**EXPLORING THE RELATIONSHIP OF THE CLOSENESS OF A GENETIC
ALGORITHM'S CHROMOSOME ENCODING TO ITS PROBLEM SPACE**

A Thesis
presented to the Faculty
of California Polytechnic State University,
San Luis Obispo

In Partial Fulfillment
of the Requirements for the Degree
Master of Science in Computer Science

by
Kevin McCullough
March 2010

© 2010
Kevin McCullough
ALL RIGHTS RESERVED

COMMITTEE MEMBERSHIP

TITLE: Exploring the relationship of the closeness of a genetic algorithm's chromosome encoding to its problem space

AUTHOR: Kevin McCullough

DATE SUBMITTED: March 2010

COMMITTEE CHAIR: Franz Kurfess, Ph. D.

COMMITTEE MEMBER: Alexander Dekhtyar, Ph. D.

COMMITTEE MEMBER: Hasmik Gharibyan, Ph. D.

ABSTRACT

Exploring the relationship of the closeness of a genetic algorithm's chromosome encoding to its problem space

Kevin McCullough

For historical reasons, implementers of genetic algorithms often use a haploid binary primitive type for chromosome encoding. I will demonstrate that one can reduce development effort and achieve higher fitness by designing a genetic algorithm with an encoding scheme that closely matches the problem space. I will show that implicit parallelism does not result in binary encoded chromosomes obtaining higher fitness scores than other encodings. I will also show that Hamming distances should be understood as part of the relationship between the closeness of an encoding to the problem instead of assuming they should always be held constant. Closeness to the problem includes leveraging structures that are intended to model a specific aspect of the environment. I will show that diploid chromosomes leverage abeyance to benefit their adaptability in dynamic environments. Finally, I will show that if not all of the parts of the GA are close to the problem, the benefits of the parts that are can be negated by the parts that are not.

ACKNOWLEDGMENTS

I would like to acknowledge the love and support of my beautiful wife Christy, without whom none of this would have been possible.

Table of Contents

List of Figures.....	viii
List of Graphs.....	x
I. INTRODUCTION	1
A. Biological Evolution.....	1
1. Retaining Genetic Traits	1
2. Refining Genetic Traits	3
B. Genetic Algorithms	5
1. Retaining Beneficial Aspects of a Solution	6
2. Refining Beneficial Aspects to a Solution	7
C. Chromosome Design in Genetic Algorithms.....	8
II. BIOLOGICAL EVOLUTION AND GENETIC ALGORITHMS.....	10
A. Population and Generations	11
B. Chromosomes	12
C. Encoding Schemes.....	14
D. Fitness Evaluation and Mate Selection	16
E. Crossover	19
F. Mutation	22
G. Haploid and Diploid Organisms	23
H. Advantages of Genetic Algorithms	26
I. Disadvantages of Genetic Algorithms	27
III. CHROMOSOME ENCODING PRINCIPLES	30
A. Closeness to the Problem Space	30
B. Implicit Parallelism	33
C. Hamming Distance	38
D. Abeyance.....	41
IV. THESIS	47
V. EXPERIMENT IMPLEMENTATION.....	48
A. Problems and Goals	50
1. Binary Problem.....	51
2. Byte Problem.....	51
3. Float Problem.....	52
4. Base Implementation	54
B. Dynamic Goals	55
C. Chromosome Implementation	57
1. Haploid Chromosomes	57
2. Diploid Chromosomes.....	61
D. Chromosome Initializers	62
E. Chromosome Crossover.....	64
F. Chromosome Mutators	66
G. Binary Chromosome	67
H. Byte Chromosome	71
I. Floating-Point Chromosome	74
J. GAlib Modifications	76
1. Abstract fitness interface.....	77
2. Ability to disable elitism	78
VI. RESULTS.....	80
A. Nadir Initialized Chromosomes with Static Goals.....	82
1. Floating point architecture and non-float chromosomes.....	91
B. Nadir Initialized Chromosomes with Dynamic Goals.....	96
1. Results from Experiments of Nadir Initialized Chromosomes	107
C. Optimum Initialized Chromosomes with Static Goals.....	109
D. Optimum Initialized Chromosomes with Dynamic Goals.....	113
1. Results from Experiments of Optimum Initialized	120
E. Uniformly Initialized Chromosomes with Static Goals	121
F. Uniformly Initialized Chromosomes with Dynamic Goals	128

1.	Results from Experiments of Uniformly Initialized Chromosomes	135
VII.	FUTURE WORK	137
A.	Code Improvements	137
B.	New Chromosome Encodings	139
VIII.	SUMMARY AND CONCLUSIONS	141
A.	Matching on Primitive Type	142
1.	Hamming Distances	143
2.	Implicit Parallelism	144
3.	Bringing all parts of a GA close to the problem	145
B.	Diploidy Benefits from Abeyance	147
IX.	Bibliography	150

List of Figures

Figure 1. Theoretical bit pattern that a GA is trying to identify.....	13
Figure 2. Example chromosome representing the solution to a problem	13
Figure 3. Example chromosome representing a solution using bytes	14
Figure 4. Example chromosome with a float encoding scheme	15
Figure 5. Example chromosomes attempting to solve the same problem	17
Figure 6. Mother and father for crossover example	20
Figure 7. Resulting sister and brother for crossover example	21
Figure 8. Brother's chromosome after mutation.....	23
Figure 9. Eight possible combinations for a 3 bit chromosome	34
Figure 10. Example 3 bit chromosome with wild card	34
Figure 11. Example 3 bit chromosome with wild card in the second position	35
Figure 12. Example 3 bit chromosome with wild card in the first position	35
Figure 13. Example 3 bit chromosome with wild card in the second and third positions	35
Figure 14. Example 3 bit chromosome with wild card in the first and third position	35
Figure 15. Example 3 bit chromosome with wild card in the first and second position	35
Figure 16. Decimal, Binary, Gray encodings, and Hamming Distances	40
Figure 17. Hollstein-Holland triallelic dominance map	43
Figure 18. Diploid chromosome example	43
Figure 19. Expressed phenotypes	43
Figure 20. XOR logic.....	44
Figure 21. The 36 tests. Chromosomes vs Problems.....	48
Figure 22. Length of chromosomes in their own primitive type	60
Figure 23. Optimum and Nadir goal values	64
Figure 24. Example Haploid Binary Chromosome for the Binary Goal	69
Figure 25. Example Diploid Binary Chromosome for the Binary Goal.....	69
Figure 26. Example Haploid Binary Chromosome for the Byte Goal	69
Figure 27. Example Diploid Binary Chromosome for the Byte Goal.....	70
Figure 28. Example Haploid Byte Chromosome for the Float Goal.....	70
Figure 29. Example Diploid Byte Chromosome for the Float Goal.....	70
Figure 30. Example Haploid Byte Chromosome for the Binary Goal	72
Figure 31. Example Diploid Byte Chromosome for the Binary Goal.....	72
Figure 32. Example Haploid Byte Chromosome for the Byte Goal	72
Figure 33. Example Diploid Byte Chromosome for the Byte Goal.....	73
Figure 34. Example Haploid Byte Chromosome for the Float Goal.....	73
Figure 35. Example Diploid Byte Chromosome for the Float Goal.....	73
Figure 36. Example Haploid Float Chromosome for the Binary Goal.....	75
Figure 37. Example Diploid Float Chromosome for the Binary Goal	75
Figure 38. Example Haploid Float Chromosome for the Byte Goal.....	75
Figure 39. Example Diploid Float Chromosome for the Byte Goal	75
Figure 40. Example Haploid Float Chromosome for the Float Goal.....	76
Figure 41. Example Diploid Float Chromosome for the Float Goal	76
Figure 42. Result charts' line styles	81
Graph 2. Nadir initialized haploid chromosomes vs static binary goal where byte chromosome uses RandomMutator()	83
Graph 5. Nadir initialized haploid chromosomes vs static byte goal where byte chromosome uses RandomMutator()	87
Figure 43. Floating-point architecture	93
Figure 44. Floating-point representation of 240.....	93
Figure 45. Integer representation of 240.....	93
Figure 46. Integer representation of 112.....	93
Figure 47. Floating-point representation of 7.05297×10^{-37}	94
Figure 48. Floating-point representation of 240.031	94
Figure 49. Floating-point representation of 240.373.....	94
Graph 15. Nadir initialized diploid chromosomes vs dynamic float goal run for twice as long	106

Graph 16. Optimum initialized diploid chromosomes vs dynamic float goal run for twice as long	107
---	-----

List of Graphs

Graph 1. Nadir initialized haploid chromosomes vs static binary goal	82
Graph 3. Nadir initialized diploid chromosomes vs static binary goal.....	85
Graph 4. Nadir initialized haploid chromosomes vs static byte goal	86
Graph 6. Nadir initialized diploid chromosomes vs static byte goal.....	88
Graph 7. Nadir initialized haploid chromosomes vs static float goal	89
Graph 8. Nadir initialized diploid chromosomes vs static float goal.....	90
Graph 9. Nadir initialized haploid chromosomes vs dynamic binary goal	96
Graph 10. Nadir initialized diploid chromosomes vs dynamic binary goal.....	98
Graph 11. Nadir initialized haploid chromosomes vs dynamic byte goal	100
Graph 12. Nadir initialized diploid chromosomes vs dynamic byte goal.....	101
Graph 13. Nadir initialized haploid chromosomes vs dynamic float goal	103
Graph 14. Nadir initialized diploid chromosomes vs dynamic float goal.....	104
Graph 17. Optimum initialized haploid chromosomes vs static binary goal	110
Graph 18. Optimum initialized diploid chromosomes vs static binary goal.....	110
Graph 19. Optimum initialized haploid chromosomes vs static byte goal	111
Graph 20. Optimum initialized diploid chromosomes vs static byte goal.....	111
Graph 21. Optimum initialized haploid chromosomes vs static float goal	112
Graph 22. Optimum initialized diploid chromosomes vs static float goal.....	112
Graph 23. Optimum initialized haploid chromosomes vs dynamic binary goal	113
Graph 24. Optimum initialized diploid chromosomes vs dynamic binary goal.....	114
Graph 25. Optimum initialized haploid chromosomes vs dynamic byte goal	116
Graph 26. Optimum initialized diploid chromosomes vs dynamic byte goal.....	117
Graph 27. Optimum initialized haploid chromosomes vs dynamic float goal	118
Graph 28. Optimum initialized diploid chromosomes vs dynamic float goal.....	119
Graph 29. Uniformly initialized haploid chromosomes vs static binary goal.....	121
Graph 30. Uniformly initialized diploid chromosomes vs static binary goal.....	123
Graph 31. Uniformly initialized haploid chromosomes vs static byte goal	124
Graph 32. Uniformly initialized diploid chromosomes vs static byte goal	125
Graph 33. Uniformly initialized haploid chromosomes vs static float goal.....	126
Graph 34. Uniformly initialized diploid chromosomes vs static float goal.....	127
Graph 35. Uniformly initialized haploid chromosomes vs dynamic binary goal.....	128
Graph 36. Uniformly initialized diploid chromosomes vs dynamic binary goal	129
Graph 37. Uniformly initialized haploid chromosomes vs dynamic byte goal.....	132
Graph 38. Uniformly initialized diploid chromosomes vs dynamic byte goal.....	133
Graph 39. Uniformly initialized haploid chromosomes vs dynamic float goal.....	134
Graph 40. Uniformly initialized diploid chromosomes vs dynamic float goal.....	135

I. INTRODUCTION

John H. Holland first developed the genetic algorithm (GA) in 1975. GAs are algorithms that are designed to mimic the theory of evolution in order to replicate the adaptability and success of biological organisms in a non-deterministic environment [Holland 1975]. GAs have been applied to many different types of problems in many different disciplines, from the classic traveling salesman problem [Buckland 2002 pg 118-141] to problems relating to sonar signal processing [Montana 1991], schedule optimization [Syswerda 1991], NP-Complete problems [Claudio et al. 2000] [Corcoran, Wainwright 1992] and encryption [Bagnall et al. 1997].

A. Biological Evolution

1. Retaining Genetic Traits

Evolution is a process by which a species adapts over successive generations by retaining and refining beneficial genetic traits. Through reproduction, two parent organisms combine their genetic material to create a child from that material. The child retains genetic material from both of its parents. This passing on of genetic material over generations is called heredity.

In evolution, the success of a species depends on the ability of its individual organisms to survive and to reproduce. A trait is considered “beneficial” if it enables organisms to achieve these two objectives: survival and reproduction.

Organisms that achieve these objectives are said to be “fit.” Because the organisms that survive and reproduce are, by definition, fit, those organisms are more likely to pass on beneficial traits to their children. In this way, the overall fitness of the entire population is increased.

Each organism contains a structure that stores the organism’s genetic material. This structure is called a chromosome. Chromosomes are comprised of one or more genes, each of which stores one of the organism’s genetic traits as an encoded DNA sequence. The particular DNA sequence coded by the gene (which defines the genetic trait) is its allele value; the set of allele values for a gene is all of the possible values the gene could represent. A phenotype is an expressed or observable trait. Although multiple allele values can map to a single phenotype, more commonly different allele values will correspond to different phenotypes.

During reproduction, genes from each parent’s chromosomes are passed on to the child’s chromosome(s). Because the parent’s genes are passed on to the child, the species retains these genes after the parent’s death. In particular, because the parent’s genes included beneficial traits that allowed it to reproduce, the species retains these beneficial traits.

2. Refining Genetic Traits

Because a child's genes are a combination of genes from both parents, new combinations of genes can exist in the child that did not exist in either parent. If these new combinations produce a beneficial trait (perhaps as an interplay between two or more genetic traits) then the organism may pass on this new trait to its own children. Because new genetic traits are developed and only the beneficial ones are retained in the population through reproduction and heredity, the organisms of the population become more and more fit over successive generations. This is how evolution refines genetic traits.

Genetic diversity is a measure of variation within a population. Organisms vary from each other by having different genes in their chromosomes and by having their genes be in different configurations in their chromosomes. The opposite of genetic diversity is genetic homogeneity, which occurs when all the chromosomes of a population are the same. Homogeneity severely impedes a population's ability to adapt: when all organisms in a population have identical chromosomes, no new traits will be developed or passed on to children. Genetic homogeneity causes a population to lose adaptability over time. The more homogeneous the population becomes, the less able it is to adapt because it is losing the ability to develop new genetic traits. Therefore, genetic diversity, or variation, improves a population's adaptability by facilitating the refinement of genetic traits.

One phenomenon that can promote genetic diversity is mutation. Mutation occurs where an organism develops a gene that it did not receive from its parents. Mutation allows new genes be introduced into a population, thereby increasing genetic diversity.

Another way to promote genetic diversity is through diploidy. "Ploidy" refers to the number of chromosomes that are present in a given cell of an organism. In biology, haploid organisms have only one chromosome per cell, while diploid organisms have two. Examples of haploid organisms include bacteria and human sperm and eggs; most plants and animals are diploid. Because diploid organisms have two chromosomes, they also have two genes that could express any given trait. However, only one gene is expressed, and therefore determines the phenotype. The expressed gene is said to be dominant, and the unexpressed gene is recessive.

Diploidy promotes genetic diversity through this dominant/recessive gene relationship. Recessive genes may represent traits that are not beneficial to the organism. If these genes were in a haploid organism's chromosome they would lessen its chance to mate because their disadvantageous traits would necessarily be expressed. For diploid organisms, however, such genes could be shielded behind a dominant gene that represents a beneficial trait. Because the disadvantageous traits are not expressed in the diploid organism, the genes representing those traits will not lessen the organism's chance to mate. Further,

if the environment changed such that genes that once were disadvantageous become advantageous, the diploid organisms could adapt to that change more readily than the haploids. The diploid organisms are better suited to adapt to that change because they have greater genetic diversity and may be retaining the newly-advantageous genes.

B. Genetic Algorithms

Genetic algorithms also use the techniques of retaining and refining beneficial traits to cause individuals to adapt over generations. This paper uses the term organism to refer to a biological organism and individual to refer to the GA counterpart to the biological organism.

While evolution's objective is to propagate a species, GAs are used to solve a specific problem. Each GA has a problem that it is applied to (or "run against") for the purpose of developing the optimal solution to that problem. The optimal solution to the GA's problem is called its goal. The GA attempts many solutions to the problem and rates solutions according to how well they solved the problem. The measure of a solution's performance, as in biological competition, is called its fitness. A solution is said to be more "fit" the closer it approaches the optimal solution. The "problem space" is the collection of all possible solutions to the problem.

In a GA, chromosomes are the structures that contain solutions to the GA's problem. A GA has a population of many individuals, each individual containing at least one chromosome, and each chromosome encodes a single solution to the problem. Each chromosome has genes that contain aspects of its solution and, as in evolution, the individuals in a GA's population develop new aspects over successive generations. Every generation, the population is evaluated against the problem by determining the fitness of each individual in the population. The results of that evaluation determine which individuals will mate to create offspring for the next generation. By mimicking the structures and mechanics of biological evolution, GAs retain and refine beneficial aspects of solutions in an attempt to develop more fit solutions to the problem.

1. Retaining Beneficial Aspects of a Solution

A GA evaluates a population by assigning a fitness score to each individual based on the genes within the individual's chromosome. Because the chromosome encodes a solution to the GA's problem, the GA determines fitness by running the solution against the problem and scoring the solution based on how well the solution solved the problem. How well the solution solved the problem is based on the values encoded in the chromosome's genes, which represent aspects of the solution. In this paper, the optimum solution to the problem is called its goal, and the fitness score of an individual that achieves the optimum solution is called the optimum fitness.

After all of the individuals in a given generation are assigned a fitness score, some are chosen to mate based on those scores. Individuals with higher fitness scores are more likely to mate than those with lower fitness scores. Individuals with higher fitness scores have genes that encode beneficial aspects of the problem's solution. As in biological reproduction, because the individuals with genes that encode beneficial aspects to the solution are more likely to mate, these beneficial genes are more likely to be retained by the population.

2. Refining Beneficial Aspects to a Solution

After individuals have been chosen for mating they are paired up and crossover is performed on each pair. Crossover is the process by which the GA creates new individuals from the genes of the parent individual's chromosomes. Like biological reproduction, new aspects to the solution can be created by combining genes in configurations that were not found in either of the new individual's parents. Similarly, because new aspects are being created but only beneficial ones are being retained, individuals become increasingly fit over the generations.

Mutation is also modeled in GAs. As with biological mutation, mutation in GAs results in a new gene that was not passed down from an individual's parents. Also like biological mutation, mutation of a GA's individuals increases the genetic diversity of the GA's population. This genetic diversity protects the GA's ability to develop new genetic traits and facilitates the refining of genetic traits.

Additionally, haploid and diploid chromosomes are modeled in GAs. As in biology, diploid chromosomes in a GA have the ability to retain unexpressed genes. The ability to retain unexpressed genes allows the GA to retain disadvantageous genes by not expressing them but rather expressing more beneficial genes. The functionality of retaining disadvantageous genes by shielding them with beneficial genes is known as abeyance. Diploid chromosomes are expected to adapt better to dynamic problems (problems in which the goal changes from one value to another) than haploid chromosomes, because by holding genes in abeyance diploid chromosomes increased genetic diversity promotes adaptation.

C. Chromosome Design in Genetic Algorithms

In a GA, the chromosome is the structure that contains a solution to the problem. Often GA implementers employ a haploid binary chromosome encoding, because this encoding has a long history of the use and conveys certain theoretical benefits. However, if the structure and encoding of a chromosome is tailored to the GAs problem, the GA can achieve a population that displays better adaptability and develop individuals with higher fitness scores. The degree to which a chromosome is tailored to the GA's problem is called the chromosome's "closeness" to the problem.

In this paper, I explore the effects of the closeness of chromosome encodings to their problems. I implement chromosomes with encodings that are tailored to a

problem and compare their performance against the problem with the performance of chromosomes that are not tailored to the problem. I repeat this experiment with different static and dynamic problems.

In the following sections, I explain the biological structures that GAs attempt to replicate and how GAs implement these structures in a way that satisfies the conditions for evolutionary adaptation. Next, I discuss the historical development and theoretical advantages of four chromosome encoding principles: closeness to the problem space, implicit parallelism, Hamming distance, and abeyance. I then discuss the implementation of my experiments and present graphs and explanations of my results. Next, I include a section on future work that could be done to continue this line of experimentation. Finally, I summarize my conclusions.

II. BIOLOGICAL EVOLUTION AND GENETIC ALGORITHMS

We should begin by understanding some of the underlying principles of evolution, the architecture of genetic algorithms, and the relationship between evolutionary theory and GA design. Evolution is the process through which a species adapts over successive generations by retaining and refining beneficial genetic traits. By mimicking this process of biological adaptation, GAs are similarly able to retain and refine beneficial aspects of a solution to a problem. While the ultimate goal of evolution is the continued propagation of the species, a GA's goal is to obtain an optimal solution, or a solution as close to optimal as possible, to a particular problem.

Banzhaf, Nordin, Keller, and Francone stated that there are four essential conditions for evolution. These conditions are reproduction, heredity, variation, and scarcity [Banzhaf et al.1998 pg 35]. Reproduction is when parents combine their genetic material to create an offspring from that material. Heredity is the passing on of genetic material from one generation to another. Variation includes all of the ways that organisms differ that help them gain these scarce resources. Variation is represented by differences in gene values and differences in gene configurations between chromosomes. Scarcity encompasses all of the things that organisms are competing for, for example a mate, food or habitat. GAs incorporate all four of these conditions in order to properly mimic evolution and evolve better solutions.

A. Population and Generations

In biology, a population is generally a collection of concurrently-living organisms that are defined by a common trait such as location. A generation is also a collection of organisms, and it is usually defined by the proximity of the organisms' dates of birth. The processes of reproduction and death are asynchronous, happening independently of each other, and, as a result, organisms of different generations have overlapping life spans. New organisms join the population while organisms from previous generations are still part of the population. This allows for organisms of multiple generations to coexist concurrently, and therefore reproduction can occur between organisms of different generations.

In GAs, there is less of a distinction between population and generation. As in biology, a population is comprised of many individuals, but unlike biology the current generation is simply the count of how many populations the GA has evaluated. Also, unlike biology, GAs do not tend to allow individuals from different generations to coexist. Reproduction and death usually occur synchronously, that is, all of the individuals in one population are replaced by the individuals from the next population. Individuals of different generations do not coexist with each other and therefore cannot reproduce with them. Through the implementation of populations and generations, GAs satisfy the evolutionary requirements of reproduction and heredity.

Gene homogeneity is when all of the genes in a population are the same. The closer a population is to being homogeneous the less able its organisms or individuals are to adapt and developing new genetic traits. Dasgupta said that once a population has become homogenous it loses its ability to search for a new optimum [Dasgupta 1993]. Therefore, a population must contain enough organisms or individuals (and, consequently, enough distinct genes) to mitigate against the loss of genetic diversity. For this reason, GA populations tend to have a large number of individuals and to randomly initialize gene values for each individual. By using these techniques to preserve gene diversity, GAs satisfy the evolutionary requirement of variation.

B. Chromosomes

To pass genetic information from one generation to the next, an organism must have a mechanism for storing that genetic information. Chromosomes are an organism's (or, in the GA context, an individual's) mechanism for storing genetic information. Chromosomes are comprised of one or more genes, each of which defines a value for a genetic trait. For example, a child with blue eyes has a chromosome that contains a gene that defines its eyes as having a blue color.

The possible values that a gene can store for a given trait are called alleles, and the set of alleles is the set of all possible values that the gene can store. Using the eye color example, the gene has the allele that codes the DNA sequence for blue eyes, but the set of alleles that define the child's eye color consist of the

coded DNA sequence for all possible eye colors (e.g., blue, brown, hazel, and green).

When a gene's trait is expressed, or observable, in an organism, that expressed trait is called a phenotype. Following the above example, the child's blue eyes are the phenotype, i.e., the expressed trait, which reveals that the child's chromosome's eye-color gene contains the blue eye allele coding.

Like biological organisms, GAs also store genetic information for an individual in a chromosome. Each chromosome is a single attempted solution to the problem that the GA is trying to solve. For example, suppose that the problem a GA is trying solve is to identify the bits that make up a specific bit pattern. The individuals' chromosomes could be implemented as bit arrays, where the length of the array is equal to the number of bits in the bit pattern.

1	1	0	0	1
---	---	---	---	---

Figure 1. Theoretical bit pattern that a GA is trying to identify

Gene 1	Gene 2	Gene 3	Gene 4	Gene 5
1	0	1	0	1

Figure 2. Example chromosome representing the solution to a problem

In a GA, each chromosome contains genes that represent aspects of an attempted solution. The value of a gene is the allele, and the set of all possible

values for a gene is the set of all of the alleles. In the above example each bit is a gene and there are two possible alleles per gene: 0 or 1.

C. Encoding Schemes

In a genetic algorithm, the implementation details of an individual's chromosome are referred to as its encoding scheme. Many early GAs used a single one-dimensional array of bits to encode their solutions. However, there are countless ways to design a chromosome, and choice of the encoding scheme is vitally important. If the chromosome's encoding scheme is not well-suited to represent the solution it can introduce new difficulties to the GA

An encoding scheme may not be well-suited to represent a solution if it mismatches the number of allele values and phenotypes. In the previous example a binary encoding scheme was used to encode the chromosome's genes [Figure 2]. However, the chromosome could be implemented using genes that are more complex than single bits. For example, the chromosome could be implemented as an array of bytes, where each gene is represented by a single byte and considered "on" if its value is greater than half of the maximum value of a byte and "off" otherwise [Figure 3].

	Gene 1	Gene 2	Gene 3	Gene 4	Gene 5
Chromosome	0xAF	0x1D	0x97	0x5D	0xD2
Phenotype	On	Off	On	Off	On

Figure 3. Example chromosome representing a solution using bytes

In this example the number of possible phenotypes per gene is the same as the bit-encoded chromosome example [Figure 2]: each gene will be considered either “on” or “off” so there are only two phenotypes. However, the number of alleles in the byte encoding scheme is 2^8 per gene, a much greater number than the number of alleles per gene in the bit-encoded chromosome. The discrepancy in the byte encoding scheme between the number of possible allele values and number of possible phenotypes could lead to inefficiencies. Many of the bits used to encode the chromosome’s genes have no effect on the chromosome’s ability to represent its solution, so effort spent to evolve those bits is wasted.

Similarly, suppose that a floating point primitive was used instead of a byte to represent each gene, and like the byte encoding each gene is considered “on” if its value is greater than half of the maximum value of a float and considered “off” otherwise.

	Gene 1	Gene 2	Gene 3	Gene 4	Gene 5
Chromosome	175.35	29.87	151.14	93.44	210.12
Phenotype	On	Off	On	Off	On

Figure 4. Example chromosome with a float encoding scheme

The float encoding not only has the same problems as the byte encoding when trying to represent a bit, but introduces new ones. For example, it is possible for a float to achieve a value of “Not a Number” (NaN) or positive or negative infinity ($\pm\text{Inf}$). The GA would have to determine a mechanism for dealing with these

values, because the values don't correspond to the defined range for "on" and "off".

Encoding schemes that are poorly suited to a problem can introduce inefficiencies and unnecessary complexities that a better suited scheme would not introduce. A GA that uses a poorly suited encoding must become more complicated in order to deal with these difficulties.

D. Fitness Evaluation and Mate Selection

In biology, fitness is a measure of an organism's ability to pass on its genes. An organism that mates is considered more fit than one that does not because it is through mating that genes are passed on. In genetic algorithms the relationship between fitness and mating is slightly different. Instead of mating determining fitness, in GAs an organism's fitness determines its ability to mate.

In a GA an individual's fitness is a measure of how well its solution, encoded in its chromosome, solves the GA's problem. An organism's ability to mate is dependent on how well its fitness score compares to the other individuals in the population. The higher an individual's fitness score is, the more likely that individual will be able to mate.

Following the example of trying to identify a bit pattern, suppose there are two individuals with the following chromosomes:

	Gene 1	Gene 2	Gene 3	Gene 4	Gene 5
Chromosome 1	1	0	1	0	1
Chromosome 2	0	1	0	1	0

Figure 5. Example chromosomes attempting to solve the same problem

Now also suppose that the bit pattern that the GA is trying to guess is when all five bits are “on”. I refer to the objective of the problem that the GA is attempting to solve as its goal; and the optimum solution to a problem is when a chromosome perfectly encodes allele values for its genes such that its phenotypes express that goal. For this example the goal would be a string of five genes which are all “on”, and the optimal solution for this encoding would be where each gene has a value of 1.

However, GAs typically do not know the optimal value ahead of time (if they did why would they need to evolve?), Instead, all the GA can do is calculate how well its current individuals’ chromosomes solve the problem. This comparison of the individual’s encoded solution to the problem is its fitness. In the example of **[Figure 5]**, an easy way to establish fitness for these two individuals is to sum the number of bits that match the corresponding value of the bits in the goal. Chromosome 1 has three bits that match the goal, but Chromosome 2 only has two that match the goal. The values of the fitness evaluation are called the fitness score of the individuals and calculation of the fitness score is usually done in a single function known as the fitness function. Now that it is possible to say that Chromosome 1 is more fit than Chromosome 2, there exists a basis for mating selection.

GAs must have a method for selecting which individuals mate. After all of the individuals in a population have their fitness determined they can be ranked, in order from best to worst, where higher fitness is usually considered better and lower, worse. A top percentage of the fittest individuals are selected for mating and the next generation's population is created from their chromosomes. This is how highly fit genes are retained in the population and poorly fit genes are discarded. Because of its influence on determining which individuals mate, the fitness evaluation function has the greatest influence on what direction the populations will evolve and is arguably the most important part of a GA .

While individuals may not be fighting over a scarce resource, it is through fitness evaluation and mating selection that Banzhaf et al.'s requirement for scarcity is modeled in GAs.

There are a few caveats to the preceding discussion analogizing biology and GAs. First, in biology because fitness is determined by reproduction, desirable genetic traits can be lost if they do not lead to the organism reproducing. For example, a tree that produces larger than average sized fruit may be more desirable than trees that produce smaller fruit, but if it is destroyed before it is able to spread its seeds then that desirable trait is lost. In a GA, fitness is not determined by reproduction and so fitness can be related to any trait that is desirable to optimize.

Also, most mating selection techniques used in GAs always ensure that the most fit individual not only is allowed to mate but is copied into the next generation. This ensures that the GA will always have an individual in its population that is as good as or better than the previous generation's, and so fitness scores can only improve or at least hold constant. This practice of retaining the best individual is known as elitism.

Lastly, GAs do not solely select a top percentage of the individuals for mating for fear of population becoming homogenized. A population is homogeneous when all of the genes in all of the individuals' chromosomes have the same allele values. As a population gets closer to being homogeneous it loses more and more of its ability to adapt and develop new solutions to the problem that the GA is operating on. If mating is only occurring between a small sub-set of the overall population and every generation that subset is mating with individuals with the same genes as previous generations then the purpose of variation is being lessened and genetic diversity is stagnating. Most GAs incorporate some form of defense against this loss of genetic diversity, often by leveraging mutation and selecting some of the less fit organisms to reproduce.

E. Crossover

Crossover is modeled after sexual reproduction as found in biology. It is the process, within a GA, of using genes from each parent's chromosomes to create a new individual whose chromosome is comprised of those genes. The parents

are the individuals from the current population that are chosen using the mating selection techniques described above. The children become the individuals in the population that replace their parents' population.

Many different techniques exist for implementing crossover and how many offspring to produce, but often the process of crossover creates the same number of children as there were parents, thus keeping the number of individuals consistent between all generations. Because not all of the individuals from the parents' generation are selected for mating, some individuals will get to mate multiple times. It is also common for all of the genes of both parents to be used to create two children.

An example of crossover is single-point crossover. This is where a single point within the chromosome is chosen to bisect them and the latter of the resulting partitions are swapped, creating the new children. Continuing the example of using arrays of bits to encode chromosomes let us suppose the following individuals were chosen for crossover:

Mother	1	0	1	0	0
Father	0	1	1	1	0

Figure 6. Mother and father for crossover example

And let us also suppose that for single-point crossover the location between the second and third bits was chosen. All of the genes before the point of one parent, let's say the mother, should go to one of the children, here the sister.

The genes before the point of the other parent go to the other child and vice versa for the genes after the crossover point. The resulting children would look like:

Sister	1	0	1	1	0
Brother	0	1	1	0	0

Figure 7. Resulting sister and brother for crossover example

Various other crossover techniques exist, including random bit swapping, two-point crossover, and custom crossover schemes.

Unsurprisingly, because crossover is modeled after sexual reproduction, it is the main way in which GAs satisfy Banzhaf et al.'s requirement of reproduction. However it also contributes to satisfying the requirements of heredity and variation. Heredity is being satisfied because parents are passing on genes to their children, and variation is being satisfied because the genes are being combined in combinations that may not have existed in their parents.

One thing GA implementers must keep in mind is potential configurations of genes that are illegal with respect to the problem. Take, for example, the sister individual that was created from our crossover example. Suppose that the problem requires values between 0 and 20 and that the bit arrays of the chromosome are interpreted at integers. The values of the mother and father chromosomes would have all been legal (or valid) since they were within the problem's range of valid values. This range of values is known as the problem's

solution space. The sister's chromosome however would evaluate to a value of 22, which is outside the solution space and therefore the chromosome is illegal.

Davis stated that “an algorithm that generates many illegal solutions will perform worse than one that generates no illegal solutions” [Davis 1991 pg 88]. For this reason, crossover algorithms should leverage the design of the chromosome to avoid illegal solutions.

F. Mutation

Mutation is rare in biology, but because of its usefulness for keeping a population from becoming homogeneous it is used quite commonly in genetic algorithms. Mutation is a change in a gene's allele value that is not a result of its parents mating. This new gene may not have been present in either parent, and is a way variation can enter a population outside of heredity.

In a genetic algorithm mutation is usually a very minor influence when compared to the effects of crossover. However, mutation accounts for a meaningful percentage of the determination of a child's chromosomes. The exact percentage is usually determined by the GAs implementer. GAs often perform mutation just after crossover, and one way it can be accomplished is by randomly flipping a small number of bits in the entire population.

As an example, let us assume that the mutation function selected the first bit of the brother's chromosome in the previous example [Figure 7]. The resulting chromosome would look like this:

Brother	1	1	1	0	0
---------	---	---	---	---	---

Figure 8. Brother's chromosome after mutation

Again, implementers must take care to avoid mutating illegal solutions. Mutation is another way that GAs can satisfy Banzhaf et al.'s requirement of variation.

G. Haploid and Diploid Organisms

Another way to preserve genetic diversity is through diploidy. In biology the number of chromosomes in a cell is called ploidy. Haploid means that there is only one chromosome per cell. Diploid means that there are two chromosomes per cell; usually, each of the two parents contributes one chromosome to the organism. For diploid organisms both of their chromosomes contain the same number of genes, and are sufficient to define all of the phenotypes, but each chromosome may have different allele values for any given gene.

Using the child's eye color example, suppose one parent passed down a chromosome that included the brown-eyes gene for the eye color gene, and the other parent passed down a chromosome that included the blue-eyes gene. Both chromosomes have an eye color gene but the allele values of those genes are different. Because there are two genes that could be expressed, a decision

mechanism is needed to select which gene to express and which not to. The expressed gene is said to be dominant, and the unexpressed gene is said to be repressed.

Complete dominance is when only one of the gene's phenotype is expressed. Incomplete dominance is when at least some of both phenotypes are expressed, and co-dominance is when both are expressed [Calabretta et al. 1997]. To build on the previous example, the child could have both eyes be brown, which would be complete dominance. Bluish-brownish eyes would be incomplete dominance; a brown and a blue eye would be co-dominance.

When a diploid organism mates it only passes down one gene per gene location to its offspring. The organism usually does not have the ability to choose which gene is passed down, but rather one of the genes is randomly selected. This allows for a parent to pass down its recessive genes to its offspring. In this way a child could express a gene's phenotype even though its parent did not.

Passing down unexpressed genes is an advantage that diploid organisms have over haploid organisms. Genes that are not currently beneficial, in terms of scarcity and reproduction, to the organism are not lost over time, but can be retained and passed down to future generations.

For example, if for some reason brown eyes become the pinnacle of attraction and only brown-eyed individuals end up mating, then it is easy to see how a haploid chromosome structure would, in a single generation, lose all blue-eyes genes, because no blue-eyed individuals would mate and pass down the blue-eyes gene. However in a diploid organism, even though the brown-eyes gene is expressed (making the organism more likely to mate) the blue-eyes gene may still be passed onto further descendants. If, in the future, the blue-eyes gene becomes favorable again, it will have been completely bred out of the haploid population but not necessarily the diploid one.

Because diploid organisms can retain genes (even disadvantageous genes) that haploid organisms do not, gene diversity is greater in populations of diploid organisms. This diversity helps fight homogeneity and helps satisfy Banzhaf et al's condition of variety. Also, if an organism's environment changes, previously disadvantageous genes may become beneficial to the organism's ability to survive and reproduce. This is how diploid organisms are better equipped to adapt than haploids.

Many GAs use a haploid design in conjunction with a static problem. However, for a dynamic environment, one where the goal changes, diploid GAs show greater ability to adapt than their haploid counterparts. Goldberg called the idea of protecting unexpressed genes abeyance. He postulated that, like in nature, diploidy in GAs allows the retaining of potentially useful gene values [Goldberg

1989 pg 149-150]. He also said that abeyance allows those values to not be destroyed in an environment when they are not currently as useful, and others have reinforced the benefits and necessity of abeyance in diploid implementations [Smith and Goldberg 1992][Syslo et al. 1983].

Goldberg showed that, in a changing environment, dominance and abeyance allow the diploid algorithm to converge on new goals quicker than haploid algorithms [Goldberg 1989 pg 154-161]. Others have also demonstrated the benefits of diploid implementations in dynamic environments, including: [Collingwood 1996][Greene 1994][Grefenstette 1992][Hadad and Eick 1997][Ryan 1996][Syslo et al. 1983].

H. Advantages of Genetic Algorithms

One advantage of genetic algorithms is that their implementers do not need to know what the optimal solution for the problem is beforehand. Instead, by comparing one solution against another, using fitness, GAs are able to identify a comparatively optimal solution. As the GA runs it identifies good solutions and keeps them even without the implementers knowing how the solutions were arrived upon. Buckland said, "The best thing about genetic algorithms is that you do not need to know how to solve a problem; you only need to know how to encode it in a way the genetic algorithm mechanism can utilize" [Buckland 2002 pg 99].

Because fitness is constantly being evaluated, a GA may be terminated at any point and a solution will exist and its fitness will be known. This is not always true for other approaches to problem solving. A designer can even set an ending condition, for example, a fitness threshold or after a number of generations.

GAs are especially proficient at developing solutions to problems with extremely large solution sets, or where the optimal solution is not known. Although GAs do not guarantee the development of an optimal solution, by exploring a large solution set randomly a GA can take advantage of patterns or structures in the solution set that may not be known by the GA's implementer.

I. Disadvantages of Genetic Algorithms

In spite of the reasons for why someone would choose to use a genetic algorithm as a problem solving technique, there are also reasons why GAs are not suitable to all problem solving tasks.

GAs derive their power from non-deterministically exploring a solution space via repeated trial and error. This can limit the problems GAs can be applied to. If a real world or scientifically theoretical problem existed that could not be repeated or there was no known way to model it accurately, a GA would simply not be an appropriate solution technique.

GAs can be difficult to implement correctly, because the numerous design choices (including chromosome encoding scheme, fitness function, mating selection, crossover function and mutation rate) can often be overwhelming. Their specific designs are often chosen by trial and error or when the implementer has a good feel for the problem domain, rather than through a scientific, objective determination.

Also, because of the great variability of the output of a GA and the fact that trends, rather than an individual's state, represent a GA's performance, if there is an error in the design of a GA it may not make itself apparent for many generations. For similar reasons, debugging a GA can be incredibly difficult. Banzhaf et al. stated that, "...in evolutionary programming it is vitally important that the definition of the fitness function and the way parameters of adaptation are defined are done well for the results to be meaningful. Coupled with a potentially long iteration cycle, failure may not be detected until far down into the project." [Banzhaf et al. 1998]

GAs also take a significant amount of computing power and time to execute and are not generally used in time, memory or power sensitive situations. Often GAs are run off-line or used to pre-compute values which are used in a later program. Special consideration and optimization techniques exist and are used to overcome this shortcoming.

Despite these shortcomings, GAs have proven themselves to be suitable algorithms for many difficult problems, and implementers must decide for their given problem whether or not a GA should be used.

III. CHROMOSOME ENCODING PRINCIPLES

A. Closeness to the Problem Space

A chromosome's closeness to the problem space is the degree to which the chromosome's encoding is tailored to the underlying problem. Though there is no precise way to measure a chromosome's closeness to its problem space, examples of ways to achieve closeness include: having the same number of possible solutions in the chromosome as the problem, having a gene per phenotype, or having allele cardinality match the number of phenotype possibilities.

Bringing an encoding closer to the problem space can ease the design and implementation of a genetic algorithm by using the concrete problem as an example to conceptualize the design against. A further benefit is that no special conversion code is needed to convert the encoded values to value that are meaningful to the problem. This can help avoid illegal solutions, which, as mentioned before, should be avoided [Davis 1991 pg 88]. Not only should the encoding be brought as close to the problem space as possible but also the other part of the GA, for example the crossover and mutation functions. Banzhaf et al. put it this way, "A representation should always reflect fundamental facts about the problem at hand. This not only makes understanding of the search easier, but it is often a precondition of successful GA runs. Correspondingly, genetic

operators have to be chosen that allow unrestricted movement in the problem space spanned by the chosen representation” [Banzhaf et al. 1998 pg. 97].

The idea that a chromosome’s encoding should in some way echo the underlying structures of what it is representing is not a new one. Davis said designers should, “use the current encoding” [Davis 1991 pg. 56], and Goldberg proposed “the principle of meaningful building blocks”, which is stated as, “the user should select a coding so that short, low-order schemata are relevant to the underlying problem...” [Goldberg 1989 pg. 80]. Goldberg also proposed a second principle: “The user should select the smallest alphabet that permits a natural expression of the problem” [Goldberg 1989 pg 80]. Although Goldberg was making an argument for binary encodings, the phrase “natural expression of the problem” does not necessarily imply a binary alphabet, but rather the smallest alphabet that can reasonably represent the problem [Goldberg 1989 pg 80].

Despite early support for choosing encodings closer to the problem space, much of early genetic algorithm work focused solely on binary representation. Other representations, like real-value encodings, have been used with much success [Janikow and Michalewicz 1991][Michalewicz 1996][Montana 1991][Wright 1991]. Likewise, success has been shown when chromosomes incorporate structures that are designed to take into account specific aspects of the problem. For example, some have demonstrated that diploid encodings perform well when the goal oscillates between two values [Calabretta et al. 1997][Goldberg 1989 pg

153-157][Greene 1994][Ryan 1996], or that a list structure representation performs well for a scheduling optimization problem [Syswerda 1991].

Wright used real numbers for optimizing functions that involved division and fractions [De Jong 1975], and thus real-value numbers existed in the problem space [Wright 1991]. Wright did notice binary encoding out-performing the real value encodings for some experiments, but only, as he points out, when the problem contained inherent attributes that the binary encoding could take advantage of [Wright 1991].

Janikow and Michalewicz consistently identify the benefits of making the encoding and problem closer saying, “the floating point representation was introduced especially to deal with real parameter problems and we see no drawbacks of tailoring the operators to such domains” [Janikow and Michalewicz 1991]. Later Michalewicz continues work with real-encoded chromosomes on “multidimensional, high-precision numerical problem”, where he makes an even more direct argument for encoding as close to the problem space as possible saying, “The main objective behind such implementations is to move the genetic algorithm closer to the problem space” [Michalewicz 1996 pg. 97-98].

Closely matching the problem space can also ease chromosome designers’ efforts, since no conversion operations are needed to convert between the encoding scheme and the problem space. Janikow and Michalewicz note that

the search space of the floating point representation is nearly identical to that of the problem space, and that the length of the chromosome and problem are the same [Janikow and Michalewicz 1991]. They recognize that a bit representation could be extended to sufficiently represent all possible values that the real encoding represents, but observe a considerable slowdown from their mutation operator having to iterate over all of the bits in the binary representation. Development time and effort as well as computational cycles are therefore saved, by avoiding the conversion work. It makes sense that an encoding that closely matches the underlying problem is easier to conceptualize and implement than an encoding that is significantly divergent from the underlying problem, assuming the designer is already knowledgeable of the problem.

B. Implicit Parallelism

When John Holland first developed genetic algorithms, he proposed an idea called implicit parallelism, whereby information learned about a single chromosome during fitness evaluation can lend information about other similarly structured chromosomes. Holland argued that implicit parallelism held the key to optimal GA performance and that GAs using implicit parallelism had a significant advantage over those that did not [Holland 1975 pg 66-74]. A key attribute of genetic algorithms is their ability to take advantage of hidden structures and patterns, and implicit parallelism is an effort to take advantage of that.

Given an alphabet having a defined set of characters and a chromosome of some defined length, there is only a finite set of concrete individuals that can be made. For example, using a binary alphabet and a chromosome of length three, there are only 8 possible combinations [Figure 9].

Gene1	Gene2	Gene3
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

Figure 9. Eight possible combinations for a 3 bit chromosome

Now suppose the 8th combination is selected, but replace the least significant bit by a “don’t care” or wild card operator [Figure 10]. Here * represents the wild card.

Gene1	Gene2	Gene3
1	1	*

Figure 10. Example 3 bit chromosome with wild card

This instance of our example chromosome represents the set of all length three binary chromosomes that begin with ones in the first two positions: 111 and 110. Holland theorized that if any chromosome in this set, say 111, has its fitness evaluated, that fitness is related to the fitness of all of the other chromosomes in this set and contains information about their fitness. In the same way, the evaluation of 111 may contain information about other sets in which it is included.

The chromosome 111 is in the set of chromosomes that begin and end with a one [Figure 11] and the set of chromosomes that have a one in the second and third positions [Figure 12]. Therefore, according to Holland, the fitness of 111 not only contains information about 110 but also 101 and 011.

Gene1	Gene2	Gene3
1	*	1

Figure 11. Example 3 bit chromosome with wild card in the second position

Gene1	Gene2	Gene3
*	1	1

Figure 12. Example 3 bit chromosome with wild card in the first position

Additionally 111 is in the set of chromosomes that begin with a one in the first position [Figure 11] that have a one in the second position [Figure 12], and that have a one in the third position [Figure 13].

Gene1	Gene2	Gene3
1	*	*

Figure 13. Example 3 bit chromosome with wild card in the second and third positions

Gene1	Gene2	Gene3
*	1	*

Figure 14. Example 3 bit chromosome with wild card in the first and third position

Gene1	Gene2	Gene3
*	*	1

Figure 15. Example 3 bit chromosome with wild card in the first and second position

In this way, a single chromosome, 111, when evaluated, provides information about seven different chromosome and six different sets of chromosomes. Holland called these sets schemata and said that by evaluating a single individual the GA is implicitly gaining information about all of the other schemata that the individual is a part of. This gaining of information is considered to be done in parallel with the evaluation of the individual. Goldberg said, “Even though each generation we perform computation proportional to the size of the population, we get useful processing of [far more] schemata in parallel with no special bookkeeping or memory other than the population itself.” [Goldberg 1989 pg 40]. The information gained through implicit parallelism is not explicitly retained in the GA. But rather the argument is that through the fitness score, mating selection and crossover the information is guiding the evolution of the population.

Holland argued that implicit parallelism is maximized when cardinality of the encoding language is minimized. In his example he evaluates a binary chromosome of length 20 against a decimal chromosome of length 6. The binary chromosome has a cardinality of 2 resulting in a total number of approximately 1.05×10^6 combinations. This is approximately equal to the total number of combinations that the decimal chromosome can make: 10^6 . The decimal alphabet has a cardinality of 10 [Holland 1975 pg 71].

If cardinality is represented by the variable v and the length of the chromosome is represented by k then the number of schemata for that alphabet and chromosome is $(v + 1)^k$, where one is added to the cardinality to represent the “don’t care” character. For Holland’s example binary chromosome, there would be $(2 + 1)^{20}$ schemata, or approximately 3.48×10^9 . Because a chromosome matches a schemata if it has the same value or a * in a corresponding position, any single chromosome should contain information about 2^{20} schemata. For the decimal example there are $(10 + 1)^6$ schemata, or 1.77×10^6 , and a single chromosome would only match 2^6 of them. Since the binary chromosome matches a significantly greater number of schemata than the decimal chromosome, the evaluation of a binary chromosome should lend information about a much greater number of other binary chromosomes. If cardinality is minimized then the length of a chromosome must be increased to represent the same number of combinations, which will maximize k and the number of matching schemata. This is how implicit parallelism is maximized when cardinality is minimized.

Because binary is the alphabet with the least cardinality, implicit parallelism is an argument for the inherent superiority of binary encoding for chromosomes. However, implicit parallelism has not gone without opposition. Antonisse [Antonisse 1989] argued that binary encoding is not necessary to optimize implicit parallelism, and Fogel [Fogel 1995 pg. 93] stated that even if implicit parallelism is maximized it can not guarantee optimal performance of a GA, with

respect to discovering an optimal solution. Because of findings like these, some have been prompted to explore other chromosome implementations besides binary. Michalewicz showed that the use of floating-point variables can outperform binary implementations for certain problems (particularly, continuous domain optimization problems [Michalewicz 1996]), and Montana, Syswerda and Wright all used lists of real values on difficult problems with reasonable results [Montana 1991] [Syswerda 1991] [Wright 1991]. Davis also points out that bit representations are often used by theoretical GA implementations because they appear robust and are a good general solution to optimization, but that specific tailoring of the encoding may be more beneficial in specific concrete problem sets [Davis 1991 pg 63-64].

Implicit parallelism was arguably the first postulated chromosome design principle and an attempt to establish theoretical justification for chromosome design decisions. While the techniques for satisfying it are disputed, as is whether satisfaction actually produces the claimed benefits, understanding implicit parallelism and the issues surrounding it is important for chromosome designers because they will need to understand the justifications for their design decisions, and what benefits they should expect from their decisions.

C. Hamming Distance

Another early, theoretically beneficial chromosome design is to use Gray code instead of normal binary encoding. Gray code, which was first patented by Frank

Gray for use in shaft encoders [Gray 1953], is a system of binary encoding where consecutive values differ by only a single bit. For example, the numbers 7 and 8 as represented in binary are 0111 and 1000, respectively. Thus, while 7 and 8 differ by only a single value in decimal, they differ by four bits in binary. In Gray code, however, 7 and 8 are 0100 and 1100: differing by only a single value in decimal and a single bit in Gray code.

The number of bit positions that contain different values is known as the Hamming distance. In standard binary representations the Hamming distance between each value is not constant, but the advantage of Gray codes is that the Hamming distance between each value is constant and always one.

An example of how a four bit binary value is represented in binary and Gray code is presented in the table below [**Figure 16**].

Decimal	Binary code	Gray code	Binary Hamming Distance	Gray Hamming Distance
0	0000	0000		
1	0001	0001	1	1
2	0010	0011	2	1
3	0011	0010	1	1
4	0100	0110	3	1
5	0101	0111	1	1
6	0110	0101	2	1
7	0111	0100	1	1
8	1000	1100	4	1
9	1001	1101	1	1
10	1010	1111	2	1
11	1011	1110	1	1
12	1100	1010	3	1
13	1101	1011	1	1
14	1110	1001	2	1
15	1111	1000	1	1

Figure 16. Decimal, Binary, Gray encodings, and Hamming Distances

Theoretically, a Gray-encoded chromosome should more closely match the problem that the genetic algorithm is attempting to solve if the values that the problem considers a single distance away are represented as such by the chromosome. The reasoning behind the theoretical benefits of Gray-encoded chromosomes is similar to the reasoning behind implicit parallelism: information known about a given chromosome implicitly lends information about other chromosomes. In both implicit parallelism and Gray codes the bit structure of a chromosome can lend information about similarly structured chromosomes.

Because normal binary encodings have variable and potentially large Hamming distances, the bit structure of a given chromosome may not be very similar to a chromosome that has a similar result or phenotype. If the phenotypes are in

order and a single unit of distance apart, then chromosomes that are encoded with genes that also order their alleles to be a single Hamming distance apart should outperform alternative encodings which do not. However if the phenotypes are not a single unit apart, allele values which are a single Hamming distance apart may not outperform other encodings. For example, there is an infinite number of values between any two real numbers, so the distance between two phenotypes is more complex than a single conceptual unit. By tailoring an encoding to closely match the complexity of the relationship between real-value phenotypes (for example, by using floating-point primitives), the implementation of that design should be easier to conceptualize and outperform an encoding of single Hamming distances.

Although Hollstein was probably the first to suggest the superiority of Gray codes over standard binary encoding in genetic search [Hollstein 1971], the advantages of Gray codes have been pointed out by many researchers [Goldberg 1989], [Davis 1991], [Hopgood 2001 pg 185], [Michalewicz 1996 pg 98], [Caruana and Schaffer 1988], [Schaffer 1984], [Schaffer 1989], [Lucasius et al. 1991], [Rana 1997] and more.

D. Abeyance

I showed previously that dominance is an important part of a diploid organism because it protects unexpressed genes. The protecting of unexpressed genes is

called abeyance. Through abeyance genetic diversity is protected, which increases adaptability.

While it is possible to implement dominance in haploid individuals [Vekaria and Clack 1997], this paper will focus on how to implement the dominant/recessive relationship in diploid individuals. One simple implementation could be to assign each gene one bit that determines dominance. If the bit is “on”, the gene is dominant. If the bit is “off”, the gene is recessive. The dominance bit itself is not considered apart of the allele value. Although this mapping of alleles to dominance is straightforward, it is also problematic because of potentially frequent dominance conflicts. If a gene received two dominant or two recessive alleles their dominance would conflict, and it would be beneficial to resolve this conflict in a way that works with the genetic algorithm’s adaptation.

Instead of having the dominance value be apart from the allele value, different encoding schemes incorporated the dominance value in an effort to allow it to evolve with its gene. Bagley proposed a method where each allele also has associated with it an evolvable dominance value, and that the highest dominance value is the dominant allele [Bagley 1967 pg 136], but in practice these dominance values tended to converge quickly, resulting in many genes having the same dominance values [Goldberg 1989 pg 151].

A combination of Hollstein and Holland's work made a dominance scheme where a simple tertiary structure was introduced to determine dominance. Each allele is 0, 1₀, or 1, where 1 is always dominant, 1₀ is always recessive, and 0 falls between the two [Hollstein 1971] [Holland 1975 pg 112-115]. This allows for both the allele value and dominance information to be held in the same gene, and through the evolution of the gene the relative dominance evolves as well.

Goldberg called this the Hollstein-Holland triallelic and showed the dominance map for the allele values [Goldberg 1989 pg 152, 154]. Note that chromosomes encoded with Hollstein-Holland triallelic values cannot be binary encoded, as they require three allele values.

	0	1 ₀	1
0	0	0	1
1 ₀	0	1	1
1	1	1	1

Figure 17. Hollstein-Holland triallelic dominance map

As an example, suppose the following two chromosomes described an individual:

Chromosome 1	1	0	1	1 ₀	1 ₀	0	1
Chromosome 2	1	1 ₀	0	1 ₀	1	0	0

Figure 18. Diploid chromosome example

Then, after applying dominance, the individual's expressed phenotypes would be:

Phenotypes	1	0	1	1	1	0	1
------------	---	---	---	---	---	---	---

Figure 19. Expressed phenotypes

Goldberg says that the Hollstein-Holland triallelic structure is the simplest diploid structure because it contains both the allele value and its dominance value, with the minimum amount of overhead per gene [Goldberg 1989 pg 152]. Goldberg and others showed that it is important for the dominance relationship to be allowed to evolve along with the individual in order for the full benefits of a diploid design to be realized [Goldberg 1989 pg 154-161][Syslo et al. 1983]. This is done in the Hollstein-Holland triallelic structure, as each allele can change its dominance independent of the rest of the chromosome structure.

There are two immediate design problems with the Hollstein-Holland triallelic. The first is that the dominance mapping is biased towards 1s over 0s. **[Figure 17]** shows that there are twice as many 1s than 0s in the expressed phenotypes [Ryan 1996]. Calabretta et al. use an XOR **[Figure 20]** operation on normal binary alleles as their dominance mechanism, which results in an unbiased phenotype expression, although they do not cite the Hollstein-Holland triallelic's bias as a motivation for doing so [Calabretta et al. 1997].

XOR	0	1
0	0	1
1	1	0

Figure 20. XOR logic

The second problem with the Hollstein-Holland triallelic is that it does not provide a dominance mapping for genes with more than two allele values, or rather it only defines a dominance mapping for binary genes. For more complex genes

encodings the GA implementer must either devise their own extension of the Hollstein-Holland triallelic, perhaps by applying it to each bit of a multi-bit gene, or define their own dominance mapping in some other way. After acknowledging this problem, Ryan used incomplete dominance as an alternative to the Hollstein-Holland triallelic [Ryan 1996].

Goldberg observed that the Hollstein-Holland triallelic showed no significant improvement in static environments (problems where the goal does not change) and also stated that previously many diploid designs were tested in non-dynamic environments and performed poorly. He did several tests of his own of diploid GAs on dynamic environments, which showed that the individuals did in fact adapt to changing environments [Goldberg 1989 pg 154-161]. However, in Goldberg's experiments as in many others the dynamic environment alternated between two goals repeatedly over time [Calabretta et al. 1997], [Greene 1994] and [Ryan 1996] are a few examples.

It should be expected that diploid individuals would do very well in dynamic environments where the goal alternates between two values repeatedly. There are two goal values and two sets of genes, thus the chromosome design matches the underlying problem very well. During the run of the GA the genes that are beneficial to the previous (and next) goal are being held in abeyance, while the currently expressed genes are beneficial to the current goal. When the goal value changes the genes in abeyance become the expressed genes and the

previously expressed genes will be held in abeyance until the next goal value change. It will be of interest to see diploid algorithms applied to problems whose goals do not alternate between the same two values, but rather change to unanticipated values, similar to Pettit and Swigger who used a randomly fluctuating environment for their GA [Pettit and Swigger 1983].

Abeyance is an advantage that diploid chromosomes have over haploid chromosomes because genetic diversity is protected, which aids adaptation. However, this advantage demonstrates itself the most in dynamic environments where the goal value returns to a previous value.

IV. THESIS

This thesis explores the importance of a the closeness of a chromosome's encoding scheme to the GA's problem space through experimentation. I hypothesize that chromosome encoding schemes that more closely match their problem spaces will perform better than competing encoding schemes. Specifically, I have two parts to my hypothesis. First, I hypothesize that chromosomes with gene encodings of the same primitive type as the problem's primitive type will outperform chromosomes with gene encodings of different primitive types. Second, I hypothesize that, because of abeyance, diploid individuals will outperform haploid individuals in dynamic problems, even dynamic problems that do not simply alternate between two goals. I measure chromosomes' relative performance by comparing the fitness score of their highest scoring individuals at the same generation.

V. EXPERIMENT IMPLEMENTATION

To test my hypothesis, I created three problems corresponding to three different primitive types. For each primitive type, I created a static and dynamic version for a total of six problems. I then created a chromosome for each primitive and a haploid and diploid version for a total of six chromosomes. I then ran all six chromosomes against each of the six problems, resulting in thirty-six tests [Figure 21].

Problem \ Chromosome		Binary		Byte		Float	
		Static	Dynamic	Static	Dynamic	Static	Dynamic
Binary	Haploid	1	2	3	4	5	6
	Diploid	7	8	9	10	11	12
Byte	Haploid	13	14	15	16	17	18
	Diploid	19	20	21	22	23	24
Float	Haploid	25	26	27	28	29	30
	Diploid	31	32	33	34	35	36

Figure 21. The 36 tests. Chromosomes vs Problems

I added further rigor to the test suite by implementing three initializers for each chromosome: one that would initialize the chromosome to its optimal result; one that would initialize the chromosome to its worst result; and one uniform or random initializer characteristic of what a genetic algorithm implementer would use. With these initializers, the total number of tests is one hundred and eight.

For my experiments, a run of a GA was when I selected a single chromosome primitive and ploidy and created a population of individuals of that type of chromosome, then evolved the individuals over a number of generations against

a specific problem. For each problem I chose a primitive type and whether the problem was static or dynamic. I ran each test for 1000 generations, as defined by the constant `NumberOfGenerations`. For the dynamic problems the goal changed 4 times, as defined by `NumberOfDynamicGoalValues`. I called the generation that the goal changes a change point. The generation of a change point is determined by dividing the number of generations by the number of dynamic goals. For all of my dynamic problems this means that the goal changed every 250 generations. If the number of generations was increased, for example to 2000, then the change points would change as well, in this case to every 500 generations.

According to my hypothesis, the individuals with chromosomes of a given primitive that were the same type as the test's goal's primitive should outperform the other individuals whose primitives were not same type as the goal's. Likewise, on tests which use dynamic goals, the individuals with diploid chromosomes that had the same primitive type as the goal should outperform haploid chromosome encodings that do not.

For each generation, I output the fitness score of the highest scoring individual. After the run I graphed those scores over the duration of the test. Because each test ran for 1000 generations, each test result contained 1000 fitness scores. I compared the performance of the chromosomes by running each chromosome against the chosen problem and then comparing the graphs of the fitness scores.

Graphs are compared in the following ways: the highest score after a given number of generations, how many generations until a GA evolves an individual with an optimal fitness score, and how quickly the scores increase (or the graph's slope). Also for dynamic goals the fitness scores just after a change point will help us detect abeyance. If abeyance is occurring then the fitness scores just after the third change point should be higher a diploid chromosome than for the haploid chromosome of the same primitive type. In all cases a higher fitness score is better than a lower fitness score.

The majority of the code I developed pertains to the fitness function and chromosome parts of the GA. For the remainder of the functionality I used the GA library GALib version 247 [GALib 2007]. I made some modifications to GALib, which I will discuss after I have described the parts of my experiments.

A. Problems and Goals

As mentioned above, each problem uses one of three primitive types, binary, byte, or float. Remember that a goal is the optimal solution to the given problem, but the optimal fitness score is the highest value a fitness function can award to a chromosome when comparing the chromosome to the goal. The chromosome should achieve an optimal fitness score when it perfectly matches the goal.

1. Binary Problem

For the binary problem I created a const value, `BINARY_GOAL`, set its value to 300 and used it as the goal for the problem. The fitness function simply counts how many genes are “on” in the chromosome, and so a chromosome will achieve an optimum fitness score if 300 of its genes are “on”.

2. Byte Problem

For the byte problem, the goal is a vector of byte values that I randomly generated once, and then hard-coded to be the optimum solution for every run of the test. I named the vector `goalValuesVector`. The fitness function asks each chromosome for a byte vector representing the chromosome’s solution, then compares each byte of the chromosome’s solution against the corresponding byte in `goalValuesVector`. These bytes are compared by subtracting the two values then subtracting the absolute value of that result from the max value of a byte.

For example, suppose the third byte of the chromosome’s byte vector is 82 and the third byte of `goalValuesVector` is 129. Because bytes have 8 bits, the max value of an unsigned byte is 255, so the fitness for the chromosome’s third gene would be 208 ($255 - \text{abs}(129 - 82)$). If the chromosome had the exact same value in its gene as the byte in `goalValuesVector` then the gene’s fitness score would be 255 ($255 - \text{abs}(129 - 129)$), which is the highest fitness score a gene can receive.

The chromosome's fitness score is calculated by summing the fitness scores of all of its genes. I defined `CHROMOSOME_LENGTH` to be 28, which is used as the length of both the `goalValuesVector` and chromosome's vector, so the optimal fitness score is 7140 ($255 * 28$)

3. Float Problem

For the float problem, the goal is also a vector of hard-coded, randomly determined values named `goalValuesVector`, but it is a two-dimensional vector. Like the byte problem, the float problem's fitness function compares each gene of the chromosome's vector to the corresponding gene in `goalValuesVector`; summing the gene's scores to calculate the chromosome's score. Both `goalValuesVector` and a float chromosome's vector's length is `CHROMOSOME_LENGTH`, and have a width of `FLOAT_GOAL_WIDTH`, which I defined to be 3. Therefore, there are 84 ($28 * 3$) genes per float chromosome.

I defined two more constants for the float problems: `MIN_FLOAT_VALUE` and `MAX_FLOAT_VALUE`, and assigned them the values of 0.0 and 255.0 respectively. I require all float chromosomes' genes and the float problem's goal values to stay within this range. Given this range and the fact that there are 84 genes per chromosome in a float problem, the optimal fitness score for a chromosome is 21420 ($255 * 84$).

The valid float value range was chosen for two reasons. Originally, I had allowed the genes in the float chromosomes to be any value that a float could represent. However, in the fitness function of the float problem the summation of gene fitness scores quickly grew to values larger than language primitives could hold. By limiting the range to between 0 and 255, I limited the summation to 21420, which could easily be stored in a language primitive. In GALib all fitness scores are stored in floats [GALib 2007].

Also, limiting the float chromosome's gene's range to be within 255 allowed genes to be easily cast to bytes when a float chromosome was used in the byte problem tests. Because floats can represent many values outside the range of values that a byte can represent, if genes were not limited to this range I would have had many floats that would require some sort of processing to make their values be meaningful with respect to the byte problem. By limiting the float value's range I avoided these many troublesome float genes.

The float problem's fitness function uses a function called `forceValidValues()` to ensure that a chromosome's float value for any given gene is valid, where valid values are defined to be inside the bounds of `MIN_FLOAT_VALUE` and `MAX_FLOAT_VALUE`, and not `NAN` or `INF`. If a float value is illegal then a new random value is generated to replace the illegal gene.

4. Base Implementation

I created a Goal base class from which all the classes that represent the problems inherit. The Goal base class defines some virtual functions that are called by the genetic algorithm when creating the chromosomes. These functions must be implemented or overridden by the derived classes to ensure the created chromosomes have the structure that the derived classes expect as input to their fitness functions. Examples of the information these virtual functions provide are how many genes the fitness function expects a chromosome to have, and how many bits are in the primitive type of the current problem. By making these functions virtual the GA does not need to know which problem it is working with when creating the chromosomes, and is guaranteed to have access to this information regardless of which problem it is working with.

In the Goal class I also defined some static functions for retrieving the static and dynamic goal values by problem type. I also defined static functions for determining which dynamic goal value a fitness function should be using, given the current generation. I also wrote a GoalClient interface that all of the chromosome classes implement. This ensures that the fitness functions have a uniform interface to the chromosomes they receive, regardless of chromosome type.

B. Dynamic Goals

In the Goal base class I implemented a function, `getGoalValuesByGoalType()` that returns a class that represents the goal for the given problem. The class I called `GoalValue`, which for static problems holds only a single goal value, but for dynamic problems holds an instance of the class `DynamicGoalValue`, which holds an array of `GoalValues`. The function `getGoalValuesByGoalType()` is called within a problem's fitness function and the `GoalValue` that it returns is compared to the current chromosome's gene values to determine the chromosome's fitness.

As stated above, for the dynamic goals there are four goal values that the problem will change between, in order. The second and fourth goals are always the same. This allows me to test if abeyance is occurring and, if so, if it is benefiting the diploid chromosomes as theorized. Remember that haploid chromosomes do not have abeyance, but because a diploid chromosome will have encountered the fourth goal previously (as the second goal), genes that were beneficial against the second goal should be held in abeyance. These abeyed genes should benefit the diploid chromosome during the fourth goal. The diploid chromosomes should exhibit higher fitness scores than the haploid chromosomes just after the third change point.

I choose not to alternate between two goals, as seen in the literature previously, because the alternating between two goals customizes the problem to fit the

chromosome design, and I wanted to test the diploid chromosome design against a more general problem. As I said before, in a problem that alternates between two goals, a diploid chromosome will store good genes to one goal in one of its chromosomes and good genes for the other goal in the other chromosome. The larger question is whether abeyance will help diploid individuals retain good genes in problems whose goals are not always the same two.

If letters are assigned to the goal values, then a problem whose goal alternates between two values would look like the string: ABABABAB . . . , repeating as long as the GA runs. Because my dynamic problems have four goals and the second and fourth are the same, their string would look like: ABCB.

Like the static problems, the goal values for the dynamic problems are hard-coded and the same for every test run. Except for the binary values they were randomly generated once before being hard-coded. I chose the binary problem's goals to be 300, 0, 150 and 0, which is all gene's "on", all "off", half "on", and all "off" again.

The fitness function contains the mechanism for changing goals in the dynamic goal problems. The problem's fitness function will check if the current goal is dynamic and if the current goal changed. This is done via a call to the function `didGoalChange()` that I added to the `GAGeneticAlgorithm` object in `GAlib`. The function `didGoalChange()` will return a value of true whenever change point is

crossed. If the problem is of the binary primitive type then the fitness function will load the new goal value into the goalValue variable, and if the problem is of the byte or float primitive type then the fitness function will load the new goal value into the goalValuesVector. From then on the new goalValue or goalValuesVector will be used for purposes of determining fitness until the next change point is crossed.

GAlib uses elitism by default, which became an important detail when implementing the dynamic goals. As noted before, elitism is the practice of preserving the highest scoring individual so as to never lose the highest fitness score this ensures that evolution always improves or at least stays constant.

I turned the elitism feature off for the first generation after a change point, on tests that have dynamic goals. If elitism was not turned off at change points GAlib would not re-evaluate the individual that was kept for elitism and the individual would have kept its old “best” score even though against the new goal, it would score worse.

C. Chromosome Implementation

1. Haploid Chromosomes

I implemented all of the haploid chromosomes as classes with a vector of genes. Each class was named after its primitive type: BinaryChromosome, ByteChromosome, and FloatChromosome. The BinaryChromosome derived

from GAlib's GA1DBinaryStringGeneome class, which kept an array of bits for me. The ByteChromosome and FloatChromosome classes are both specializations of the GA2DArrayGenomeWithGoals template class that I wrote and that inherits from GAlib's GA2DArrayGenome class. Like GA1DBinaryStringGeneome, GA2DArrayGenome keeps a two-dimensional array of genes of my classes' specified primitive type. I had the ByteChromosome and FloatChromosome inherit from GA2DArrayGenome so that both chromosomes would be able to handle the byte and float problems' requirement of one-dimensional and two-dimensional arrays. This allowed me to avoid a custom design of each chromosome for each goal; however the binary chromosome was already going to require a custom design (as I'll show below) so its class did not follow this same inheritance structure.

In all of the chromosomes I implemented three functions for use by the fitness functions. These three functions are how the fitness functions get the genes from the chromosomes that it uses for evaluating the chromosome's fitness score. These functions allow a fitness function to operate identically regardless of the chromosome it was evaluating.

The first function, numAllelesOn(), returns the number of alleles that are "on" for the binary goal. The second function, getByteArray(), returns an array of bytes as a solution to the ByteGoal. The third function, getFloatArray(), likewise returns an array of floats for the FloatGoal. In all of the chromosomes I

implemented another function, `setFloatValue()`, that is also used by the `FloatGoal`. In the case where the `forceValidValues()` function determines that a gene is illegal, the new valid value is set in the chromosome by a call to `setFloatValue()`.

I overrode each chromosome's base class's `clone()` function to ensure proper cloning behavior. Without overriding this, when the chromosome's `clone()` function would get called, the base class's `clone()` function would return an object of the type of the base class, not the derived class. Any additional members of the derived class would therefore not be cloned. I also had to override the `copy()` function for the diploid chromosomes, to correctly copy both chromosomes.

I created a `ChromosomeFactory` class to hide the ugly details of specific chromosome type creation and the handling of the 108 possible combinations of chromosome, goal and initializer types. The `ChromosomeFactory` calls a `create()` function that I implemented for each chromosome type. The `create()` function takes a concrete instance of a problem type and uses the problem's virtual functions to get information about the problem that the `create()` function then uses to setup the chromosome correctly.

The `BinaryChromosome` gets the length, width and number of bits in the problem's primitive (what I call bit depth), and multiplies them together to get a chromosome length value. The chromosome length value is how many bits the

BinaryChromosome needs to represent a solution for the given problem. For example, for the binary problem I've already stated that BINARY_GOAL is defined to be 300 and the problem is a one-dimensional array. Therefore, the length is 300 and the width is 1. Because it only takes a single bit to represent a bit, the chromosome length is 300 ($300 * 1 * 1$) for the binary problem. For the byte problem I defined the length to be CHROMOSOME_LENGTH, which I defined as 28. The byte problem is also a one-dimensional array, but there are 8 bits in a byte. So the chromosome length of the BinaryChromosome with the byte goal is 224 ($28 * 1 * 8$). Because the float problem uses a two-dimensional array and floats have 32 bits the BinaryChromosome length for the float problem is 2688 ($28 * 3 * 32$).

Because the ByteChromosome and FloatChromosome inherit from GA2DArrayGenome they do not need to multiply the width and length, but the base class accepts both as input parameters. The ByteChromosome must still use 4 bytes per float when that chromosome is used on the float problem, but in all other combinations of chromosomes and problems the width and length of the chromosomes is the same as the problem's.

Problem \ Chromosome	Binary	Byte	Float
Binary	300	224	2688
Byte	300	28	336
Float	300	28	84

Figure 22. Length of chromosomes in their own primitive type

2. Diploid Chromosomes

I implemented the diploid chromosomes by defining a new class, `DiploidChromosome`, and having that class hold references to two haploid chromosomes. I also modified the fitness functions to evaluate each chromosome, and the higher fitness score of the two is the fitness score for the `DiploidChromosome`. This is how a chromosome is expressed in the diploid architecture.

The design of a single individual having two chromosomes and comparing the chromosome's fitness before expressing the individual's fitness is the same design as Greene's [Greene 1994], and he offers three arguments for its benefit. First, what he calls "dimensional consistency" is simply that the individual's fitness has the same units as the fitness of either of the chromosomes. Second, if the recessive chromosome's fitness worsens it does not affect the individual's current fitness score. Greene called this "shielding of the recessive allele" and explained that this may help preserve abeyance in "a lengthy or radical, change in environment". This is precisely the type of experiment I have tried to implement. Finally, Greene argues that this diploid implementation will benefit from "identification of global optima", which is where if either chromosome achieves optimal fitness then the individual will also achieve optimal fitness.

D. Chromosome Initializers

I implemented three initializers for each of the chromosome primitive types. Initializers fill in the gene's allele values for the population of the first generation of the genetic algorithm's run. The initializer functions are only used on the first population and every generation after that the chromosomes evolve by the use of crossover and mutation.

I implemented an optimum and nadir (meaning "opposite") initializer for all three chromosome types: `BinaryChromosome`, `ByteChromosome`, and `FloatChromosome`. I also implemented a uniform initializer for the `ByteChromosome` and `FloatChromosome`, but did not have to for the `BinaryChromosome` because `GAlib` already provided a binary uniform initializer. The uniform initializers set a chromosome's genes to random valid values. This is standard practice for GA chromosome initialization because random initialization helps the GA explore the whole solution space and helps prevent premature convergence to a less than optimal solution.

The optimum initializers initialize the chromosomes' genes to the target goal values for the problem, causing that chromosome to achieve an optimum fitness score in the first generation. The nadir initializers initialize the chromosomes' genes to the worst values for the Goal class. This does not always mean that an individual that has been nadir initialized will receive a fitness score of 0. Rather, the nadir-initialized individual will receive the lowest possible fitness score for a

valid solution encoded in its chromosome. The reason for this is that the nadir function will initialize genes to either the minimum or maximum valid value depending on which is further from the goal value.

For example, for a byte gene running against the byte goal the range of valid values for both the gene and goal is between 0 and 255. If the goal was 175, the worst value that the nadir initializer could set the gene to would be 0, because 0 is as far away from 175 as possible within the set of valid values. In contrast, if goal was 20 the worst value that the nadir initializer could set the gene to would be 255, because 255 is the valid value that is furthest away from 20.

Earlier I showed that the fitness score is calculated by taking the absolute value of the difference between the gene's value and the goal value, and then subtracting that value from the highest possible value. For the first example the fitness score of the gene would be 80 ($255 - \text{abs}(175 - 0)$), and the second example's fitness score would be 20 ($255 - \text{abs}(20 - 255)$). Even though the genes were initialized to their worst possible values they still have a fitness score that is positive and greater than or equal to zero. This is true of all genes' fitness scores. Because the fitness score of the chromosome is the sum of the fitness for each gene, all chromosomes' fitness scores will always be positive and greater than or equal to zero.

Goal Type	Optimum Value	Nadir Value
Binary	300	0
Byte	28	1683
Float	21420	6020.1665...

Figure 23. Optimum and Nadir goal values

Originally I only implemented the optimum and nadir initializers for debugging purposes. However, the results of running the GAs with these seeded values revealed unexpected chromosome behavior, especially in the dynamic problem cases. Because of this I integrated the initializers as a permanent part of the implementation's interface and recorded the results of running the GA with the optimum and nadir initializers along with the results from GA runs that were initialized with the uniform initializers.

E. Chromosome Crossover

I used GALib's `UniformCrossover()` function to implement crossover for all chromosomes. The `UniformCrossover()` function randomly select a parent for each gene loci and copies that parent's gene value into one child and the other parent's gene into the other child. This is an algorithm that promotes diversity and avoids problems that other crossover algorithms have. For example, the single-point crossover method mentioned previously will often leave the endpoint values constant while more frequently crossing the genes in the middle of the chromosome. This is an uneven application of crossover that can lead to worse GA performance than a more even crossover algorithm like the one that the `UniformCrossover()` function implements.

Because each diploid individual contains two chromosomes, crossing diploid chromosomes means that there are four chromosomes on which to perform crossover. In the `DiploidChromosome` class I implemented a crossover function that crossed the first chromosome of the parents with each other and then crossed the second chromosomes with each other. The `DiploidChromosome`'s `crossover()` function uses the selected chromosomes' `UniformCrossover()` function, and is similar to Greene's implementation. Greene randomly chooses which of the parent's chromosomes to cross with the other parent's whereas my crossover method always crosses the first chromosomes with each other and the second chromosomes with each other [Greene 1994]. By forcing the first chromosomes to only cross with first and the second to only cross with second chromosomes, I helped enforce abeyance because new good solutions do not pollute the chromosomes of the old solutions. This is on top of the three benefits mentioned earlier that Greene offers for this diploid design.

I did not implement the Hollstein-Holland triallelic as a dominance scheme for two main reasons. As I showed in the literature, it is a biased dominance and I was concerned with its effects on abeyance. Also, it is a scheme that is only defined for binary primitive types and my experiments included other primitive types than only binary. While I could have used the Hollstein-Holland triallelic for the binary chromosomes, that would have meant that my chromosomes were not all using the same dominance scheme which adds another variable when comparing performance results.

F. Chromosome Mutators

The mutator function adds variation to a population by modifying a number of genes in an effort to promote genetic diversity and fight homogeneity. In my experiments the mutator function is called just after crossover but before the new generation has its fitness scores determined.

I used three different mutators in my implementation, one for each of the chromosome primitive types. The BinaryChromosome uses the FlipMutator() provided by GAlib, which randomly flips a bit to the logical opposite of its current value. This method of mutation is a popular choice for binary mutators.

For the ByteChromosome class I used GAlib's SwapMutator() function that randomly selects two genes at different locations within the chromosome and swaps their values. This was the only mutator provided by GAlib for GA2DArrayGenomes. The SwapMutator() has the drawback of not introducing new gene values because it can only swap the positions of existing genes. However, the advantage of such a mutator is that it does not mutate illegal genes because it is always swapping valid genes and not changing the allele values. The SwapMutator() could generate illegal genes if the position of a gene in the chromosome affected legality, or if the SwapMutator() did not swap all of the bits of a gene. In my experiments the position of a gene does not affect legality. However, for the float problem the ByteChromosome uses multiple gene's to represent a single float, so illegal values can be mutate by the SwapMutator()

when the ByteChromosome is run against the float problem. This is not an issue when the ByteChromosome is run against the binary and byte problems.

For the FloatChromosome I implemented a RandomMutator(), which randomly selects a gene and replaces its value with a new randomly generated one. This algorithm is similar to GAlib's flip mutators, but works on GA2DArrayGenomes and their derived classes, which the flip mutators do not.

DiploidChromosomes simply call the mutate function on both of their chromosomes, which are defined by whatever primitive type the chromosome is using.

G. Binary Chromosome

The BinaryChromosome is the most complex of all the chromosomes. To ensure that the chromosome had no loss of precision when it is converted to byte and float genes it has to have sufficient bits to emulate the byte and float chromosome structures. This required custom conversion code to convert the bit array into the primitive type required by the problem's fitness function. Also, because the BinaryChromosome inherits from GAlib's GA1DBinaryStringGenome, it uses a one-dimensional array even for problems whose fitness functions require two-dimensional arrays. I implemented even more custom conversion code to convert the correct parts of the one-dimensional array into the corresponding parts of the required two-dimensional array.

When a chromosome is run against the binary problem the function `numAllelesOn()` is called on the chromosome by the problem's fitness function. `numAllelesOn()` counts how many genes are "on" in the chromosomes. For the `BinaryChromosome` `numAllelesOn()` simply counts the number of bits that have the value 1.

When a chromosome is run against the byte problem the function `getByteArray()` is called. The fitness function expects `getByteArray()` to return an array of bytes that it can compare to the goal to determine fitness. In the `BinaryChromosome` class I implemented `getByteArray()` with the help of a second function `getBytes()`. The function `getBytes()` takes an index as an argument and converts the next 8 bits into a byte. Also in `getByteArray()` I had to be careful to skip every 8 bits as `getByteArray()` iterates over the bit array to ensure that the same bit is not included in multiple returned bytes.

The function `getBytes()` is an example of a custom conversion function that was needed for converting between the array of bits and the required byte output type. Likewise, the fact that the function `getByteArray()` has to skip bits while iterating is a good example of extra complexity required for implementation. These are both examples of code that could easily become buggy if future unfamiliar developers were to work on this section. This is especially true because hidden requirements of the interaction of the two functions are not enforced in either function. I could work around this problem by including

getBytes()'s code within getByteArray(), or to have getByte() keep track of which index it had already processed. However, both solutions would severely degrade getByte()'s encapsulation or ability to be used independently from getByteArray().

I implemented a similar mechanic for when the BinaryChromosome is used against the float problem. In the getFloatArray() function I used another helper function called getFloatValue(). The getFloatValue() function also takes an index as input and converts the next 32 bits (the size of a float in bits) into a float. It leverages a union I created and named floatConverter. Because ints and floats have the same length in bits the union floatConverter is a union of an unsigned int and a float.

Below are example instances of the BinaryChromosome against each problem type.

	Gene 1	Gene 2	Gene 3	Gene 4	Gene 5	Gene 6	...	Gene 28
Chromosome	0	1	1	0	1	1	...	1
Phenotype	0	1	1	0	1	1	...	1

Figure 24. Example Haploid Binary Chromosome for the Binary Goal

	Gene 1	Gene 2	Gene 3	Gene 4	Gene 5	Gene 6	...	Gene 28
Chromosome 1	0	1	1	0	1	1	...	1
Chromosome 2	1	1	0	1	1	0	...	0

Figure 25. Example Diploid Binary Chromosome for the Binary Goal

	Gene 1	Gene 2	Gene 3	Gene 4	Gene 5	...	Gene 28
Chromosome	0111 0110	1000 0010	1011 0011	0010 1110	1011 1001	...	1011 0011
Phenotype	118	130	179	46	185	...	179

Figure 26. Example Haploid Binary Chromosome for the Byte Goal

	Gene 1	Gene 2	Gene 3	Gene 4	Gene 5	...	Gene 28
Chromosome 1	0111 0110	1000 0010	1011 0011	0010 1110	1011 1001	...	1011 0011
Phenotype 1	118	130	179	46	185	...	179
Chromosome 2	1111 0000	1011 1110	0011 0011	1111 0110	0010 0101	...	1010 0010
Phenotype 2	240	190	51	246	37	...	162

Figure 27. Example Diploid Binary Chromosome for the Byte Goal

	Array	Gene 1				Gene 2			...	Gene 28			
Chromosome 1	1	0100 0010	1110 1011	1111 1010	1110 0001	0100 0010	1011 1010	1110 0001	...	0100 0010	0001 0101	0100 0001	1000 1001
	2	0100 0010	0001 1101	1000 0000	0000 0000	0100 0011	0000 0001	1111 0100	...	0100 0010	1101 0001	1000 1001	0011 0111
	3	0100 0011	0001 0101	0101 0110	1100 1001	0011 1111	0100 1010	0011 1101	...	0100 0010	1100 1001	1000 0110	1010 1000
Phenotype	1	117.99				93.441			...	37.314			
	2	39.375				129.954			...	104.768			
	3	149.339				0.79			...	100.763			

Figure 28. Example Haploid Byte Chromosome for the Float Goal

	Array	Gene 1				Gene 2			...	Gene 28			
Chromosome 1	1	0100 0010	1110 1011	1111 1010	1110 0001	0100 0010	1011 1010	1110 0001	...	0100 0010	0001 0101	0100 0001	1000 1001
	2	0100 0010	0001 1101	1000 0000	0000 0000	0100 0011	0000 0001	1111 0100	...	0100 0010	1101 0001	1000 1001	0011 0111
	3	0100 0011	0001 0101	0101 0110	1100 1001	0011 1111	0100 1010	0011 1101	...	0100 0010	1100 1001	1000 0110	1010 1000
Phenotype 1	1	117.99				93.441			...	37.314			
	2	39.375				129.954			...	104.768			
	3	149.339				0.79			...	100.763			
Chromosome 2	1	0011 1111	0001 1100	0010 1000	1111 0110	0100 0011	0000 1001	0111 1110	...	0100 0011	0101 1111	1011 0100	1111 1110
	2	0100 0011	0010 0011	1110 0101	0001 1111	0100 0010	1001 1111	1010 1101	...	0100 0010	1000 0000	1111 1101	0111 0001
	3	0100 0011	0100 1011	0011 0010	0010 1101	0100 0010	0000 1110	1011 1101	...	0100 0010	1010 0101	0001 1001	1001 1010
Phenotype 2	1	0.61				137.495			...	223.707			
	2	163.895				79.838			...	64.495			
	3	203.196				35.685			...	82.550			

Figure 29. Example Diploid Byte Chromosome for the Float Goal

For clarity of comparison, all of the example chromosomes that follow have the same phenotypes as the ones in the corresponding chromosomes of the above examples.

H. Byte Chromosome

I implemented the ByteChromosome to use bytes to represent genes and a two-dimensional array of bytes to represent chromosomes. For the BinaryGoal, the ByteChromosome uses a byte per bit in the problem's length. To determine if a byte is "on" or "off", ByteChromosome's numAllelesOn() function masks the most significant bit of a gene and uses that bit's value to determine the gene's value.

By using bytes to represent bits the ByteChromosome wastes seven bits for every bit its gene represents when run against the binary problem. There are three reasons that I decided to design the ByteChromosome this way. First, by using the byte primitive for a gene I am keeping the implementation of the ByteChromosome consistent with my thesis question of comparing how well encodings of other primitive types perform, with respect to fitness, against problems of other primitive types. Second, the BinaryChromosome also wastes seven bits per chromosome bit because GALib typedefs unsigned chars as bits for use in the BinaryChromosome's base class GA1DBinaryStringGenome. So ByteChromosome wastes no more bits than BinaryChromosome against the binary problem. Lastly, if I had wanted to not waste bits I could have encoded 8

bits per byte and used bit shifting operations to obtain or set a specific bit. However, this would have required more custom conversion functions for a primitive that is more than capable of representing the two binary alleles.

The fitness function of the byte problem calls `getBytesArray()` on the chromosome that it is determining the fitness score for. Because the `ByteChromosome` stores its genes as an array of bytes in its implementation of `getBytesArray()` it simply needs to return that array to the problem's fitness function.

However, like the `BinaryChromosome` the `ByteChromosome` must combine multiple bytes to create a single float when operating on the float problem. Four bytes are required per float, and the `floatConverter` union mentioned previously, is again used for the combining of the bytes.

	Gene 1	Gene 2	Gene 3	Gene 4	Gene 5	Gene 6	...	Gene 28
Chromosome	55	177	128	103	161	162	...	134
Phenotype	0	1	1	0	1	1	...	1

Figure 30. Example Haploid Byte Chromosome for the Binary Goal

	Gene 1	Gene 2	Gene 3	Gene 4	Gene 5	Gene 6	...	Gene 28
Chromosome	55	177	128	103	161	162	...	134
Phenotype	0	1	1	0	1	1	...	1
Chromosome	179	175	79	151	227	22	...	90
Phenotype	1	1	0	1	1	0	...	0

Figure 31. Example Diploid Byte Chromosome for the Binary Goal

	Gene 1	Gene 2	Gene 3	Gene 4	Gene 5	Gene 6	Gene 7	Gene 8	Gene 9	...	Gene 28
Chromosome	118	130	179	46	185	209	33	133	116	...	179
Phenotype	118	130	179	46	185	209	33	133	116	...	179

Figure 32. Example Haploid Byte Chromosome for the Byte Goal

	Gene 1	Gene 2	Gene 3	Gene 4	Gene 5	Gene 6	Gene 7	Gene 8	...	Gene 28
Chromosome 1	118	130	179	46	185	209	33	133	...	179
Phenotype 1	118	130	179	46	185	209	33	133	...	179
Chromosome 2	240	190	51	246	37	13	102	73	...	162
Phenotype 2	240	190	51	246	37	13	102	73	...	162

Figure 33. Example Diploid Byte Chromosome for the Byte Goal

	Array	Gene 1	Gene 2	Gene 3	Gene 4	...	Gene 28
Chromosome	1	0x42EBFAE1	0x42BAE1CB	0x43753687	0x42ED06A8	...	0x42154189
	2	0x421D8000	0x4301F439	0x4378FFBE	0x4361249C	...	0x42D18937
	3	0x431556C9	0x3F4A3D71	0x4203D70A	0x43588C8B	...	0x42C986A8
Phenotype	1	117.99	93.441	245.213	118.513	...	37.314
	2	39.375	129.954	248.999	225.143	...	104.768
	3	149.339	0.79	32.96	216.549	...	100.763

Figure 34. Example Haploid Byte Chromosome for the Float Goal

	Array	Gene 1	Gene 2	Gene 3	Gene 4	...	Gene 28
Chromosome 1	1	0x42EBFAE1	0x42BAE1CB	0x43753687	0x42ED06A8	...	0x42154189
	2	0x421D8000	0x4301F439	0x4378FFBE	0x4361249C	...	0x42D18937
	3	0x431556C9	0x3F4A3D71	0x4203D70A	0x43588C8B	...	0x42C986A8
Phenotype 1	1	117.99	93.441	245.213	118.513	...	37.314
	2	39.375	129.954	248.999	225.143	...	104.768
	3	149.339	0.79	32.96	216.549	...	100.763
Chromosome 2	1	0x3F1C28F6	0x43097EB8	0x4357378D	0x42113A5E	...	0x435FB4FE
	2	0x4323E51F	0x429FAD0E	0x432FFCEE	0x42D12F1B	...	0x4280FD71
	3	0x434B322D	0x420EBD71	0x428FC083	0x431C9C29	...	0x42A5199A
Phenotype 2	1	0.61	137.495	215.217	36.307	...	223.707
	2	163.895	79.838	175.988	104.592	...	64.495
	3	203.196	35.685	71.876	156.61	...	82.550

Figure 35. Example Diploid Byte Chromosome for the Float Goal

I. Floating-Point Chromosome

Like the ByteChromosome I implemented the FloatChromosome to use floats to represent genes and a two-dimensional array of floats to represent a chromosome. Because floats are the largest of the primitives that I tested, there is no need to compose multiple together to form a type required by any of the problems' fitness functions.

When used against the binary problem the FloatChromosome uses a float to represent each bit, which is a waste of more bits than the BinaryChromosome or ByteChromosome. However this is consistent with the previously discussed design of using the chromosome's primitive type for bit representation. In order for the numAllelesOn() function of the FloatChromosome to determine if a gene was "on" or "off", I defined a constant MIDDLE_FLOAT as the midpoint between MIN_FLOAT_VALUE and MAX_FLOAT_VALUE. I defined a gene to be "on" if the float was above MIDDLE_FLOAT and "off" it was less than or equal to MIDDLE_FLOAT.

The FloatChromosome's implementation of getByteArray() truncates each gene by casting the gene to a single byte. This causes the gene to lose any decimal values it may have had, but because I defined MIN_FLOAT_VALUE and MAX_FLOAT_VALUE to be within the valid range of values for a byte, this casting will not change the gene's value by more than 1. Because the float genes can represent many fractional values that are discarded when getByteArray()

truncates the gene the FloatChromosome may waste some effort evolving values that do not end up affecting its fitness score.

Like the ByteChromosome when run against the byte problem, the FloatChromosome simply returns its internal two-dimensional array of float genes when the float problem's fitness function calls the FloatChromosome's getFloatArray().

	Gene 1	Gene 2	Gene 3	Gene 4	Gene 5	Gene 6	...	Gene 28
Chromosom	55.67	141.73	213.104	35.16	225.180	248.119	...	231.161
Phenotype	0	1	1	0	1	1	...	1

Figure 36. Example Haploid Float Chromosome for the Binary Goal

	Gene 1	Gene 2	Gene 3	Gene 4	Gene 5	Gene 6	...	Gene 28
Chromosom	55.67	141.73	213.104	35.16	225.180	248.119	...	231.161
Phenotype 1	0	1	1	0	1	1	...	1
Chromosom	216.3	181.52	10.203	175.141	156.70	63.189	...	36.16
Phenotype 2	1	1	0	1	1	0	...	0

Figure 37. Example Diploid Float Chromosome for the Binary Goal

	Gene 1	Gene 2	Gene 3	Gene 4	Gene 5	Gene 6	Gene	Gene 8	...	Gene 28
Chromosome	118.16	130.206	179.242	46.252	185.219	209.247	33.44	133.218	...	179.165
Phenotype	118	130	179	46	185	209	33	133	...	179

Figure 38. Example Haploid Float Chromosome for the Byte Goal

	Gene 1	Gene 2	Gene 3	Gene 4	Gene 5	Gene 6	Gene 7	...	Gene 28
Chromosome 1	118.16	130.206	179.242	46.252	185.219	209.247	33.44	...	179.165
Phenotype 1	118	130	179	46	185	209	33	...	179
Chromosome 2	240.71	190.17	51.49	246.229	37.20	13.182	102.232	...	162.117
Phenotype 2	240	190	51	246	37	13	102	...	162

Figure 39. Example Diploid Float Chromosome for the Byte Goal

	Array	Gene 1	Gene 2	Gene 3	Gene 4	Gene 5	Gene 6	Gene 7	...	Gene 28
Chromosome	1	117.99	93.441	245.213	118.513	56.560	243.199	183.754	...	37.314
	2	39.375	129.954	248.999	225.143	178.26	209.556	99.818	...	104.768
	3	149.339	0.79	32.96	216.549	177.790	130.428	13.448	...	100.763
Phenotype	1	117.99	93.441	245.213	118.513	56.560	243.199	183.754	...	37.314
	2	39.375	129.954	248.999	225.143	178.26	209.556	99.818	...	104.768
	3	149.339	0.79	32.96	216.549	177.790	130.428	13.448	...	100.763

Figure 40. Example Haploid Float Chromosome for the Float Goal

	Array	Gene 1	Gene 2	Gene 3	Gene 4	Gene 5	Gene 6	Gene 7	...	Gene 28
Chromosome 1	1	117.99	93.441	245.213	118.513	56.560	243.199	183.754	...	37.314
	2	39.375	129.954	248.999	225.143	178.26	209.556	99.818	...	104.768
	3	149.339	0.79	32.96	216.549	177.790	130.428	13.448	...	100.763
Phenotype 2	1	117.99	93.441	245.213	118.513	56.560	243.199	183.754	...	37.314
	2	39.375	129.954	248.999	225.143	178.26	209.556	99.818	...	104.768
	3	149.339	0.79	32.96	216.549	177.790	130.428	13.448	...	100.763
Chromosome 2	1	0.61	137.495	215.217	36.307	249.811	175.546	83.47	...	223.707
	2	163.895	79.838	175.988	104.592	216.997	241.899	113.344	...	64.495
	3	203.196	35.685	71.876	156.61	175.294	181.936	180.935	...	82.550
Phenotype 2	1	0.61	137.495	215.217	36.307	249.811	175.546	83.47	...	223.707
	2	163.895	79.838	175.988	104.592	216.997	241.899	113.344	...	64.495
	3	203.196	35.685	71.876	156.61	175.294	181.936	180.935	...	82.550

Figure 41. Example Diploid Float Chromosome for the Float Goal

J. GAlib Modifications

As I stated previously, I used GAlib version 247 for the GA functionality that I did not implement myself [GAlib 2007]. GAlib provides a framework that I customized for my experiments. I defined settings like the mutation probability (1 in 1000 genes) and population size (50 individuals per generation). Below is a diagram of the class hierarchy of the relevant classes in the version of GAlib that I used.

Although a full explanation of the GAlib glasses is outside the scope of this paper, I do want to identify key modifications I made to the library for my experiments.

1. Abstract fitness interface

As mentioned previously, I added four functions to the chromosome's base classes as an abstraction mechanism: `numAllelesOn()`, `getByteArray()`, `getFloatArray()` and `setFloatValue()`. These functions allow the fitness functions to retrieve the information necessary for evaluating the chromosomes without having to know which chromosome the fitness function is evaluating. Because all chromosomes inherit from `GAGenome`, I added these functions as virtual functions to the `GAGenome` object.

2. Ability to disable elitism

The largest set of changes that I made to GALib were to allow my GAs to disable elitism during the dynamic tests. GALib was not designed to allow a GA to disable and enable elitism dynamically. Also, GALib did not have a concept of a dynamic problem. GALib always expected elitism to be defined to be on or off before the test started and to not be changed, and GALib did not expect the goal to change during a test. I added these functionalities.

In the GAGeneticAlgorithm class I added a variable and accessor functions for tracking whether the goal changed in the current generation. In the GASimpleGA class I added a variable and accessor functions for setting and retrieving whether or not this test is dynamic. I also implemented the ability to make the test dynamic using the GAList class, which is GALib's recommended way to initialize settings for a test.

GASimpleGA uses a function called step() to create and evaluate a new population from the previous one. In step() I added code that would check if the current generation was a change point and if so it would tell the population to reset their flags that indicated that they had already been evaluated. In order to do this I had to change the GAPopulation class to tell the individuals in the population to reset their flags. I likewise had to change the GAGenome class to allow the GAPopulation class to indicate to the GAGenome class that the goal

had changed and that they should reset the flags that indicate that they are evaluated.

Besides telling the population that each individual would need to be reevaluated after a change point, I also had to modify the `step()` function to not carry over the best individual from the previous generation. The step function tests if the current generation is a change point and does not retain the best scoring individual from the previous population.

I also had to modify the `GAStatistics` class when I disabled elitism. The `GAStatistics` class keeps an array of the best individuals from past generations, and I wrote code to clear this array after every change point. This array is used for outputting the top scoring individual of the current generation. Because I am using this statistic when evaluating my results, I needed to ensure the array accurately reflected the fitness scores of the individuals in the tests. If the array of best individuals was not cleared after a change point then the best scoring individual of the current generation might not (and usually didn't) score as high as the individuals that were already in the array. Therefore, the `GAStatistics` class would report the fitness scores of individuals when they were evaluated against the previous goal, not the current goal.

VI. RESULTS

In this section I describe all 36 of the tests, first using the nadir initializer, then using the optimum initializer, and finally using the uniform initializer. The experiments that were initialized with the uniform initializer should be the most similar to a typical genetic algorithm test case because optimum and nadir values are typically not known beforehand. Although these results are from specific runs of the tests, they are representative of the results that are normally obtained. The following graphs are representative of my results in general in that they display no significant deviation from other runs of the same tests.

As discussed earlier in the section on chromosome initializers, the goal values for the binary, byte and float goals are 300, 7140, and 21420, respectively [**Figure 23**]. Each graph title indicates the problem it relates to and the goal for a given problem is indicated along the Y axis.

To establish whether or not abeyance is occurring (i.e., if chromosomes re holding onto past good genes) I compare the graphs of the haploid and diploid runs of the same experiments. My expectation is that the first few generations of the fourth goal for the diploid chromosomes will have a higher fitness score than the corresponding generations of the haploid chromosomes.

Comparing the fitness scores between the haploid and diploid chromosomes is a better indicator of abeyance than just evaluating the scores of a single

chromosome run. For example even if the fitness scores just after the third change point are higher than the scores just after the other change points within the same test, this does not necessarily indicate abeyance. This is because chromosomes may evolve genes during the generations of the previous goals that benefit the fitness score after the third change point as opposed to abeyance of previous genes.

The graphs show the fitness score for the top-scoring individual of each generation. Each generation had a population size of 50 individuals. only the top scoring individual is shown. As stated before, this appears to be consistent with the standard practice among GA researchers in evaluating their results. All of the graphs use the following line styles for each of the chromosome primitive types.

Figure 42. Result charts' line styles

A. Nadir Initialized Chromosomes with Static Goals

Graph 1. Nadir initialized haploid chromosomes vs static binary goal

This first graph shows that all three chromosomes types began with a nadir fitness score of 0 and began to improve towards the goal value of 300 [Graph 1]. As per my hypothesis, the binary chromosome outperformed the other chromosomes. I expect the binary chromosome to perform best in this test because the test is also of the binary primitive type.

The byte chromosome kept a fitness score of 0 for the entire run. This is because the byte chromosome uses the swap mutator which can only swap existing genes, not mutate new ones. Because these chromosomes were initialized with the nadir initializer all of the chromosomes started with all of their bits being “off”, so there were no “on” bits to swap. Neither mutation nor crossover could generate a single “on” bit, and the individuals never evolved over

the generations. This shows the importance of a random initialization and a mutation that can introduce new values, and how using methods like these can allow individuals to avoid converging to a non-optimal solution. It also shows the influence that the mutator can have over a GAs performance.

To ensure that the swap mutator is the reason for the byte chromosomes' odd behavior (i.e. constant fitness score of 0), I re-ran the test with the byte chromosome using the RandomMutator() function [Graph 2].

Graph 2. Nadir initialized haploid chromosomes vs static binary goal where byte chromosome uses RandomMutator()

The swap mutator is indeed to blame for the byte chromosome's poor performance. When the byte chromosome used the RandomMutator() function it performed much better than when it used the SwapMutator() function, but it still

performed the worst of the three chromosomes. Even though the byte chromosome's performance is the worst of the chromosomes' regardless of mutator, when the byte chromosome used the swap mutator it demonstrated a complete lack of adaptation not just poor performance. These results are still indicative of the influence of mutator choice on a GAs performance. For the remaining results the byte chromosome uses the SwapMutator() function to demonstrate its effects on the chromosome's performance.

The float chromosome has a similar curve as the binary chromosome, only performing a little worse. Neither the float chromosome nor the binary chromosome reached the optimum. Instead, as they approached 300 they start to look asymptotic. This is because every bit needs to be turned "on" to achieve the optimum score, and thus they only have a single way to achieve optimum fitness. This is not uncommon because many problems have only a single optimum solution. If the chromosomes had more genes and could "overshoot" the goal, there would be more than one way to achieve a perfect score of 300, and likely arrive at the optimum quicker than the chromosomes in [Graph 1].

Graph 3. Nadir initialized diploid chromosomes vs static binary goal

All three nadir initialized diploid chromosomes exhibit similar performance as their haploid versions against the same test. The binary chromosome performed the best. The byte chromosome again held a score of 0 for the entire test because of the use of the swap mutator, and the float chromosome performed slightly worse than the binary chromosome [Graph 3].

So far, the haploid and diploid chromosomes have not demonstrated much difference between each other even though the diploid chromosomes use twice as many genes and require both chromosomes to be evaluated. One might theorize that diploids would outperform haploids in all situations because the excess genes would allow for more variation in the population, but since diploid chromosomes' primary proposed benefit is abeyance, which is not in this static goal, it is not surprising that performance differences are minimal.

Graph 4. Nadir initialized haploid chromosomes vs static byte goal

This graph shows that all three chromosomes perform similarly until around the 200th generation, at which point they diverge [Graph 4]. Surprisingly the byte chromosome performs worse than the other two. This is surprising because my hypothesis says that because the byte chromosome uses the same primitive type as the byte test (and therefore being closer to the problem than the other chromosome types) the byte chromosome should perform better than the other chromosome types. After a number of generations (about 300) the byte chromosome's fitness plateaus. The cause of this is again the swap mutator. Because it can only rearrange the chromosomes' genes it removes the chromosome's ability to mutate new gene values. I re-ran the test and modified the byte chromosome to use the RandomMutator() function [Graph 5].

Graph 5. Nadir initialized haploid chromosomes vs static byte goal where byte chromosome uses RandomMutator()

Although the byte chromosome performed much better with the random mutator than the swap mutator, it still did not outperform the other two chromosomes. This pattern persists throughout all of the remaining results. The byte chromosome is never the highest performing chromosome against a byte problem (regardless of mutator), and when the byte chromosome uses the random mutator it performs very similarly to the float chromosome against the byte problems. If the byte chromosome uses the random mutator then the byte and float chromosomes have very similar crossover and mutator functions, and their set of valid gene values is also almost identical. This explains why these two chromosomes would perform similarly against the byte problems.

The binary chromosome is always the highest performing chromosome against the binary tests. The reason for this is likely due to the fact that the binary

chromosome's crossover and mutator functions work on elements that are smaller than the required primitive type: a byte. The binary chromosomes can retain parts of a byte that match the solution and evolve the parts that do not, whereas the byte and float chromosomes can only replace whole bytes with new ones, without being able to guarantee if the new value is any better of a choice for that gene than the previous value.

Although the byte chromosome was not the best performing chromosome regardless of mutator the effects of the swap mutator can again be seen here. When the byte chromosome used the swap mutator its fitness scores were lower and they plateaued very early. This test presents more evidence that the choice of mutator can impact fitness scores more than the choice of encoding scheme.

Graph 6. Nadir initialized diploid chromosomes vs static byte goal

The results of the Diploid Chromosomes in the static Byte Goal case are nearly identical to their Haploid counterparts [**Graph 6**].

Graph 7. Nadir initialized haploid chromosomes vs static float goal

The final problem is the float test. This graph shows that the float chromosome performs better than the binary and byte chromosomes, but not until after about the 450th generation [**Graph 7**].

Graph 8. Nadir initialized diploid chromosomes vs static float goal

As in the byte problem, the diploid and haploid chromosomes' exhibit similar performance against the float problem [Graph 8].

Given my hypothesis, it is not surprising that the float chromosomes have higher final fitness scores and their graphs have sharper slopes than the binary or byte chromosomes. The float chromosome is closest to the problem because it uses the same primitive type as the problem and so its superior performance is expected. The pattern of the float chromosome outperforming the other encodings against the float problem is one that recurs often in the following results. Although this trend is in-line with my overall theory of encoding closeness matching the problem, it was worth some investigation to better understand the three chromosome's behavior against the float problem.

1. Floating point architecture and non-float chromosomes

For diagnostic purposes I modified the static float goal test to count the number of times the `forceValidValues()` function had to assign a new float value to a gene that had become illegal. I then re-ran the test against the three chromosomes. The float chromosome was never assigned a new value and therefore always operated on valid values. This is because the float chromosome is initialized with all valid values, and both its crossover and mutation functions produce valid values. Crossover simply swaps whole, valid genes between chromosomes, and the mutation function replaces a whole gene with a new, random, valid float.

The binary and byte chromosomes did not always operate on valid values. The binary chromosome crosses bits and likewise can mutate any bit, which resulted in many illegal genes. When running this test, the nadir initialized binary chromosome versus the static float goal, I observed over 40,000 illegal genes that needed to be replaced in the `forceValidValues()` function. The total number of genes in a single run of the float test is 4.2 million (the structure returned to the float problem's fitness function is a $3 * 28 \text{ array} * 50 \text{ individuals per generation} * 1000 \text{ generations}$). This means that less than 1% of the genes needed to be replaced.

The byte chromosome performed much worse than both the binary and float chromosomes. Because the swap mutator can only rearrange the bytes in a gene there are only a limited number of combinations a byte chromosome can

create before it plateaus from lack of genetic diversity. Ironically the `forceValidValues()` function is the only means by which the byte chromosome could get new genetic material as it replaces illegal genes with valid random ones. On this test the byte chromosome had over one million illegal genes replaced by the `forceValidValues()` function (almost 25%). This large number indicates just how many illegal genes were created by the swap mutator and crossover function and how little the random genes from the `forceValidValues()` function helped. This is a strong argument in support of the assertion that generating lots of illegal genes results in poor GA performance [Davis 1991 pg 88]. When I ran the byte chromosome against the float problems but had it use the random mutator the chromosome still performed very poorly. The poor performance of the byte chromosome is likely due to the incredibly large number of illegal genes that the byte chromosome generates on these tests.

I also realized that the architecture of the floating-point primitive was working against the binary and byte chromosomes. Although a detailed explanation of floating-point architecture is outside the scope of this paper, it should suffice to know that single-point precision floats use a total of 32 bits. The most significant bit is a sign bit, the next eight bits are called exponent bits, and the remaining twenty three bits are the mantissa bits. The exponent bits determine the value of the number, and the mantissa bits determine placement of the decimal point [Figure 43].

Figure 43. Floating-point architecture

To help demonstrate why the floating-point architecture impeded the binary and byte chromosomes' performance, consider the following example where both a float and int represent the value 240 [**Figure 44** and **Figure 45**].

Figure 44. Floating-point representation of 240

Figure 45. Integer representation of 240

The largest valid value change that a single bit change could cause in the binary case would be to turn off the 7th bit, resulting in a value of 112 or a change of 128 [Figure 46].

Figure 46. Integer representation of 112

Turning off the most significant bit that is on in the floating representation of 240, results in the value of 7.05297×10^{-37} [Figure 47]. This is a valid value for the float goal test because it is between MIN_FLOAT_VALUE and MAX_FLOAT_VALUE, but as it is so close to 0 that the change in value is essentially 240.

possible for genes to spend a lot of effort crossing and mutating bits that hardly affect fitness score.

The binary and byte crossover and mutation functions evolve a new gene value that differs from its previous value by less than one through affecting the mantissa bits. Because the float chromosome's crossover and mutation functions do not operate on only part of a float, they do not only affect the mantissa bits of a float gene.

To summarize, non-float primitive chromosomes experience two problems that the float primitive chromosomes do not. First, the standard mutation and crossover functions for non-float types tend to create illegal floats which must be replaced by valid, but random values. Second, many valid combinations of bits only change the gene's value by fractional amounts and therefore affect the fitness score by an inconsequential amount.

B. Nadir Initialized Chromosomes with Dynamic Goals

Graph 9. Nadir initialized haploid chromosomes vs dynamic binary goal

As stated before, the goal changes every 250 generations in the dynamic problems. For the dynamic binary goal the goal's value fluctuates between 300, 0, 150 and back to 0. When graphed, the byte chromosome's results display as a clock-like pattern [Graph 9]. Because it is not evolving and initialized to all zeros, its score is always exactly 0, 300, 150, and 300 for the different goal values. The fact that the byte chromosome returns to the optimal fitness during the generations of the fourth goal is not an example of abeyance or adaptability, but rather a coincidence that the goal changed to the chromosomes' values.

The binary and float populations are more interesting to observe than the byte chromosome and more similar to the expected output of a GA. Both populations grow towards the current goal; as soon as the goal is changed, they begin to

grow towards the new goal. The binary chromosome performs better than the float chromosome during the first goal. However, because the second goal is the exact opposite of the first goal, when the goal change occurs the float chromosome outperforms the byte chromosome.

During the second goal it becomes even more apparent that the binary chromosome is better suited for this test, as its results have a steeper slope and so they begin to close the gap with the float chromosome's fitness scores. At the third goal the binary chromosome actually achieves the optimal fitness, while the float chromosome just barely misses it. At the start of the fourth goal neither chromosome begins where it left off at the end of the second goal, the last time the chromosome encountered this goal value, but the binary chromosome again outperforms the float chromosome and almost returns to the fitness score that it had achieved at the end of the second goal.

This graph demonstrates the superior performance of the binary chromosome over the float chromosome against the binary goal, not just by producing an individual with a higher fitness score each generation, but also by adapting to goal changes quicker, as seen by the slope of the plot of the best individuals per generation.

Graph 10. Nadir initialized diploid chromosomes vs dynamic binary goal

For this test the byte chromosome behaved exactly like the haploid byte chromosome [Graph 10]. The binary and float chromosomes show that abeyance is occurring and because of that fitness scores are higher.

During the first goal both the binary and float chromosomes perform about as well as their haploid counterparts, but after the first change point their performance increases greatly. They both reach higher fitness scores almost immediately. As in the haploid test, the float chromosome starts off with a higher fitness score than the binary, but the binary chromosome's results has a steeper slope and so it is closing up the gap between the two chromosomes' fitness scores. At the third goal change the binary chromosome almost immediately achieves optimal fitness. The float chromosome achieves optimal fitness only 70 generations after the binary chromosome.

Interestingly, the float chromosome outperforms the binary chromosome during the generations of the fourth goal, both starting and finishing with a higher fitness score than the binary chromosome. This is a clear sign of abeyance. The float chromosome kept many of its beneficial genes from the generations of the second goal. The binary chromosome, however, did not score as highly during the second goal generations, so it did not have as many beneficial genes to hold in abeyance.

Another reason the float chromosome kept more of the beneficial genes in abeyance than the binary chromosome is that the binary chromosome achieved optimal fitness earlier during the generations of the third goal and could have “bred out” some of the second generation scores. This is the trend toward homogeneity that I referenced in the section on populations and generations. Although the float chromosome was optimal during the generations of the third goal, it was optimal for fewer generations than the binary chromosome and therefore would have bred out fewer individuals’ second goal genes.

But why did the diploid chromosomes have higher fitness scores after the change to the second goal value than the haploid chromosomes? The answer again is abeyance. For the diploid chromosomes, the nadir initialized values were held in abeyance during the generations of the first goal. In this case, the initialized values are all zeros. When the goal changed, many genes were still holding a

value of zero, many more than in the haploid GAs. Because the diploid chromosomes were holding zeros in abeyance, and the second goal for the dynamic binary problem is zero, the diploid chromosomes achieved a greater fitness score than the haploid chromosomes.

Even though the float chromosome demonstrated greater adaptability via abeyance of the second goal for the generations of the fourth goal, that does not indicate that the float chromosome is better suited for the binary problem. The float chromosome gained an advantage by performing worse during the generations of the first goal and kept those nadir values (again 0) in abeyance all the way until the fourth goal.

Graph 11. Nadir initialized haploid chromosomes vs dynamic byte goal

This graph shows all three chromosomes behaving in an expected way for a GA in a dynamic environment. All three chromosomes begin at their nadir values and climb towards the goal [Graph 11]. Once the goal changes, the chromosomes initially score worse against the new value, but begin to evolve towards the new goal. This pattern is repeated at each goal change. The binary chromosome outperforms the other chromosomes at each goal value change, with the float chromosome a close second.

Interestingly, for each goal change the byte chromosome rises rapidly but then plateaus. While there is still some slow growth, it is easy to see that the population has only so many bytes with which the swap mutator can move around to create as close to optimal individuals as possible.

Graph 12. Nadir initialized diploid chromosomes vs dynamic byte goal

In the diploid case the graph looks almost identical to the haploid performance until the fourth goal value change [**Graph 12**]. Both the binary and byte chromosomes readily show abeyance. However, the float chromosome's beginning and ending fitness scores are almost identical to the haploid float chromosome's showing the least abeyance.

When comparing the fitness scores just after the third change point, the binary chromosome starts with a higher fitness score than any of the other diploid or haploid chromosomes. This shows that the binary chromosome kept solutions to the second goal in abeyance for the fourth goal. The binary chromosome's fitness score quickly converges towards the goal, and only comes up slightly short.

The byte chromosome's fitness score at the beginning of the generations of the fourth goal is essentially right back where it left off at the end of the second goal. This shows that the byte chromosome kept solutions to the second goal in abeyance for the fourth goal.

The float chromosome does not show an advantage from abeyance in this test, but does in the optimum and uniform initialized dynamic byte goal tests, which shows that the float chromosome is retaining the nadir initialized values which are working against its fitness score for this test. Also, as mentioned in the section problems and goals, the float chromosome's genes are truncated when

cast to a byte, so many all of the fractional values that the float genes may be representing are being discarded. The effort to evolve those fractional values is not leading to improved fitness scores.

Graph 13. Nadir initialized haploid chromosomes vs dynamic float goal

In the dynamic float test the binary chromosome displays the highest fitness scores during the first three goals, but is passed up by the float chromosome during the generations of the fourth goal [Graph 13]. The byte chromosome has the lowest fitness scores right before each change point, and its scores again plateau for many generations. The float chromosome exhibits the sharpest slope of all the chromosomes, during the generations of all of the goals. This sharp slope suggests that if the chromosomes were allowed to run for more generations against any of the goals, the float chromosome would eventually surpass the binary.

Graph 14. Nadir initialized diploid chromosomes vs dynamic float goal

Surprisingly, the diploid chromosomes have nearly identical performance as the haploid chromosomes against the same problem [Graph 14][Graph 13]. According to my hypothesis I expected the diploid chromosomes to have higher fitness than the haploid chromosomes scores just after the third change point, but for this test they do not.

The byte and binary chromosomes do not show abeyance against the dynamic float goal in any of the later tests. This is because of the reasons discussed earlier: many combinations of valid bits are not beneficial to fitness and genes that should be held in abeyance may evolve to become illegal and be replaced.

However, the float chromosome will demonstrate abeyance against the dynamic float goal when it is initialized by the other initializers. Therefore the float

chromosome does not have higher fitness scores after the third change point against this problem because its genes were nadir initialized. If these nadir initialized genes are held in abeyance and the fourth goal value is similar to the first goal value then the nadir initialized chromosome would have a lower fitness score against the second and fourth goals.

An alternative theory that could explain the lack of increased fitness after the third change point is that the gene values during the generations of the third goal value may have simply bred out the values of the generations of the second goal. To test this theory I re-ran this test, but doubled the number of generations and goal values. I ran this new test for 2000 generations with 8 goal values. The goal values are the same four goals as in the standard version of this test, but they are now encountered twice. So the 2nd, 4th, 6th, and 8th goal values were all the same [**Graph 15**].

Graph 15. Nadir initialized diploid chromosomes vs dynamic float goal run for twice as long

After the 7th change point the float chromosome still shows almost no abeyance. To further test the theory that the initialized values are the reason for the lack of benefit from abeyance I ran this same test with optimally initialized values [Graph 16].

Graph 16. Optimum initialized diploid chromosomes vs dynamic float goal run for twice as long

On this test a large increase in fitness is seen not only after the 7th change point, but after every change point where the goal returned to a value that the GA had previously experienced (3rd, 5th, 7th). This reaffirms the conclusion that the initializer is having a great effect on fitness scores, sometimes even more so than encoding type.

1. Results from Experiments of Nadir Initialized Chromosomes

Some of the chromosomes that match on primitive type to the problem's primitive type outperformed those that did not. The binary chromosomes outperformed the other chromosomes when run against the binary problems [Graph 1][Graph 3][Graph 9][Graph 10], and the float chromosome's outperformed the other chromosomes when run against the float problems [Graph 7][Graph 8][Graph

13][Graph 14]. However, the byte chromosomes did not outperform the other chromosomes against the byte problems **[Graph 4][Graph 6][Graph 11][Graph 12]**. The reason for this was because of details relating to the use of a swap mutator and initializing the chromosomes to nadir values. These results show that other parts of the GA (initialization function, crossover function, mutation function, etc) can impact fitness scores more than the choice of encoding scheme. The benefits of an encoding scheme being close to the problem can be negated by poor choice of the other parts of the GA.

The binary and byte chromosomes performed poorly against the float problems. This demonstrates that chromosomes that are further away from the problem experience unnecessary difficulties. The binary and byte chromosomes evolved illegal genes, which the float chromosomes never did, and the binary and byte chromosomes spent excess effort evolving values that did very little to improve their fitness scores.

I also observed that the diploid chromosomes did not perform any better than the haploid chromosomes when tested against static environments **[Graph 1][Graph 3][Graph 4][Graph 6][Graph 7][Graph 8]**. Abeyance does not appear to help an individual in a static environment, which is consistent with Goldberg's observations [Goldberg 1989 pg 154-161].

Some chromosomes demonstrated abeyance against the binary test and byte test, by having higher fitness score's just after the third change point [**Graph 9**][**Graph 10**][**Graph 11**][**Graph 12**]. However, these results also showed that the swap mutator and nadir initialization hampered the benefit of abeyance for several chromosomes [**Graph 9**][**Graph 10**][**Graph 11**][**Graph 12**][**Graph 13**][**Graph 14**]. Like the benefits of matching the chromosome's and problem's primitive were negated by other parts of the GA, the benefits of abeyance can also be negated by poor choices of those parts.

C. Optimum Initialized Chromosomes with Static Goals

The chromosomes graphed in this section were initialized with the optimum gene value for the goal. Thus, each chromosome received the optimum fitness score in the first generation. In the case of the static goals, no evolution is observed, or necessary, because all of the individuals in each population have optimum genes [**Graph 17**][**Graph 18**][**Graph 19**][**Graph 20**][**Graph 21**][**Graph 22**].

Graph 17. Optimum initialized haploid chromosomes vs static binary goal

Graph 18. Optimum initialized diploid chromosomes vs static binary goal

Graph 19. Optimum initialized haploid chromosomes vs static byte goal

Graph 20. Optimum initialized diploid chromosomes vs static byte goal

Graph 21. Optimum initialized haploid chromosomes vs static float goal

Graph 22. Optimum initialized diploid chromosomes vs static float goal

D. Optimum Initialized Chromosomes with Dynamic Goals

In the dynamic goal context, even though the chromosomes are initialized with the optimum genes for the first goal, they must adapt for each subsequent goal change.

Graph 23. Optimum initialized haploid chromosomes vs dynamic binary goal

As expected, all three chromosomes achieve optimum fitness during the first goal [Graph 23]. At the second goal they all begin with a fitness of 0 because the first and second goal values are at opposite ends of the valid range of values.

Like the nadir initialized chromosomes of the same test the binary chromosome results in the highest final fitness score, and the graph of its fitness scores has a steeper slope than the float chromosomes. This again reinforces my hypothesis

that the chromosome with the same primitive type as the problem performs better than those chromosomes that do not.

The byte chromosome has no “off” bits to swap so maintains a fitness score of 0 for the generations of the second and fourth goals because those goals require all genes to be “off”. Likewise, during the third generations of the goal the byte chromosome keeps a fitness score of exactly half of the optimum, because the goal requires half of the gene’s to be “on” and half “off”.

In this test the float chromosome performs similarly but slightly worse than the binary chromosome.

Graph 24. Optimum initialized diploid chromosomes vs dynamic binary goal

The results of running the diploid chromosomes against the same test surprisingly display similar behavior to the haploid chromosomes' results, especially during the fourth goal generations [**Graph 24**]. Although the fourth goal is supposed to indicate where abeyance helped the diploid chromosomes retain genes that would benefit the individual for this goal value, the performance of the diploid binary and float chromosomes is very close to that of their haploid implementations. This suggests that the diploid chromosomes did not retain any greater number of beneficial genes during the second generations of the goal than did the haploid chromosomes.

The most straightforward explanation for this result is the inverse of the explanation for exceptional abeyance performance on these same tests in the nadir initialized cases. In the optimum initialized cases all genes are initialized to "on" values. By the fourth goal very few genes with an "off" value are being held in abeyance because the population was so heavily weighted towards "on" genes early on. Therefore, the fitness scores of the diploid chromosomes are not higher than the haploid chromosomes' at the same generation.

Like the nadir initialized dynamic binary goal tests, the above results for the optimum initialized chromosomes show that initialization plays a major role in resulting fitness as it can affect which genes are held in abeyance.

Graph 25. Optimum initialized haploid chromosomes vs dynamic byte goal

The optimum initialized haploid chromosomes against the dynamic byte goals display expected behavior: during the first generations of the first goal all chromosomes are at optimum, then after each changeover the chromosomes climb towards the new goal values [Graph 25]. The byte chromosome again does not outperform the other chromosomes, for reasons mentioned earlier, but the byte chromosome's fitness scores do not plateau in these tests. This is because there are sufficient byte values for the swap mutator to work with and not enough time, before a goal change, to exhaust the combinations of good-performing genes.

Graph 26. Optimum initialized diploid chromosomes vs dynamic byte goal

When the same test is run with the diploid chromosomes, all three chromosomes have higher fitness scores just after the third change point than their counterpart haploid chromosomes had at the same generation [Graph 26][Graph 25]. This supports my hypothesis that diploid chromosomes will outperform haploid chromosomes in dynamic problems because they can retain previous solution's genes in abeyance.

Graph 27. Optimum initialized haploid chromosomes vs dynamic float goal

Against the dynamic float problem the float chromosome performs the best of all three chromosomes [Graph 27]. The float chromosome has the highest fitness score at the generations before each change point, and the float chromosome's fitness score's graphs have the steepest slopes of all the chromosomes'. This again supports my hypothesis that closeness of primitive type between chromosome and problem will result in better performance.

Graph 28. Optimum initialized diploid chromosomes vs dynamic float goal

Of the diploid chromosomes, only the float chromosome benefits from abeyance against the dynamic float problem [Graph 28]. Not much abeyance was observed against the same problem with nadir initialized chromosomes. My theory was that nadir initialized genes were being held in abeyance, and if the fourth goal value is similar to the first goal value then a nadir chromosome would have a low fitness score on both the first and fourth goals.

The test results with the optimum initialized chromosomes tend to confirm this theory. As theorized, optimum initialized chromosomes perform well on the fourth goal, because the initialized value score higher on the fourth goal, so if some of the genes are not lost over the generations before the fourth goal, the chromosome benefits from those genes.

The binary and byte chromosomes both perform worse than the float chromosome and again do not show abeyance for the reasons talked about in the discussion of the nadir version of the same goal.

1. Results from Experiments of Optimum Initialized

Because the optimally initialized chromosomes could not evolve in a static environment, only the dynamic tests revealed useful information.

The binary chromosomes again outperformed the byte and float chromosomes against the dynamic binary tests [**Graph 23**][**Graph 24**], and the float chromosomes outperformed the binary and byte chromosomes against the dynamic float tests [**Graph 27**][**Graph 28**].

Many of the diploid chromosomes in the optimally initialized tests exhibited higher fitness scores after the third change point [**Graph 25**][**Graph 26**][**Graph 27**][**Graph 28**]. The times they did not were attributable to the effects of the initialization or mutator functions [**Graph 23**][**Graph 24**] or the difficulties that the non-float chromosomes experienced against the float problems [**Graph 27**][**Graph 28**].

E. Uniform Initialized Chromosomes with Static Goals

Uniform initialization is the most common form of chromosome initialization for a number of reasons. The optimum solution is not often known so the chromosomes usually cannot be initialized to the optimum value. If the optimum solution was known then why would a GA be necessary? The nadir value is furthest from the goal, and initializing chromosomes to the nadir solution may cripple a GA depending on other design decisions (e.g. mutator choice). Random initialization is the best guarantee for genetic diversity without requiring specific knowledge of good solutions from the implementer. As I described in the previous section on my implementation of chromosome initializers, all of the chromosome's uniform initializers initialize genes to random valid values.

Graph 29. Uniformly initialized haploid chromosomes vs static binary goal

Comparing the results of the uniformly initialized chromosomes against the static binary goal shows that the byte chromosome outperforms the binary and float chromosomes against the goal [**Graph 29**]. The byte chromosome achieved an optimum fitness score hundreds of generations (400+) before either of the other two chromosomes. This behavior is contrary to my hypothesis and the previous evidence of the binary chromosome outperforming the other chromosomes against the binary goal. However, the byte chromosome's superior fitness is explained by the swap mutator. Because the swap mutator does not change the value within a gene, if ever a gene is "on" in the byte chromosome that gene will remain "on", but the binary and float chromosomes' mutation functions can change the value of a gene and therefore could generate an "off" gene.

The swap mutator has so far been detrimental to the byte chromosome's fitness and ability to adapt, but in this test it caused superior performance. This again shows that the mutator choice can have more effect on chromosome performance than encoding choice alone.

Graph 30. Uniformly initialized diploid chromosomes vs static binary goal

Like the haploid chromosomes, in the case of the diploid chromosomes run against the static binary goal, the byte chromosome again reaches optimum fitness more than four hundred generations before the binary or float chromosomes [Graph 30].

Graph 31. Uniformly initialized haploid chromosomes vs static byte goal

All three chromosomes perform almost identically to each other in the static byte goal test [Graph 31]. Even though the binary chromosome had slightly better fitness scores for the entire test, these results are an example of when encoding choice did not greatly affect the chromosomes' relative performance.

Graph 32. Uniformly initialized diploid chromosomes vs static byte goal

The diploid chromosomes against the static byte problem show similar behavior to the haploid chromosomes against the same problem [Graph 32][Graph 31].

Graph 33. Uniformly initialized haploid chromosomes vs static float goal

The float goal test results show the best examples of closeness of encoding to the problem translating into higher fitness scores [Graph 33]. My hypothesis was that chromosomes that are closer to the problem would have higher fitness scores than those that are not close to the problem. The float chromosome is closest to the problem because it has the same primitive type as the problem, and the float chromosome has the highest fitness scores of all three chromosomes.

Graph 34. Uniformly initialized diploid chromosomes vs static float goal

The diploid chromosomes against the static float problem behave in a similar way as the haploid tests [Graph 34][Graph 33]. The float chromosome produces higher fitness scores than the other two.

F. Uniformly Initialized Chromosomes with Dynamic Goals

Graph 35. Uniformly initialized haploid chromosomes vs dynamic binary goal

The uniformly initialized binary and float chromosomes performed similarly to their behavior when they were nadir and optimally initialized and run against the dynamic binary problem [Graph 35][Graph 9][Graph 23]. The binary chromosome achieved a higher fitness score than the float chromosome, and the binary chromosome adapted more quickly (as indicated by the comparative slopes of their graphs) after a change point than the float chromosome.

However, for the first time the result of the byte chromosome against the dynamic binary problem does not produce the clock-pattern fitness scores seen in the dynamic binary tests where the byte chromosome is nadir or optimum initialized.

In fact, the byte chromosome has the highest fitness score at the end of the test. The reason for this is the same as for why the byte chromosome performed better than the other chromosomes in the static version of this test; because the byte chromosome's swap mutator can not turn "on" or "off" a gene's value it is not accidentally harming the chromosome's fitness.

Graph 36. Uniformly initialized diploid chromosomes vs dynamic binary goal

The results of running the uniformly initialized diploid chromosomes against the dynamic binary goal were unexpected in two ways [Graph 36]. First, the binary and float chromosome's fitness scores had more similarities to the nadir initialized binary and float chromosome results against the dynamic binary goal than they did with the uniformly initialized haploid chromosomes or optimally initialized chromosomes run against the same test. Second, the byte

chromosome ended the test with the lowest fitness score of the three chromosomes, even though the haploid byte chromosome for the same test ended with the highest fitness score of the three haploid chromosomes.

During the generations of the second goal the binary and float chromosomes results were similar in two ways to the results of the binary and float nadir-initialized chromosomes run against the same goal. First the binary chromosome had lower fitness scores than the float chromosome throughout the generations of the second goal. Second, both the binary and float chromosomes fitness scores in the first generation after the change point were greater than 140. These two similarities are not true of the binary and float chromosomes that were optimally initialized or the uniformly initialized haploid chromosomes run against this same problem.

During the generations of the third goal the binary and float chromosomes' results were most similar to the nadir-initialized haploid binary and float chromosome's results run against the same problem. They were again similar in that in both cases the binary chromosomes had higher fitness scores and achieved optimal fitness earlier than the float chromosomes. Another similarity that the uniformly-initialized diploid binary and float chromosomes share with the nadir-initialized haploid binary and float chromosomes is that they all achieved optimal fitness after the 600th generation.

During the generations of the fourth goal the binary and float chromosomes' results were most similar to the nadir-initialized diploid binary and float chromosome's results run against the same problem. In both cases the binary chromosomes had lower fitness scores than the float chromosomes for the remainder of the tests. Also in both cases the chromosomes had fitness scores above 210 just after the final change point. These two similarities are not shared with any of the other binary or float chromosomes against the dynamic binary problem.

Given the similarities to the nadir initialized results it seems reasonable to assume that the diploid binary and float chromosomes are keeping more "off" genes in abeyance than "on" genes for the uniformly initialized chromosomes against this test.

Because the byte chromosome displayed the clock-pattern results for the nadir and optimum initialized dynamic binary goal tests, it is hard to compare those results to the results of the uniformly initialized tests. However, the uniformly-initialized results do show that uniform initialization allowed the byte chromosome to perform in a way that enabled its individuals to evolve.

Graph 37. Uniformly initialized haploid chromosomes vs dynamic byte goal

For the test of the uniformly initialized haploid chromosomes against the dynamic byte goal, the binary chromosome had the highest fitness at the end of the test [Graph 37]. Next was the byte chromosome, and lastly the float chromosome. Despite the order of the fitness performance of the chromosomes, like the optimum initialized static byte test, all three chromosomes performed similarly. This reinforces that for this problem encoding choice did not greatly affect the chromosomes' relative performance.

Graph 38. Uniformly initialized diploid chromosomes vs dynamic byte goal

The uniformly initialized diploid chromosomes against the dynamic byte goal excellently demonstrate that abeyance is occurring [Graph 38]. The graph looks very similar to the haploid graph [Graph 37] except for the beginning of the fourth generations of the goal, when all three chromosomes start with a higher fitness score than their haploid counterparts.

Graph 39. Uniformly initialized haploid chromosomes vs dynamic float goal

The results of running the uniformly initialized haploid chromosomes against the dynamic float goal show the float chromosome's fitness score is higher than the other chromosomes' and has a steeper slope than the other two chromosomes [Graph 39]. These results are consistent with the results from running the uniformly initialized chromosomes against the static float goal [Graph 33][Graph 34].

Graph 40. Uniformly initialized diploid chromosomes vs dynamic float goal

The results of running the uniformly initialized diploid chromosomes against the dynamic float goal test show that the binary and byte chromosomes display little to no abeyance after the third change point [Graph 40]; similar to their results against the other dynamic float goal problems [Graph 14][Graph 28]. The effects of abeyance is easily seen in the diploid float chromosome against the haploid on the same test. The diploid float chromosome's fitness score just after the third change point bests the haploid by over 2500 [Graph 39].

1. Results from Experiments of Uniformly Initialized Chromosomes

In the uniformly initialized chromosome tests the float chromosomes again outperformed the binary and byte chromosomes against the float problems

[Graph 33][Graph 34][Graph 40][Graph 39]. However, the byte chromosome outperformed the other two chromosomes against the binary problems [Graph 29][Graph 30][Graph 35][Graph 36]. This was again because of the swap mutator, which this time was not hampered by the initializer values of the nadir and optimum initializers, but capitalized on the random initialization values of the uniform initializer. This reinforces that the other parts of the GA, such as the initializer and mutator functions, have just as much influence on the individual's fitness score as the individual's encoding's closeness to the problem. The influence of the initialization function was further seen by the fact that when the diploid binary and float chromosomes were ran against the dynamic binary test their results were similar to the results of the nadir initialized binary and float chromosomes against the dynamic byte test [Graph 36][Graph 10].

Higher fitness scores as the result of abeyance were seen in almost all of the combinations of chromosomes and dynamic problems except the binary and byte chromosomes against the float problem [Graph 39][Graph 40]. For that test, the poor fitness scores after the third change point of those two chromosomes is attributable to the difficulties they experience from their encodings being poorly suited for the floating-point architecture.

VII. FUTURE WORK

Future work on this thesis could include refinements to the techniques and implementations already described. It could also include new experiments. Personally, I would enhance the performance and information revealed by the current implementation before developing new experiments.

A. Code Improvements

A simple start would be to track more than just the fitness score of the most fit individual per generation. GAlib tracks many different data points and could easily be modified to track any custom data the implementer desired. For example, an evaluation metric that I saw often in the literature is to track the average fitness of the population for each generation not just the fittest individual.

Also, I previously mentioned that the binary chromosomes' processor cycle performance was especially poor for the floating-point goals. This was caused by the necessity to convert the binary representation into the float primitive and back again. This is another example of how extra development effort is needed for a chromosome because its encoding is not very close to its underlying problem.

I implemented a temporary solution to this issue early on in my development by caching float values as they were converted; resetting the cache when

necessary. I eventually removed this code because it was unclear whether it achieved any appreciable performance benefit, and at the time I needed to simplify the code for debugging purposes. A better optimization may be to store the bits in a float and use bit operations to extract the bits, which would make conversion unnecessary. However, this method might result in loss of adherence to the theoretically pure binary chromosome implementation as well as wasted bits for solutions that are not evenly divisible by the number of bits in a float. Additionally a smarter `forceValidValues()` function could be developed to help alleviate the performance of the non-float performance against the float goal. Preferably, a design could be made to ensure that the chromosomes could not evolve illegal values and the `forceValidValues()` function could be removed altogether.

I ran performance profiling software on select tests and noticed that a lot of allocating and deallocating was occurring. All goal classes allocate and eventually deallocate a vector for each `fitnessFunction()` call. Because the `fitnessFunction()` is called on every individual for every generation, this is an excellent candidate for memory reuse.

Two other functions are called for every individual evaluated by a call to `fitnessFunction()`: `getGoalValuesByGoalType()` and `getCurrentGenerationsDynamicGoalValue()`. These two functions only need to

be called once per generation. Their return values could be cached for the current generation, and changed at the beginning of the next generation.

I would also change the ByteChromosome class to use the RandomMutator() function instead of the SwapMutator(). While the use of the SwapMutator() function has been informative, and a further analysis is presented in the conclusions section below, I believe that the swap mutator's limitations outweigh the few times it resulted in higher fitness score performance, and in general caused non-preferred behavior.

I also considered making more varied goals. For example the byte or float chromosome tests could have some threshold that the fitness score must fall outside of as each gene is evaluated, or else the fitness function would stop evaluation and return whatever the result at the stopping point was. If one imagines that the array of gene solutions are control commands to an algorithm or robot and if the unit being controlled deviated too much from its objective it would consider to have failed. For example, a robot navigating a maze on top of a table would fail if it deviated off of the table.

B. New Chromosome Encodings

Besides improvements to the current implementation, new chromosome encodings and new goals could be added to the tests. The new chromosomes should be designed to be closer to one of the new goals than the other

chromosomes, and all of the chromosomes should be run against all of the goals. The encoding schemes discussed in the background section of this paper, such as , Gray-code encoding and the Hollstein-Holland triallelic dominance mapping, would be a good start. The literature also suggests many other schemes: grammatical encodings [Mitchell 1996 pg 72-76] and [Antonisse 1991], rule based encodings [Grefenstette et al. 1990], order based encodings [Davis 1991 pg 77-90], [Davis 1991 pg 72-90] and [Delahaye et al. 1995], the structured GA [Dasgupta 1993], tree encodings [Mitchell 1996 pg 158] and [Banzhaf et al. 1998], and the variable-length chromosome [Hopgood 2001 pg 186] and [Schaffer 1984].

VIII. SUMMARY AND CONCLUSIONS

I hypothesized that chromosomes with encoding schemes that more closely match the problem space will perform better than chromosomes with encoding schemes that do not match the problem space. This hypothesis incorporates two parts: first that chromosomes with gene encodings of the same primitive type as the problem's primitive type will outperform chromosomes with gene encodings of different primitive types, and secondly that, because of abeyance, diploid individuals will outperform haploid individuals in dynamic problems. My experiments explored the relationship between the closeness of a chromosome's encoding to the problem space and the GA's performance.

I measured each GA's performance by running the GA against a given problem and then graphing the fitness score of the fittest individual across every generation. I then compared the graphs of different GA runs to each other. I considered the following points of comparison: the highest fitness score for a chromosome after a given number of generations; the number of generations before each GA evolved an individual with an optimal fitness score; and the slope of each graph, which indicated the speed with which a chromosome's scores increased. I also compared the graphs of haploid and diploid chromosomes run against the dynamic goal problems to determine whether abeyance was occurring. Specifically, if the fitness scores just after the third change point were higher for the diploid chromosome than for the haploid chromosome of the same primitive type, that indicated abeyance was occurring.

A. Matching on Primitive Type

In the first part of my hypothesis I theorized that chromosomes with gene encodings of the same primitive type as the primitive type of the problem would outperform chromosomes with gene encodings of primitive types that differed from those of the problem. Overall, my results confirmed this part of my hypothesis. In most of the binary problems [**Graph 1**][**Graph 3**][**Graph 9**][**Graph 23**][**Graph 24**] the binary chromosomes outperformed the other chromosomes. All of the float chromosomes outperformed the other chromosomes against the float problems [**Graph 7**][**Graph 8**][**Graph 13**][**Graph 14**][**Graph 27**][**Graph 28**][**Graph 33**][**Graph 34**][**Graph 39**][**Graph 40**].

The difficulties that the binary and byte chromosomes experienced against the float problems reinforced my conclusions about the first part of my hypothesis. When run against the float problems, the crossover and mutation functions of the binary and byte chromosomes generated illegal gene values; the crossover and mutation functions of the float chromosome did not. Also, the binary and byte chromosomes wasted effort evolving genes that had little or no effect on their fitness scores because, as discussed above in the section “Floating point architecture and non-float chromosomes”, they evolved gene values that differed from each other by less than one, so these gene’s fitness’ also differed by less than one. In contrast, the float chromosomes did not because their crossover and mutation functions operated on whole floats. The binary and byte chromosomes are considered further from the float problem because they do not

have the same primitive type as the problem. Their poor performance when run against the float problem is a direct result of their encoding not representing problem specific details. This supports my conclusion that closeness to the problem by matching chromosome primitive type to problem primitive will benefit the GA's fitness scores.

1. Hamming Distances

As discussed in the section on Hamming distances, above, one view within the literature holds that Gray-codes are an improvement over binary encoding in the GA context. Because Gray-code values are a single Hamming distance apart, this position implicitly assumes that the solutions in the problem space are also a single value apart. I believe this assumption is not always correct. If the assumption is true (i.e., if the problem space values actually are a single value apart), encodings with single value Hamming distances, like Gray-codes, closely match the problem space. However, if the problem space has values that are not a single unit apart, single Hamming distances between values may not lend performance benefits.

The assumption about solutions being single Hamming distances apart was not true for the problems that used floats as their primitive type. The relationship between one float's value and the next is a complex relationship that has many and varied Hamming distances between values. Of the three primitive type I used for chromosomes (binary, byte, float) only the float chromosome matched

the complex Hamming distance relationships that exist between the float problems' solutions, and was therefore closest to the problem. As a result, the float chromosome's outperformed the non-float chromosomes on against every float problem. Instead of concluding that Hamming distances should always be consistent and of a singular unit apart, the discussion on Hamming distances turns out to be another argument for bringing a chromosome's encoding closer to the problem space.

2. Implicit Parallelism

Earlier I presented the idea of implicit parallelism, which is a theory that a GA leverages hidden information about an unexpressed gene when it evaluates another gene of the same schemata. I showed that by arguing that implicit parallelism is maximized when alphabet cardinality is minimized, implicit parallelism is inherently an argument for the superiority of binary encoding for chromosomes. My own results do not support this theory. Although I did not design these tests to isolate implicit parallelism and attempt to test it as an encapsulated concept, my results show that simply choosing binary encoding for the benefits of implicit parallelism may not afford the GA designer better performance. The binary encoded chromosomes never outperformed the float chromosomes against the float problems, and the binary chromosomes' superior performance against the byte problems is inconclusive.

3. Bringing all parts of a GA close to the problem

According to the first part of my hypothesis the byte chromosome should have been the best performing chromosome against the byte problems, but the byte chromosome was never the best performer against the byte problem. The byte chromosome performed poorly on many of the other tests as well, and most of its poor performance can be attributed to its use of the swap mutator. Because the swap mutator only swaps genes from one location to another it does not create new genes and therefore does not improve genetic diversity as much as a mutator that does generate new genes. This shortcoming caused the byte chromosome's fitness scores to result in a clock-like pattern in some tests or plateau in others. It also generally caused the byte chromosome to perform worse than it would have with a random mutator.

While these results might lead one to assume that a swap mutator should never be used, the swap mutator did lead to superior fitness performance for the byte chromosome during the uniformly initialized binary tests. A swap mutator can also help guarantee against illegal gene values. If the swap mutator is swapping a whole valid problem space primitive then the result of the swap will be a valid problem space primitive. The byte chromosome did not swap a whole problem space primitive in the float problems because the swap mutator swapped bytes. However, if the byte chromosome's swap mutator had swapped floats like the float chromosome's crossover function, then the swap mutator would not have generated any illegal gene values, just like the float chromosome's crossover

function did not. The lesson I learned was that mutator choice affects chromosome performance more than I previously believed; the closer the mutator matched the constraints of the problem, the better the chromosome performed.

I expanded upon the lesson I learned about the swap mutator and applied it to other components of the GA. I saw that initialized values were sometimes held in abeyance for many generations, and could have a real impact on the diploid chromosomes' performance. Also, the uniformly initialized byte chromosome outperformed the binary chromosome against many of the binary problems. When initialized with the nadir or optimum initializer the byte chromosome's results displayed the clock-pattern. The byte chromosome's performance was greatly affected by the initializer choice, arguably more so than the encoding choice.

Overall, my results confirmed that not only should the chromosome's encoding be brought as close to the problem space as possible, but all of the parts of the GA should be brought close to the problem as well. As Banzhaf et al's said, "A representation should always reflect fundamental facts about the problem at hand. . . . Correspondingly, genetic operators have to be chosen that allow unrestricted movement in the problem space spanned by the chosen representation." [Banzhaf et al. 1998 pg. 97].

B. Diploidy Benefits from Abeyance

The second part of my hypothesis stated that because of abeyance, diploid individuals would outperform haploid individuals when run against dynamic problems. Because diploid chromosomes have two genes that could express a given trait a dominant/recessive mechanism is used to express one gene and repress the other. If the recessive gene is disadvantageous to the individual it does not lower the individual's fitness score because it is not expressed. In this way diploid chromosomes can retain disadvantageous genes that haploid chromosomes cannot. This retaining of disadvantageous genes is called abeyance. Through abeyance diploid chromosomes are able to increase genetic diversity by retaining genes that would otherwise be bred out of the population. Abeyance benefits the chromosome in a dynamic environment, not only by increasing genetic diversity but also by retaining genes that were beneficial to a previous goal value. If the same value becomes the goal again the diploid chromosomes may be holding genes in abeyance that were beneficial to that goal.

To determine whether the second part of my hypothesis was true, I compared the performance of haploid and diploid chromosomes in dynamic environments. A dynamic problem's goal changes values and I specifically caused the goal value to return to the same value that it was previously to test whether abeyance was benefiting the diploid chromosomes. I looked to see whether the diploid chromosomes had a higher fitness score immediately after the goal return to a

previous value than the haploid chromosomes. If the diploids did have a higher fitness score, that indicated that the diploids had successfully held past gene values in abeyance, and that abeyance had benefited their fitness score. Many of my results confirmed this. Specifically, for the nadir initialized chromosomes the tests that supported the second part of my hypothesis were: the binary and float chromosomes against the binary problem [**Graph 9**][**Graph 10**] and the binary and byte chromosomes against the byte problem [**Graph 11**][**Graph 12**]. For the optimum initialized chromosomes the tests that supported the second part of my hypothesis were: all three chromosomes against the byte problem [**Graph 25**][**Graph 26**] and the float chromosome against the float problem [**Graph 27**][**Graph 28**]. For the uniformly initialized chromosomes the tests that supported the second part of my hypothesis were: the binary and float chromosomes against the binary problem [**Graph 36**][**Graph 35**], all three chromosomes against the byte problem [**Graph 37**][**Graph 38**], and the float chromosome against the float problem.[**Graph 39**][**Graph 40**].

There were some tests where the diploid chromosomes' fitness scores after the third change point were not higher than the scores of the haploid chromosomes. For the byte chromosomes this was sometimes attributable to the swap mutator. Sometimes the cause of this was that initialized values were held in abeyance. These values caused worse fitness scores than the haploids which have effectively bred out the initialized values. Examples of initialized values lowering the diploid chromosomes' fitness scores after the third change point include the

float chromosome nadir initialized and run against the float problem [**Graph 13**][**Graph 14**], and the binary and float chromosomes optimally initialized and run against the binary problem [**Graph 23**][**Graph 24**]. As discussed with respect to the first part of my hypothesis, the comparison of the diploid chromosomes' results to the haploid chromosomes' demonstrates that the mutator and initializer can overcome the benefits of abeyance. This reinforces the conclusion that all of the parts of the GA, not only the chromosome's encoding, should be brought as close to the problem space as possible .

My hypothesis was that the chromosome encoding schemes that more closely match their problem spaces will perform better than chromosomes whose encoding schemes do not. My experiments have shown that this hypothesis is correct. However, my experiments have also shown that the other parts of the GA can greatly affect a GA's performance as well. If not all of the parts of a GA are close to the problem, the benefits of the parts that are can be negated by the parts that are not.

IX. Bibliography

[Antonisse 1989] Antonisse, J. "A New Interpretation of Schema Notation that Overturns the Binary Encoding Constraint" Proc. of the third Int. Conf. on Genetic Algorithms. San Mateo, CA. 1989.

[Antonisse 1991] Antonisse, J. "A Grammar-Based Genetic Algorithm." Foundations of Genetic Algorithms. 1991 pp 193 – 203.

[Bagley 1967] Bagley, J. D. "The behavior of adaptive systems which employ genetic and correlation algorithms." (doctoral dissertation, University of Michigan). Dissertation Abstracts International, 28(12), 5106B. (University Microfilms No. 68-7556)

[Bagnall et al. 1997] Bagnall, A., McKeown, G., and Rayward-Smith, V. "The Cryptanalysis of a Three Rotor Machine Using a Genetic Algorithm." Proceedings of the seventh international conference on Genetic Algorithms. Morgan Kaufmann Publishers, Inc. San Francisco, CA 1997

[Banzhaf et al.1998] Banzhaf, Wolfgang, Nordin, Peter, Keller, Robert E., and Francone, Frank D. Genetic Programming An Introduction. Morgan Kaufmann Publishers, Inc. San Francisco, CA. 1998

[Buckland 2002] Buckland, Mat. AI Techniques for Game Programming. Premier Press. Cincinnati, Ohio. 2002

[Calabretta et al. 1997] Raffaele Cabretta, Riccardo Galbiati, Stefano Nolti, and Domenico Parisi. "Investigating the role of diploidy in simulated populations of evolving individuals." Electronic Proceedings of the 1997 European Conference on Artificial Life. 1997

[Caruana and Schaffer 1988] Caruana, Richard A., and Schaffer J. David. "Representation and Hidden Bias: Gray vs. Binary Coding for Genetic Algorithms." Proceedings of the Fifth International Conference on Machine Learning. The University of Michigan. Ann Arbor. June 1988 pp 153-161

[Claudio et al. 2000] Rossi, Claudio, Marchiori, Elena, and Kok, Joost N. "An Adaptive Evolutionary Algorithm for the Satisfiability Problem." Proceedings of the 2000 ACM symposium on applied computing. March 2000. pp 463 - 469.

[Collingwood 1996] Collingwood, E., Corne, D. and Ross, P. "Useful Diversity via Multiploidy". Proceedings of IEEE 3rd International Conference on Evolutionary Computation. Nagoya, Japan. 1996.

[Corcoran, Wainwright 1992] Corcoran III, Arthur L. and Wainwright, Roger L. "A Genetic Algorithm for Packing in Three Dimensions" Proceedings of the 1992 ACM/SIGAPP symposium on Applied computing: technological challenges of the 1990's. 1992. Pp. 1021 – 1030.

[Dasgupta 1993] Dasgupta, Dipankar. "Optimisation in Time-Varying environments using Structured Genetic Algorithms" Technical Report No. IKBS-17-93. University of Strathclyde, Glasgow. Dec. 1993.

[Davis 1991] Davis, Lawrence. Handbook Of Genetic Algorithms. Van Nostrand Reinhold, New York. 1991.

[De Jong 1975] De Jong, K. An analysis of the behavior of a class of genetic adaptive systems. PhD thesis, University of Michigan. 1975.

[Delahaye et al. 1995] Delahaye, Daniel, Alliot, Jean-Marc, Schoenauer, Marc, and Farges, Jean-Loup. "Genetic Algorithms for Automatic Regrouping of Air Traffic Control Sectors." Proceedings of the Fourth Annual Conference on Evolutionary programming. The MIT Press, Cambridge, MA. 1995.

[Fogel 1995] Fogel, David B. Evolutionary Computation. IEEE Press. Piscataway, NJ. 1995.

[GAlib 2007] Wall, Matthew. "GAlib:Matthew's C++ Genetic Algorithms Library." Massachusetts Institute of Technology. 6 Mar. 2007 <<http://lancet.mit.edu/ga/>>.

[Goldberg 1989] Goldberg, David E. Genetic Algorithms in search Optimization & Machine Learning. Addison-Wesley Publishing Company, Inc. 1989.

[Gray 1953] F. Gray, "Pulse Code Communication", U. S. Patent 2 632 058, March 17, 1953.

[Greene 1994] F Greene. "A Method for Utilizing Diploid/Dominance in Genetic Search." World Conference on Computational Intelligence IEEE. 1994. Pp. 439 – 444.

[Grefenstette et al. 1990] Grefenstette, J. J., Ramsey, L. C., and Schultz, C. A. "learning sequential decision rules using simulation models and competition." Machine Learning, 4(5). 1990. pp 137-144.

[Grefenstette 1992] Grefenstette, J. John, "Genetic Algorithms for changing environments." Proceedings of Parallel Problem Solving From Nature (PPSN-2), Brussels, Sept. 1992. pp 137-144.

[Hadad and Eick 1997] Hadad, Ben S. and Eick, Christoph F. "Supporting Polyploidy in Genetic Algorithms Using Dominance Vectors." Proceedings of the 6th International Conference on Evolutionary Programming VI. 1997. pp 223-234

[Holland 1975] Holland, J.H. Adaptation in Natural and Artificial Systems. Ann Arbor: University of Michigan Press. 1975.

[Hollstein 1971] Hollstein, R. B. "Artificial genetic adaptation in computer control systems. (Doctoral dissertation, University of Michigan). Dissertation Abstracts International, 32(3), 1510B. (University Microfilms No. 71-23,773).

[Hopgood 2001] Hopgood, Adrian A. Intelligent Systems for Engineers and Scientists. CRC Press. New York. 2001.

[Janikow and Michalewicz 1991] Janikow, Cezary and Michlewicz, Zbigniew. "An Experimental Comparison of Binary and Floating Point Representations in Genetic Algorithms." Proc. 4th Int. Conf. on Genetic Algorithms. San Mateo, CA, Morgan Kaufmann, 1991, pp. 31–36.

[Lucasius et al. 1991] Lucasius, C., Blommers, M., Buydens, L., and Kateman, G. "A Genetic Algorithms for Conformational Analysis of DNA." Handbook of Genetic Algorithms. Van Nostrand Reinhold. 1991. pp 251-281.

[Michalewicz 1996] Michalewicz, Zbigniew. Genetic Algorithms + Data Structures = Evolution Programs. Springer-Verlag Berlin Heidelberg. NY. 1996.

[Mitchell 1996] Mitchell, Melanie. An Introduction to Genetic Algorithms. MIT Press. Cambridge, Mass. 1996.

[Montana 1991] Montana, David J. "Automated Parameter Tuning for Interpretation of Synthetic Images". Handbook of Genetic Algorithms. Van Nostrand Reinhold. New York, New York. 1991.

[Pettit and Swigger 1983] Pettit, K. and Swigger, E. "An analysis of genetic based pattern tracking and cognitive-based component tracking models of adaptation." Proceedings of National Conference on AI. Morgan Kaufmann, 1983, pp 327-332.

[Rana 1997] Rana, Soraya B. and Whitley, L. Darrell. "Bit Representations with a Twist." Proceedings of the Seventh International Conference on Genetic Algorithms. 1997. pp 188-195.

[Ryan 1996] Ryan, Conor. "The Degree of Oneness." Proceedings of the ECAI workshop on Genetic Algorithms. 1996.

[Schaffer 1984] Schaffer, J. David. "Some Experiments In Machine Learning Using Vector Evaluated Genetic Algorithms." Vanderbilt University Ph.D. Dissertation. Nashville, TN. Dec. 1984.

[Schaffer 1989] Schaffer, J. David, Caruana, Richard A., Eshelman, Larry J., and Das, Rajarshi. "A Study of Control Parameters Affecting Online Performance of Genetic Algorithms for Function Optimization." Proceedings of the Third International Conference on Genetic Algorithms. 1989. pp 51 –60.

[Smith and Goldberg 1992] Smith, R. E. and Goldberg, David E. "Diploidy and Dominance in Artificial Genetic Search." Complex Systems, 6. pg 251-285.

[Syslo et al. 1983] Syslo, Maciej M., Deo, Narsingh, and Kowalik, Janusz S. Discrete Optimization Algorithms with Pascal Programs. Prentice-Hall, Englewood Cliffs, NJ. 1983.

[Syswerda 1991] Syswerda, Gilbert. "Schedule Optimization Using Genetic Algorithms". Handbook of Genetic Algorithms. Van Nostrand Reinhold. New York, New York. 1991.

[Vekaria and Clack 1997] Vekaria, Kanta and Clack, Chris. "Haploid Genetic Programming with Dominance." Department of Computer Science University College London. 1997.

[Wright 1991] Wright, Alden H. "Genetic Algorithms for Real Parameter Optimization". Foundations of Genetic Algorithms. Morgan Kaufmann Publishers, Inc. San Mateo, CA 1991.