 11 | 11 |
| 2D ($R = 20$) | 13 | 11 | 10 | 11 | 9 | 9 | 9 |
| 2D ($R = 30$) | 11 | 8 | 9 | 9 | 8 | 8 | 8 |

Table 7.1: Generation of first appearance of an optimal partition (median of 11 runs) on the 64 vertex multilevel graph problem. † indicates that the median run did not find an optimal solution within 1000 generations.

| Algorithm | $\log_2$ Population Size | | | | | |
|---|---|---|---|---|---|---|
| | 14 | 15 | 16 | 17 | 18 | 19 |
| Stochastic | † (0.57) | 165 (1.09) | 285 (2.08) | 552 (4.64) | † (9.47) | 2652 (19.50) |
| Linear Rank | 40 (0.70) | 83 (1.26) | 132 (2.54) | 200 (5.72) | 379 (11.84) | 755 (24.34) |
| 1D ($R = 1$) | 24 (0.16) | 41 (0.29) | 70 (0.56) | 139 (1.10) | 264 (2.44) | 540 (4.74) |
| 1D ($R = 5$) | 15 (0.19) | 21 (0.35) | 42 (0.67) | 81 (1.42) | 133 (2.72) | 265 (5.29) |
| 1D ($R = 10$) | 13 (0.25) | 20 (0.42) | 37 (0.74) | 65 (1.50) | 122 (3.06) | 252 (6.15) |
| 1D ($R = 20$) | 16 (0.39) | 20 (0.54) | 35 (0.95) | 72 (2.13) | 116 (3.85) | 199 (7.38) |
| 1D ($R = 30$) | 16 (0.42) | 22 (0.69) | 38 (1.18) | 71 (2.44) | 120 (4.61) | 216 (9.00) |
| 2D ($R = 1$) | 7 (0.16) | 12 (0.29) | 25 (0.59) | 51 (1.28) | 98 (2.46) | 185 (4.87) |
| 2D ($R = 5$) | 4 (0.22) | 6 (0.38) | 12 (0.73) | 26 (1.62) | 48 (3.01) | 86 (5.72) |
| 2D ($R = 10$) | 3 (0.26) | 6 (0.47) | 10 (0.87) | 22 (1.86) | 39 (3.54) | 75 (6.86) |
| 2D ($R = 20$) | 4 (0.39) | 7 (0.67) | 13 (1.22) | 23 (2.58) | 43 (4.82) | 83 (9.27) |
| 2D ($R = 30$) | 4 (0.53) | 8 (0.91) | 14 (1.60) | 26 (3.29) | 49 (6.14) | 93 (11.67) |

Table 7.2: Computation time in seconds to the first appearance of an optimal partition (median of 11 runs) on the 64 vertex multilevel graph problem on a 16K processor Connection Machine-2. The time in seconds per generation is shown in parentheses. † indicates that the median run did not find an optimal solution within 1000 generations.

Across all of the genetic algorithms, increasing the population size causes only slight speed improvements (in terms of number of generations to an optimal solution). In addition, across all selection algorithms we found that an optimal solution is either found within a couple hundred generations, or else the run times out. We never

| $\log_2 N$ | Linear Rank | Stochastic | 1D Local (all $R$ values) | 2D Local (all $R$ values) |
|---|---|---|---|---|
| 13 | 0.45 | 0.27 | 1.0 | 1.0 |
| 14 | 0.55 | 0.36 | 1.0 | 1.0 |
| 15 | 0.64 | 0.64 | 1.0 | 1.0 |
| 16 | 0.82 | 0.55 | 1.0 | 1.0 |
| 17 | 1.0 | 0.55 | 1.0 | 1.0 |
| 18 | 1.0 | 0.27 | 1.0 | 1.0 |
| 19 | 1.0 | 0.73 | 1.0 | 1.0 |
| overall | 0.78 | 0.48 | 1.0 | 1.0 |

Table 7.3: Fraction of runs finding an optimal solution within 1000 generations for the panmictic selection algorithms.

observed a run that first discovered an optimal solution between generation 200 and 1000.

The second speed comparison is based on the actual time required to find an optimal solution (Table 7.2). The run-time measurements reported here are based on implementations in C++/CM++ (Collins, 1990) that differ only in the selection/mate choice code. The data was gathered on an 16K processor Connection Machine-2 equipped with 64K bits of memory per processor and 32-bit floating point accelerators, with a Sun 4/330 front end running SunOS 4.1.1 and Connection Machine software version 6.0.

Although the run time per generation for stochastic selection is less than linear rank selection, stochastic selection still requires more than twice as much time to find optimal solutions. For local mating, although long random walks require significant computation, the fastest evolution occurs when $R$ is in the range $5 < R < 20$. Even with long random walks, the local mating algorithms run significantly faster (per generation) than the panmictic algorithms. The two effects together make the local algorithms optimize *much* faster than the panmictic schemes.

When we compare the fastest (median time to an optimal solution) panmictic algorithm that we implemented (linear rank) to our slowest local mating algorithm (1 dimensional with $R = 1$), we find that local mating is faster by about a factor of 2. When compared to the fastest local mating algorithm (2 dimensional with $R = 10$), linear rank is slower by more than an order of magnitude and stochastic selection is slower by about a factor of 25.

We measure the robustness of the various genetic algorithms in terms of the fraction of runs that find one of the two optimal solutions within 1000 generations. None of the local mating runs, across all population sizes and $R$ values, failed to find an optimal solution. Unlike the local algorithms, the panmictic algorithms are not 100% robust (Table 7.3). Of the two panmictic algorithms, linear rank selection is more robust by this measure and on this problem.

| Algorithm | Parameter Settings | Total Runs | Total Generations | Total Time (seconds) |
|---|---|---|---|---|
| Stochastic | 1 | 88 | 56,000 | 300,200 |
| Linear Rank | 1 | 88 | 27,000 | 179,000 |
| 1D | 5 | 440 | 22,000 | 43,500 |
| 2D | 5 | 440 | 16,000 | 31,600 |
| Total | 12 | 1056 | 121,000 | 554,300 |

Table 7.4: The computational requirements for the **Partition** 64-vertex multilevel graph partitioning studies. The data on the total number of runs, generations, and time are approximate. This study required approximately 18 billion fitness function evaluations. The total run time is nearly 6.5 days of Connection Machine-2 CPU time (16K processors).

### 7.4.5 Implementation Notes

**Partition** requires about 500 lines of code beyond the core library of routines. Most of this code is devoted to building the graph data structures and the scoring function. The balance of the code handles the run-time selection of instrumentation and parameter options.

The computational requirements for the **Partition** multilevel graph studies are summarized in Table 7.4. Note that while the 83.3 percent of the runs use spatial structure, these runs only consumed 13.5 percent of the CPU time. The total time for all runs is nearly 6.5 days on a Connection Machine-2 (16K processors).

### 7.4.6 Discussion

In the **Partition** experiments we have presented in this section, the panmictic selection algorithms become focused towards one of the two optimal solutions in the very early generations, and eventually converge on or near that solution. In fact, we have not observed a single run in which a panmictic selection algorithm discovered both optimal solutions. In sharp contrast, the local mating algorithms consistently discover both optimal solutions.

The allele diversity data (Figure 7.3) demonstrates quite dramatically how panmictic selection converges toward only one of the two optimal solutions, while local selection contains a nearly equal mix of both. (Remember that the two optimal solutions are bitwise complements of each other.)

Local selection also maintains much greater genotype diversity than panmictic selection (Figure 7.4). What we expect to see is a cloud of mutants surrounding (in the adaptive landscape) an optimal solution. Because local mating finds and maintains both optimal solutions but panmictic mating finds only one, we would expect local mating to exhibit more genotypic diversity than panmictic selection. This is clearly the case—local mating produces four times as much genotypic diversity than stochastic selection and ten times as much as linear rank selection. This difference is important, because it demonstrates that local selection explores many more genotypes in each generation.

The panmictic index results (Figure 7.5) are quite interesting, because they show two different sources of decreased heterozygosity. As expected, local mating is characterized by a low panmictic index throughout the experiment. The decreased heterozygosity is a result of local fixation (convergence) on a particular genotype within each deme due to selection pressure and genetic drift. The fixation is local—diversity is maintained because each locality fixes on a different genotype.

On the other hand, panmictic selection results in almost complete panmixia ($P = 1$) until the population begins to converge on a particular genotype. When this occurs, we start to observe excess homozygosity, which is due to selection for the high–fitness genotype. Linear rank selection shows significant excess homozygosity ($P \ll 1$), indicating that very few sub–optimal genotypes are chosen as parents for the next generation. In contrast, stochastic selection shows a much higher panmictic index. This indicates that the stochastic selection algorithm allows sub–optimal genotypes to be sampled with a higher frequency (which is consistent with the greater allele and genotype diversity).

The diversity and panmictic index data demonstrate the dramatic dynamic differences between local and panmictic mating. It is clear that local mating maintains a broad genetic search for thousands of generations. If the adaptive landscape or fitness function were changing over time, this diversity would allow the population to discover and exploit the genotypes with higher relative fitness values, even if earlier those genotypes had relatively low fitness.

The data on the speed of evolution shows that the dynamics of local mating result in a faster and more robust genetic algorithm, at least for this particular problem. Local mating is characterized by faster evolution both in terms of the number of fitness function evaluations and machine time required to discover an optimal solution. The robustness of local mating is rather surprising: across all population sizes, both 1 and 2 dimensional geometries, and all $R$ values, local mating *never* failed to find an optimal solution, while both of the panmictic algorithms had a significant fraction (25 to 50 percent) of their runs "time out" (no optimal solution found within 1000 generations).

The dynamics of evolution in a spatially structured population are fascinating to watch. In the initial generations, the population is made up of nearly uniformly mediocre graph partitions. As relatively high–scoring partitions are discovered, they spread, and soon that genotype dominates the local population. The result is local convergence, but global diversity; each locality converges on a different genotype.

As these successful genotypes spread through the population, they encounter other relatively successful genotypes. If one genotype is more fit than another, it will continue to march through the less fit area of the world. When two genotypes of similar fitness but encoding radically different partitions collide, a stable boundary forms between the two subpopulations. This boundary consists of low–scoring hybrid individuals that are the result of unfortunate recombinations between the two competing genotypes. In nature, these sorts of boundaries are known as ***hybrid bands***.

As you might expect, there is almost no evolution (genetic change) within the genetically homogeneous regions—almost all the genetic diversity and evolutionary innovation occurs in the hybrid bands. Almost every new, higher scoring genetic

combination that is produced first appears in a hybrid band, whether the band is on the leading edge of a spreading subpopulation or a relatively stable boundary between two demes of similar fitness. As we noted above, every run with a large, locally mating population results in a nearly even mixture of both optimal solutions. The demes form genetically homogeneous regions that are separated by hybrid bands that are stable for thousands of generations (Figure 7.6).

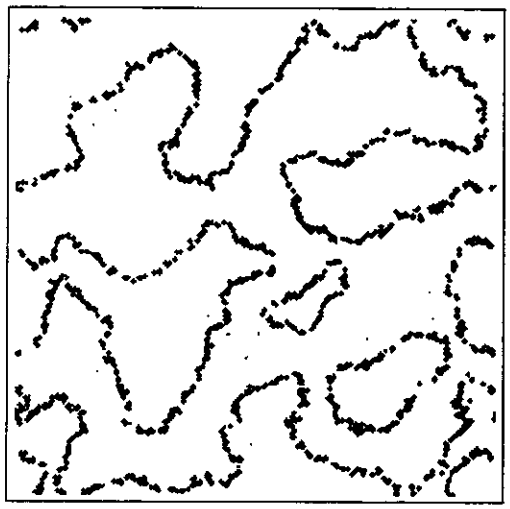

Figure 7.6: The geographical distribution of fitness scores in generation 150 a run using the 2 dimensional local mating algorithm ($R = 1$) with a population size of $2^{16} = 65,536$ on the 64–vertex multilevel graph partitioning problem. Individuals scoring less than -3 are represented by black pixels. Note that this is a toroidal population.

Naturally, the length of the random walk ($R$) affects the formation of homogeneous subpopulations and hybrid bands. With a small $R$, fit genotypes spread slowly, resulting in a large number of small demes. The hybrid bands are narrow, because the two demes are not within $R$ steps of very many individuals. With relatively longer walks (greater $R$), fewer and larger homogeneous subpopulations are maintained, and these are separated by wider hybrid bands.

## 7.5 Robustness of Spatial Structure

In the previous section, we examined a number of random walk lengths for the selection algorithms that use local mating. We found that for the 64–vertex multilevel graph partitioning problem, longer random walks (within the limits of our parameter study) always result in faster (in generations) optimization. Yet global (panmictic) selection is not very robust, so presumably there is some best walk length where the genetic algorithm is still robust, but requires the fewest generations to find an optimal solution. To explore this phenomena, we need an optimization problem that is scalable to great enough difficulty that our local mating genetic algorithm will begin to fail occasionally.

What kinds of problems are difficult for genetic algorithms? As we have seen, traditional genetic algorithms start to have trouble with problems that require a re-

liance on recombination, such as the 64–vertex multilevel graph. This problem is hard because convergence must be avoided until the global optimum has been discovered. But it is easy for genetic algorithms employing spatial structure; they discover optimal partitions quickly and reliably (eventually finding both solutions in every run we have examined) across a variety of parameter settings. The spatial structure aids in avoiding premature convergence. However, we expect that on problems of this type:

- very short walks require more generations to find an optimal solution than walks of medium length;
- very long walks result in premature convergence and fail to ever find an optimal solution;
- larger populations can tolerate longer random walks and still remain robust.

To test these hypotheses, we need a scalable problem, but the multilevel graph problem does not scale well.

Ackley (1987) scales the multilevel graph problem from 32 to 64 vertices, but in doing so has changed the character of the adaptive landscape. Each connected component of the 32–vertex graph consists of four 4–vertex clumps connected in a ring; each clump is connected to two others. When he constructs the connected components of the 64–vertex graph, eight 4–vertex clumps are connected in a cube; each clump is connected to *three* others. If we continue scaling the problem in this manner, in the 128–vertex multilevel graph, each clump is connected to *four* others; in the 256–vertex graph, each clump is connected to *five* others; etc. In the small multilevel graphs most of the edges are concentrated in the clumps, while larger graphs contain an ever increasing proportion of interclump edges. This means that as we scale up the problem size, the proportion of nonlocal (in terms of chromosomal distance) epistasis (fitness–altering interactions between genes) increases. In our experience, the epistatic characteristics of a problem has a significant impact on the efficacy of a given genetic algorithm. This means that the multilevel graph partitioning problem is not a very good testbed for robust genetic algorithms, because each different size problem is also qualitatively different. Another practical problem is that the multilevel graphs all must have a size that is a power of two, which means that relatively small increments in problem difficulty are not possible.

### 7.5.1 Clumpy Rings: A Scalable Graph Partitioning Problem

In this section, we introduce a new, scalable graph design for partitioning problems that is based on *clumpy rings*, and use it as a testbed for the genetic algorithms described above. The clumpy ring graph consists of two connected components. Each connected component is made up of a ring of clumps, and each vertex is fully connected to all other vertices in its clump (see Figure 7.7). The three parameters that describe a clumpy ring graph of vertices $V$ are $c$, the number of vertices in each clump; $r$, the number of rings (connected components); and $k$, the number of clumps

in each ring. Not only do we constrain $c$, $r$, and $k$ to be integers, but we also require $r$ to be even and $c > 2$, $r > 1$, and $k > 1$.

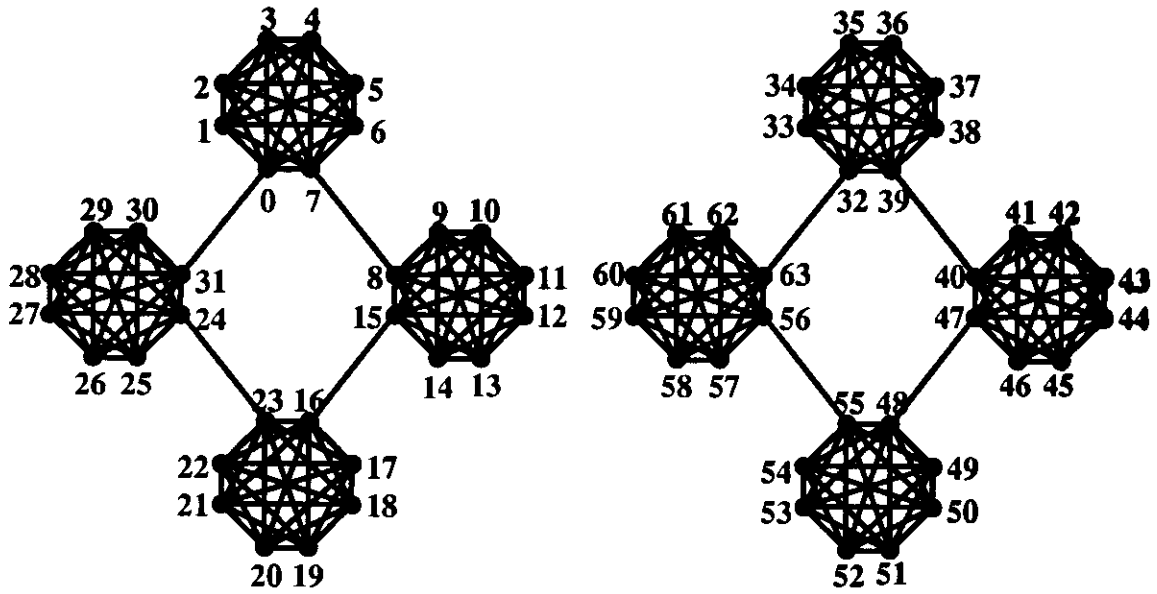


Figure 7.7: The 64-vertex clumpy ring graph. The number associated with each vertex indicates the locus (bit position) in the chromosome that specifies on which side of the partition it lies.

The size of the graph is the product of these parameters: $|V| = ckr$. In general, the size of the problem is scaled by modifying $k$ while $c$ and $r$ are held constant. This scales the size of the graph in increments of $r$ clumps ($cr$ vertices), by adding one clump to each of the rings. As we increase the size of the graph, each new clump adds exactly $c(c-1)/2$ intraclump edges and exactly one interclump edge, so the adaptive landscape remains qualitatively the same. In the empirical study that we describe below, we use $c = 8$, rather than the clumps of four of the multilevel graphs, and $r = 2$. We use larger clumps to make the problem more difficult. The large ratio of intraclump to interclump connections (in this case 14:1) makes it very unlikely that a single-bit mutation can improve the fitness of a partition (see the discussion in Section 7.4).

### 7.5.2 Empirical Studies

The obvious way to test our hypotheses concerning the relationship between population size, problem difficulty, and the random walk length is with a large parametric study. Instead of a full parametric study, we have used **Partition** to explore a portion of the parameter space to determine the limits of the robustness of the genetic algorithm with respect to the three parameter dimensions. (A full parametric study is probably not worth the months of Connection Machine time that would be required.)

The **Partition** genetic algorithm is not changed, except we will now be using clumpy ring graphs. The other relevant parameters (which are held constant) are

- recombination rate: $\rho = 1.0/l$ per bit, where $l$ is the chromosome length (vertices in the graph);
- mutation rate: $\mu = 0.0$ per bit.

Since we are studying the ability of the genetic algorithm to exploit recombination, so we have eliminated mutation entirely. This reduces the number of free parameters to study. In any case, the clump–size is so large that a mutation is unlikely to ever result in improved fitness.

For present purposes we define the genetic algorithm to be *robust* for a given set of parameters if the median run of 21 replications finds at least one optimal solution by generation 5000. The *limit of robustness* along a given parameter dimension is the first set of parameters for which the genetic algorithm is not robust (i.e. the point where more than half of the runs fail). As we stated above, we want to explore the limit of robustness as the difficulty of the problem increases (for partitioning clumpy ring graphs, larger graphs are more difficult). For each population size and random walk length, we increase the size of the clumpy ring graph partitioning problem until the limit of robustness is discovered.

We expect that for a given value of $R$, larger populations will be able to handle larger, more difficult problems. Larger populations inherently contain greater searching power (more individuals per generation), more resistance to convergence due to selection (longer distances for genes to flow to take over the whole population), and more resistance to convergence due to drift.

For a given population size, smaller values of $R$ (shorter random walks) are expected to lead to a more robust genetic algorithm than larger values of $R$. This prediction is based on the hypothesis that slower gene flow leads to more robust evolution. Implicit in this prediction is the assumption that our time limit of 5000 generations is sufficient to allow any run that is actively approaching an optimal solution to do so. For example, a small $R$ leads to slow gene flow, which tends to slows down the rate of evolution (and thus the movement towards convergence), which we demonstrated empirically above. If we had put a time limit of 100 generations on the runs in Section 7.4, all of the 1–dimensional, $R = 1$ median runs would have timed out, although the evolution clearly was moving towards discovery of the optimal solutions. A good way to check for a too stringent time limit is to examine the runs with the same parameters but the next smaller graph. If the successful runs for the smaller problem just barely finish before time runs out, then it is likely that the limit is too low. This is where it is important to have a scalable optimization problems that allows relatively small difficulty increments. By this measure, 5000 generations appears to be sufficient for the range of parameters of this study.

Figure 7.8 summarizes the data for the 2–dimensional selection algorithm. The data is consistent with our predictions. Within each curve (constant $R$, the length of the random walks), the limit of robustness increases with population size. Between curves (constant $N$, the population size), the limit of robustness decreases with increasing $R$.

At the limit of robustness, why is local mating failing to discover an optimal solution? With traditional panmictic genetic algorithms, failure is almost always

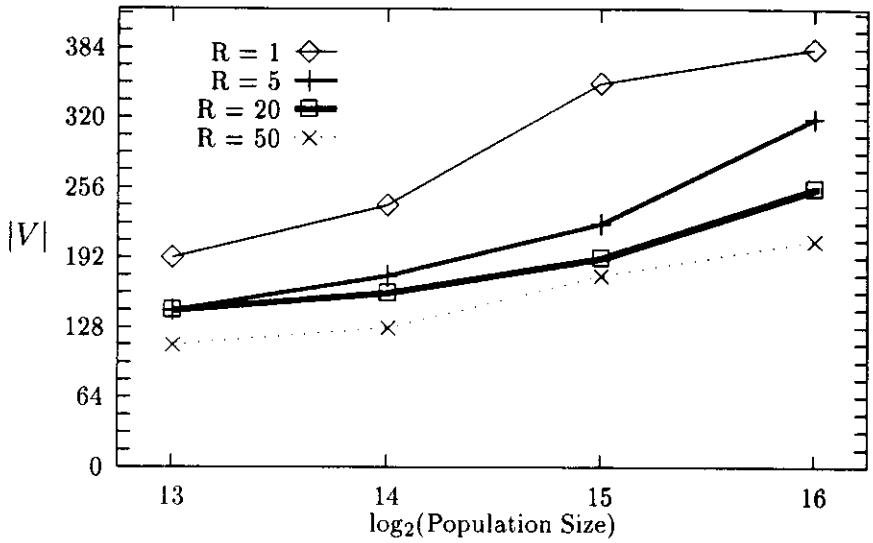


Figure 7.8: The limit of robustness for 2-dimensional local mating for the various population sizes and values of $R$, the length of the random walks.

| Total Runs | Total Generations | Total Partitions | Total Time (seconds) |
|---|---|---|---|
| 1757 | 2,383,961 | 157,101,113,344 | 2,047,600 |

Table 7.5: The computational requirements for the **Partition** clumpy ring graph partitioning studies. "Total Partitions" refers to the number of fitness function evaluations. The data on the total time are approximate. The total run time is nearly 24 days of Connection Machine-2 CPU time.

because of convergence. With local mating, convergence is not always the culprit (Figure 7.9); we typically observe a high degree of variation (both in terms of alleles and genotypes) in the population throughout the low-$R$ runs that fail. The allele variation data in Figure 7.9 shows that a significant amount of variation is retained when $R = 1$, but the population converges almost immediately when $R = 50$.

### 7.5.3 Implementation Notes

The addition of the clumpy ring graphs to **Partition** required only about 80 additional lines of code. Despite our best efforts to minimize the amount of computation, the computational requirements for the clumpy ring graph studies were severe (Table 7.5). Due to the difficulty of the optimization problem, we set the time-out generation at 5000, requiring a large number of generations to pin down the limit of robustness. This portion of the **Peacock** simulations evaluated more than 150 billion partitions, requiring nearly 24 days of 16K processor Connection Machine-2 time.

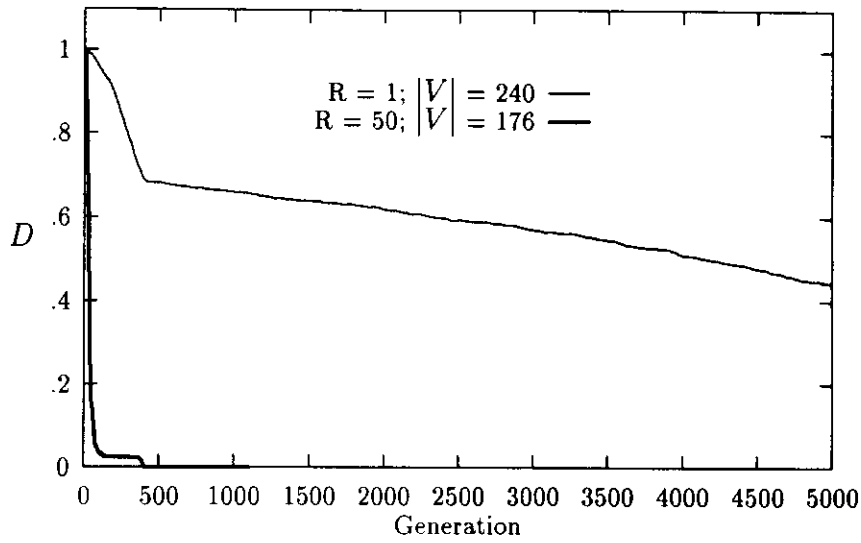


Figure 7.9: The diversity of alleles $D$ maintained by the 2 dimensional local selection algorithm. $N = 2^{14} = 16,384$ and the task is the clumpy ring graph. The size of the graph is chosen to be 48 vertices larger than the limit of robustness. Note that none of these runs discovered either of the optimal solutions. Each curve is the average of 7 runs.

## 7.6 Discussion

Artificial evolution simulations operate on large populations in complex and changing ecosystems. The adaptive landscapes are generally enormous (hundreds or thousands of orders of magnitude larger than the population size) and constantly changing. Artificial evolution requires a genetic algorithm that is resistant to convergence and can simultaneously explore different parts of the adaptive landscape. Spatial structure (local mating) provides domain-independent convergence resistance.

Local mating is resistant to convergence, because each deme can explore different peaks in the adaptive landscape. Small demes allow more effective exploration of the adaptive landscape. Using **Partition**, we have demonstrated the dramatic differences in the evolutionary dynamics between spatially structured and panmictic populations. Local mating results in a faster and more robust genetic algorithm.

The fact that genetic algorithms that use local mating can be significantly faster and more robust than traditional genetic algorithms is an important result, and suggests that further investigation is in order. While important theoretical results have been developed for panmictic mating algorithms, it is not clear that any of these results can be applied directly to local mating. In addition, we have only examined one very simple local mating algorithm, and it is almost certainly not the best that can be found.

The studies in this chapter also point out one of the problems of parametric studies: incredible computational requirements. The empirical data in this chapter required the generation and testing of approximately 200 billion graph partitions,

which took more than a month of CPU time on the UCLA 16K processor Connection Machine-2. Parametric studies should be performed only when absolutely necessary.

# Chapter 8

# Contributions, Conclusions, and Future Work

## 8.1 Contributions and Conclusions

The main theme that runs throughout this dissertation is low-level, bottom-up simulation of evolution. Our basic approach is to model in detail each individual, its genetics, and the sexual recombination and mutation of the genetic material as it is passed on to the next generation. By manipulating the genetics in ways similar to the way natural genomes are manipulated, biologically realistic evolutionary dynamics emerge at the level of the population. We have applied this simulation methodology to three very different areas: the study of natural evolution, the evolution of complex behaviors in artificial life forms, and the application of evolution to optimization tasks.

In the study of natural evolution, we presented two simulations: **Peacock** and **Parasite**. The **Peacock** studies simulate variations on an analytic population genetics model: Kirkpatrick's (1982) model of sexual selection and female choice. **Peacock** models the well-known situation where females have mating preferences for maladaptive male traits: the females of the species prefer to mate with males that possess a secondary sexual characteristic that is so exaggerated that it reduces the male's ability to survive to adulthood. A typical example is the peacock, where the female's preferences have resulted in the evolution of extremely long and colorful tail feathers in the males. The male's long tail makes him more likely to be discovered and captured by predators, because it increases his visibility and decreases his mobility. Until ten years ago, the evolution and maintenance of preferences for maladaptive traits had been a well-known, but completely open problem in biology for more than 100 years. It has become a hot topic recently.

In formulating his analytic model, Kirkpatrick was forced to make many simplifying assumptions to make the mathematics tractable. The assumptions include an infinite population, panmictic mating, haploid genetics, no mutation, etc. Based on these assumptions, the equilibrium allele frequencies form a curve. With **Peacock**, we have simulated an approximation of this model with large, finite populations ($N = 131,072$ individuals per generation) with a low mutation rate ($\mu = 0.00001$

per locus). After 500 generations of evolution, all of our simulated populations lie very near to the equilibrium predicted by Kirkpatrick, providing empirical verification of his analysis. We then proceed to simulate other variations on his model: a stepping–stone model of spatial structure and diploid genetics. The equilibrium for the diploid, panmictic population matches the haploid equilibrium (when viewed in terms of phenotype frequencies). However, spatial structure changes the equilibrium states from a curve to a set of points that are bound by a region.

We also simulated the non–equilibrium case of the invasion of the male trait (e.g. long tail allele) into a population of short–tailed males, but with enough females with a genetic predisposition to long–tailed males that there will be strong selection for the mutant, long–tailed males. In the haploid/panmictic model, the invasion is rapid, and the population moves quickly to the equilibrium. In the haploid/spatial structure model, the invasion is also fairly quick, but spatial structure slows down the rate of invasion by about a factor of two. In the diploid/panmictic model, with the trait (long tail) and the preference for the trait alleles recessive, the invasion is very slow, and often does not occur for thousands of generations. In the diploid/spatial structure model (again with long tails and the preference for long tails recessive), the invasion is slow, but much faster than the panmictic case. This not only demonstrates that the simplifying assumptions have dramatic effects on the non–equilibrium dynamics of the system, but also that the variations on the model can interact. In this example, spatial structure slows down the invasion in a haploid population, but dramatically speeds it up in a diploid population.

Our other natural evolution simulation is **Parasite**, which simulates multiple, interacting species. The problem of sex, the origin and maintenance of sexual reproduction, is one of the biggest open questions in evolutionary biology. The problem is that asexual reproduction is much more efficient than sexual reproduction (by as much as a factor of two). What advantages does sexual reproduction convey that are enough to offset the costs of sex? **Parasite** models the parasite hypothesis: that host–parasite coevolution may favor the maintenance of sexual reproduction in the host species.

The parasite hypothesis is one of many that have been proposed. With **Parasite**, we have obtained empirical evidence that parasites can cause selection for higher recombination rates in the host species. The higher recombination rate in the host translates into greater mixis, which is the main effect of sexual reproduction. Also, for the range of parameters that we studied, faster parasite evolution results in higher equilibrium recombination rates in the host. We manipulate the rate of parasite evolution by adjusting the parasite recombination rate and/or the number of parasite generations per host generations (the parasites reproduce more frequently than the hosts).

Together, **Peacock** and **Parasite** demonstrate that low–level simulations of large, evolving populations can be used to study problems in natural evolution. These studies were completed using only a total of 16 days on a 16K processor Connection Machine-2.

The second application area that we study is the evolution of behavior in artificial organisms that live in a complex, shared environment. **AntFarm** simulates the

evolution of foraging behavior in colonies of ant–like artificial organisms. Simulations of this sort have been successful in the past, but the organisms were rather simple. **AntFarm** attempts to scale up the size and complexity of the environment and organisms. The **AntFarm** organisms have dozens of sensory inputs and outputs, and efficient foraging behavior is quite complex.

The central problem in the evolution of the behavior of artificial organisms is the representation of the behavior function both as an executable program and as a bit-string chromosome on which evolution can operate effectively. Other researchers have evolved simple programs using similar evolutionary techniques, using a wide variety of organism representations. These have included the successful use of parameterized functions, Lisp S–expressions, finite state automata, fully connected neural networks, rule systems, etc. Unfortunately, none of these representations scale to the environment/organism complexity that is necessary for **AntFarm**.

As part of the **AntFarm** simulation, we have developed the symmetric connection descriptor ANN encoding scheme, which empirically is capable of supporting the evolution of ant–like foraging behaviors in the **AntFarm** world. The connection descriptor encoding has a number of attractive properties by placing both the placement and strengths of the connections under genetic control. This encoding scheme decouples the number of connections from the number of neural units, and allows unrestricted connectivity patterns. Also, connection descriptors are position independent, having the same effect wherever they reside in the chromosome. The connection descriptor encoding scheme has been used successfully in other studies involving the evolution of complex behavior (Wieland, 1991a; Wieland, 1991b). We have also found that the artificial morphology (sensor and effector design and layout) of the organisms can have a large impact on the behaviors that evolve.

We have identified six properties that an artificial organism representation should possess: (1) syntactic closure of the genotype under the genetic operators, (2) smoothness of the behavior function under the genetic operators, (3) scalability to a large number of inputs and outputs, (4) symmetric behaviors, (5) the ability to evolve both continuous and discrete behaviors, and (6) a computational model that is uniform. The only representation that we have found that possesses all of these properties is the symmetric connection descriptor ANN with input suppression.

Why was it necessary to invent this new ANN encoding? Much of the literature in the area of ANNs describes the programming of ANNs via various supervised learning techniques involving the backpropogation of errors in the output vector ("backprop"). The use of supervised learning requires a training set which consists of input/output pairs that describe the correct output for each of a representative sample of inputs. Backprop is usually successfully applied to ANN architectures that consist of multiple layers of neural units that are fully forward connected (no cycles or lateral connections). Backprop incrementally improves the performance of the ANN by using the detailed error information provided by the training set to incrementally update the strength of each connection toward an appropriate value. Early attempts to apply evolution to the task of programming ANNs used these fully–connected architectures which are so successful for backprop. In general, these attempts involved toy problems, and were quite successful. But when we attempted to scale this technique to

the complex **AntFarm** problem, we were unsuccessful. Apparently, it is difficult to program fully connected networks to calculate complex functions like the **AntFarm** foraging algorithm given only the sparse feedback and random weight update that is characteristic of the process of evolution. Where backprop simultaneously updates *every* weight in the ANN in the *appropriate* direction, evolution can only modify a *small number* of weights in any given generation and the changes are made in *random* directions. The lack of detailed error information and the sparse update procedure that characterizes evolutionary programming of ANNs makes it unlikely that a viable evolutionary path exists from an initial population of random, fully connected ANNs to an ANN that performs well on the task at hand. The connection descriptor encoding method appears to be much more amenable to the programming of ANNs via evolution.

While it is clear that many artificial life applications require the ability to program ANNs with evolution, this technique of evolving ANNs goes beyond the artificial life domain. As we noted above, backprop requires a detailed and representative training set. Therefore, backprop can only be used on problems where we can calculate the correct response for a given input. It cannot be applied to problems where we do not know the best output for each input, nor to problems that require a sequence of outputs before the quality of the outputs can be determined (delayed feedback). There is thus a large class of problems for which supervised training cannot be used, simply because we cannot generate the detailed error information that backprop requires. On the other hand, evolution does not require detailed error information; it only requires us to provide a relative ranking of the individuals in a population after many invocations of each ANN.

In the **AntFarm** simulations, we also examined variable–length chromosomes. We have evolved successful behaviors, but have not observed selection for either longer or shorter chromosomes. This suggests that the initial size of the chromosome was sufficient to support the evolution of ant–like behaviors. Further study will be required before we can draw any conclusions about variable–length chromosomes.

Our **Partition** simulation attempts to discover optimal graph partitions via a genetic algorithm. We (and others) have had success with genetic algorithms that use spatial structure (locality) in the process of selection and mating. Sewall Wright's (1931) shifting balance theory of evolution suggests that spatial structure in populations should lead to faster and more robust evolution. **Partition** applies Wright's ideas to an optimization problem, and we find that spatial structure is very beneficial. Spatial structure speeds optimization and provides a domain–independent method of convergence avoidance.

We have used **Partition** to perform a head–to–head comparison of panmixia and spatial structure on Ackley's (1987) multilevel graph partitioning problem. We used several metrics to quantify the dynamic differences and optimization success of the different selection schemes, including diversity of alleles, diversity of genotypes, the panmictic index, speed, and robustness. We compared two panmictic selection algorithms (stochastic selection with replacement and linear rank selection) and 1 and 2 dimensional local mating, varying the population size, from 8,192 to 524,288 individuals per generation.

When we examine the *diversity of alleles* through time, both local algorithms maintain nearly perfect diversity, while both panmictic algorithms lose most of the population's diversity after about 200 generations. Of the two local methods, the 2 dimensional algorithm loses diversity faster. Linear rank selection loses diversity of alleles sooner than stochastic selection with replacement, and stabilizes at a lower level.

We also track the *diversity of genotypes*, which is a measure of the breadth of the genetic search (number of unique genotypes present in the population). Both local algorithms examine four times as many 10–bit genotype samples each generation as stochastic selection with replacement, and an order of magnitude more than linear rank selection.

The *panmictic index* (borrowed from biology) measures the degree of panmixia (random mating) in the population. Both local mating algorithms are characterized by a high degree of inbreeding (low panmictic index). During early generations, both panmictic algorithms show a high degree of panmixia, but once convergence occurs linear rank selection is characterized by a low panmictic index (non–random mating). Stochastic selection with replacement maintains a fairly high panmictic index even after convergence.

We measure the *speed of optimization* in two ways: (1) by counting the number of generations required to find an optimal solution (implementation independent) and (2) by run–time required to find an optimal solution (implementation dependent). Our data show that across all population sizes that we examined, both spatially structure genetic algorithms beat the panmictic algorithms on both speed measures. In terms of the number of generations to an optimal solution, linear rank is significantly faster than stochastic selection with replacement, and it is faster by a factor of two when our Connection Machine-2 implementation is considered. At most of the parameter settings (degree of locality), the local mating algorithms require half as many generations and are five times faster (real time) to an optimal solution.

We also measure the *robustness* (success rate) of the genetic algorithms. Overall, only 48 percent of the stochastic selection with replacement runs found one of the two optimal solutions, while 78 percent of the linear rank selection runs were successful. On the other hand, the local selection algorithms were 100 percent successful. Not only did the local methods always find an optimal solution, every run we examined found ***both*** optimal solutions. We have not observed any panmictic run that discovered both solutions.

Our local selection/mating algorithms are parameterized by $R$, which adjusts the degree of locality (deme size): a small $R$ results in a small neighborhood of competition and mating, while a larger $R$ results in a larger neighborhood. Our results suggest that smaller neighborhoods result in slower evolution. Because panmixia simulates an infinite–sized neighborhood, presumably large neighborhoods are less robust. Therefore, $R$ probably trades off evolution speed and robustness, with small $R$ being slow and robust, while larger $R$ is faster, but more prone to convergence. To test this hypothesis, we developed the *clumpy ring* graph partitioning problem, which is scalable in small increments of problem difficulty. We scale the clumpy ring graph problem until it is hard enough to reach the *limit of robustness* for the 2 dimensional

local selection algorithm (for a given $R$ value). The results of this study indicate that a small $R$ is more robust, and can successfully search larger, more difficult adaptive landscapes. Also, as population size increases, more difficult problems can be robustly solved with a given $R$ value.

## 8.2 Future Work

This dissertation is among the first in the area of artificial life, and apparently the first to address the problem of realistic artificial evolution. As such, it is appropriate to address a broad spectrum of issues and problems, though resolve none of them fully. In this section, we indicate the directions for future research that appear to the most important and potentially fruitful.

### 8.2.1 Studying Natural Evolution

We have demonstrated that artificial evolution can be used to study macroevolutionary phenomena. While our examples have produced interesting results, they are so far just scratching the surface of what is possible. It is time to get these techniques into the hands of population geneticists and evolutionary biologists, who are qualified to apply them to the most pressing problems in their fields. Unfortunately, these studies are so computationally intensive that (in today's technology) supercomputers are required. So far, few biologists have the computational knowledge and experience required to implement simulations of their models on a massively parallel computer. An important step towards providing this technology to biologists will be the development of an artificial evolution toolkit.

This toolkit should allow the biologist to mix and match models of various genetic systems (haploid, diploid, haplodiploid, multiple chromosomes, sex chromosomes etc.), sexual systems (asexual, sexual, self fertilization, single sex, multiple sex, hermaphrodites, etc.), selection and mating systems (truncation selection, frequency dependent selection, sexual selection, etc.), spatial structure (isolation by distance, stepping stone, island, hybrid models, etc.), age structure (overlapping generations), life histories (stages of development), etc. By providing the basic building blocks, we will free biologists from the need to program much beyond the scope of their specific models. A critical part of the toolkit will be the instrumentation tools that will allow the biologist to determine what is going on in an evolving population.

In terms of our specific simulations in the area of natural evolution, we have plans for several extensions. An obvious extension to **Peacock** is to implement a model of sexual selection based on diploid genetics with polygenic (multiple loci) inheritance, with multiple alleles at each locus. With this more realistic model, we could more graphically demonstrate Fisher's runaway selection. We will also use this extended model to test the hypothesis that runaway sexual selection can cause reproductive isolation (and thus the potential for speciation) even in the absence of geographic barriers.

The **Parasite** model of the parasite hypothesis is quite simple, and with modifications can lead to more general conclusions. The first extension we plan is to include a number of loci in the host organisms that are not involved in competition with the parasites. This would make the host organisms more realistic, because there would be other components of fitness than the host-parasite interactions. We could then explore what portion of the host's fitness must be determined by interactions with parasites before we observe selection for higher recombination rates.

### 8.2.2 Evolving Artificial Organisms

In this dissertation, we have made progress towards evolvable artificial organism representations, and in particular evolvable ANNs. The connection descriptor ANN encoding that we have developed is almost certainly not the best possible method. To improve on it, we will have to develop a suite of complex tasks (such as **AntFarm**) to act as a testbed for empirical studies. In applying the evolution of ANNs in an engineering domain, the important problems appear to revolve around the ability to lead the evolution from ANNs with the ability to solve a simple problem through progressively more complex situations until the ANNs have been programmed to handle the target task. This will require progress in a number of areas. For instance, we will need heuristics or a theory for designing the trajectory of ever more complex fitness functions. In addition, we will need an ANN encoding scheme that places the complexity of the ANN architecture under genetic control, so complexity can be evolved as needed.

Another avenue for research is towards more complex development functions (which translate the bitstring genetic encoding into the ANN), especially towards development functions that are mostly under genetic control. This corresponds to ontogeny in natural organisms. This will allow complex ANNs to be encoded in a relatively small genome.

While the **AntFarm** model is complex enough to evolve a wide variety of behaviors, there are a number of extensions that would make the evolve behaviors look more like real ants. For instance, currently, at the beginning of the generation, all of the ants leave the nest simultaneously. In nature, the workers trickle out of the nest, and in many cases the initial foragers have already returned with food before the majority of the workers begin foraging. This might be an advantage, if the successful foragers leave a trail leading to a large patch of food. In this way, the ants that leave the nest later are able to immediately begin transporting food, without having to search for either food or a pheromone trail.

Another area that we should address is the modeling of the pheromones and food in the environment. We should at least go back to the **AntFarm I** model of food and pheromones. In **AntFarm I**, the environment kept an actual count of the number of units of food and pheromone in each environment location. We should allow the ants to sense the relative amounts of food and pheromones. This would not only make the ant's senses more realistic, but also allow more realistic pheromone diffusion. In the later **AntFarm** simulations, we fake pheromone diffusion probabilistically (in an effort to reduce memory and computational requirements).

### 8.2.3 Evolution for Optimization

In the area of parallel genetic algorithms, we have presented an example where local mating in large populations leads to a very robust and fast genetic algorithm that is significantly better than simply scaling up traditional techniques to large populations. While this result is significant, this study tested only one local mating strategy on one type of optimization problem. This area is ripe for more empirical and theoretical investigations aimed at understanding the operation of genetic algorithms and improving these optimization techniques. We have introduced several metrics for quantifying various aspects of the dynamics of genetic algorithms, some of which were borrowed directly from the biological literature. The development of additional tools for studying massively parallel genetic algorithm dynamics is likely to be very important. Note that many of these tools will be similar to those that are needed in the artificial evolution toolkit that we describe above.

# Acknowledgments

First and foremost, I must thank **David Jefferson** for getting me involved in artificial life, and getting me through this dissertation. David also did his best to teach me how to write well. I hope some of it sticks with me. The rest of my committee, **Mike Dyer**, **Andrew Kahng**, **Chuck Taylor**, and **Bill Schopf**, were also very helpful and contributed greatly to this work. Chuck kept me straight on the biology, and Mike made sure I wrote the right dissertation. Mike also made sure that my workstation kept working.

Without the continuing upbeat support of **Valerie Aylett**, I probably would never have been able to stick with this dissertation, especially through the months when the ants were stupid. Valerie, **Peter Trajmar**, **Mark Cooper**, and **Leslie Phillipsen** helped by making me stop working and play bridge.

**Joe Pemberton** helped me with all my stupid math questions, and bothered me at all the right times (although I could have done without some of the really bad jokes). **Greg Werner** and **Alexis Wieland** gave me a lot of good input and ideas about neural networks. **David Wells** and **Chris McRae** did a great job of keeping the Connection Machine and front ends up and running. **Curt Powley** and the rest of the UCLA Connection Machine users were kind enough to allow me to consume an enormous amount of time on the machine, without complaining *too* much.

I owe thanks to many others who have contributed ideas and comments to this dissertation, including **Chris Langton**, **Doyne Farmer**, **Don Feener**, **Liane Gabora**, **Danny Hillis**, **Adam King**, **John Lighton**, **Ernst Mayr**, and **John McInerney**. Comments and suggestions from the members of the Center for the Study of Evolution and the Origin of Life (**CSEOL**) were greatly appreciated, as was the financial support. Thanks also go to **Russell Leighton** and The MITRE Corporation for use of the Aspirin/MIGRAINES neural network simulation package, which was used for all supervised learning described in this dissertation. **Rich Wales'** LaTeX style files were also greatly appreciated.

This work was supported in part by W. M. Keck Foundation grant number W880615, University of California Los Alamos National Laboratory award number CNLS/89–427, and University of California Los Alamos National Laboratory award number UC–90–4–A–88. The empirical data was gathered in part on a Connection Machine-2 computer at UCLA under the auspices of National Science Foundation Biological Facilities grant numbers BBS8714206 and DIR9024251, and a Connection Machine-2 computer at the Advanced Computing Laboratory of Los Alamos National

Laboratory under the auspices of the U.S. Department of Energy, contract W-7405-ENG-36.

# References


Ackley, David H. (1987). *Stochastic Iterated Genetic Hillclimbing.* PhD thesis, Carnegie Mellon Univeristy.

Anderson, R. M. and R. M. May (1982). Coevolution of hosts and parasites. *Parasitology*, 85:411–426.

Barrett, J. A. (1983). Plant–fungus sympioses. In Futuyama, D. J. and M. Slatkin, editors, *Coevolution.* Sinauer Associates Inc.

Barrett, J. A. (1985). The gene–for–gene hypothesis: Parable or paradigm. In Rollinson, D. and R. M. Anderson, editors, *Ecology and Genetis of Host–Parasite Interactions.* Academic Press.

Belew, Richard K. and Lashon B. Booker, editors (1991). *Proceedings of the Fourth International Conference on Genetic Algorithms.* Morgan Kaufmann.

Bell, Graham (1982). *The Masterpiece of Nature: The Evolution and Genetics of Sexuality.* University of California Press.

Bremermann, H. J. (1980). Sex and polymorphism as strategies in host–pathogen interactions. *Journal of Theoretical Biology*, 87:671–702.

Bremermann, H. J. and J. Pickering (1983). A game–theoretical model of parasite virulence. *Journal of Theoretical Biology*, 100:411–426.

Brooks, Lisa D. (1987). The evolution of recombination rates. In Michod, Richard E. and Bruce R. Levin, editors, *The Evolution of Sex: An Examination of Current Ideas*, pages 87–105. Sinauer Associates Inc.

Brown, A. H. D. and M. T. Clegg (1983). Analysis of variation in related DNA sequences. In Weir, B., editor, *Statistical Analysis of DNA Sequence Data*, pages 107–132. Marcel Dekker, New York.

Cohoon, J. P., W. N. Martin, and D. S. Richards (1991). A multi–population genetic algorithm for solving the k–partition problem on hyper–cubes. In Belew, Richard K. and Lashon B. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 244–248. Morgan Kaufmann.

Collins, Robert J. (1990). CM++: A C++ interface to the Connection Machine. In *Proceedings of the Symposium on Object Oriented Programming Emphasizing Practical Applications.* Marist College.

Collins, Robert J. and David R. Jefferson (1991a). AntFarm: Towards simulated evolution. In Langton, Christopher G., Charles Taylor, J. Doyne Farmer, and Steen Rasmussen, editors, *Artificial Life II*, volume 10 of *Santa Fe Institute Studies in the Sciences of Complexity*, pages 579–601. Addison–Wesley.

Collins, Robert J. and David R. Jefferson (1991b). Representations for artificial organisms. In Meyer, Jean-Arcady and Stewart W. Wilson, editors, *Proceedings of the First International Conference on Simulation of Adaptive Behavior: From Animals to Animats*, pages 382–390. The MIT Press/Bradford Books.

Collins, Robert J. and David R. Jefferson (1991c). Selection in massively parallel genetic algorithms. In Belew, Richard K. and Lashon B. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 249–256. Morgan Kaufmann.

Coulson, Robert N., Joseph Folse, and Douglas K. Loh (1987). Artificial Intelligence and natural resource management. *Science*, 237:262–267.

Crosby, J. L. (1963). Evolution by computer. *New Scientist*, 327:415–417.

Crow, James F. (1986). *Basic Concepts in Population, Quantitative, and Evolutionary Genetics.* W. H. Freeman and Company, New York.

Darwin, Charles (1859). *On the Origin of Species by Means of Natural Selection.* Murray, London.

Darwin, Charles (1871). *The Descent of Man and Selection in Relation to Sex.* Murray, London.

Davidor, Yuval (1991). A naturally occuring niche & species phenomenon: The model and first results. In Belew, Richard K. and Lashon B. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 257–263. Morgan Kaufmann.

Day, P. R. (1974). *The Genetics of Host–Parasite Interactions.* W. H. Freeman.

De Jong, Kenneth A. (1975). *An Analysis of the Behavior of a Class of Genetic Adaptive Systems.* PhD thesis, University of Michigan.

Deb, Kalyanmoy and David E. Goldberg (1989). An investigation of niche and species formation in genetic function optimization. In Schaffer, J. David, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 42–50. Morgan Kaufmann.

Dobzhansky, T. (1956). What is an adaptive trait? *The American Naturalist*, 190:337–347.

Eshelman, Larry J. and J. David Schaffer (1991). Preventing premature convergence in genetic algorithms by preventing incest. In Belew, Richard K. and Lashon B. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 115–122. Morgan Kaufmann.

Falconer, Douglas S. (1981). *Introduction to quantitative genetics.* Longman, London, 2 edition.

Feldman, M. W., F. B. Christiansen, and Lisa D. Brooks (1986). Evolution of recombination in a constant environment. *Proceedings of the National Academy of Science, USA*, 77:4838–4841.

Felsenstein, Joseph (1987). Sex and the evolution of recombination. In Michod, Richard E. and Bruce R. Levin, editors, *The Evolution of Sex: An Examination of Current Ideas*, pages 74–86. Sinauer Associates Inc.

Felsenstein, Joseph and S. Yokoyama (1976). The evolutionary advantage of recombination. *Genetics*, 83:845–859.

Fewell, Jennifer H. (1988). Energetic and time costs of foraging in harvester ants, *pogonomyrmex occidentalis. Behav. Ecol. Sociobiol.*, 22:401–408.

Fisher, R. A. (1930). *The Genetical Theory of Natural Selection.* Dover Press, New York.

Fisher, Ronald A. (1958). *The Genetical Theory of Natural Selection.* Dover, New York, 2nd edition.

Flor, H. H. (1956). The complementary genic systems in flax and flax rust. *Advances in Genetics*, 8:29–54.

Franklin, I. and Richard C. Lewontin (1970). Is the gene the unit of selection? *Genetics*, 65:707–734.

Fry, John, Charles E. Taylor, and U. Devgan (1989). An expert system for mosquito control in Orange County California. *Bulletin of the Society of Vector Ecology*, 14(2):237–246.

Ghiselin, Michael T. (1987). The evolution of sex: A history of competing points of view. In Michod, Richard E. and Bruce R. Levin, editors, *The Evolution of Sex: An Examination of Current Ideas*, pages 7–23. Sinauer Associates Inc.

Gilmour, J. S. L. and J. W. Gregor (1939). Demes: A suggested new terminology. *Nature*, 144:333.

Glesener, R. R. and D. Tilman (1978). Sexuality and the components of environmental uncertainty: Clues from geographic parthenogenesis in terrestrial animals. *American Naturalist*, 112:659–673.

Goldberg, David E. (1989a). *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison–Wesley Publishing Company, Inc.

Goldberg, David E. (1989b). Sizing populations for serial and parallel genetic algorithms. In Schaffer, J. David, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 70–79. Morgan Kaufmann.

Goldberg, David E. and Jon T. Richardson (1987). Genetic algorithms with sharing for multimodal function optimization. In Grefenstette, John J., editor, *Genetic Algorithms and Their Applications: Proceedings of the Second International Conference on Genetic Algorithms*, pages 41–49. Lawrence Erlbaum Associates.

Goodenough, Ursula (1984). *Genetics.* Saunders College Publishing, 3 edition.

Gorges-Schleuter, Martina (1989). \SPARAGOS an asynchronous parallel genetic optimization strategy. In Schaffer, J. David, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 422–427. Morgan Kaufmann.

Grefenstette, John J., editor (1987). *Genetic Algorithms and Their Applications: Proceedings of the Second International Conference on Genetic Algorithms.* Lawrence Erlbaum Associates.

Grefenstette, John J. and James E. Baker (1989). How genetic algorithms work: A critical look at implicit parallelism. In Schaffer, J. David, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 20–27. Morgan Kaufmann.

Haigh, J. (1978). The accumulation of deleterious genes in a population: Muller's ratchet. *Theoretical Population Biology*, 14:251–257.

Hamilton, William D. (1980). Sex versus non–sex versus parasite. *Oikos*, 35:282–290.

Hamilton, William D. (1982). Pathogens as causes of genetic diversity in their host populations. In Anderson, R. M. and R. M. May, editors, *Population Biology of Infectious Diseases.* Springer–Verlag.

Hamilton, William D. (1986). Instability and cycling of two competing hosts with two parasites. In Karlin, S. and E. Nevo, editors, *Evolutionary Process Theory.* Academic Press.

Hamilton, William D. (1990). Sexual reproduction as an adaptation to resist parasites. *Proceedings of the National Academy of Science, USA*, 87:3566–3573.

Hamilton, William D., P. A. Henderson, and N. A. Moran (1981). Fluctuation of environment and coevolved antagonist polymorphisms as factors in the maintenance of sex. In Alexander, R. D. and D. W. Tinkle, editors, *Natural Selection and Social Behavior.* Chiron Press.

Harp, Steven Alex, Triq Samad, and Aloke Guha (1989). Towards the genetic synthesis of neural networks. In Schaffer, J. David, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 360–369. Morgan Kaufmann.

Hartl, Daniel L. and Andrew G. Clark (1989). *Principles of Population Genetics.* Sinauer Associates, Inc., Sunderland, Massachusetts.

Harvey, Inman (1991). The puzzle of the persistent question marks: A case study of genetic drift. Unpublished manuscript.

Hillis, W. Daniel (1985). *The Connection Machine.* The MIT Press, Cambridge, Massachusetts.

Hillis, W. Daniel (1991). Co–evolving parasites improve simulated evolution as an optimization procedure. In Langton, Christopher G., Charles Taylor, J. Doyne Farmer, and Steen Rasmussen, editors, *Artificial Life II*, volume 10 of *Santa Fe Institute Studies in the Sciences of Complexity*, pages 313–324. Addison–Wesley.

Hillis, W. Daniel and Joshua Barnes (1987). Programming a highly parallel computer. *Nature*, 326(6108):27–30.

Hillis, W. Daniel and Guy L. Steele, Jr. (1986). Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183.

Holland, John H. (1975). *Adaptation in Natural and Artificial Systems.* The University of Michigan Press.

Hölldobler, Bert and Edward O. Wilson (1990). *The Ants.* Harvard University Press.

Jaenike, J. (1978). An hypothesis to account for the maintenance of sex within population. *Evolutionary Theory*, 3:191–194.

Jefferson, David, Robert Collins, Claus Cooper, Michael Dyer, Margot Flowers, Richard Korf, Charles Taylor, and Alan Wang (1991). The Genesys System: Evolution as a theme in artificial life. In Langton, Christopher G., Charles Taylor, J. Doyne Farmer, and Steen Rasmussen, editors, *Artificial Life II*, volume 10 of *Santa Fe Institute Studies in the Sciences of Complexity*, pages 549–578. Addison–Wesley.

Johnson, Leslie K., Stephen P. Hubbell, and Donald H. Feener, Jr. (1987). Defense of food supply by eusocial colonies. *Amer. Zool.*, 27:347–358.

Kauffman, Stuart and Simon Levin (1987). Towards a general theory of adaptive walks on rugged landscapes. *Journal of Theoretical Biology*, 128:11–45.

Keightley, Peter D. and William G. Hill (1989). Quantitative genetic variability maintained by mutation–stabalizing selection balance: Sampling variation and response to subsequent directional selection. *Genetical Research*, 54:45–57.

Kimura, Motoo (1968). Evolutionary rate at the molecular level. *Nature*, 217:624–626.

Kimura, Motoo and Takeo Maruyama (1971). Pattern of neutral polymorphism in a geographically structured population. *Genetical Research*, 18:125–131.

Kimura, Motoo and Tomoko Ohta (1971). *Theoretical Aspects of Population Genetics.* Princeton University Press, Princeton, New Jersey.

Kimura, Motoo and George H. Weiss (1964). The stepping stone model of population structure and the decrease of genetic correlation with distance. *Genetics*, 49:561–576.

Kirkpatrick, Mark (1982). Sexual selection and the evolution of female choice. *Evolution*, 36(1):1–12.

Kirkpatrick, Mark and Michael J. Ryan (1991). The evolution of mating preferences and the paradox of the lek. *Nature*, 350:33–38.

Koza, John R. (1990). Genetic programming: A paradigm for genetically breeding populations of computer programs to solve problems. Technical report, Department of Computer Science, Stanford University.

Langton, Christopher G. (1989a). Artificial life. In Langton, Christopher G., editor, *Artificial Life*, volume 6 of *Santa Fe Institute Studies in the Sciences of Complexity*, pages 1–47. Addison–Wesley.

Langton, Christopher G., editor (1989b). *Artificial Life*, volume 6 of *Santa Fe Institute Studies in the Sciences of Complexity.* Addison–Wesley.

Langton, Christopher G., Charles Taylor, J. Doyne Farmer, and Steen Rasmussen, editors (1991). *Artificial Life II*, volume 10 of *Santa Fe Institute Studies in the Sciences of Complexity.* Addison–Wesley.

Levin, D. A. (1975). Pest pressure and recombination in plants. *American Naturalist*, 109:437–451.

Lewontin, Richard C. (1964). The interaction of selection and linkage. i. general considerations; heterotic models. *Genetics*, 49:49–67.

Lighton, John R. B. (1990). Energetics of foraging and recruitment in the giant tropical ant *paraponera clavata* (hymenoptera: Formicidae). (unpublished manuscript).

Lippmann, Richard P. (1987). An introduction to computing with neural nets. *IEEE ASSP Magazine.*

Manderick, Bernard and Piet Spiessens (1989). Fine–grained parallel genetic algorithms. In Schaffer, J. David, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 428–433. Morgan Kaufmann.

Martin, F. G. and C. C Cockerham (1960). Hish speed selection studies. In Kempthorne, O., editor, *Biometrical Genetics*. Pergamon.

May, R. M. (1983). Parasitic infections as regulators of animal populations. *American Scientist*, 71:36–44.

Maynard Smith, J. (1987). The evolution of recombination. In Michod, Richard E. and Bruce R. Levin, editors, *The Evolution of Sex: An Examination of Current Ideas*, pages 106–125. Sinauer Associates Inc.

Mayr, Ernst (1983). How to carry out the adaptationist program? *The American Naturalist*, 121(3):324–334.

Meyer, Jean-Arcady and Stewart W. Wilson, editors (1991). *Proceedings of the First International Conference on Simulation of Adaptive Behavior: From Animals to Animats*. The MIT Press/Bradford Books.

Michod, Richard E. and Bruce R. Levin, editors (1987). *The Evolution of Sex: An Examination of Current Ideas*, Sunderland, Massachusetts. Sinauer Associates Inc.

Miller, Geoffrey F., Peter M. Todd, and Shailesh U. Hegde (1989). Designing neural networks using genetic algorithms. In Schaffer, J. David, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 379–384. Morgan Kaufmann.

Mühlenbein, Heinz (1989). Parallel genetic algorithms, population genetics and combinatorial optimization. In Schaffer, J. David, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 416–421. Morgan Kaufmann.

Mühlenbein, Heinz, M. Schomisch, and J. Born (1991). The parallel genetic algorithm as function optimizer. In Belew, Richard K. and Lashon B. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 279–287. Morgan Kaufmann.

Muller, H. J. (1964). The relation of recombination and mutational advance. *Mutation Research*, 1:2–9.

O'Donald, Peter (1980). *Genetic Models of Sexual Selection*. Cambridge University Press.

Ohta, Tomoko (1987). Simulating evolution by gene duplication. *Genetics*, 115:207–213.

Ohta, Tomoko (1989). Time for spreading of compensatory mutations under gene duplication. *Genetics*, 123:579–584.

Ohta, Tomoko and Hidenori Tachida (1990). Theoretical study of near neutrality. I. Heterozygosity and rate of mutant substitution. *Genetics*, 126:219–229.

Pettey, Chrisila B., Michael R. Leuze, and John J. Grefenstette (1987). A parallel genetic algorithm. In Grefenstette, John J., editor, *Genetic Algorithms and Their Applications: Proceedings of the Second International Conference on Genetic Algorithms*, pages 155–161. Lawrence Erlbaum Associates.

Pettey, Chrisila C. and Michael R. Leuze (1989). A theoretical investigation of a parallel genetic algorithm. In Schaffer, J. David, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 398–405. Morgan Kaufmann.

Price, M. V. and N. M. Waser (1982). Population structure, frequency–dependent selection and the maintenance of sexual reproduction. *Evolution*, 36:35–43.

Provine, William B. (1986). *Sewall Wright and Evolutionary Biology.* University of Chicago Press.

Read, Andrew F. (1988). Sexual selection and the role of parasites. *Trends in Ecology and Evolution*, 3(5):97–102.

Rennie, John (1992). Trends in parasitology: Living together. *Scientific American*, 266(1):122–133.

Rice, W. R. (1983). Parent–offspring pathogen transmission: A selective agent promoting sexual reproduction. *American Naturalist*, 121:187–203.

Rumelhart, David E. and James L. McClelland, editors (1986). *Parallel distributed processing: Explorations in the microstructure of cognition.* MIT Press/Bradford Books, Cambridge, Massachusetts.

Schaffer, J. David, editor (1989). *Proceedings of the Third International Conference on Genetic Algorithms.* Morgan Kaufmann.

Schull, William J. and Bruce R. Levin (1964). Monte Carlo simulations: Some uses in the genetic study of primitive man. In Gurland, J., editor, *Stochastic Models in Medicine and Biology*, pages 179–196. The University of Wisconsin Press, Madison.

Seger, Jon and William D. Hamilton (1987). Parasites and sex. In Michod, Richard E. and Bruce R. Levin, editors, *The Evolution of Sex: An Examination of Current Ideas*, pages 176–193. Sinauer Associates Inc.

Shields, William M. (1987). Sex and adaptation. In Michod, Richard E. and Bruce R. Levin, editors, *The Evolution of Sex: An Examination of Current Ideas*, pages 253–269. Sinauer Associates Inc.

Spiessens, Piet and Bernard Manderick (1991). A massively parallel genetic algorithm: Implementation and first analysis. In Belew, Richard K. and Lashon B. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 279–287. Morgan Kaufmann.

Steele, Jr., Guy L. (1984). *Common LISP: The Language.* Digital Press.

Sudd, John H. and Nigel R Franks (1987). *The Behavioural Ecology of Ants.* Chapman & Hall, New York.

Swartzman, Gordon L. and Stephen P. Kaluzny (1987). *Ecological Simulation Primer.* Macmillan Publishing Company.

Tanese, Reiko (1987). Parallel genetic algorithm for a hypercube. In Grefenstette, John J., editor, *Genetic Algorithms and Their Applications: Proceedings of the Second International Conference on Genetic Algorithms*, pages 177–183. Lawrence Erlbaum Associates.

Tanese, Reiko (1989). Distributed genetic algorithms. In Schaffer, J. David, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 434–440. Morgan Kaufmann.

Taylor, Charles E. (1983). Evolution of resistance to insecticides: The role of mathematical models and computer simulations. In Georghiou, George P. and Tetsuo Saito, editors, *Pest Resistance to Pesticides.* Plenum Press.

Taylor, Charles E., David R. Jefferson, Scott R. Turner, and Seth R. Goldman (1989a). RAM: Artificial life for the exploration of complex biological systems. In Langton, Christopher G., editor, *Artificial Life*, volume 6 of *Santa Fe Institute Studies in the Sciences of Complexity*, pages 275–295. Addison–Wesley.

Taylor, Charles E., L. Muscatine, and David R. Jefferson (1989b). Maintenance and breakdown of the *hydra–chlorella* symbiosis: A computer model. *Proceedings of the Royal Society of London*, 238:277–289.

Tooby, J. (1982). Pathogens, polymorphism, and the evolution of sex. *Journal of Theoretical Biology*, 97:557–576.

Weiss, Gerhard (1990). Combining neural and evolutionary learning: Aspects and approaches. Technical report, Institute für Informatik, Technische Universität München.

Werner, Gregory M. (1991). Personal communication.

Werner, Gregory M. and Michael G. Dyer (1991). Evolution of communication in artificial organisms. In Langton, Christopher G., Charles Taylor, J. Doyne Farmer, and Steen Rasmussen, editors, *Artificial Life II*, volume 10 of *Santa Fe Institute Studies in the Sciences of Complexity*, pages 659–687. Addison–Wesley.

Whitley, Darrell (1989). The GENITOR algorithm and selection pressure: Why rank–based allocation of reproductive trials is best. In Schaffer, J. David, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 116–124. Morgan Kaufmann.

Whitley, Darrell and Thomas Hanson (1989). Optimizing neural networks using faster, more accurate genetic search. In Schaffer, J. David, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 391–397. Morgan Kaufmann.

Wieland, A.P. (1991a). Evolving controls for unstable systems. In Touretzky, D., J. Elman, T. Sejnowski, and G. Hinton, editors, *Proceedings of the 1990 Connectionist Models Summer School*, pages 91–102, San Diego 1990. Morgan Kaufmann, San Mateo.

Wieland, A.P. (1991b). Evolving neural network controllers for unstable systems. In *International Joint Conference on Neural Networks*, volume 2, pages 667–673, Seattle 1991. IEEE, New York.

Williams, George C. (1980). Kin selection and the paradox of sexuality. In Barlow, G. W. and J. Silverberg, editors, *Sociobiology: Beyond Nature/Nurture*. Westview.

Wright, Sewall (1931). Evolution in Mendelian populations. *Genetics*, 16:97–159.

Wright, Sewall (1932). The roles of mutation, inbreeding, crossbreeding and selection in evolution. In *Proceedings of the Sixth International Congress of Genetics*, volume 1, pages 356–366.

Wright, Sewall (1943). Isolation by distance. *Genetics*, 28:114–138.

Wright, Sewall (1968). *Evolution and the Genetics of Populations. Volume 1: Genetic and Biometric Foundations.* University of Chicago Press.

Wright, Sewall (1969). *Evolution and the Genetics of Populations. Volume 2: The Theory of Gene Frequencies.* University of Chicago Press.

Wright, Sewall (1977). *Evolution and the Genetics of Populations. Volume 3: Experimental Results and Evolutionary Deductions.* University of Chicago Press.

Wright, Sewall (1978). *Evolution and the Genetics of Populations. Volume 4: Variability Within and Among Natural Populations.* University of Chicago Press.

Zahavi, Amotz (1975). Mate selection—a selection for a handicap. *Journal of Theoretical Biology*, 53:205–214.

# Appendix A

# Chromosome Implementation

This appendix contains the code for our basic chromosome–handling algorithms. We implement both constant–length and variable–length chromosomes. This code is written in C++/CM++ for the Connection Machine-2, but many of the low–level routines are written in Paris, the Connection Machine-2 "assembly language." These algorithms are discussed in Section 2.4.1 of the text.

```
class Chromosome : public CM_bitstring {
  public:
    Chromosome(const unsigned length);
    Chromosome(const Chromosome&);
    virtual ~Chromosome(void);
    virtual void reproduce(const Chromosome& c);
    virtual void recombine(const Chromosome& c) = 0;
    virtual void mutate(void) = 0;
};

Chromosome::Chromosome(const unsigned length) : CM_bitstring(0, length) { }

Chromosome::Chromosome(const Chromosome& c) : CM_bitstring(c) { }

Chromosome::~Chromosome(void) { }

void Chromosome::reproduce(const Chromosome& c)
{
    recombine(c);
    mutate();
}

class Chromosome_constant : public Chromosome {
  private:
    float the_xover_rate;
    float the_mutation_rate;
  protected:
    void xover_rate(const float rate);
    void mutation_rate(const float rate);
    Chromosome_constant(const CM_field& f, const unsigned offset,
                        const unsigned length, const float xover,
                        const float mutate);
```

```
    Chromosome_constant(const Chromosome_constant&);
  public:
    virtual ~Chromosome_constant(void);
    virtual void recombine(const Chromosome& c);
    virtual void mutate(void);
};

Chromosome_constant::Chromosome_constant(const CM_field& f, const unsigned offset,
                                         const unsigned length, const float xover,
                                         const float mutate)
  : CM_field(f, offset, length), Chromosome(length), the_xover_rate(xover),
    the_mutation_rate(mutate) { }

Chromosome_constant::Chromosome_constant(const Chromosome_constant& c)
  : CM_field((const CM_field &) c), Chromosome(c), the_xover_rate(c.xover_rate()),
    the_mutation_rate(c.mutation_rate()) { }

Chromosome_constant::~Chromosome_constant(void) { }

void Chromosome_constant::recombine(const Chromosome &c)
{
    unsigned l = length();
    if (l == 1) {
        CM_boolean parent;
        CMSSL_u_fast_rng_1L(parent, parent.length(), 0);
        CM_if (parent) {
            CM_move(*this, c);
        } CM_end_if;
    }
    else if (l > 1) {
        int i;
        CM_boolean parent;
        CMSSL_u_fast_rng_1L(parent, parent.length(), 0);
        CM_float rnd;
        CM_boolean context;
        CM_store_context(context);
        for (i = 0; i < l; i++) {
            CM_logand_context(parent);
            CM_u_move_1L(CM_add_offset(*this, i), CM_add_offset(c, i), 1);
            CM_load_context(context);
            if (i ≠ l - 1) {
                CMSSL_f_fast_rng_1L(rnd, rnd.slength(), rnd.elength());
                CM_f_lt_constant_1L(rnd, the_xover_rate, rnd.slength(), rnd.elength());
                CM_logand_context_with_test();
                CM_lognot_1_1L(parent, parent.length());
                CM_load_context(context);
            };
        };
    };
    assert((CM_clear_all_flags(), 1));
}

void Chromosome_constant::mutate(void)
{
```

```
    if (the_mutation_rate == 0.0) {
        return;
    };
    int i;
    CM_float rnd;
    unsigned l = length();
    CM_boolean context;
    CM_store_context(context);
    for (i = 0; i < l; i++) {
        CMSSL_f_fast_rng_1L(rnd, rnd.slength(), rnd.elength());
        CM_lt(rnd, the_mutation_rate);
        CM_logand_context_with_test();
        invert(i);
        CM_load_context(context);
    };
    assert((CM_clear_all_flags(), 1));
}

class Chromosome_variable : public Chromosome {
  private:
    float the_xover_rate;
    float the_mutation_rate;
    float the_length_mutation_rate;
    CM_u_int the_var_length;
  protected:
    void xover_rate(const float rate);
    void mutation_rate(const float rate);
    void length_mutation_rate(const float rate);
    Chromosome_variable(const CM_field& f, const unsigned offset,
                        const unsigned length, const float xover,
                        const float mutate, const float mutate_length);
    Chromosome_variable(const Chromosome_variable&);
  public:
    virtual ~Chromosome_variable(void);
    virtual void recombine(const Chromosome& c);
    virtual void mutate(void);
    virtual void length_mutate(void);
};

Chromosome_variable::Chromosome_variable(const CM_field& f, const unsigned offset,
                                         const unsigned length, const float xover,
                                         const float mutate,
                                         const float length_mutate)
  : CM_field(f, offset, length), Chromosome(length), the_xover_rate(xover),
    the_mutation_rate(mutate), the_length_mutation_rate(length_mutate),
    the_var_length(0, lg(length) + 1) { }

Chromosome_variable::Chromosome_variable(const Chromosome_variable& c)
  : CM_field((const CM_field &) c), Chromosome(c), the_xover_rate(c.xover_rate()),
    the_mutation_rate(c.mutation_rate()),
    the_length_mutation_rate(c.length_mutation_rate()),
    the_var_length(c.the_var_length) { }

Chromosome_variable::~Chromosome_variable(void) { }
```

```
void Chromosome_variable::recombine(const Chromosome &c)
{
    unsigned l = length();
    if (l == 1) {
        CM_boolean parent;
        CMSSL_u_fast_rng_1L(parent, parent.length(), 0);
        CM_if (parent) {
            CM_move(*this, c);
        } CM_end_if;
    }
    else if (l > 1) {
        int i;
        CM_boolean parent;
        CM_u_int len0(the_var_length);
        CM_u_int len1((*((Chromosome_variable *)&c)).the_var_length);
        CM_random(parent);
        CM_if (parent) {
            CM_swap_2_1L(len0, len1, len0.length());
            assert((CM_clear_all_flags(), 1));
        } CM_end_if;
        CM_float rnd;
        CM_boolean context;
        CM_store_context(context);
        CM_boolean not_done;
        CM_store_context(not_done);
        for (i = 0; i < l; i++) {
            CM_logand_context(parent);
            CM_u_move_1L(CM_add_offset(*this, i), CM_add_offset(c, i), 1);
            CM_load_context(not_done);
            CM_u_le_constant_1L(len0, i + 1, len0.length());
            CM_logand_context_with_test();
            CM_u_move_1L(the_var_length, len0, len0.length());
            CM_u_move_zero_1L(not_done, not_done.length());
            CM_load_context(not_done);
            if (i ≠ l - 1) {
                CMSSL_f_fast_rng_1L(rnd, rnd.slength(), rnd.elength());
                CM_f_lt_constant_1L(rnd, the_xover_rate, rnd.slength(), rnd.elength());
                CM_logand_context_with_test();
                CM_u_gt_constant_1L(len1, i + 1, len1.length());
                CM_logand_context_with_test();
                CM_lognot_1_1L(parent, parent.length());
                CM_swap_2_1L(len0, len1, len0.length());
                CM_load_context(not_done);
            };
        };
        CM_load_context(context);
    };
    assert((CM_clear_all_flags(), 1));
}

void Chromosome_variable::mutate(void)
{
    if (the_mutation_rate == 0.0) return;
```

```
    int i;
    CM_float rnd;
    unsigned l = length();
    CM_boolean context;
    CM_store_context(context);
    for (i = 0; i < l; i++) {
        CMSSL_f_fast_rng_1L(rnd, rnd.slength(), rnd.elength());
        CM_lt(rnd, the_mutation_rate);
        CM_logand_context_with_test();
        invert(i);
        CM_load_context(context);
    };
    assert((CM_clear_all_flags(), 1));
}

void Chromosome_variable::length_mutate(void)
{
    if (the_length_mutation_rate == 0.0) return;
    int i;
    unsigned l = length();
    CM_boolean context;
    CM_store_context(context);
    for (i = 0; i < l; i++) {
        CM_u_lt_constant_1L(the_var_length, i, the_var_length.length());
        CM_logand_context_with_test();
        CMSSL_u_fast_rng_1L(CM_boolean_alias(*this, i), 1, 0);
        CM_load_context(context);
    };
    assert((CM_clear_all_flags(), 1));
    CM_float rnd;
    CM_random(rnd);
    CM_if (rnd < the_length_mutation_rate) {
        CM_int delta(0, 6);
        CM_random(delta);
        CM_if (delta < 0) {
            delta++;
        } CM_end_if;
        CM_int tmp(0, 32);
        CM_u_move_2L(tmp, the_var_length, tmp.length(), the_var_length.length());
        tmp += delta;
        CM_if (tmp < 0) {
            the_var_length = 0;
        }
        CM_elif (tmp > length()) {
            the_var_length = length();
        }
        CM_else {
            the_var_length = tmp;
        } CM_end_if;
    } CM_end_if;
}
```

# Appendix B

# Connection Descriptor ANN Implementation

This appendix contains the code for the connection descriptor ANN encoding. This code is written in C++/CM++ for the Connection Machine-2, but some of the code is written in Paris, the Connection Machine-2 "assembly language." This ANN encoding is discussed in Chapters 5 and 6 of the text.

```
class Decision {
  public:
    Decision(const unsigned num_states,
             Genome_variable *genome);
    virtual ~Decision(void);

    // Run the decision one step.
    virtual void update(void);

    // State access functions.
    unsigned num_states(void) const;
    CM_boolean &state(const unsigned i) const;
    CM_bitstring &state_vector(void) const;

    // Input access functions.
    unsigned num_inputs(void) const;
    CM_boolean& carry(void) const;
    CM_boolean& not_carry(void) const;
    CM_boolean& food(void) const;
    CM_boolean& food0(void) const;
    CM_boolean& food1(void) const;
    CM_boolean& nest(void) const;
    CM_boolean& nest0(void) const;
    CM_boolean& nest1(void) const;
    CM_boolean& pheromone(void) const;
    CM_boolean& pheromone0(void) const;
    CM_boolean& pheromone1(void) const;
    CM_int& compass_left(void) const;
    CM_int& compass_right(void) const;
    unsigned num_random(void) const;
    CM_bitstring& random_vector(void) const;
```

```
    CM_boolean& random(const unsigned i) const;

    // Output access functions.
    unsigned num_outputs(void) const;
    CM_boolean& grab_food(void) const;
    CM_boolean& drop_food(void) const;
    CM_int& turn_left(void) const;
    CM_int& turn_right(void) const;
    CM_int& move(void) const;
    CM_boolean& drop_pheromone(void) const;

  protected:
    // The genome.
    Genome_variable *the_genome;

    // The state stuff.
    unsigned the_num_states;
    CM_bitstring the_state_vector;
    CM_boolean_alias **the_state;

    // The input stuff.
    CM_boolean the_carry;
    CM_boolean the_not_carry;
    CM_boolean the_food;
    CM_boolean the_food0;
    CM_boolean the_food1;
    CM_boolean the_nest;
    CM_boolean the_nest0;
    CM_boolean the_nest1;
    CM_boolean the_pheromone;
    CM_boolean the_pheromone0;
    CM_boolean the_pheromone1;
    CM_int the_compass_left;
    CM_int the_compass_right;
    unsigned the_num_random;
    CM_bitstring the_random_vector;
    CM_boolean_alias **the_random;

    // The output stuff.
    CM_boolean the_grab_food;
    CM_boolean the_drop_food;
    CM_int the_turn_left;
    CM_int the_turn_right;
    CM_int the_move;
    CM_boolean the_drop_pheromone;
};

class ANN : public Decision {
  public:
    ANN(const unsigned num_states, Genome_variable *genome);
    ~ANN(void);
    // Run the decision one step.
    void update(void);
  protected:
```

```
    // The total number of units.
    unsigned num_units(void) const;
    unsigned the_num_units;
    // The weights.
    CM_bitstring the_weight_matrix;
    CM_float_alias **the_w;
    // The accumulators.
    CM_float *the_input_acc;
    CM_float *the_output_acc;
    CM_float *the_state_acc;
  private:
    void init(void);
};

inline unsigned ANN::num_units(void) const { return the_num_units; }

ANN::ANN(const unsigned num_states, Genome_variable *genome)
  : Decision(num_states, genome),
    the_num_units(num_inputs() + num_outputs() + num_states),
    the_input_acc(new CM_float[num_inputs()]),
    the_output_acc(new CM_float[num_outputs()]),
    the_state_acc(new CM_float[num_states]),
    the_w(new CM_float_alias*[((num_inputs() + num_states) * (num_inputs() + num_states
                                                              + num_outputs()))]),
    the_weight_matrix(0, (((num_inputs() + num_states) * (num_inputs() + num_states
                                                          + num_outputs())) * 32))
{
    init();
    for (int i = 0;
         i < ((num_inputs() + num_states) * (num_inputs() + num_states
                                             + num_outputs()));
         i++)
        the_w[i] = new CM_float_alias(the_weight_matrix, i * 32);
}

void ANN::init(void)
{
    const unsigned from_addr_length = lg(16);
    assert(num_outputs() + num_states() + num_inputs() ≤ 32);
    const unsigned to_addr_length = lg(32);
    const unsigned weight_length = 5;
    const unsigned connection_length = from_addr_length + to_addr_length
                                       + weight_length);

    //
    // We are going to build up a matrix of weights by interpreting
    // the genome as a sequence of connection descriptors.
    //
    CM_u_int num_connections(the_genome→var_length(0));
    num_connections /= connection_length;
    // transcribe to fill the matrix
    int i;
    CM_u_int addr(0, 10);
    CM_int w(0, weight_length);
```

```
CM_float tmp, tmp1;
for (i = 0; i < (the_genome→length() / connection_length); i++) {
    CM_if (i < num_connections) {
        CM_u_int_alias from(*the_genome, i * connection_length + 0, from_addr_length);
        CM_u_int lfrom(0, 16);
        lfrom = from;
        CM_u_int_alias to(*the_genome, i * connection_length + from_addr_length,
                          to_addr_length);
        CM_u_int lto(0, 16);
        lto = to;
        CM_int_alias weight(*the_genome,
                            i * connection_length + from_addr_length
                            + to_addr_length,
                            weight_length);
        // remove bias from weight
        w = weight;
        CM_if (w < 0) {
            w++;
        } CM_end_if;
        CM_to_float(tmp1, w);
        //
        // Include this weight in the matrix.
        //
        // We are using a bilaterally symmetric network, which
        // means that we interpret the connection descriptors
        // as 2 connections.  Naturally, this gets rather grim
        // rather quickly.  I hope this code is right, because
        // I have no idea how to document it.
        //
        unsigned num_asym_in = 5;       // carry + f + n + p
        unsigned num_sym_in = 4 + num_random() / 2;   // c + f + n + p + r/2
        unsigned num_sym_hid = num_states() / 2;
        unsigned num_asym_out = 4;      // grab f + drop f + drop p + move
        unsigned num_sym_out = 1;       // turn
        // Make sure it is not a nop descriptor
        CM_if ((lfrom < (num_asym_in + num_sym_in + num_sym_hid))
               && (lto < (num_asym_in + 2 * num_sym_in + num_asym_out
                          + 2 * num_sym_out + 2 * num_sym_hid))) {
            //
            // First connection.
            //
            CM_if (lfrom < (num_asym_in + num_sym_in)) {
                // it is from an input
                addr = lfrom;
            }
            CM_else {
                // it is from a hidden
                addr = lfrom + num_sym_in;
            } CM_end_if;
            addr * = (num_inputs() + num_outputs() + num_states());
            addr += lto;
            CM_aref32(tmp, the_weight_matrix, addr);
            tmp += tmp1;
            CM_aset32(tmp, the_weight_matrix, addr);
```

```
//
// Second connection.
//
CM_if (lfrom < num_asym_in) {
    // it is from an asymmetric input
    addr = lfrom;
}
CM_elif (lfrom < (num_asym_in + num_sym_in)) {
    // it is from a symmetric input
    addr = lfrom + num_sym_in;
}
CM_else {
    // it is from a symmetric hidden
    addr = lfrom + (num_sym_in + num_sym_hid);
} CM_end_if;
addr *= (num_inputs() + num_outputs() + num_states());
CM_if (lto < num_asym_in) {
    // it is to an asymmetric input
    addr += lto;
}
CM_elif (lto < (num_asym_in + num_sym_in)) {
    // it is to a symmetric input on this side
    addr += lto + num_sym_in;
}
CM_elif (lto < (num_asym_in + 2 * num_sym_in)) {
    // it is to a symmetric input on the other side
    addr += lto - num_sym_in;
}
CM_elif (lto < (num_asym_in + 2 * num_sym_in + num_asym_out)) {
    // it is to an asymmetric output
    addr += lto;
}
CM_elif (lto < (num_asym_in + 2 * num_sym_in + num_asym_out
                + num_sym_out)) {
    // it is to a symmetric output on this side
    addr += lto + num_sym_out;
}
CM_elif (lto < (num_asym_in + 2 * num_sym_in + num_asym_out
                + 2 * num_sym_out)) {
    // it is to a symmetric output on the other side
    addr += lto - num_sym_out;
}
CM_elif (lto < (num_asym_in + 2 * num_sym_in + num_asym_out +
                2 * num_sym_out + num_sym_hid)) {
    // it is to a symmetric hidden on this side
    addr += lto + num_sym_hid;
}
CM_else {
    // it is to a symmetric hidden on the other side
    addr += lto - num_sym_hid;
} CM_end_if;
CM_aref32(tmp, the_weight_matrix, addr);
tmp += tmp1;
CM_aset32(tmp, the_weight_matrix, addr);
```

```
            } CM_end_if;
        } CM_end_if;
    };
    CM_transpose32(the_weight_matrix);
}

ANN::~ANN(void)
{
    delete [num_inputs()] the_input_acc;
    delete [num_outputs()] the_output_acc;
    delete [num_states()] the_state_acc;
    for (int i = 0;
         i < ((num_inputs() + num_states()) * (num_inputs() + num_states()
             + num_outputs()));
         i++)
        delete the_w[i];
    delete the_w;
}

void ANN::update(void)
{
    int i, j;
    float bounds;

    //
    // Set up the inputs.
    //
    CM_save_context context;
    for (i = 0; i < num_inputs(); i++) the_input_acc[i] = 0.0;
    i = 0;
    CM_load_context(carry());
    the_input_acc[i++] = 1.0;
    CM_load_context(not_carry());
    the_input_acc[i++] = 1.0;
    CM_load_context(food());
    the_input_acc[i++] = 1.0;
    CM_load_context(nest());
    the_input_acc[i++] = 1.0;
    CM_load_context(pheromone());
    the_input_acc[i++] = 1.0;
    CM_load_context(context);
    CM_to_float(the_input_acc[i++], compass_left());
    CM_load_context(food0());
    the_input_acc[i++] = 1.0;
    CM_load_context(nest0());
    the_input_acc[i++] = 1.0;
    CM_load_context(pheromone0());
    the_input_acc[i++] = 1.0;
    for (j = 0; j < num_random() / 2; j++) {
        CM_load_context(random(j));
        the_input_acc[i++] = 1.0;
    };
    CM_load_context(context);
    CM_to_float(the_input_acc[i++], compass_right());
```

```
CM_load_context(food1());
the_input_acc[i++] = 1.0;
CM_load_context(nest1());
the_input_acc[i++] = 1.0;
CM_load_context(pheromone1());
the_input_acc[i++] = 1.0;
for (j = num_random() / 2; j < num_random(); j++) {
    CM_load_context(random(j));
    the_input_acc[i++] = 1.0;
};
CM_load_context(context);

//
// Initialize the input, state and output accumulators.
//
CM_float **the_pre_input_acc = new CM_float*[num_inputs()];
for (i = 0; i < num_inputs(); i++) the_pre_input_acc[i] = new CM_float(0.0);
CM_float **the_pre_state_acc = new CM_float*[num_states()];
for (i = 0; i < num_states(); i++) the_pre_state_acc[i] = new CM_float(0.0);
for (i = 0; i < num_states(); i++) {
    the_state_acc[i] = 0.0;
    CM_load_context(state(i));
    *(the_pre_state_acc[i]) = 1.0;
    CM_load_context(context);
};
for (i = 0; i < num_outputs(); i++) the_output_acc[i] = 0.0;

//
// Run the inputs->hidden and hidden->hidden.
//
for (i = 0; i < num_inputs(); i++) {
    int aindex = (i * (num_inputs() + num_outputs() + num_states()));
    for (j = 0; j < num_inputs(); (j++, aindex++)) {
        CM_f_mult_add_always_1L(*(the_pre_input_acc[j]),
                                the_input_acc[i],
                                *(the_w[aindex]),
                                *(the_pre_input_acc[j]),
                                the_w[aindex]->slength(),
                                the_w[aindex]->elength());
    };
};

//
// Threshold the inputs.
//
for (i = 0; i < num_inputs(); i++) {
    CM_load_context(*(the_pre_input_acc[i]) < 0.0);
    the_input_acc[i] = 0.0;
    CM_load_context(context);
};
for (i = 0; i < num_inputs(); i++) delete the_pre_input_acc[i];
delete the_pre_input_acc;

//
```

```
// Run the inputs->hidden and hidden->hidden.
//
for (i = 0; i < num_inputs(); i++) {
    int aindex = (i * (num_inputs() + num_outputs() + num_states())
                  + num_inputs() + num_outputs());
    for (j = 0; j < num_states(); (j++, aindex++)) {
        CM_f_mult_add_always_1L(the_state_acc[j],
                                the_input_acc[i],
                                *(the_w[aindex]),
                                the_state_acc[j],
                                the_w[aindex]→slength(),
                                the_w[aindex]→elength());
    };
};
for (i = 0; i < num_states(); i++) {
    int aindex = ((i + num_inputs())
                  * (num_inputs() + num_outputs() + num_states())
                  + num_inputs() + num_outputs());
    for (j = 0; j < num_states(); (j++, aindex++)) {
        CM_f_mult_add_always_1L(the_state_acc[j],
                                *(the_pre_state_acc[i]),
                                *(the_w[aindex]),
                                the_state_acc[j],
                                the_w[aindex]→slength(),
                                the_w[aindex]→elength());
    };
};

//
// Threshold the states.
//
for (i = 0; i < num_states(); i++) {
    *(the_pre_state_acc[i]) = 0.0;
    CM_load_context(the_state_acc[i] > 0.0);
    *(the_pre_state_acc[i]) = 1.0;
    CM_load_context(context);
};

//
// Run the inputs->outputs and hidden->outputs.
//
for (i = 0; i < num_inputs(); i++) {
    int aindex = (i * (num_inputs() + num_outputs() + num_states())
                  + num_inputs());
    for (j = 0; j < num_outputs(); (j++, aindex++)) {
        CM_f_mult_add_always_1L(the_output_acc[j],
                                the_input_acc[i],
                                *(the_w[aindex]),
                                the_output_acc[j],
                                the_w[aindex]→slength(),
                                the_w[aindex]→elength());
    };
};
for (i = 0; i < num_states(); i++) {
```

```
        int aindex = ((i + num_inputs())
                      * (num_inputs() + num_outputs() + num_states())
                      + num_inputs());
        for (j = 0; j < num_outputs(); (j++, aindex++)) {
            CM_f_mult_add_always_1L(the_output_acc[j],
                                    *(the_pre_state_acc[i]),
                                    *(the_w[aindex]),
                                    the_output_acc[j],
                                    the_w[aindex]->slength(),
                                    the_w[aindex]->elength());
        };
    };
    for (i = 0; i < num_states(); i++) delete the_pre_state_acc[i];
    delete the_pre_state_acc;

    //
    // Copy out the states.
    //
    for (i = 0; i < num_states(); i++) state(i) = (the_state_acc[i] > 0.0);

    //
    // Copy out the outputs.
    //
    i = 0;
    // Grab food.
    grab_food() = (the_output_acc[i++] > 0.0);
    // Drop food.
    drop_food() = (the_output_acc[i++] > 0.0);
    // Move.
    bounds = (float) ((1 << (move().length() - 1)) - 1);
    CM_min(the_output_acc[i], bounds);
    CM_max(the_output_acc[i], -bounds);
    CM_truncate(move(), the_output_acc[i]);
    i++;
    // Drop pheromone.
    drop_pheromone() = (the_output_acc[i++] > 0.0);
    // Turn left.
    bounds = (float) ((1 << (turn_left().length() - 1)) - 1);
    CM_min(the_output_acc[i], bounds);
    CM_max(the_output_acc[i], -bounds);
    CM_truncate(turn_left(), the_output_acc[i]);
    i++;
    // Turn right
    bounds = (float) ((1 << (turn_right().length() - 1)) - 1);
    CM_min(the_output_acc[i], bounds);
    CM_max(the_output_acc[i], -bounds);
    CM_truncate(turn_right(), the_output_acc[i]);
}
```