Genetic Algorithms with Chaotic Population Dynamics

Thomas Harrison Nelson
University of Colorado at Boulder, thomasharrisonnelson@gmail.com

Follow this and additional works at: http://scholar.colorado.edu/csci_gradetds

Part of the Theory and Algorithms Commons

Recommended Citation
Genetic Algorithms with Chaotic Population Dynamics

by

Thomas Nelson

B.S., University of Texas, 2007

A thesis submitted to the
Faculty of the Graduate School of the
University of Colorado in partial fulfillment
of the requirements for the degree of
Masters of Science in Computer Science

December 8, 2010
This thesis entitled:
Genetic Algorithms with Chaotic Population Dynamics
written by Thomas Nelson
has been approved for the Department of Computer Science

__________________________
Elizabeth Bradley

__________________________
Robin Dowell

__________________________
Richard Byrd

Date ______________________
The final copy of this thesis has been examined by the signatories, and we find that both the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline.
Nelson, Thomas (M.S., Computer Science)
Genetic Algorithms with Chaotic Population Dynamics
Thesis directed by Elizabeth Bradley

In this thesis I demonstrate a novel application of chaotic dynamics to evolutionary algorithms, specifically in population size management. Typical evolutionary algorithms require a population size to be set as a parameter, which remains constant throughout execution. I created a new algorithm that can vary the population size chaotically or periodically, and do a series of performance tests comparing static, periodic, and chaotic population control. The problems targeted in these tests are chosen from both continuous and discrete multi-dimensional domains. I find that both chaotic and static population control perform well in certain situations; my evidence suggests that periodic population control is rarely a good choice. I also present additional analysis on the effects of the population dynamics and how they relate to mean population size and variance in the performance results.
# Contents

1 Introduction .......................................................... 1

2 Genetic Algorithms .................................................. 3

3 Chaotic Population Dynamics in a Genetic Algorithm .......... 7

4 Experimental Design ................................................ 13

5 Results and Discussion .............................................. 20
   5.1 Bitstring problem .................................................. 20
   5.2 Sine, Schwefel, and Traveling Salesman problems ............. 23

6 Conclusion .................................................................. 29
List of Figures

3.1 The continuous logistic map, showing exponential growth followed by asymptotic growth over time .............................................................. 8
3.2 trajectories of the logistic map for various values of r ..................................... 10
3.3 Bifurcation Diagram of the logistic map ............................................................ 11

4.1 Sine function optimization example ................................................................. 16
4.2 Two of the ten axes of the Schwefel function optimization example. All ten dimensions are symmetrical and look the same as above. .................................................. 18

5.1 Static GA fitness progress over time for the bitstring problem: two example runs with different population sizes ................................................ 21
5.2 The effect of population size and number of function evaluations on fitness for the bitstring problem .............................................................. 22
5.3 The effect of population size and growth rate (r) on fitness for the bitstring problem .............................................................. 22
5.4 Boxplots comparing fitness distributions for different population dynamics for the sine function .............................................................. 24
5.5 Boxplots comparing fitness distributions for different population dynamics for the Schwefel function .............................................................. 25
5.6 Boxplots comparing fitness distributions for the traveling salesman problem .............................................................. 25
5.7 Fitness distribution histograms for periodic and chaotic GAs for the sine problem, pop = 12 .............................................................. 27
5.8 Typical example GA runs over time for the sine problem, r = 2.8 (static), 3.3 (periodic), and 3.8 (chaotic) .............................................................. 28
Chapter 1

Introduction

Optimization is the general problem of finding the maximum or minimum value of a function: given some $f : A \rightarrow \mathbb{R}$, we wish to find some element $x \in A$ such that $\forall y \in A, f(x) \leq f(y)$. Many optimization strategies exist; quadratic fitting algorithms, the Nelder-Meade Algorithm, Simulated Annealing, genetic algorithms, and convex programming are all optimization techniques with slightly different goals and methods (see Nocedal 1999 [7] and Kincaid 2002[4] for a thorough treatment). This thesis describes a novel application of chaos theory to genetic algorithms, and evaluates its practical application.

Rather than a single algorithm for solving a specific problem, genetic algorithms may be thought of as a general scheme for solving a wide class of problems, and that is their great strength. With small modifications a genetic algorithm may be applied to the traveling salesman problem, automatic code generation, continuous function optimization, or simulations. As a cost for this generality, genetic algorithms are often outperformed by more domain-specific solvers. Genetic algorithms do solve the traveling salesman problem, for instance, but not nearly as well as programs designed for that specific task. In domains where derivatives are available, other methods making use of that information are usually more appropriate. But in domains where derivatives do not exist, or problem-specific solvers cannot be constructed, a genetic algorithm offers a good chance at finding the best solution.

Rather than attempt to compare genetic algorithms (GAs) to other optimization algorithms, this thesis focuses specifically on the problem of population size in the context of genetic algorithms. I introduced a new genetic algorithm which, rather than maintaining the same population size at each generation, can vary the population size either periodically or chaotically. I did this to see how a variable population size affects performance on a variety of problems. Although every genetic algorithm user must make a choice about
population size, little research or supporting theory exists to help with this choice. GA users typically rely on instinct in choosing parameters for their system. In particular, the choice to keep the population size static throughout the GA run is simple, but not always correct, as I have shown.

I used my new genetic algorithm to compare the performance of different population dynamics. If one population strategy typically solves a problem more efficiently than another, genetic algorithm users can use that strategy in solving new problems and expect it to do well. To that end, the main experiments of the paper involve running the genetic algorithm on a few test problems many times, and using statistical analysis to estimate the expected solutions the algorithm will produce. Additionally I analyzed a subset of the results and specific runs in more detail to gain some insight into how dynamic changes create specific effects in the population. That understanding contributes to an important to the goal of this thesis: even if the new method appeared significantly better than a traditional genetic algorithm, knowing that without understanding why is an unsatisfying result. Rather than just show how black box A performs compared to black box B on some test function, I wanted to understand the causal link between changing population dynamics and an increased or decreased likelihood of success. To this end I applied some descriptive statistics to my results and drew a few interesting conclusions from them.

In chapter 2 I provide background material about how genetic algorithms work, and describe some of the key choices in genetic algorithm design. In chapter 3, I motivate and explain the use of chaotic dynamics in genetic algorithms. In Chapter 4, I describe the experimental design, including difficulties in choosing test problems and evaluation metrics and how I address those problems. Chapter 5 explains my results and discusses their significance. I conclude in chapter 6 with an overview and possibilities for future work.
Chapter 2

Genetic Algorithms

Genetic algorithms are a class of optimization tools influenced by the study of genetics and natural selection in biology. There are many algorithms that can fit under this heading, but they have the following basic structure in common:

1. Choose a set of points in the search space, called organisms to draw the parallel to biology,

2. Evaluate each organism by some criterion, called the fitness function, and

3. Create a new working set using the best performing organisms, usually using crossover.

A typical genetic algorithm thus needs three things: a way to generate the initial organism, a fitness function, and a crossover operation. For example, imagine our organism consists of a string of 8 bits. The initial generation would generate a set of random bitstrings using any random number generator. The fitness function could be any function that takes a byte and returns a real value as a result. It could be something very simple, such as matching specific bits, or something more complex, where perhaps the bits encode a cellular automata or Turing machine that then solves some problem. In terms of crossover, say we have two bitstrings, which look like so:

```
11011110
00100110
```

One method would be to choose a random integer between 0 and 8, take that many bits from the first string, and the rest from the second string. Say we choose 3:

```
110-----
00100110
```
This is typically called single-point crossover. An alternate method might be to choose randomly from each string at each bit position, giving something like this:

```
1-0----0
-0-0011-
```

```
10000110
```

This is called uniform crossover. Note that either way, the final “110” is preserved, since it appears in both parents. Binary representations have the advantage of having a clear and well-studied crossover, and almost any solution type can be converted into a binary representation. However, for many problems a conversion to bits is either impractical or loses important information about the problem space, so sometimes GA users create a custom crossover function to suit the problem.

It’s important also to consider how encoding the problem creates a mapping from the problem space to the fitness function. Often a problem can be expressed with more than one organism encoding and crossover function, which may cause the fitness function to have very different searchability properties.

Here is the pseudocode for a naive genetic algorithm:

```python
generate random initial population
while not finished:
    for organism in population:
        calculate organism fitness
    sort population by fitness
    pair randomly top K organisms in population
    run crossover to generate new population
```

There are a few problems with this approach. First, most crossover operators remove variation from the population, but never introduce it. Consider our bit example above. If the whole population has “110” at the end, the algorithm cannot introduce “111”, even if that is the optimal value. Because of this, most algorithms introduce a small amount of randomness after crossover, often called a mutation operator. For
example, we might with small probability $P$ flip each bit in the child produced. For some problems it can be assumed that the initial population contains enough variation that a mutation operator is not necessary.

Second, and perhaps more seriously, choosing the top $K$ organisms in the population at every step will also reduce variation, and risks reaching a local optimum too soon. Typically the problems that genetic algorithms are used to solve contain many non-optimal peaks and plateaus, so it is important to discourage the algorithm from converging too early. There are a few approaches to solving this problem. One, called the roulette method, is to choose each organism for crossover with a probability proportional to its fitness: essentially spinning a roulette wheel where each organism’s slice is the size of its relative fitness. This way even the worst organism in the population has some chance at being selected, but most of the selections will be of high-fitness organisms. Another method, called the tournament approach, is to randomly choose $N$ organisms and pull the best organism from that selection. So if $N$ equals 3, we will randomly choose three organisms from the population and choose the one with highest fitness to be a parent.

Choices for which organisms to cross, how to cross them, and how to mutate them all have an effect on algorithm performance. Another point not yet discussed is the choice of population size. As population size decreases, the risk of premature convergence increases. But if population size is too large, many expensive evaluations will be spent on organisms that do not contribute to final solution. I conjecture that the best population size depends not only on the problem, but where the solver is in the search space, which changes over the course of the algorithm. For this reason some researchers have looked into population sizes that change dynamically over the course of optimization, and that is the focus of this thesis. One can view an increase in population size as allowing increased variability into the strategy set being searched; similarly, a decrease in population size restricts the search to only the very best known strategies. Sometimes larger populations are necessary to overcome local optima, but if the population is too large, search will be very slow. Dynamic population size may be a way to balance these effects.

Several genetic algorithm variants have been proposed with non-constant population size. One of the first successful ones was GAVaPS (Arabas 1994[2]), which assigned each individual in the population a lifespan based on its fitness. (Koumousis 2006[5]) used a saw-tooth approach, where the population slowly declines and then is repopulated with a random influx. Taking an idea from evolutionary biology, Hu and Banzaf [3] try to estimate whether the population is undergoing positive selection (primarily adding new traits) or negative selection (primarily removing deleterious traits). This is done by measuring the rate at which new mutations are accepted into the population. Generally positive selection does better with a larger population, and negative selection does better with a smaller population. Hu and Banzaf count the rate of
genetic substitutions in successful offspring at each generation and use that to alter population size. All three of these approaches show some improvement over static-population genetic algorithms, although irritatingly all three use different metrics for evaluation.
Chapter 3

Chaotic Population Dynamics in a Genetic Algorithm

The chaotic genetic algorithm that was investigated in this project draws on an idea from population ecologists; in natural biological populations, population size often varies chaotically over time. Robert May in 1976[6] wrote one of the first papers to discuss chaotic dynamics, and did so in the context of population ecology. He showed how a simple model could replicate the complex dynamics often seen in real-world populations. The basic logistic growth model comes from the idea that population growth is determined by two factors: the population size and the amount of resources available in the local environment. Let \( x \) be population size, and \( t \) be time. If the change in population size is some constant factor \( r \) of the current population, that can be expressed using differential equations as:

\[
\frac{dx}{dt} = rx
\]  

(3.1)

With the solution:

\[
x(t) = ce^{rt}  
\]

(3.2)

This is of course classic exponential growth. By adding another term describing the limiting resources of the environment, we get the following:
\[
\frac{dx}{dt} = rx(1 - x) \tag{3.3}
\]

Which has the solution:

\[
x(t) = \frac{e^{rt}}{e^{rt} + e^c} \tag{3.4}
\]

This logistic function begins with exponential growth, then slows and begins asymptotically approaching a maximum population limit, defined by \(c\) and \(r\). Note that the although we interpret \(t\) as time, the function runs from negative infinity to positive infinity, so there will always be negative values in \(t\). When someone fits a logistic model to data, they typically have some finite data set and arbitrarily call the first point in the set \(t = 0\). The function approximates the data, and \(t < 0\) represents the population before the data was collected. Setting \(e^c = 1\) and \(r = 1\) we have the following function, Which travels from 0 at \(-\infty\) to 1 at \(+\infty\).

\[
x(t) = \frac{e^t}{e^t + 1} \tag{3.5}
\]

This particular logistic curve (see figure 3.1) is sometimes called the sigmoid function. Some natural
populations display this pattern, like bacteria in a petri dish with limited food, but some do not, either growing and declining periodically or varying erratically.

Instead of using the continuous logistic equation, May studied the effect of modeling populations with a discrete difference equation:

\[ x_{n+1} = rx_n(1 - x_n) \]  

(3.6)

Whereas the differential equation describes the behavior of a continuous derivative, the difference equation explains the change as a discrete shift from time \( n \) to time \( n + 1 \). Given some initial \( x_0 \) we apply the function repeatedly to get each new point in the series. Unlike the continuous equation, the discrete logistic equation can display a variety of behaviors, including chaos.

In the discrete logistic map \( x \) can be thought of as the current population’s proportion of the maximum possible size; it varies from 0 to 1. Now \( r \) acts as a growth parameter; varying the growth rate gives different patterns of behavior as shown in figure 3.2. For \( r < 1 \), the population eventually dies out to 0. For \( 1 \leq r < 2 \), the population grows asymptotically towards the population limit (itself a function of \( r \)), behaving exactly like the continuous logistic function. For \( 2 < r < 3 \), the population eventually reaches equilibrium, but first oscillates around the value. This behavior has a biological interpretation: the population has temporarily grown beyond the carrying capacity, and crashes below capacity before recovering and rising again. If \( r \) is increased beyond 3, the population oscillates permanently, without falling into a single equilibrium. Eventually when \( r \) is increased beyond about 3.45, it oscillates between four values. The number of points in the oscillation continue to increase in a period-doubling cascade, until around \( r = 3.57 \) the behavior becomes chaotic. It remains mostly chaotic for higher \( r \), but occasional periodic behavior resurfaces. Both oscillating and chaotic behavior have been observed in natural populations (though the distinction between oscillation and chaos is sometimes difficult to make in the approximate counts that come with real data). The full pattern of the logistic map’s behavior can be described by a bifurcation diagram, as shown in figure 3.3. The x-axis shows values of \( r \), and the y-axis shows values that \( x_n \) will take on after the map has settled into its attractor, a set of points (possibly infinite) through which it will travel through for all \( n \) greater than some constant. This diagram summarizes at what point the logistic map begins displaying periodic behavior, chaotic behavior, and so on.

What makes this population model interesting from an optimization point of view is that by varying a single parameter we can smoothly transition between qualitatively different dynamics, from static to periodic to chaotic. By trying to solve a problem using a variable population size genetic algorithm, with several different values of \( r \), we can hopefully find a population dynamic that is well suited to that task and get a
Figure 3.2: trajectories of the logistic map for various values of $r$
Figure 3.3: Bifurcation Diagram of the logistic map
better solution than we would have otherwise.

The pseudocode of the genetic algorithm in this thesis is given below. The basic algorithm is quite similar to a standard genetic algorithm. The variables $r$ and $x$ are the same as in equation (3.6). At each step the population proportion $x$ is multiplied by a $\text{popMax}$ constant, and the result is used to determine the number of organisms in the next generation. $\text{evaluate_fitness}$ calculates the fitness of each member of the population and stores it. $\text{next_generation}$ creates the next generation using tournament selection as described in section 2, and does mutation and crossover as specified by the user, depending on the problem. The best organism of each generation ($\text{bestYet}$) is preserved unchanged into the next generation.

There are many challenges in evaluating the success of this method, beginning with its free parameters. At its core, the algorithm makes use of two parameters: $r$, the growth rate, and $\text{popMax}$, the maximum population and the multiplier for $x$. A key challenge in this thesis project was determining proper values for these two parameters, and especially evaluating the influence of $r$ on the outcome. This involved choices about different assessment metrics and different test problems; these choices, and the overall experimental design, are the focus of the next chapter.

---

Program 1 genetic algorithm pseudocode

```python
def evolve(problem, population, numEvals, r, popMax, x, bestYet):
    while numEvals > 0:
        x = r * x * (1 - x)
        nextgensize = round(x * popMax)
        evaluate_fitness(problem.fitness, population)
        nextGen = next_generation(problem, population, nextgensize)
        population = [bestYet] + nextGen
        numEvals -= nextgensize - 1
        bestYet = min(population)
    return bestYet
```
Chapter 4

Experimental Design

The overall goal of these experiments was to investigate the effects of population dynamics on the efficiency of genetic algorithms. To do this I needed to test the performance of my chaotic genetic algorithm on a suitable set of problems. My analysis of the different population dynamics breaks into three phases. First, I considered a bitstring problem in some depth, with the goal of studying the effect of changing the number of fitness function evaluations, the population size, and the population dynamics, independently and together. The second phase was a statistical analysis of three more-complex problems, testing them at three different values of $r$ and two different values of $\text{popMax}$, the two critical variables. This test was the most important, because it checked the general viability of my genetic algorithm compared to traditional genetic algorithms, and evaluated the effects of the free parameters in more depth. The results of this test were mixed, leading to the third phase. In it, I examined the distributions of the results of each experiment, and patterns from individual runs, in hopes of uncovering some underlying pattern.

Each run of the GA program requires several things. First, it requires a problem definition, including organism type, fitness function, and operators for mutation, generation, and crossover. Second, it requires values for the $r$ and $\text{popMax}$ parameters, and a specification of how many evaluations of the fitness function the algorithm is allowed to take. Because this research focuses mainly on the effect of varying $r$, I chose $\text{popMax}$ and the total evaluation cap by hand for each problem. Harder problems need more function evaluations in general: I tried to choose the smallest number of evaluations that made each problem solvable.

To make comparisons relevant, I scaled the population maximum for each value of $r$ to keep mean population value the same. For a given $r$, the mean value of $x$ is $\frac{r-1}{r}$, the fixed point of $x_{n+1} = rx_n(1-x_n)$, so I multiply the population by $\frac{r}{r-1}$ to adjust for that population difference. Finally, when running the
experiment, the program throws away the first 1000 points of the logistic map in an attempt to ensure that
the GA run begins the experiment inside the attractor, the set of points in which the population will remain.

There are two main schools of thought for the best stopping methods for comparing two optimization
algorithms: one can either let the algorithms run until they plateau and stop improving, and see which can
obtain the best overall value, or one can limit each algorithm to a set number of function evaluations and
see which performs best with the allotted number of evaluations (here the experiments assume evaluation of
the fitness function to be the most expensive aspect of the optimization, as it will be in almost all cases).
For my thesis I chose the second approach, for two reasons. Firstly, it is often difficult or impossible to
determine, in general, that an algorithm has really gotten stuck. One can choose to stop the algorithm if no
improvement in fitness has been found after some number of generations. However, often a genetic algorithm
will show no overall improvement, because the best organism in the population isn’t changing, while many
other organisms, currently less fit, continue to cross over and improve. One of those may jump into a new
area of the search space and cause an improvement after many generations of apparent stagnation. Secondly,
from a practical viewpoint, a user may need to limit the number of evaluations because of allocated time
or computing resources, or they simply want to have a good idea of how long the experiment will take, so
allowing an algorithm to run “until it’s done” may not be feasible. In situations where evaluations are fast
or processing power is less constrained, allowing the algorithm to run until it shows no improvement could
be much more feasible, but one can of course never prove that a global optimum has been reached.

The next question was which test problems to use. I chose a bitstring problem, a 1-dimensional continuous
function, a multi-dimensional continuous function, and the traveling salesman problem. These problems were
selected because they were used as test problems in previous research, and because they present a range of
difficulties and problem types.

The GA literature contains many different test problems and methods of evaluation. The bit-matching
problem was included as an easy way to make sure the algorithm was working and provide some small
cases that could be easily analyzed. To make sure the genetic algorithm was working properly and test its
responsiveness to simple problems, I started with a basic matching function: match a bitstring containing
all ones. In this context, GA organisms are bitstrings; the bitstrings “1100” and “1011” have fitness 2.0
and 3.0, respectively. Crossover is single point crossover, exactly as described in chapter 2. After crossover,
each bit of each child is flipped with probability of 0.001. Therefore, the number of bits flipped per instance
follows a binomial distribution, with an average of 6.4 bits flipped per 100 organisms. First I wanted to
make sure my GA could successfully solve the problem (that is, evolve a bitstring of only ones, starting
from random initial bitstrings), and determine approximately how many function evaluations are required
to accomplish this for a static-population GA. I wanted to check first the effect of varying the population
size for a static-population GA, and then see how inserting chaotic dynamics would affect that result.

For the statistical testing I chose three optimization problems that provided distinct challenges and gave
some opportunity for in-depth analysis. Rather than exploring the entire search space of \( r \) and \( \text{popMax} \), I
focused on three values for \( r \), one for each major mode of the logistic map: static, periodic, and chaotic
population dynamics. For each problem, I considered two different population sizes, adapting based on the
difficulty of the particular problem. Because some of the problems I tested were significantly more difficult
than others, I found that larger or smaller population sizes were necessary to get an accurate result. I solved
this by choosing the population sizes by hand for each problem, using preliminary tests to make sure the
population sizes generated reasonably efficient performance on the problem. If the GA performed poorly I
would adjust the values. Testing all combinations of growth rate and population gives a total of six solution
strategies per problem.

Because genetic algorithms are stochastic, multiple runs of the solver on the same problem produce
different results. Although a typical user would only run the genetic algorithm once, and then use whatever
best solution was found during the run, an effective evaluation of different algorithms needs to compare the
output across multiple runs of the same genetic algorithm, differing only in the seed of the pseudo-random
number generator. If we then take the best solution produced by each run, those solutions will fall into
a distribution with some mean and variance. We can repeat this process for several different population
strategies, and compare the mean of each. This gives an estimate of which population strategy is expected
to perform best on the target problem for any particular run. If one population strategy outperforms others
for several different kinds of problems, that is evidence in favor of using that strategy for new problems that
come along.

The standard statistical way to approach this type of problem is the t-test. Given the means and
variances for two samples, the t-test estimates the probability that the means are the same. In my case, the
two samples are the results from two different types of population control, for example chaotic versus static.
I have a null hypothesis that the two types have the same mean fitness result for a given problem. Given
the results, I calculate the probability that such a null hypothesis could be true. If the probability is small,
I can safely assume that the two methods have different mean fitness results and that one is significantly
better than the other. If the probability is large, we cannot make that assumption and it remains likely that
the two methods produce the same result.
Figure 4.1: Sine function optimization example
The first problem on which I ran t-tests is the sine function from [2], \(-x \sin(10\pi x) + 1\). This function, shown in figure 4.1, has a relatively simple GA representation and an easy-to-find answer with a small number of iterations, which makes it useful for comparing different population dynamics. It still presents a challenge, since it has several local maxima that can trap the solver. The organism for this problem is a double-precision floating-point number. Crossover is just choosing one of the two parents, so in some sense no information is common to both parents, which is unusual for a GA. I discuss this further in chapter 6. After the child is chosen, with probability 0.8 noise is added to that child point, uniformly distributed from -0.2 to 0.2, for the mutation. Each run of the GA is allowed 200 total function evaluations.

The sine function problem implementation did not use crossover in the traditional sense. A more typical approach would be to convert the doubles into bitstrings and apply crossover as described in chapter 2. I did some preliminary testing with this method, as well as a crossover that chose a random number uniformly distributed between the two parents. In general the performance was worse than the method I chose. By forcing the crossover to choose one point or the other, I can increase the mutation rate without losing too much information from the parents. Since each child is guaranteed to be a relatively fit organism before mutation, we can do more mutation than otherwise. It is worth asking whether or not these results are still applicable to general genetic algorithms, given that the key crossover operation has been removed. The experiments still show how changing population dynamics affect optimization, and the selection method is the same as for the other problems I test. It may be that the value of crossover is lost in such a domain: a single floating point number does not have multiple separate parts that can be combined to create a greater whole. But it is a testament to the flexibility of genetic algorithms that we can so simply adjust the algorithm to handle such a problem.

My next test problem was the 10-dimensional Schwefel function, described in [5]:

\[
\sum_{i=1}^{10} -x_i \sin(\sqrt{|x_i|})
\] (4.1)

This function, shown in figure 4.2, has many more local maxima and requires many more iterations to solve, making it a more realistic test. Very similar to the approach in the sine problem, the GA represents the problem domain (the organisms) as lists of 10 double-precision numbers. Crossover involves randomly choosing one double for each dimension: the uniform crossover described for bits is extended so that each double is analogous to a single bit. Then with probability 0.8 I mutate each dimension by adding uniformly distributed noise in \((-10, 10)\).
Figure 4.2: Two of the ten axes of the Schwefel function optimization example. All ten dimensions are symmetrical and look the same as above.
Table 4.1: Summary of Problem Parameters

<table>
<thead>
<tr>
<th>Problem</th>
<th>Evaluation Limit</th>
<th>Genome</th>
<th>Crossover</th>
<th>Mutation Prob.</th>
<th>Mutation Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bitstring</td>
<td>varied</td>
<td>64 bits</td>
<td>single point crossover</td>
<td>0.001 per bit</td>
<td>flip</td>
</tr>
<tr>
<td>Sine</td>
<td>200</td>
<td>1 double</td>
<td>choice of parent</td>
<td>0.8</td>
<td>uniform +/- 0.2</td>
</tr>
<tr>
<td>Schwefel</td>
<td>20000</td>
<td>10 doubles</td>
<td>choice for each double</td>
<td>0.8</td>
<td>uniform +/- 10</td>
</tr>
<tr>
<td>TSP</td>
<td>1000</td>
<td>28 doubles</td>
<td>choice for each double</td>
<td>0.8</td>
<td>uniform +/- 0.2</td>
</tr>
</tbody>
</table>

Finally I tested my GA on the traveling salesman problem, which is used in many GA tests, for example [1]. This is a discrete problem, an important class of problems to test here because its members are not amenable to solution by derivative-based optimizers. For the genome representation there are many ways to encode a tour, each with pros and cons. A simple implementation as a list of cities runs into difficulties with crossover: since no two cities can be visited twice, an elementwise crossover of two lists may generate an invalid tour. I chose to solve this problem by defining each organism as a list of cities, with each city associated with a floating point number. The tour is generated by traveling through each city in ascending order of associated value. This has the advantage that a typical mutation and crossover operator will work as intended, changing the floating point values as with the Schwefel problem. TSP priorities are arbitrarily forced to stay within the range (0,1). For this problem, each GA run uses 1000 evaluations. As in the Schwefel experiments, crossover chooses one value from each dimension, and mutation, with probability 0.8, adds uniformly distributed noise in the range (−0.2, +0.2) to that chosen value.

A summary of problem parameters is given in table 4.1 for convenience. The evaluation limit is the number of function evaluations a single run of the GA is allowed to perform. The details of genome mapping and crossover and mutation are given in the text.
Chapter 5

Results and Discussion

5.1 Bitstring problem

I began with an in-depth study of the bitstring function. This is a very simple problem: easy to understand, easy to maximize. First I ran the genetic algorithm with static population size. Examples of fitness improvement over time for two runs of the GA on the bitstring problem are given in figures 5.1a and 5.1b. Note that there is a general trend of steadily improving fitness. This is a good sign. To explore the effect of population size, I ran the GA once with a low population and once with a high population, to see a clear contrast. These two runs had population sizes 10 and 200 respectively. It is interesting to note that for a higher population size, the growth is more erratic, but actually improves slightly faster. Most likely the erratic behavior of the higher population is because there is room for many less fit organisms in the population, so occasionally the most fit organisms will not get a chance to reproduce. However the variance also makes it easier for crossover to produce new good traits (in this case 1s in new positions), so the overall growth is faster.

My next step was to verify that population size and number of evaluations have some positive effect on the genetic algorithm performance. Since this involves two independent variables, I show the results using heatmaps. Typically heatmaps, which color the dependent variable, using two spatial dimensions for two independent variables. (It helps to imagine a topological map with higher areas colored red, and valleys colored blue.) This allows the viewer to visualize complex correlations and relationships in higher dimensions. In this case the two independent variables are the number of times the genetic algorithm is allowed to call the fitness function and the number of calls per generation. If the algorithm is working properly, allowing the solver to call the fitness function more times will generally result in a better solution, and that is what is
visible in figure 5.2. The bright orange band in the region around 700 - 1000 evaluations and population cap 0 - 0.1 suggests that the solver works best when it can use more function evaluations, and when the population at each generation is less than 10 percent of the total number of evaluations allocated. This provides some evidence that there is a trade-off in population size, and the algorithm needs at least 10 generations to reach a good solution. The best performance seems to appear when the GA uses the largest population size that will still allow 10 to 20 iterations. It is not clear whether or not this will extrapolate to problems with more evaluations, however.

Next I evaluated the effect of chaotic dynamics in the population size. A heatmap plot of the results is shown in figure 5.3. The two variables in this case are chaotic growth (the $r$ variable from the logistic map) and maximum population size, so the axes have changed from the previous figure, where we compared population size and number of evaluations. For this experiment each run is only allowed to use the evaluation function 1000 times, which was chosen so that the GAs could often, but not always, find the optimal solution. As with the last experiment, the GA performs best on the bitstring problem when the population size is about 50, or 5% of the total number of evaluations. This allows 20 iterations of the algorithm. However, the chaotic dynamics do not seem to have any effect at all. This could be because the problem itself is too simple for the genetic algorithm to need any complicated population policy.
Figure 5.2: The effect of population size and number of function evaluations on fitness for the bitstring problem

Figure 5.3: The effect of population size and growth rate (r) on fitness for the bitstring problem
Table 5.1: Experiments where Static Performed Best

<table>
<thead>
<tr>
<th>problem</th>
<th>popSize</th>
<th>static</th>
<th>periodic</th>
<th>chaotic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Schwefel</td>
<td>200</td>
<td>0.9928</td>
<td>0.9915</td>
<td>0.9842</td>
</tr>
<tr>
<td>TSP</td>
<td>20</td>
<td>0.5760</td>
<td>0.5710</td>
<td>0.5640</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem</td>
</tr>
<tr>
<td>Schwefel</td>
</tr>
<tr>
<td>TSP</td>
</tr>
</tbody>
</table>

5.2 Sine, Schwefel, and Traveling Salesman problems

In the bitstring problem, changing the population size of the genetic algorithm caused a clear change in the algorithm’s performance, but changing the population dynamics did not. I next tested the genetic algorithm on three more difficult problems, with an emphasis on finding the variation caused by different population dynamics. For these three problems I found a greater variance in the effect of those population dynamics than I found for the bitstring problem, so I analyzed them in more detail, including statistical significance. The results for each of the three test problems are given in tables 5.1 and 5.2, with two population sizes hand selected for each problem, giving six total experiments. The table contains the problem type, population size, and probabilities comparing static, periodic, and chaotic population sizes. Also for each problem there is a set of boxplots given in figs. 5.6-5.8 for each of the six solution types, showing the approximate distribution and outliers for each of the solutions.

Of the six tests I ran, the chaotic population GA outperformed the periodic population GA in four and outperformed the static population GA in three (one of these is borderline significant, p=0.07.) These results show no clear winner between static and chaotic GAs, but they suggest that periodic population variation is not effective. There is some evidence against the chaotic performance, however. Of the three situations where chaotic is better than static, two occur in a large population size. But in both of those cases the static GA at the smaller population size outperforms the chaotic GA at the highest population size for the same problem. For example, figure 5.5b shows that the chaotic GA has the highest performance of the three population size strategies when the average population size is 500. But figure 5.5a shows that lowering the average population to 200 reverses the trend and makes the static GA perform the best. Furthermore, the performance of the static GA when population is 200 is better than the chaotic GA performance at 500 (4179.3 and 4177.5 respectively). So the static GA can be said to be “best overall” in the two of the three problems, Schwefel and traveling salesman, while chaotic performs the best on the sine problem.
Table 5.2: Experiments where Chaotic Performed Best

<table>
<thead>
<tr>
<th>problem</th>
<th>popSize</th>
<th>static</th>
<th>periodic</th>
<th>chaotic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sine</td>
<td>12</td>
<td>0.9499</td>
<td>0.9290</td>
<td>0.9620</td>
</tr>
<tr>
<td>Sine</td>
<td>18</td>
<td>0.9564</td>
<td>0.9451</td>
<td>0.9589</td>
</tr>
<tr>
<td>Schwefel</td>
<td>500</td>
<td>0.9919</td>
<td>0.9918</td>
<td>0.9924</td>
</tr>
<tr>
<td>TSP</td>
<td>200</td>
<td>0.5510</td>
<td>0.5551</td>
<td>0.5613</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Problem</th>
<th>popSize</th>
<th>P(static &gt; chaotic)</th>
<th>P(periodic &gt; chaotic)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sine</td>
<td>12</td>
<td>0.0217</td>
<td>8.6607e-07</td>
</tr>
<tr>
<td>Sine</td>
<td>18</td>
<td>0.3179</td>
<td>0.0062</td>
</tr>
<tr>
<td>Schwefel</td>
<td>500</td>
<td>2.9882e-07</td>
<td>0.0015</td>
</tr>
<tr>
<td>TSP</td>
<td>200</td>
<td>0.0757</td>
<td>0.0173</td>
</tr>
</tbody>
</table>

Figure 5.4: Boxplots comparing fitness distributions for different population dynamics for the sine function

(a) population = 12
(b) population = 18
Figure 5.5: Boxplots comparing fitness distributions for different population dynamics for the Schwefel function

Figure 5.6: Boxplots comparing fitness distributions for the traveling salesman problem
Given the inconclusiveness of the results described above, I spent some time trying to further analyze the reasons behind the differing performance. First I considered the sine problem. This problem is relatively easy, so I chose to allocate a small number of evaluations per run (200) and used small population sizes (20 and 50). Recall that the organism in this instance is just one double-precision number. This has the advantage of making it quick to generate data for analysis. As mentioned above, the results indicate that the periodic population-size GA performed worse on this problem than either the chaotic or static versions. The best performance was for the chaotic GA with an average of 12 organisms per generation. Some sample runs for each of the r values are shown in figure 5.8. This comparative plot shows that the algorithm comes fairly quickly to the optimal neighborhood, around 2.95. Then it bounces around underneath that ceiling, dropping in fitness and then re-approaching the ceiling. Histograms comparing static, periodic, and chaotic results appear in figure 5.7. These histograms plot the frequency of fitness results over the course of the experiment (450 runs total). The same data is summarized in the boxplots in figure 5.4. In these, the box has lines at the first quartile, median, and third quartile values. The distance from the first quartile to the third quartile is called the interquartile range ($IQR$), and is a useful measurement of variance similar to standard deviation. Any point point greater than $1.5IQR$ from the median is called an outlier. Such outliers are displayed with a red + sign. The whiskers extending from the box cover all points within the $1.5 \times IQR$ range. Boxplots are designed to easily visualize distribution shape and outliers. The histograms and the boxplots in 5.4 are highly skewed towards the optimal value at 2.95, with a long tail stretching into lower fitness values. This shows that the algorithm often found the optimal value, and only occasionally got stuck in some lower local optimum.

One important feature that shows up in the histogram of the sine function optimizations is a small spike in frequency for the local maxima at 2.75 and 2.55 which correspond to the two next-closest local maxima, and a larger spike at 1.95, which corresponds to the local maximum on the far right side of the function. It seems that with a small number of evaluations, if the initial population is mostly on the right side of the search space, the solver has a hard time getting across to the correct side of the function. It seems likely that the chaotic variation has some effect on allowing the function to move across the otherwise troublesome local maxima, since it performs significantly better than static or periodic population GAs.

The Schwefel function, in contrast to the sine function, is a 10-dimensional continuous problem that poses a significant challenge; it has even more local optima that trap optimizers. As with the sine function I compared the performance of three versions of the genetic algorithm upon this problem: one with a constant population, one with a population alternating between two values, and one with a chaotic population. Where
the organism was a double for the sine instance, it was an array of 10 doubles for the Schwefel problem, with mutation and crossover occurring elementwise. First I ran each of the three GAs with an overall population mean of 200 per cycle. The results are given in the boxplot in figure 5.5a. In this case the static population algorithm has the best performance and is significantly more consistent than the varying ones (although the difference between static and periodic is not statistically significant due to the high variation in the performance of the periodic GA). The boxplot clearly shows the very small variance of the static GA results, and the extremely large variance of the chaotic GA results. Then I ran the experiment again after increasing the population to 500, shown in figure 5.5b. In this case the results are nearly opposite, with the best performance achieved by the chaotic genetic algorithm. However, the sample sizes are not really large enough to say that one is conclusively better than the other, as shown by the results of the t-tests in 5.1. Unlike in the sine problem, the fitness values here are mostly within the area of the global maximum, and not trapped in local maxima. If they had stuck in other local maxima, there would be corresponding spikes in the histogram. Most of the results are in the hill of the largest maxima. It seems that all three genetic algorithms were usually able to find the right area, but were not very efficient in climbing the last hill to the globally optimal point. In a situation like that a high-performance algorithm could switch from a GA search to a fast gradient climber that is guaranteed to find the local maximum quickly. A hybrid approach might out-perform both a GA or a gradient-based method individually.

The next problem was the traveling salesman problem, in particular a 29-city instance from TSPLIB. Unlike the Schwefel example, this is a discrete fitness function, so traditional optimizers using derivatives
will not be effective. The significance results in the table and in figure 5.6 suggest a similar outcome to the Schwefel problem: in both cases the chaotic population GA performs worst at low populations, and best at high populations. It seems that if the population size is too low, large population swings destroy too much of the information contained in the gene pool. At higher populations, the swings can provide a benefit, allowing the chaotic GA to outperform the static and periodic GAs. Interestingly, unlike in the Schwefel and sine problems, which both have fairly skewed distributions of fitness due to frequently finding the optimal value, the distribution of the results for the traveling salesman problem was more normal and symmetrical. This suggests the algorithm was rarely hitting the optimum, and perhaps more evaluations were needed.
Chapter 6

Conclusion

In my thesis work, I tried to solve two closely related problems. First, I wanted to create a practical genetic algorithm that had noticeable performance benefits compared to the standard GA. Second, I wanted to better understand how population size affects genetic algorithm performance in general. To tackle both of these problems, I introduced a logistic-map based genetic algorithm. This allowed me to easily vary the GA population dynamic, using a single free parameter. I then tested the different population dynamics on a set of test problems. This addressed my first goal by determining whether dynamic-population algorithm could outperform the static-population algorithm; it also addressed my second goal by letting me examine what response the GA had to different dynamics for each problem. That response revealed something about the underlying structure of the algorithms.

I made good progress on the first problem, with some qualifications. By choosing two different population sizes for three different problems of varying difficulty, I created a set of six performance tests, from three diverse problems. In four of those six tests, the chaotic-population GA outperformed the standard static-population GA by a statistically significant margin. This suggests that, in some situations, the chaotic-population GA is a useful improvement over the standard GA. My initial hypothesis, however, was that the chaotic-population GA would outperform the standard GA for almost all problems. That hypothesis turned out to be false, and I cannot recommend the chaotic-population GA to be used exclusively. An important caveat to this research is the hand-tuned nature of the population sizes, and the hand-picked problems on which I tested. To address these issues, future work would need to be more methodical in both areas, looking at a larger set of test problems over a larger range of population sizes.

Another important limitation to this research was the choice in genome and crossover methods. For
any genetic algorithm, the mapping from problem to genome has effects that are difficult to predict. My choices of mapping for the test problem certainly influenced the results I found. I think the problems and genome choices among my tests are different enough from one another to make my conclusions generally applicable to most genomes; it seems unlikely that the four mappings I chose are all biased in some way that would invalidate the results on population dynamics. However, I would like to extend the work further by performing statistical analysis on the same problems, but with different genome mappings: if the results I found were consistent across those mappings, it would significantly strengthen the conclusions of this research.

The second problem was to understand how population dynamics affect GAs. The ideal result would have been a deep understanding of how changing populations over time affect the algorithm’s ability to find a solution. On this problem I had less success. By looking at which algorithms had best performance for which problems, I was able to form some preliminary hypotheses about what kinds of effects might be at work. Using the bitstring problem, I confirmed that the optimal population size depends on overall number of function evaluations allocated. Although there were no statistically-significant population-dynamic effects for that problem, there were such effects for other problems. Overall, chaotic-dynamic GAs performed best with larger population sizes, while periodic-dynamic GAs were never the best performers for any given problem. These are not final answers, but they provide some important leads for future research.

I think there are two important next steps in this line of research. First, I think the set of tests needs to be expanded into more problems, especially problems where genetic algorithms may be the only viable approach. I have begun work on testing my GA on compiler optimization. I hope to extend this work to include those new results. Second, I think the problems that showed significant performance variance for different population dynamics need in-depth analysis, similar to the approach I took with the bitstring problem. Ideally such research would show not only the optimal population size, but also the optimal value of $r$, the logistic-map variable. However, this requires a nearly exhaustive search over the parameter space. This is prohibitively expensive for most difficult problems. But if those results could be obtained, I think they would provide critical insight into the population dynamic problem.
Bibliography


