The Open University of Israel
Department of Mathematics and Computer Science

Analysis of the Effects of Lifetime Learning on Population Fitness using Vose Model

Thesis submitted in partial fulfillment of the requirements
towards an M.Sc. degree in Computer Science
The Open University of Israel
Computer Science Division

By

Roi Yehoshua

Prepared under the supervision of
Dr. Mireille Avigal, The Open University of Israel
Prof. Ron Unger, Bar-Ilan University

June 2010

I wish to express my gratitude to Dr. Mireille Avigal and Prof. Ron Unger for their devoted counsel, guidance and patience.

Contents

List of Figures

Abstract

Vose's dynamical systems model of the simple genetic algorithm (SGA) is an exact model that uses mathematical operations to capture the dynamical behavior of genetic algorithms. The original model was defined for a simple genetic algorithm. This thesis suggests how to extend the model and incorporate two kinds of learning, Darwinian and Lamarckian, into the framework of the Vose model. The extension provides a new theoretical framework to examine the effects of lifetime learning on the fitness of a population. We analyze the asymptotic behavior of different hybrid algorithms on an infinite population vector and compare it to the behavior of the classical genetic algorithm on various population sizes. Our experiments show that Lamarckian-like inheritance - direct transfer of lifetime learning results to offsprings - allows quicker genetic adaptation. However, functions exist where the simple genetic algorithms without learning as well as Lamarckian evolution converge to the same local optimum, while genetic search based on Darwinian inheritance converges to the global optimum. The main results of this thesis are included in a paper that was accepted in the Genetic and Evolutionary Computation Conference (GECCO 2010).

Chapter 1

Introduction

Genetic algorithms (GA) have been shown to be very efficient at exploring large search spaces. However, they are often incapable of finding the precise local optima in the region where the algorithm converges. A hybrid genetic algorithm uses local search to improve the solutions produced by the genetic operators. Local search in this context can be regarded as a kind of learning that occurs during the lifetime of an individual string.

Evolution and learning are two forms of biological adaptation that differ in space and time. Evolution is a process of selective reproduction and mutations occurring within a geographically-distributed population of individuals over long periods of time. Learning, however, is a set of modifications taking place within each single individual during its own life time.

Learning can guide and accelerate the evolutionary process in different aspects. First, learning allows the maintenance of more genetic diversity in the population, since different genes have more chances to be preserved in the population if the individuals who incorporate these genes are able to learn the same fit behaviors. Second, learning provides the evolutionary process with rich amount of information from the environment to rely upon when deciding whether the individual is fit to its environment. Whereas evolutionary adaptation relies on a single value which reflects how well an individual coped with its environment (the number of offspring in the case of natural evolution and the fitness value in the case of artificial evolution), learning can rely on a huge amount of feedback from the environment that reflects how well an individual is doing in different moments of its life.

However, learning has some costs. Learning individuals may suffer from a

sub-optimal behavior during the learning phase. As a result, they will collect less fitness than individuals who have the same behavior genetically inherited. Moreover, since learned behavior is determined mostly by the environment, if a vital behavior-defining stimulus is not encountered by a particular individual, then it may hamper its development.

Another important aspect of combining learning and evolution is known as the *Baldwin effect* [1]. Baldwin's argument was that learning accelerates evolution because sub-optimal individuals can reproduce by acquiring during life necessary features for survival. However, since learning requires time, evolution tends to select individuals who have already at birth those useful features which would otherwise be learned. Hence, Baldwin's effect explains the indirect genetic assimilation of learned traits, even when those traits are not coded back into the genome. A number of researches have replicated the Baldwin effect in population of artificial organisms [2, 3, 4].

Our goal in this research was to find a mathematical model that would help us understand the effects of combining lifetime learning and genetic search on population fitness. We have chosen Vose's dynamical systems model for the simple genetic algorithm [5] as the theoretical framework for our analysis. A detailed description of the model will follow.

We compare two forms of hybrid genetic search. The first uses Darwinian evolution, in which the improvements an individual acquired during its lifetime contribute to its fitness, but are not transformed back into the genetic encoding of the individual. The second uses Lamarckian evolution, in which the results of the learning process (i.e. the acquired features) are coded back onto the strings processed by the genetic algorithm. We also compare these two approaches to a simple genetic algorithm without any learning.

1.1 Related Work

Hinton and Nowlan [6] were among the first researchers to describe a simple computational model that shows how learning could help and guide evolution. They illustrate the Baldwin effect using a genetic algorithm and a simple random learning process that develops a simple neural network.

In their experiment individuals have genotypes with 20 genes which encode a neural network with 20 potential connections. Only a neural network that is

connected in exactly the right way can provide added reproductive fitness to an organism. Genes can have three alternative values: 1, 0, or ?, which specifies, respectively, the presence of a connection between two neurons in the network, the absence of a connection, and a modifiable state (presence or absence of a connection) that can be changed according to a learning mechanism. The learning mechanism is a simple random process that keeps changing modifiable connections until a good combination (if any) is found during the lifetime of the individual. Any guessed values are lost at the end of the individual's life.

The experiments compared the performance of a population endowed with learning to one without. Results showed that the non-learning population was not capable of finding optimal solutions to the problem. In contrast, once learning was applied, the population converged on the problem solution. The addition of learning made the fitness surface area smoother around the good combination which could be discovered and easily climbed by the genetic algorithm. As can be seen in figure 1.1, without learning the fitness surface is flat, with a thin spike corresponding to the good combination of alleles (the thick line). When learning is enabled, the fitness surface has a nice hill around the spike which includes the alleles' combinations which have some right fixed values and some unspecified (learnable) values.

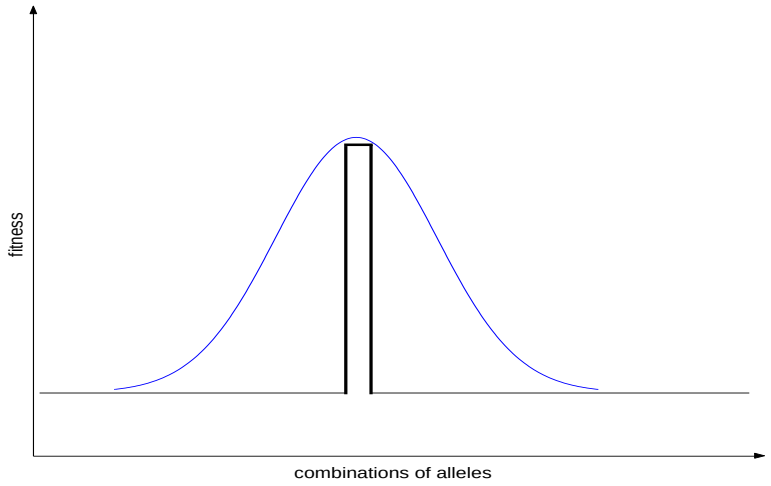

Figure 1.1: Fitness surface with and without learning. Redrawn from Hinton and Nowlan [6].

The model also showed that once individuals which have part of their genes

fixed on the right values and part of their genes unspecified are selected, individuals with less and less learnable genes tend to be selected. In other words, characters that were first acquired through learning tend to become genetically specified later on, which is supported by the Baldwin effect.

Number of other researchers have since explored the interactions between evolution and learning, showing that the addition of individual lifetime learning can improve the population's fitness and diversity [7, 8, 9, 10].

Some of the researchers also compared the performance of Lamarckian evolution to Darwinian evolution on various test sets [11, 12]. In some cases, when the problems were relatively easy to solve using stochastic hill-climbing methods, Lamarckian learning led to quicker convergence of the genetic algorithm and to better solutions than by leaving the chromosome unchanged after evaluation. However, in more complex, non-linear problem domains, forcing the genotype to equal the phenotype caused the algorithm to converge prematurely to one of the local optima, and consequently in those domains the Lamarckian search has suffered from inferior performance relative to the Darwinian learning.

Houck et al. [13] have shown that neither a pure Lamarckian nor a pure Darwinian search strategy was found to consistently lead to quicker convergence of the GA to the best known solution for a series of test problems, including the location-allocation problem and the cell formation problem. Only partial Lamarckianism search strategies (i.e., updating the genetic representation for only a percentage of the individuals) yielded the best mixture of solution quality and computational efficiency.

Sasaki and Tokoro [14] found when evolving artificial neural networks that Lamarckian inheritance of weights learned in a lifetime was harmful in changing environments or when different individuals were exposed to different learning experiences, but beneficial in stationary environments. Recently, Paenke et al. [15] developed a simple stochastic model of evolution and learning that explains why Lamarckian inheritance should be less common in environments that oscillate rapidly compared to stationary environments. However, their model was limited to a genotype consisting of only one gene and two possible phenotypes.

1.2 Theoretical Models of GA

One of the earliest theoretical models which was devised to explain the behavior of genetic algorithms was the schemata theory, originally introduced by Holland [16] and further developed by Goldberg [17]. A schema is a similarity pattern describing a subset of strings with similarities at certain string positions. For example, the schema *11*0, represents all strings that have 1 at positions 2 and 3, and 0 at position 5. The schema theory provides a mathematical model which estimates how the number of individuals in the population belonging to a certain schema can be expected to grow in the next generation.

The fundamental schema theorem states that short, low-order, fitter-than-average schemata are allocated exponentially increasing trials over time. However, the conventional schema theorem provides us with only a lower bound for the expected number of schemata at the next generation, because it accounts only for schema disruption and survival, not schema creation (by genetic operators such as crossover). Thus its predictions may be difficult to use in practice.

In contrast, the Vose model [5] is an exact mathematical model that captures every detail of the simple genetic algorithm in mathematical operators, and thus enables us to prove certain theorems about properties of these operators. The model defines a matrix G as the composition of the fitness matrix F and a recombination operator M that mimics the effects of crossover and mutation. By iterating G on the population vector, it is possible to give an exact description of the expected behavior of the Simple Genetic Algorithm (SGA). Our goal in this work was to extend the Vose model by adding a learning operator L that mimics the effects of lifetime learning on the population vector and to investigate its influence on the evolutionary process. A formal definition of the model will follow.

The Vose model assumes an infinite population size. In any finite population, sampling errors will cause deviations from the expected values. However, since the infinite population vector is equivalent to the sampling distribution probabilities used by finite Markov models of the SGA, a general agreement between the behavior of the infinite population vector and the behavior of random finite population vectors is observed in experiments in a reasonable population size.

We have found that the Vose dynamical systems model has several advantages over other theoretical models of the genetic algorithms:

- The schema theory makes predictions about the expected change in frequencies of genetic patterns from one generation to the next, but it does not directly make predictions regarding the population composition, the speed of population convergence or the distribution of the fitnesses in the population over time. The Vose model, by predicting the exact evolution of an infinite population vector, enables us to explore these aspects of the genetic algorithm.

- Traditional schema theory does not support Lamarckian learning, since Lamarckian learning disrupts the schema processing of the genetic algorithm. In contrast, it is possible to model the integration of both forms of lifetime learning (Darwinian and Lamarckian) within the framework of the Vose model using the same basic approach.

- The mathematical framework of the Vose model enables us to use techniques of matrix calculus to explore the asymptotic behavior of the hybrid genetic algorithms without relying on specific settings of the algorithm parameters (such as population size, random generator seed, etc.).

This thesis also builds on earlier work by Whitley, Gordon and Mathias [11], which explored the behavior of hybrid genetic algorithms using a model of a genetic algorithm developed by Whitley [18]. However, the Vose model we use in this thesis takes a more general approach, which includes both crossover and the mutation operators and also considers finite population models of the genetic search. Furthermore, we extend Vose's analysis of the behavior of the simple genetic algorithm near its stationary points, to include the effects of lifetime learning.

The rest of this thesis is organized as follows: Chapter 2 reviews the Vose dynamical systems model of the simple genetic algorithm, chapter 3 shows how the two forms of the hybrid genetic algorithm can be modeled in the context of the Vose model, chapter 4 describes a series of experiments performed on a binary optimization problem, chapter 5 compares the behavior of the classical genetic algorithm with the predictions of the Vose model, chapter 6 analyzes the asymptotical behavior of the SGA around its fixed points, chapter 7 tests the results on a numerical function optimization problem and chapter 8 draws some conclusions and suggests future work.

Chapter 2

Vose Model of Simple Genetic Algorithm

2.1 Representation

Let the search space Ω be defined over l-digit binary representations of integers in the interval $[0, 2^l - 1]$. $n = 2^l$ is the number of points in the search space. For example, if $l = 3$ then $n = 8$ and $\Omega = \{000, 001, 010, 011, 100, 101, 110, 111\}$

Define the simplex to be the set $\Lambda = \{p = (p_0, ..., p_{n-1}) : \mathbf{1}^T p = 1, p_j \in \Re, p_j \geq 0\}$, where $\mathbf{1}$ denotes the vector of all 1s. A vector $p^t \in \Lambda$ represents a population vector at generation t, where p_i^t is the proportion of the ith element of Ω in p^t.

For example, if $l = 3$ then the population $\{010, 000, 111, 000, 000\}$ is represented by the vector $p = (0.6, 0, 0.2, 0, 0, 0, 0, 0.2)$. Note that tuples in round brackets (...) will be regarded as column vectors in this thesis.

Given the current population vector p, the next population vector is derived from the genetic algorithm using some transition rule τ. Thus, an evolutionary run can be described as a sequence of iterations beginning from some initial population vector p:

$$p, \tau(p), \tau^2(p), ...$$

Let $\mathcal{G}$ be a probability function which, given a population vector p, produces a vector whose i^{th} component is the probability that the i^{th} element of Ω is chosen for the next generation (with replacement). That is, $\mathcal{G}(p)$ is the

probability vector which specifies the sampling distribution for the next generation. If the population is infinitely large, then $\mathcal{G}(p)$ is the exact distribution of the next population.

For example, let Ω be the set $\{00, 01, 10, 11\}$ and suppose the function $\mathcal{G}$ is

$$\mathcal{G}(p) = \frac{(p_0, 2p_1, 5p_2, 0)}{p_0 + 2p_1 + 5p_2}$$

Let the initial population be $p = (0.25, 0.25, 0.25, 0.25)$. Thus, $\mathcal{G}(p) = (1/8, 1/4, 5/8, 0)$, and the probability of sampling 00 is $1/8$, of sampling 01 is $1/4$, and of sampling 10 is $5/8$. If population size is $r = 100$, the transition rule corresponds to making 100 independent samples, with replacement, according to these probabilities. A plausible next generation is therefore $\tau(p) = (\frac{12}{100}, \frac{25}{100}, \frac{63}{100}, 0) = (0.12, 0.25, 0.63, 0)$.

Genetic algorithms can be classified according to the behavior of $\mathcal{G}$. In particular, a genetic algorithm is called *focused* if $\mathcal{G}$ is continuously differentiable and for every p the sequence

$$p, \mathcal{G}(p), \mathcal{G}^2(p), ...$$

converges. If we denote $\omega = \lim_{l\to\infty} \mathcal{G}^l(p)$, then by the continuity of $\mathcal{G}$,

$$\mathcal{G}(\omega) = \mathcal{G}(\lim_{l\to\infty} \mathcal{G}^l(p)) = \lim_{l\to\infty} \mathcal{G}^{l+1}(p) = \omega$$

Such points ω that satisfy $\mathcal{G}(\omega) = \omega$ are called *fixed points* of $\mathcal{G}$. These points have great influence on both the short-term and asymptotic behavior of focused genetic algorithms. One of the purposes of our research is to understand the effects of incorporating learning into the genetic algorithm on these fixed points and the behavior of the algorithm around those points.

The simple genetic algorithm model is now realized by defining the function $\mathcal{G}$ through steps analogous to the classical genetic algorithm. In the following sections we describe how the genetic operators of selection, mutation and crossover are defined, and then how these elements are combined in the simple genetic algorithm.

Before starting off the detailed description of the genetic operators, it is convenient to mention the algebraic notation that will be used throughout this thesis:

$\oplus$ is the bitwise *exclusive-or* operator and $\otimes$ is the bitwise *and* operator, for example:

$$5 \oplus 3 = 101 \oplus 011 = 111 = 6$$

$$4 \otimes 6 = 100 \otimes 110 = 100 = 4$$

$\overline{k}$ is the bitwise complement of k, i.e. $\overline{k} = \mathbf{1} \oplus k$. For example, $\overline{101} = 010$.

2.2 Selection

Let the vector s^t represent the population vector at generation t after selection but before any other operators (e.g. mutation and crossover) are applied.

The computation of s^t from p^t is based on a fitness function $f : \Omega \to R$. The value $f(i)$ is called the fitness of i. Through the identification $f_i = f(i)$, the fitness function may also be regarded as a vector. We also define $diag(f)$ as the diagonal matrix whose $(i,i)^{th}$ element is given by f_i.

Proportional selection is then defined for each $x \in \Lambda$ by the following selection function:

$$\mathcal{F}(x) = \frac{diag(f)x}{f^T x}$$

Thus the population vector at generation t after selection is:

$$s^t = \mathcal{F}(p^t) = \frac{diag(f)p^t}{f^T p^t}$$

where $diag(f)p^t = (f_0 p_0^t, f_1 p_1^t, ..., f_{n-1} p_{n-1}^t)$ and the population fitness weighted average is given by $f^T p^t = f_0 p_0^t + f_1 p_1^t + ... + f_{n-1} p_{n-1}^t$.

2.3 Mutation

Let the vector μ be a mutation vector in which the component μ_i is the probability with which $i \in \Omega$ is selected to be a mutation mask. The effect of mutating a vector x using mutation mask i is to alter the bits of x in those positions where the mutation mask i is 1, i.e. the result is $x \oplus i$.

When mutation is determined by a *mutation rate* $\mu \in [0, 0.5)$, the probability of selecting mask i depends only on the number of 1s that i contains, i.e. the distribution μ is defined by the following rule:

$$\begin{aligned}\mu_i &= (\mu)^{\text{the number of 1s in } i}(1-\mu)^{\text{the number of 0s in } i} \\ &= (\mu)^{\mathbf{1}^T \cdot i}(1-\mu)^{l-\mathbf{1}^T \cdot i}\end{aligned}$$

where $\mathbf{1}^T \cdot i$ denotes the inner product of the vector of 1s with the mask i, and l denotes the length of i.

The function $\mathcal{F}_\mu$ corresponding to mutating the result of selection is defined by

$$\begin{aligned}\mathcal{F}_\mu(p)_i &= \Pr[i \text{ results} \mid \text{population } p] \\ &= \sum_j \Pr[j \text{ selected} \mid \text{population } p] \Pr[j \text{ mutates to } i] \\ &= \sum_j \mathcal{F}(p)_j \mu_{j \oplus i}\end{aligned}$$

where $\mu_{j \oplus i}$ denotes the probability to choose the mutation mask $j \oplus i$, which transforms vector j to vector i, since $j \oplus (j \oplus i) = (j \oplus j) \oplus i = 0 \oplus i = i$.

2.4 Crossover

Let the vector χ be a crossover vector in which the component χ_i is the probability with which $i \in \Omega$ is selected to be a crossover mask. The application of a crossover mask i to parent vectors x, y produces offsprings by exchanging the bits of the parents in those positions where the crossover mask i is 1. The result is the pair $(x \otimes i) \oplus (\bar{i} \otimes y)$ and $(y \otimes i) \oplus (\bar{i} \otimes x)$, each created with equal probability. The application of χ to x, y is referred to as *recombining* x and y.

When crossover is determined by a *crossover rate* $\chi \in [0, 1]$, the distribution χ is specified according to the following rule:

$$\chi_i = \begin{cases} \chi c_i & i > 0 \\ 1 - \chi + \chi c_0 & i = 0 \end{cases}$$

Classical crossover types include 1-point crossover, for which:

$$c_i = \begin{cases} \frac{1}{l-1} & \exists k \in (0, l) |\, i = 2^k - 1 \\ 0 & \text{otherwise} \end{cases}$$

2.5 Mixing

Obtaining child z from parents x and y via the process of mutation and crossover is called *mixing* and has probability denoted by $m_{x,y}(z)$.

By theorem 4.3 in [5], if mutation is performed before crossover, then

$$m_{x,y}(z) = \sum_{i,j,k} \mu_i \mu_j \frac{\chi_k + \chi_{\overline{k}}}{2} [((x \oplus i) \otimes k) \oplus (\overline{k} \otimes (y \oplus j)) = z]$$

The right hand side sums the probabilities of choosing mutation masks i, j and a crossover mask k such that the result of mutating x, y using i, j and then recombining $x \oplus i$ and $y \oplus j$ using k produces offspring z.

By theorem 4.4 in [5],

$$m_{x,y}(z) = m_{y,x}(z) = m_{x \oplus z, y \oplus z}(0)$$

It was also shown by Vose that as long as the mutation rate is independently applied to each bit in the string, it makes no difference whether mutation is applied before or after crossover.

The matrix M with $(i, j)^{th}$ entry $m_{i,j}(0)$ is called the *mixing matrix*. The mixing matrix can provide mixing information for any string z just by changing how M is accessed.

Let σ_k be the permutation matrix defined by

$$(\sigma_k)_{i,j} = [i \oplus j = k]$$

The permutation σ_k corresponds to applying the mapping $i \mapsto i \oplus k$ to the subscripts of a given vector, that is

$$\sigma_k(x_0, ..., x_{n-1}) = (x_{0 \oplus k}, ..., x_{(n-1) \oplus k})$$

The mixing function $\mathcal{M}$ is now defined by the component equations

$$\mathcal{M}(x)_i = (\sigma_i x)^T M(\sigma_i x)$$

$\mathcal{M}(x)_i$ represents the probability that the string i is produced after muta-

tion and crossover are applied to a population vector x, which follows from:

$$\begin{aligned}
\mathcal{M}(x)_i &= (x_{0\oplus i}, ..., x_{(n-1)\oplus i})^T M(x_{0\oplus i}, ..., x_{(n-1)\oplus i}) \\
&= \sum_u x_{u\oplus i}(\sum_v M_{u,v} x_{v\oplus i}) \\
&= \sum_{u,v} x_{u\oplus i} M_{u,v} x_{v\oplus i} \\
&= \sum_{u,v} x_u x_v M_{u\oplus i, v\oplus i} \\
&= \sum_{u,v} x_u x_v m_{u\oplus i, v\oplus i}(0) \\
&= \sum_{u,v} x_u x_v m_{u,v}(i) \\
&= \sum_{u,v} x_u x_v \Pr[i \text{ is the child} \mid \text{parents } u, v]
\end{aligned}$$

Thus, the expected population vector at time $t+1$ can be computed from s^t by:

$$p^{t+1} = \mathcal{M}(s^t) = (\, (\sigma_0 s^t)^T M(\sigma_0 s^t), ..., (\sigma_{n-1} s^t)^T M(\sigma_{n-1} s^t)\,)$$

2.6 The SGA

Following the previous sections, the function $\mathcal{G}$, defining the simple genetic algorithm, can be formulated as the composition of the mixing and selection functions:

$$\mathcal{G} = \mathcal{M} \circ \mathcal{F}$$

Although the heuristic functions $\mathcal{F}$ and $\mathcal{M}$ can be shown to be focused under quite general conditions and formulas can be derived for their fixed points, the situation for $\mathcal{G}$ is considerably more complex. It is yet unknown when $\mathcal{G}$ is focused, although empirical evidence shows that this is often the typical case.

In section 6 we derive formulae for the differential $d\mathcal{G}_x$ to be used as the main analytical tool to explore the behavior of the genetic algorithm around its stationary points.

A computer program that computes $\mathcal{G}$ for any given mutation rate μ and crossover rate χ is included in Appendix A.

Chapter 3

Adding Learning to the Simple Genetic Algorithm

In this thesis we model the learning algorithm as a steepest ascent of each binary string in the population to the string with the highest fitness among its neighbors in the Hamming space. Each improvement changes at most one bit in the string being processed.

The learning algorithm is applied to the initial population processed by the genetic algorithm and to all successive generations just before the mutation and recombination operators are applied to obtain the next generation.

There are two learning strategies corresponding to Darwinian and Lamarckian evolution. In Darwinian evolution, lifetime events occurred to individual change its fitness but such changes are not incorporated back into its genome. In Lamarckian evolution, acquired changes are incorporated back into the genome and will be inherited to the offspring of the organism.

For both Darwinian and Lamarckian evolution, the learning algorithm can be outlined as follows:

For each vector x representing a binary string in the current population p:

1. Evaluate the fitness values of x and all its neighbors in the Hamming space.
2. Find the vector max with the best fitness out of all x's neighbors.
3. If $x = max$, no changes are applied to x.

4. If Lamarckian search strategy is used, replace x with max.

5. If Darwinian search strategy is used, change the fitness of x to equal $f(max)$.

3.1 Defining Learning Matrix

Let $d(x, y)$ be the Hamming distance between binary strings x and y. Let us now define L as a learning matrix of size $n \times n$, which represents one step of local search by:

$$L_{i,j} = \begin{cases} 1 & \text{if } i = \text{argmax}_{\{k|d(j,k)\leq 1\}} f(k) \\ 0 & \text{otherwise} \end{cases}$$

For example, let f be the following 4 bit fitness function:

$$\begin{array}{llll} f(0000) = 14 & f(0100) = 10 & f(1000) = 6 & f(1100) = 2 \\ f(0001) = 13 & f(0101) = 9 & f(1001) = 5 & f(1101) = 1 \\ f(0010) = 12 & f(0110) = 8 & f(1010) = 4 & f(1110) = 0 \\ f(0011) = 11 & f(0111) = 7 & f(1011) = 3 & f(1111) = 15 \end{array}$$

In this fitness landscape 1111 is the global maximum while 0000 is a local maximum. In this case, the matrix L is defined as follows:

$$L = \begin{pmatrix} 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \ldots & & & & & & & & & & & & & & & \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \end{pmatrix}$$

Note that the first row of the matrix has a 1 bit in those positions that represent strings that are attracted to string 0000, i.e. all strings that are different from 0000 by 1 bit for which the value of the function at 0000 is the highest compared to all other 1 bit changes. And in general, row i of the matrix flags those strings that are attracted to string i under one step of steepest ascent.

Let us define the learning rate η as the number of steps of local search applied to each population. The learning function $\mathcal{L}$ is now defined by:

$$\mathcal{L}(x) = L^{\eta} x$$

3.2 Darwinian Evolution

To model the Darwinian evolution, only the fitness function needs to be changed. A new fitness function f_D will be constructed from f by the following rule:

$$f_{D,i} = \max_{\{k|d(i,k)\leq 1\}} f(k)$$

For example, the following function f_D is constructed from the 4-bit fitness function described in section 3.1:

$$\begin{array}{ll} f_D(0000) = f(0000) = 14 & f_D(1000) = f(0000) = 14 \\ f_D(0001) = f(0000) = 14 & f_D(1001) = f(0001) = 13 \\ f_D(0010) = f(0000) = 14 & f_D(1010) = f(0010) = 12 \\ f_D(0011) = f(0001) = 13 & f_D(1011) = f(1111) = 15 \\ f_D(0100) = f(0000) = 14 & f_D(1100) = f(0100) = 10 \\ f_D(0101) = f(0001) = 13 & f_D(1101) = f(1111) = 15 \\ f_D(0110) = f(0010) = 12 & f_D(1110) = f(1111) = 15 \\ f_D(0111) = f(1111) = 15 & f_D(1111) = f(1111) = 15 \end{array}$$

Running a simple genetic algorithm on function f_D produces results identical to running a simple genetic algorithm on function f and using one iteration of steepest ascent to change the evaluation of each vector.

Therefore, in the case of Darwinian evolution the learning process is incorporated into the selection function $\mathcal{F}$, and the whole evolutionary process is described by the function $\mathcal{G} = \mathcal{M} \circ \mathcal{F}$.

3.3 Lamarckian Evolution

Under the Lamarckian strategy, the population distribution is altered at the beginning of each generation to model the effects of the local search.

Let $p^{t,L}$ be the population vector at time t after the learning process. This vector can be computed from p^t by

$$p^{t,L} = \mathcal{L}(p^t) = L^\eta p^t$$

For the example fitness function described in the previous section and learn-

ing rate $\eta = 1$, the redistribution of points in the search space occurs as follows:

$$\begin{aligned}
p^{t,L}_{0000} &= p^t_{0000} + p^t_{0001} + p^t_{0010} + p^t_{0100} + p^t_{1000} \\
p^{t,L}_{0001} &= p^t_{0011} + p^t_{0101} + p^t_{1001} \\
p^{t,L}_{0010} &= p^t_{0110} + p^t_{1010} \\
p^{t,L}_{0100} &= p^t_{1100} \\
p^{t,L}_{1111} &= p^t_{0111} + p^t_{1011} + p^t_{1101} + p^t_{1110} + p^t_{1111}
\end{aligned}$$

The probabilities of all the other vectors become 0. These vectors lie between basins of attraction, and thus have no representation after one iteration of steepest ascent.

Now we can extend the function $\mathcal{G}$ to include the effects of Lamarckian learning. $\mathcal{G}^L$ will be defined as the simple genetic algorithm function where a learning process defined by $\mathcal{L}$ is applied to each generation, i.e.

$$\mathcal{G}^L = \mathcal{M} \circ \mathcal{F} \circ \mathcal{L}$$

Chapter 4

Experiments

4.1 Empirical Results

At this stage, the hybrid genetic algorithms were used to track the expected string representation in an infinitely large population. We used the same fitness function described in section 3.1. In all experiments a one-point crossover was used, the crossover rate was 0.5 and mutation rate was 0.01. The algorithm was run until convergence was reached.

Figure 4.1 illustrates the results obtained using the SGA, SGA with Lamarckian learning and SGA with Darwinian learning. Each graph shows the proportions of strings 0000 (a local optimum) and 1111 (the global optimum) in the population during the evolutionary process.

The results indicate that both the simple genetic algorithm without learning and the Lamarckian evolution converge to a local optimum, whereas the Darwinian search strategy converges to the global optimum, but in a slower pace. The plain SGA converged after 75 generations, while SGA with Lamarckian learning converged after 60 generations and SGA with Darwinian learning converged after 95 generations.

One possible reason for the slow convergence of the Darwinian search strategy is that there is less variation in the fitness of strings in the space under the function f_D (all strings have a fitness between 10 and 15). A one-time scaling of the fitness was performed by subtracting 10 from each fitness value, which caused the genetic algorithm to converge faster (see last graph on Figure 4.1).

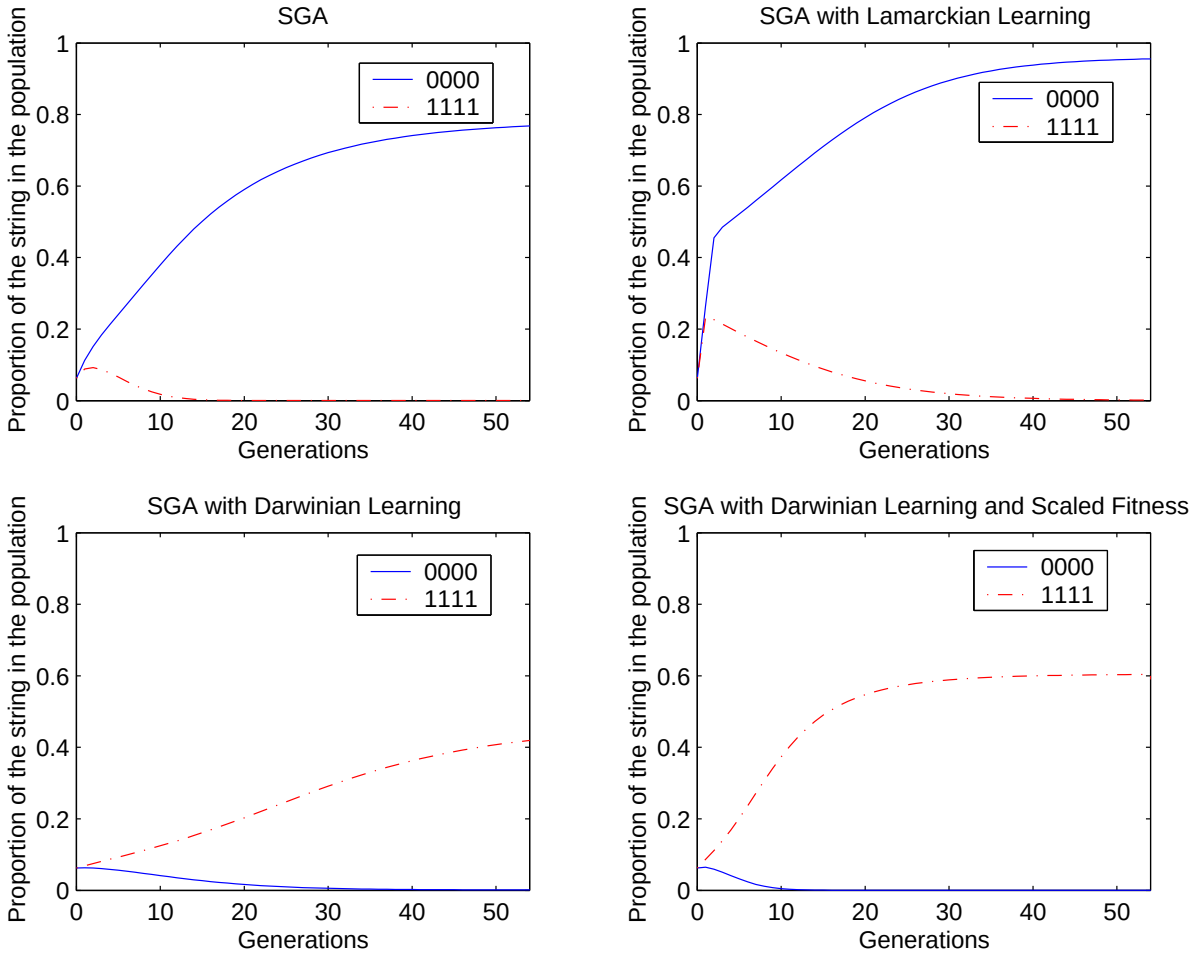

Figure 4.1: The proportion of the strings 0000 (local optimum) and 1111 (global optimum) in the population during the evolutionary process with and without Lamarckian and Darwinian learning.

4.2 Random Fitness Function

Next we created a random fitness function by assigning a random value between 0 and 20 to all points in space. Then we ran each of the three algorithms 50 times using the same fitness function, but starting from different random initial populations p. Figure 4.2 shows the number of times each of the algorithms has found the optimal solution, out of 50 runs.

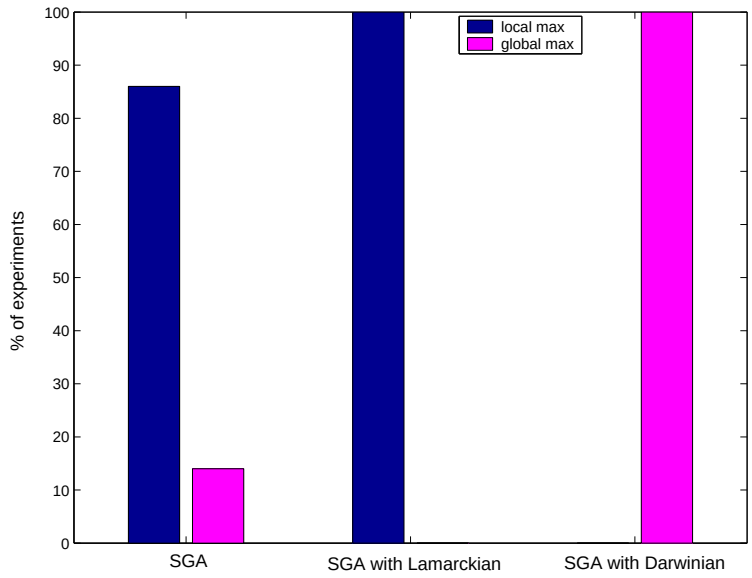

Figure 4.2: The percentage of times each of the algorithms converged to the global optimum and to a local optimum out of 20 times. The left graph represents SGA, the middle graph represents SGA with Lamarckian learning and the right graph represents SGA with Darwinian learning.

As we can see, the Darwinian strategy consistently converged to the optimal solution, while the Lamarckian strategy consistently converged to a sub-optimal solution, and the GA without learning converged most of the times to a sub-optimal solution.

4.3 Structured Fitness Function

In addition we wanted to compare the different models on a more structured fitness function, thus we chose a fitness function which gives a higher score to

strings that contain more 1 bits, i.e.

$$f_i = 2 * \text{the number of 1s in } i$$

for example, $f_{0011} = 4$ and $f_{1110} = 6$.

This function has only one local maximum, which is also its global maximum, at the point 1111. Figure 4.3 illustrates the results obtained using the SGA, SGA with Lamarckian learning and SGA with Darwinian learning averaged over 20 runs. Each graph shows the proportion of the string 1111 in the population during a typical evolutionary process, starting from a random initial population vector.

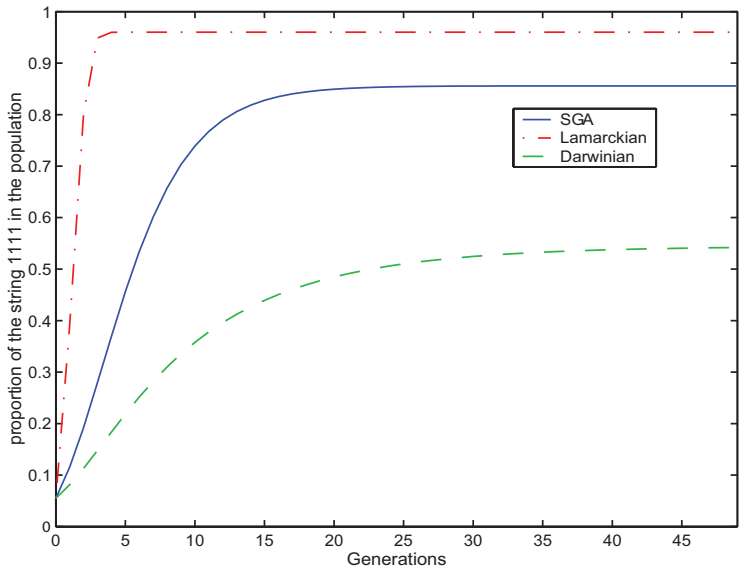

Figure 4.3: The proportion of the string 1111 (global optimum) in the population during the evolutionary process with and without Lamarckian and Darwinian learning, averaged over 20 runs.

In this case, since the fitness function contains only one global optimum, the simple genetic algorithm and the Lamarckian algorithm don't get trapped in a local minimum, and thus all three strategies converge to the global maximum in all the experiments. As in the previous experiment, the convergence rate of the Lamarckian strategy is much higher than the Darwinian strategy and the algorithm with no learning (it took only 4 generations on the average for it to converge).

The reason that the string 1111 has only 54% proportion in the final population vector of the Darwinian strategy is that all its that all its four neighbors in the Hamming space (the strings 0111, 1011, 1101 and 1110) receive the same fitness value as the vector at the global maximum, which makes the fitness function's surface flatter around this point.

Nevertheless, the vector at the global maximum receives a significantly higher share than its neighbors (see figure 4.4), due to Baldwin's effect discussed previously, which suggests that evolution tends to select individuals who have already at birth those useful features which would otherwise be learned during their lifetime.

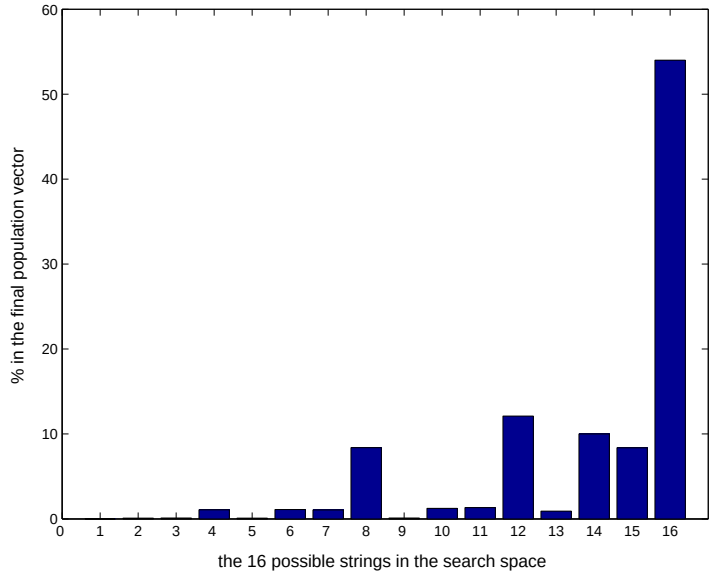

Figure 4.4: The proportion of the 16 possible strings in the final population vector of the Darwinian evolutionary process.

Chapter 5

Finite Population Model

Our next goal was to study how the standard GA converges to the analytical Vose model by exploring the behavior of the hybrid genetic algorithm on a finite population size. As defined in chapter 2, the finite genetic algorithm is represented in the Vose model by the transition rule τ. τ can be computed by sampling the function G using the following algorithm:

1. Select an initial random population vector x.
2. Compute the distribution represented by $\mathcal{G}(x)$.
3. Select $i \in 0, ..., n-1$ for the next generation with probability $\mathcal{G}(x)_i$.
4. Repeat the previous step until the next generation is complete.
5. Replace x by the population vector describing the new generation just formed.
6. If termination criterion not met, return to step 2.

This algorithm was implemented to compare the orbit $p, \tau(p), \tau^2(p), ...$ and the theoretical SGA orbit $p, \mathcal{G}(p), \mathcal{G}^2(p), ...$ for increasing population sizes under the different variants of SGA.

Figures 5.1 and 5.2 show the differences between the trajectories determined by τ and $\mathcal{G}$, as measured by the sum of squared distance between the vectors, for population sizes $r = 100$ and $r = 1000$, using a random fitness function.

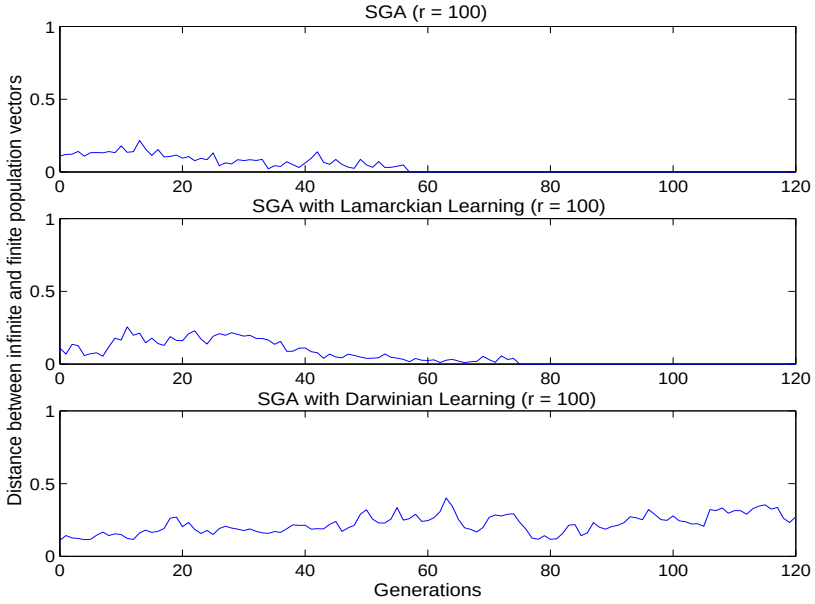

Figure 5.1: The Euclidean distance between the finite population vector τ and the infinite population vector $\mathcal{G}$ for population size $r = 100$.

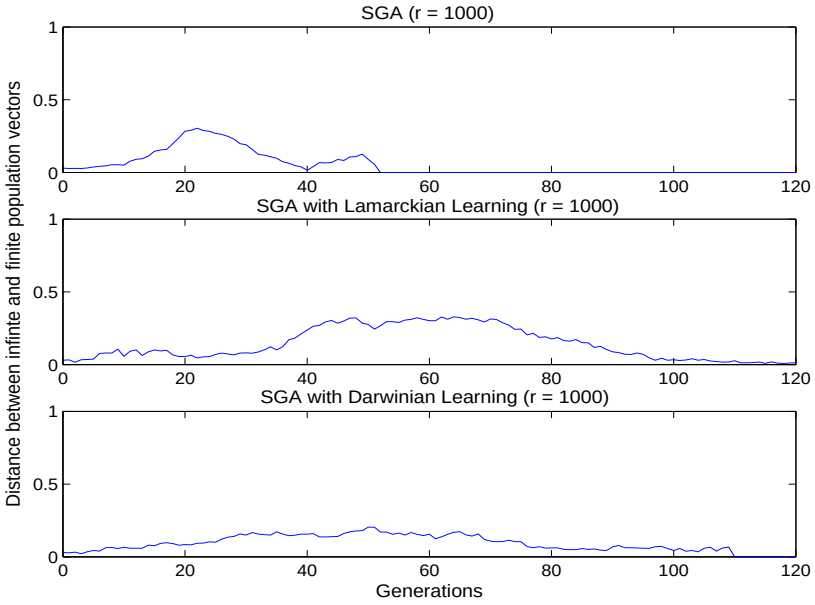

Figure 5.2: The Euclidean distance between the finite population vector τ and the infinite population vector $\mathcal{G}$ for population size $r = 1000$.

When the population size is 100, a general agreement between τ and $\mathcal{G}$ is observed for both plain SGA and SGA with Lamarckian learning, while the SGA with Darwinian learning does not converge to its asymptotical fixed point. When the population size is 1000, all three methods converge. The same experiment has been repeated for increasing population sizes $r = 3000, 10000, 100000$. As $r \to \infty$, the trajectories of τ and $\mathcal{G}$ become close to each other and all three algorithms converge to their asymptotical fixed points.

These results suggest that in small population sizes the Lamarckian evolution outperforms the Darwinian evolution, while the Darwinian evolution has a clear advantage in larger population sizes. This can be explained by the observation that low genetic diversity in small populations helps Lamarckian evolution reach its theoretical fixed point faster, while large population variety helps the Darwinian algorithm explore the search space more efficiently.

Chapter 6

The Spectrum of $d\mathcal{G}$

In this chapter we obtain the formulas for calculating the differential $d\mathcal{G}$ for the hybrid genetic algorithms described earlier.

The importance of the differential $d\mathcal{G}$ lies in the fact that it represents the best linear approximation to $\mathcal{G}$ near a given point. Therefore, the asymptotical behavior of $\mathcal{G}$ in the areas near its fixed points can often be determined by the eigenvalues of the differential $d\mathcal{G}$ at the stationary point. For instance, in a stable fixed point we expect all eigenvalues of $d\mathcal{G}$ to be less than 1. The fixed points of G indicate areas where there is little pressure for change, thus it is expected that the SGA will spend more time near such regions of the search space.

We will first obtain the formulas for the derivative of $\mathcal{G}$ for the non-Lamarckian case (i.e. plain SGA or SGA with Darwinian learning) and then we will develop the formula for $d\mathcal{G}^{\mathcal{L}}$ for the Lamarckian case. Finally, we will use these formulas to compare the behavior of the different genetic algorithms in the vicinity of their fixed points.

6.1 Non-Lamarckian Evolution

From section 2.6 we have the following formula for the function $\mathcal{G}$:

$$\mathcal{G} = \mathcal{M} \circ \mathcal{F}$$

For proportional selection,

$$\mathcal{F}(x) = \frac{diag(f)x}{f^T x}$$

Let us denote the Jacobian matrix of the function $\mathcal{F}(x)$ by $d\mathcal{F}_x$.
By theorem 7.1 in [5]:

$$d\mathcal{F}_x = \frac{f^T x \cdot diag(f) - diag(f) x f^T}{(f^T x)^2}$$

By theorem 6.13 in [5], considered as a function $\mathcal{M} : \mathcal{R}^N \to \mathcal{R}^N$, the Jacobian matrix of $\mathcal{M}(x)$ is:

$$d\mathcal{M}_x = 2 \sum_u \sigma_u^T M^* \sigma_u x_u$$

where M^* is the twist of matrix M, i.e. its i, jth entry is $M_{i \oplus j, i}$.

According to the chain rule of multivariate functions, the Jacobian matrix of a composite function is the product of the Jacobian matrices of the two functions. Thus, the differential $d\mathcal{G}$ may be calculated using the following formula:

$$d\mathcal{G}_x = d\mathcal{M}_{\mathcal{F}(x)} \cdot d\mathcal{F}_x$$

6.2 Lamarckian Evolution

As defined in section 3.2, the formula for $\mathcal{G}^L$ incorporates the learning operator $\mathcal{L}$, thus:

$$\mathcal{G}^L = \mathcal{M} \circ \mathcal{F} \circ \mathcal{L}$$

Since the learning matrix L represents a linear transformation, the Jacobian of the function $\mathcal{L}(x)$ is precisely L. Thus, the differential $d\mathcal{G}^L$ may be calculated by the following formula:

$$d\mathcal{G}^L_x = d\mathcal{M}_{\mathcal{F}(\mathcal{L}(x))} \cdot d\mathcal{F}_{\mathcal{L}(x)} \cdot L$$

6.3 Fixed Points Analysis

We used the formulas for $d\mathcal{G}$ to compare the asymptotic behavior of the different genetic algorithms near their fixed points. We conducted a series of experiments using a random fitness function and the same crossover and mutation settings as in the previous sections.

In all experiments the differential $d\mathcal{G}$ at the initial point had some eigenvalues greater than 1 whereas at the fixed point all eigenvalues were smaller

than 1, which means that in all experiments the algorithms converged to a stable fixed point.

Let us denote the vector of eigenvalues of the Jacobian matrix $d\mathcal{G}$ at generation i by $spec(dG)_i$. A typical example for the spectrum of $d\mathcal{G}$ at the beginning (generation 0) and at the end of the evolutionary process (generation n) is illustrated below:

$$\begin{aligned} spec(d\mathcal{G})_0 &= (0.000, 1.802, 1.661, 1.582, 1.406, 1.172, 1.063, \\ &\quad 0.829, 0.793, 0.513, 0.323, 0.249, 0.000, 0.068, 0.085, 0.076) \\ spec(d\mathcal{G})_n &= (0.000, 0.943, 0.734, 0.754, 0.581, 0.564, 0.525, \\ &\quad 0.387, 0.333, 0.281, 0.176, 0.000, 0.010, 0.049, 0.027, 0.028) \end{aligned}$$

As evident, all eigenvalues have modulus less than 1 at the fixed point, i.e. the spectral radius of the matrix is: $\rho(d\mathcal{G}) = \max|\lambda| < 1$, which means that in the area of the fixed point, $d\mathcal{G}$ is a convergent matrix and $\lim_{n\to\infty}(dG)^n = 0$.

Comparing the spectrums of $d\mathcal{G}$ at the fixed points has revealed some differences between the different algorithms, as illustrated in figure 6.1. This figure shows that the most significant change in the spectrum of $d\mathcal{G}$ occurs within the genetic algorithm without learning. This can be explained by the fact that the distance the SGA has to travel in order to reach a fixed point is much greater than the algorithms which use learning as a guidance.

It should also be noted that the Lamarckian evolution has the smallest eigenvalues at the fixed point. This view supports the results obtained in the previous sections which showed that the Lamarckian evolution had the strongest attraction to its fixed points and thus the fastest convergence rate.

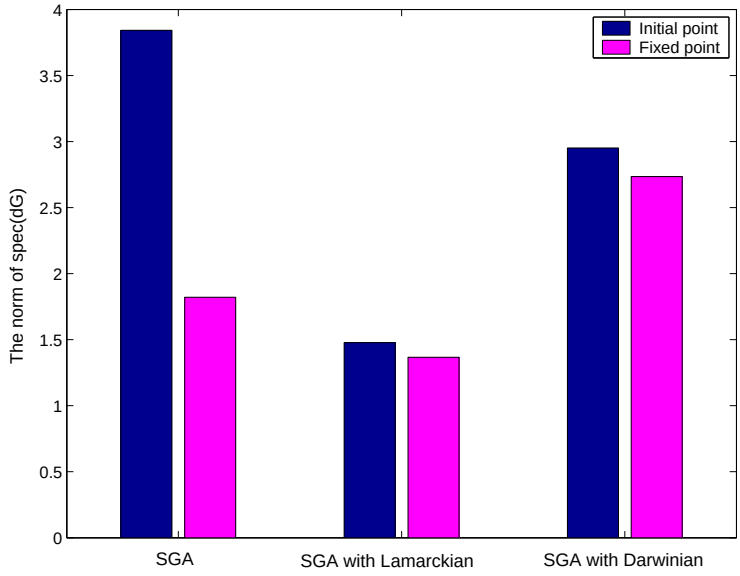

Figure 6.1: The norm of the spectrum of $d\mathcal{G}$ at the initial points and at the fixed points. The left graph represents SGA, the middle graph represents SGA with Lamarckian learning and the right graph represents SGA with Darwinian learning.

Chapter 7

Numerical Function Optimization Test

Finally, we wanted to test if the results obtained earlier, mainly that the Lamarckian evolution works faster but Darwinian evolution works better on the long run, hold for other examples and for regular GA not only in the Vose model. Thus, we tested the effects of adding Darwinian and Lamarckian search strategies to a simple genetic algorithm in a numerical function optimization test. For this test we have chosen the Schwefel numerical function [19], which has been widely used as a benchmark problem in evolutionary optimization literature:

$$f(x) = 418.989 * n + \sum_{i=1}^{n} -x_i sin(\sqrt{|x_i|}) \quad x_i \in [-512, 511]$$

where n indicates the number of variables.

The surface of Schwefel's function is composed of a great number of peaks and valleys (figure 7.1 shows the function for $n = 2$). The function has a second best minimum far from the global minimum where many search algorithms are trapped. Moreover, the global minima are located near the bounds of the domain - at the points $x = (-420.9687, ..., -420.9687)$ and $x = (420.9687, ..., 420.9687)$, where $f(x) = 0$.

We have used the Schwefel function with $n = 20$ to compare the performance of the standard genetic algorithm, Lamarckian learning and Darwinian learning. We used real-valued representations, i.e. an alphabet of floats, with uniform mutation and simple crossover. Local optimization was employed in the form of steepest descent.

Figure 7.2 shows the evolutionary graphs for each genetic algorithm, using a

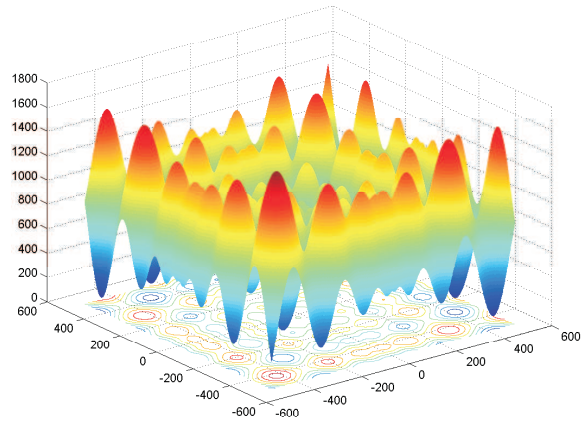

Figure 7.1: The Schwefel function for $n = 2$. The global minima are located at (-420.9687, -420.9687), (420.9687, 420.9687).

population size of 100. Similar results were observed for larger population sizes. The graph shows that the Lamarckian search results in faster improvement in the early stages of the evolutionary run, however the average best solution for the Darwinian search gains superiority after a certain period of time. This supports our earlier conclusions that if one wishes to obtain results quickly, the Lamarckian strategy is the favorable choice, however the Darwinian strategy tends to have better long-term effects.

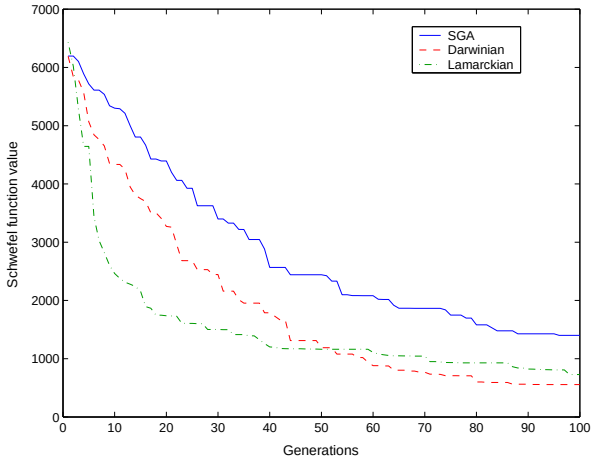

Figure 7.2: The function value of the best string in the population running the genetic algorithms on the Schwefel function for $n = 20$.

Chapter 8

Conclusions

In this thesis we have shown how learning algorithms can be integrated into the Vose dynamical systems model of the simple genetic algorithm. We have succeeded in using the integrated model to demonstrate some differences in the behavior of the SGA under different learning schemes.

The results clearly indicate that a Darwinian search strategy as defined in this thesis can be more effective in the long run than a Lamarckian strategy employing the same local search strategy, especially in large population sizes. However, in most cases, the Darwinian search strategy is slower than the Lamarckian search.

We have also supplied new mathematical formulas to analyze the asymptotic behavior of the different genetic algorithms near their fixed points. Using these formulas we have shown that the attraction of the Lamarckian strategy to its fixed points is much stronger than the attraction of the Darwinian strategy. This allows the Lamarckian strategy to make faster improvements, whereas it gives the Darwinian strategy the opportunity to explore more extensive areas of the search space, thus enabling it to reach the global optimum in cases where the Lamarckian strategy converges to a local optimum.

In future we would like to extend the analysis model in order to cover other aspects of the interaction between evolution and learning, such as changing the environmental conditions during the evolution, and including cost of learning. Another interesting aspect of the model to investigate is whether it can be used to predict on which types of functions or problems learning can accelerate evolution, and how various parameters such as mutation rate, learning rate, or population size can affect the performance of both Lamarckian and Darwinian evolutions.

Bibliography

[1] J.M. Baldwin. A new factor in evolution. *American Naturalist*, 30, 441-451, 1896.

[2] J. Watson and J. Wiles The rise and fall of learning: A neural network model of the genetic assimilation of acquired traits. In: *Proceedings of the 2002 Congress on Evolutionary Computation (CEC 2002)*, 600-605, 2002.

[3] P.D. Turney How to shift bias: Lessons from the baldwin effect. *Evolutionary Computation*, 4, 271-295, 1996.

[4] T. Arita, R. Suzuki Interactions between learning and evolution: The outstanding strategy generated by the baldwin effect. In: *Proceedings of Artificial Life VII, MIT Press*, 196-205, 2000.

[5] M. Vose. The Simple Genetic Algorithm: Foundations and Theory. MIT Press, Cambridge, MA, 1999.

[6] G.E. Hinton and S.J. Nowlan. How learning can guide evolution. *Complex Systems*, 1, 495-502, 1987.

[7] S. Nolfi and D. Parisi. Learning to adapt to changing environments in evolving neural networks. *Adaptive Behavior*, 5, 75-97, 1996.

[8] D. Floreano, F. Mondada. Evolution of plastic neurocontrollers for situated agents. *Animals to Animats*, 4, 1996.

[9] J. Paredis. Coevolutionary Life-time Learning. *PPSN*, 72-80, 1996.

[10] D. Curran, C. O'Riordan and H. Sorensen. Evolutionary and Lifetime Learning in Varying NK Fitness Landscape Changing Environments: An Analysis of Both Fitness and Diversity. *AAAI*, 706-711, 2007.

[11] D. Whitley, S. Gordon and K. Mathias. Lamarckian Evolution, the Baldwin effect and function optimization. In *Y. Davidsor, H. Schwefel, and R. Manner, editors, Parallel Problem Solving from Nature-PPSN III*, Springer-Verlag, 6-15, 1994.

[12] D. Ackley and M. Littman A case for Lamarckian evolution. In *C.G. Langdon (Ed.), Proceedings of Artificial Life III, SFI Studies in the Sciences of Complexity*, Addison-Wesley, 1994.

[13] C. Houck, J.A. Joines, M.G. Kay and J.R. Wilson. Empirical investigation of the benefits of partial Lamarckianism. In *Evolutionary Computation*, 5(1), 31-60, 1997.

[14] T. Sasaky and M. Tokoro Comparison between Lamarckian and Darwinian evolution on a model using neural networks and genetic algorithms. *Knowledge and Information Systems*, 2(2), 201-222, 2000.

[15] I. Paenke, B. Sendhoff, J. Rowe and C. Fernando On the Adaptive Disadvantage of Lamarckianism in Rapidly Changing Environments In *Advances in Artificial Life, 9th European Conference on Artificial Life*, Springer-Verlag, 355-364, 2007.

[16] J.H. Holland. Adaptation in Natural and Artificial Systems. Ann Arbor: The University of Michigan Press, 1975. Reprinted by MIT, 1992.

[17] D.E. Goldberg. Genetic Algorithms in Search, Optimization and Machine Learning. Addison-Wesley, MA, 1989.

[18] D. Whitley, R. Das and C. Crabb Tracking primary hyperplane competitors during genetic search. In *Anals of Mathematics and Artificial Intelligence*, 6, 367-388, 1992.

[19] H. P. Schwefel. Numerical Optimization of Computer Models. In *English translation of Numerische Optimierung von Computer-Modellen mittels der Evolutionsstrategie*, John Wiley & Sons, 1981.

Appendix A

Matlab code

```
% SGA.m
%
% The main function of the algorithm - it runs the SGA on an initial
% population distribution vector until convergence is reached.
function SGA()

len = 4;                  % length of the binary strings
n = len ^ 2;              % the dimension of the search space
max_iterations = 1000;    % max number of iterations
epsilon = 0.0000001;      % convergence criterion
mutation_rate = 0.01;     % mutation rate
crossover_rate = 0.5;     % crossover rate
learning_rate = 1;        % learning rate
learning_type = 1;        % a flag indicating which type of learning to use
                          % 0 - no learning, 1 - Lamarckian, 2 - Darwinian

% create an initial distribution vector
p = ones(n, 1);
p = p / n;

f = fitness(n, len)

% compute the mixing matrix (includes mutation and recombination)
mixing_mat = MixingMatrix(len, mutation_rate, crossover_rate);

% compute the learning matrix
if learning_type == 1 % Lamarckian
```

```
    learning_mat = LearningMatrix(len, f, learning_rate)
elseif learning_type == 2 % Darwinian
    f = DarwinianFitnessFunction(len, f)
end

% run the algorithm
for i = 1:max_iterations
    display(['generation #' int2str(i)]);

    % store the proprotion of the optimum points
    y(1, i) = p(1);
    y(2, i) = p(n);

    % apply selection scheme to the population vector
    p_after_selection = selection(p, f);

    % apply the mixing matrix to the population vector
    p_after_mixing = mix(mixing_mat, p_after_selection, len);

    % apply learning
    if learning_type == 1 % Lamarckian
        p_after_learning = learn(learning_mat, p_after_mixing);
    else
        p_after_learning = p_after_mixing;
    end

    new_p = p_after_learning;
    p_new = new_p' % for printing

    % check for convergence
    diff_p = sum((p - new_p) .^ 2);
    if (diff_p < epsilon)
        break;
    end
    p = new_p;
 end

 if i == max_iterations
     display('no convergence');
 else
```

```
    display(['converged after ' int2str(i) ' generations']);
end

% display graph with the proportion of the optimum points
gen = [0:1:length(y)-1];
plot(gen, y(1,:), '-b', gen, y(2,:), '-.r');
axis([0, length(y) - 1, 0, 1]);
legend('0000', '1111');
xlabel('Generations');

if learning_type == 0
   title('SGA');
elseif learning_type == 1
   title('SGA with Lamarckian Learning');
else
   title('SGA with Darwinian Learning');
end

% Define the fitness function
function f = fitness(n, len)
curr_fitness = 14;
for i = 1: n - 1
    f(i) = curr_fitness;
    curr_fitness = curr_fitness - 1;
end
f(n) = 15;
```

```
% MixingMatrix.m
%
% This function computes the mixing matrix for a given mutation rate
% and crossover rate
function M = MixingMatrix(len, mutation_rate, crossover_rate)

n = 2 ^ len;

% compute the mutation vector
u = mutation_vector(mutation_rate, len);

% compute the crossover vector
c = crossover_vector(crossover_rate, len);

M = zeros(n, n);

% iterate over all the possible pairs in the population (x, y)
for x = 0:n-1
    xb = dec2binvec(x, len);

    for y = 0:n-1
        yb = dec2binvec(y, len);

        % iterate over all the possible mutation masks
        for j = 0:n-1
            jb = dec2binvec(j, len);

            % iterate over all the possible crossover masks
            for m = 0 : len - 1
                k = 2 ^ m - 1;
                kb = dec2binvec(k, len);

                % if any of the children created after mutation and crossover
                % are applied to the pair x,y is the zero vector, add the
                % probability of the event to M
                child1 = xor(xor(xb .* kb, not(kb) .* yb), jb);
                if isequal(child1, zeros(1, len))
                    M(x+1, y+1) = M(x+1, y+1) + u(j+1) * c(k+1) / 2;
                end
```

```
                child2 = xor(xor(xb .* not(kb), kb .* yb), jb);
                if isequal(child2, zeros(1, len))
                    M(x+1, y+1) = M(x+1, y+1) + u(j+1) * c(k+1) / 2;
                end
            end
        end
    end
end

% This function computes the mutation vector
function u = mutation_vector(mutation_rate, len)

for i = 0 : 2 ^ len - 1
    ib = dec2binvec(i, len);
    u(i+1) = (mutation_rate) ^ (ones(1, len) * ib') *
     (1 - mutation_rate) ^ (len - ones(1,len) * ib');
end

% This function computes the crossover vector
function c = crossover_vector(crossover_rate, len)

c = zeros(1, 2 ^ len);
c(1) = 1 - crossover_rate;
for k = 1 : len - 1
    c(2 ^ k) = crossover_rate * (1 / (len - 1));
end
```

```
% LearningMatrix.m
%
% Build the learning matrix according to the fitness vector and the
% learning rate
function L = LearningMatrix(len, f, learning_rate)

n = len ^ 2;
L = zeros(n, n);

for i = 0: n - 1

    ib = dec2binvec(i, len);

    % Find the maximum among all neighbors of string i with hamming
    % distance 1
    max = f(i + 1);
    max_index = i + 1;

    for j = 1: len

        nb = ib;
        nb(j) = xor(nb(j), 1);
        n = binvec2dec(nb, len);

        if f(n + 1) > max
            max = f(n + 1);
            max_index = n + 1;
        end
    end

    L(max_index, i + 1) = 1;
end

L = L ^ learning_rate;
```

```
% DarwinianFitnessFunction.m
%
% Build the fitness function induced by the Darwinian learning algorithm
function fd = DarwinianFitnessFunction(len, f)

n = len ^ 2;

for i = 0: n - 1

    ib = dec2binvec(i, len);

    % Find the maximum among all neighbors of string i with hamming
    % distance 1
    max = f(i + 1);

    for j = 1: len

        nb = ib;
        nb(j) = xor(nb(j), 1);
        n = binvec2dec(nb, len);

        if f(n + 1) > max
            max = f(n + 1);
        end
    end

    fd(i + 1) = max;
end
```

```
% selection.m
%
% This function returns the result of applying proportional selection
% on x using fitness function f
function F = selection(x, f)

F = diag(f) * x / (f * x);

% mix.m
%
% This function returns the result of applying mutation and crossover
% to the vector x using the mixing matrix
function M = mix(mixing_mat, x, len)

n = 2 ^ len;

M = zeros(n, 1);
for i = 1 : n

    % compute the permutation (x0,...,x[n-1]) -> (x(0 xor i),...,
    % x(n-1 xor i))
    perm_x = zeros(n, 1);
    for j = 1 : n
        k1 = dec2binvec(i - 1, n);
        k2 = dec2binvec(j - 1, n);
        k = xor(k1, k2);

        perm_x(j) = x(binvec2dec(k, n) + 1);
    end

    % compute the ith component of the vector after mixing
    M(i) = perm_x' * mixing_mat * perm_x;
end

% learn.m
%
% This function applies the learning matrix on a given vector p
function L = learn(learning_mat, p)

L = learning_mat * p;
```

```
%F_Derivative.m
%
%This function computes the derivative of the fitness function f at
%point x
function dF = F_Derivative(f, x)

a = f' * x * diag(f) - diag(f) * x * f';
b = (f' * x) ^ 2;
dF = a / b;

%M_Derivative.m
%
%This function computes the derivative of mixing matrix M at point x
function dM = M_Derivative(tM, x, len)

n = 2 ^ len;

sum_mat = zeros(n, n);

for u = 0:n-1
    P = PermutationMatrix(u, len);
    sum_mat = sum_mat + (P * tM * P .* x(u + 1));
end

dM = 2 * sum_mat;

%G_Derivative.m
%
%This function computes the derivative of function G at point x
%using the chain rule.
function dG = G_Derivative(f, x, len, tM)

dF = F_Derivative(f, x);
Fx = selection(x, f);
dM = M_Derivative(tM, Fx, len);
dG = dM * dF;
```

```
%G_Lamarckian_Derivative.m
%
%This function computes the derivative of function G at point x
%taking into account the learning matrix L.
function dGL = G_Lamarckian_Derivative(f, x, len, tM, L)

Lx = learn(L, x);
dF = F_Derivative(f, Lx);
FLx = selection(Lx, f);
dM = M_Derivative(tM, FLx, len);
dGL = dM * dF * L;  % Jacobian of Lx = L since L is a linear map
```