An Analysis of Variation Methods Used in Genetic Algorithms

CIS 4397 Independent Research

Prepared For:	Dr. Pei Wang Associate Professor Temple University
Prepared By:	Brett Crawford Computer Science Undergraduate Temple University

Table of Contents

Introduction

This report was compiled for a course, Independent Research, taken in the spring of 2015 at Temple University under the supervision of Dr. Pei Wang. The topic for the course was Evolutionary Computation and the bulk of the research that was performed involves a branch known as Genetic Algorithms (GA). This report aims to take a closer look at an important aspect of GAs known as variation.

Genetic Algorithms

Genetic algorithms are an approach to solving many different types of problems by applying Darwinian principles in an effort to evolve a solution or optimization. This technique is generally applied to problems that do not have a straightforward way of approaching an overall solution. GAs fall under the category of Evolutionary Computation, a field that is used in various disciplines including Artificial Intelligence, Electrical and Industrial Engineering, and Operations Research among others.

While the use of evolutionary tactics in computer simulation began to gain traction in the mid 1950s, GAs became popular in the early 1970s through the work of John Henry Holland. In 1975, Holland released the first major publication on GAs in his book, "Adaption in Natural and Artificial Systems." He is also credited with the creation of Holland's Schema Theorem, otherwise know as the Fundamental Theorem of Genetic Algorithms, which is widely taken to provide the basis for the power behind GAs. Their use in practical applications has grown significantly over the years along with the improvements in computing power.

In general, GAs are composed of population-based, fitness-oriented, and variation-driven properties. The procedure of evolving a solution consists of many iterations of applying these properties. The iteration begins with an initial population of possible solutions, commonly referred to as individuals. Each individual undergoes a fitness evaluation that involves some measure of its viability as a possible solution. In most cases, only the strongest possible solutions are used in the following iteration; the weaker solutions are discarded. In an effort to increase the population back up to its original size, variation is used to create new individuals. These variations generally include crossover, the "mating" of two individuals in an effort to produce superior offspring, and mutation, the alteration of a single individual in an effort to promote diversity.

Terminology

In the following section, some of the terminology common to GAs will be reviewed. These terms are all biologically inspired, however they are greatly simplified from the concepts they represent.

The lifecycle of the GA begins with an initial population of individuals, i.e., possible solutions. This number of individuals is often referred to as the population size. In general, the population size is kept constant throughout the life of the GA, but for some specialized GAs this population size can vary.

Each individual is comprised of two main components, a location and a quality. A chromosome, sometimes referred to as a genome, represents the location of the individual, i.e., a possible solution in the search space. The quality is a measure of the viability of the solution against the rest of the population, also known as the fitness value. The calculation of the fitness varies greatly, ultimately depending upon the context of the problem.

A chromosome is composed of genes. In binary representations, bits, values of either 1 or 0, represent these genes. Each gene has a value, sometimes referred to as an allele, and a position within the chromosome, the locus. There are many different possibilities for representation of the chromosome, including binary/gray code, continuous code, permutation code, and tree code among others.

Generally, a chromosome will have both a genotype representation and a phenotype representation. The genotype is the representation that is useful in the context of the GA. The phenotype is the representation that is useful in the context of the solution. For a binary representation of a real number, the binary string would be the genotype and the real number would be the phenotype.

Variation refers to changes to the chromosome and consists of two main methods, crossover and mutation. Crossover involves two separate chromosomes that are used to form a new chromosome that inherits traits from each in the hopes of producing a superior offspring. Mutation involves the alteration of a single chromosome in an effort to increase the exploration of the search space.

Binary/Gray Code Variations

Binary code is one of the more historically common choices for chromosome representation. When GAs were initially proposed, binary representation was recommended in order to imitate the biological aspects of natural organisms. There exist many schemes for encoding various types of phenotype solutions into their genotype representations, e.g., real number to binary. When attempting to represent multiple variables, the genotype representations of the variables are generally concatenated together to form a long bit string.

However, there are a few drawbacks to using a binary code representation. The Hamming cliff is the first of these drawbacks. Consider the following real numbers and their binary representations:

Real	**Binary**				
15	0	1	1	1	1
16	1	0	0	0	0

A distance of 1 on the real number line separates the numbers 15 and 16. However, in the binary representation, these numbers are separated by a change to 5 bits. Using mutation, there would only be a very small probability that number 15 would be mutated into the number 16. The Hamming distance of two bit strings of equal length is the number of positions in which the two strings differ. With this consideration in mind, it is clear that these cliffs can present a problem with variation in GAs.

One solution to the Hamming cliff drawback is the use of Gray code. Gray code is a binary numeral system in which successive real values differ in their binary representations by only one bit. The real, binary, and gray representations of the numbers 1 through 8 are shown below.

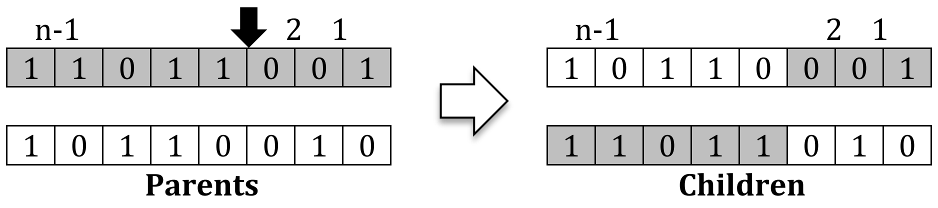
Real	**Binary**				**Gray**			
1	0	0	0	1	0	0	0	1
2	0	0	1	0	0	0	1	1
3	0	0	1	1	0	0	1	0
4	0	1	0	0	0	1	1	0
5	0	1	0	1	0	1	1	1
6	0	1	1	0	0	1	0	1
7	0	1	1	1	0	1	0	0
8	1	0	0	0	1	1	0	0

Another drawback is the loss of precision when converting between real and binary representations. There are many real numbers that cannot be properly represented using binary encoding. This drawback is generally overcome by designating a tolerance for the real value representation. The encoding scheme is modified and the length of the bits in the genotype is increased until this tolerance is accounted for. However, increasing the precision of representation can ultimately decrease the performance of the GA.

In the following sections, a few methods of variation for binary and gray code representation will be discussed.

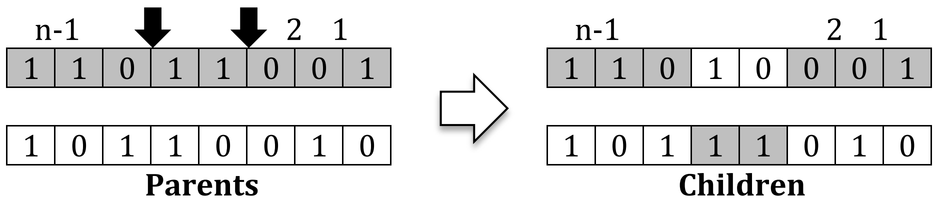
Single-Point Crossover (SPC)

The single point crossover was one of the first methods of variation used in GA and provides a somewhat decent mechanism for promoting exploration of the search space. This method of variation generally involves the creation of two new child individuals from two parent individuals. For individuals with a chromosome containing n bits, a point is selected in the range 1 to n-1 and is designated as the crossover point. Each child then receives a former half and a latter half from each of the parents.



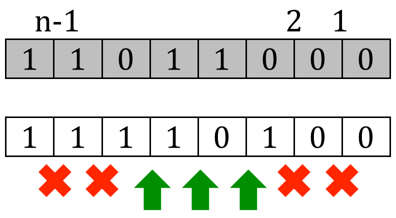
Dual-Point Crossover (DPC)

Dual-point crossover is very similar to the single-point crossover method. With dual-point crossover, the parent chromosomes are segmented into three (or more) sections. For individuals with a chromosome containing n bits, multiple points are selected in the range 1 to n-1. The division of the sections can either be handled on an alternating basis (i.e., child one gets parent one's section one and parent two's section two…) or by way of a random selection. It should be noted that while increasing the number of crossover points improves the thoroughness with which the search space is explored, it might degrade the performance of the GA.



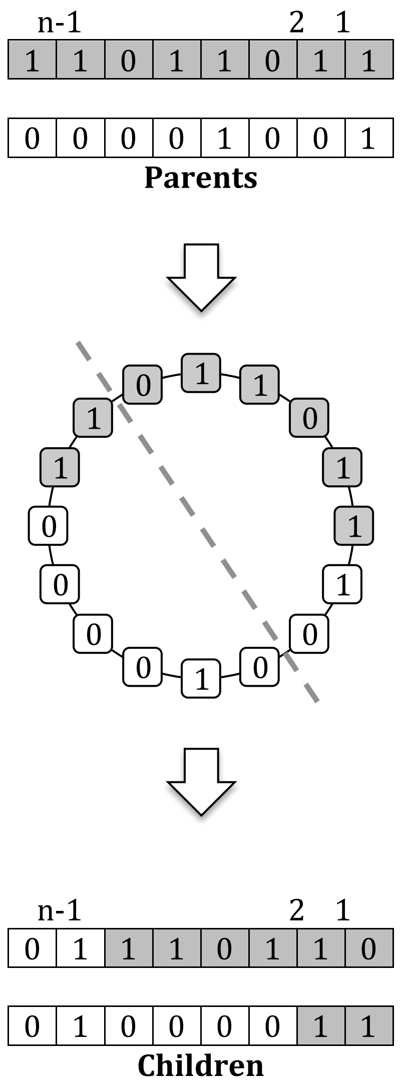
Reduced Surrogate Optimization (RS)

The reduced surrogate crossover is a technique that can be applied to the crossover methods discussed above. This technique attempts to create only new individuals wherever possible. Prior to selecting a crossover point (or points), the chromosome is checked for positions that would result in new individuals. Only these points are considered when selecting crossover points.



Ring Crossover (RC)

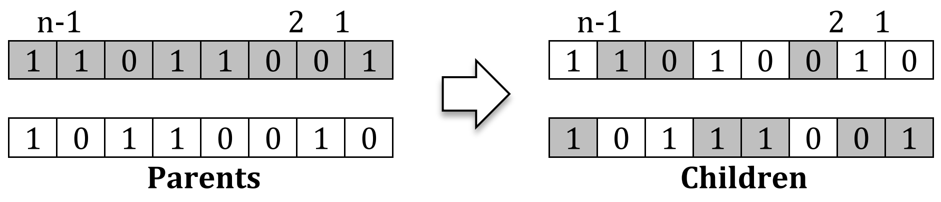
The ring crossover method attempts to increase the level of variety in the children produced. With this method, the parents are concatenated together to form a circular ring. A split point is randomly generated and the ring is segmented in two. The first child is composed of the clockwise segment relative to the split point and the second child is composed of the counterclockwise segment from the split point.



Uniform Crossover (UC)

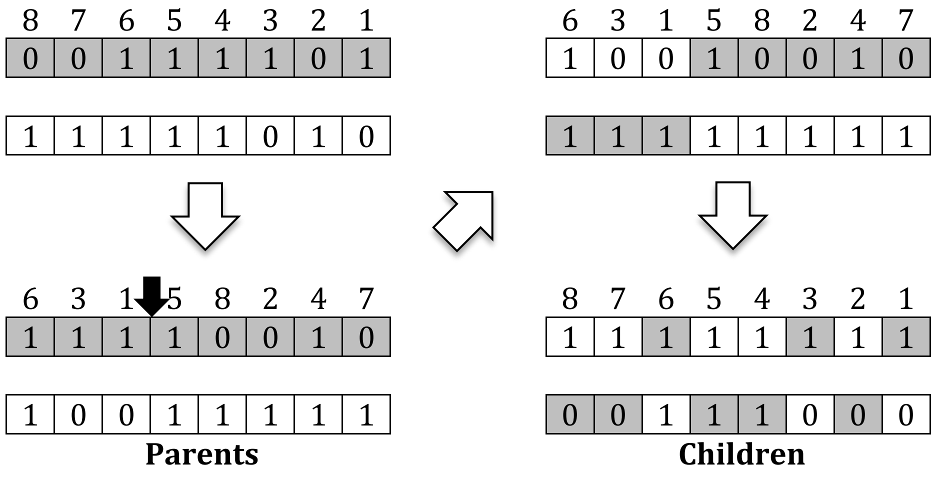
Uniform crossover is a method for extracting information from parents to children on a per gene basis. When two parents are used, each gene in the child chromosome has a 0.5 probability of inheriting that gene from the first parent. Otherwise, the gene from the second parent is inherited. In certain cases, usually when generating a single child, this probability can be altered to provide a bias towards the parent with the higher quality. When generating two offspring, a second child can be generated by inverting the genes of the first child. Uniform crossover is commonly implemented using a uniformly distributed random vector or a bit mask. In the example shown below, both the random vector and the bit mask provide the same operation. For the vector and the bit mask, a value of 0.5 or higher (or a bit of 1) causes the first child to inherit that gene from the second parent. Otherwise, it inherits from the first parent. The second child is generated as the inverse of the first.

Random Vector:	0.51	0.16	0.34	0.88	0.85	0.42	0.56	0.76
Bit Mask:	1	0	0	1	1	0	1	1



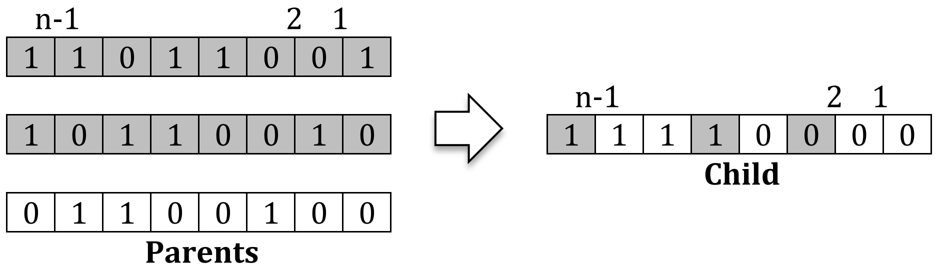
Shuffle Crossover (SC)

The shuffle crossover is similar in nature to the uniform crossover, but attempts to remove any positional bias that may occur when using a uniform crossover method. The genes in the parents are randomly shuffled in tandem before a single point crossover position is selected. The children are generated and the genes are then unshuffled.



Three-Parent Crossover (TPC)

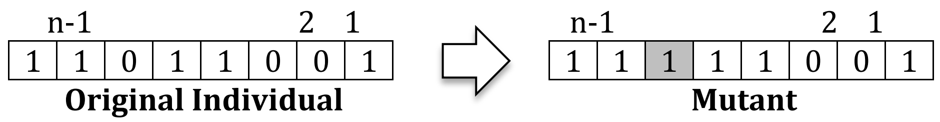
Three-parent crossover is a method that attempts to further increase the exploration mechanism in GAs. With this technique, three parents are randomly chosen and designated as parents 1, 2, and 3. With each gene in the chromosome, if parents 1 and 2 share the same gene, the child receives that gene. Otherwise, the child receives the gene of parent 3. A second child can be generated similarly by modifying the parent ordering in the operation.



Bit-Flip Mutation

Bit-flip mutation is a variation that only requires a single individual. Generally, this operation is used to alter (or mutate) an existing individual rather than to create a new one. Many GAs use mutation as a last step on children that are generated via a crossover operation. The mutation step is used in an attempt to keep the algorithm from becoming trapped in a local minimum. Bit-flip mutation works by flipping the bit of each gene with a given probability. This probability value is generally very low (~0.01) in most cases. It is quite possible that after applying the bit-flip operation that the individual is unchanged. The example below shows the bit-flip mutation using a random vector.

Random Vector:								
	0.68	0.44	0.01	0.96	0.53	0.75	0.15	0.09



Continuous Code Variations

Continuous code representation is an attempt to overcome some of the drawbacks that are inherent to binary code representations. As was discussed previously, when using binary code representations there is ultimately a loss of precision when attempting to represent real-value numbers. This can be overcome in various ways, but when the solution requires the use of several variables needing a certain level of precision the search ability of the GA is severely reduced. Using this encoding scheme, each gene in the chromosome is represented by a floating-point number.

When using a continuous code representation, the variation methods mentioned for binary code can still be used, but do not do a good job of exploring the search space. This is due to the fact that each gene now represents a variable and performing the binary variation methods will not introduce any new variables into the chromosome. For this reason, several methods of variation for continuous code representations have been devised.

Arithmetic Crossover

The arithmetic crossover is essentially a linear interpolation of the individual genes of two parents. As a result, the corresponding genes within the child will lie somewhere within the line connecting the two genes from the parents. The arithmetic crossover can be performed in one of two ways: the whole arithmetic crossover and the local arithmetic crossover. The former uses a single uniformly distributed random number for all genes and the latter uses separate uniformly distributed random numbers for each gene.

Whole Arithmetic Crossover (WAC)

$$\boldsymbol{x}^1 = (x_1^1, x_2^1, \dots, x_n^1) \qquad \Rightarrow \qquad \boldsymbol{y}^1 = \alpha\boldsymbol{x}^1 + (1-\alpha)\boldsymbol{x}^2$$

$$\boldsymbol{x}^2 = (x_1^2, x_2^2, \dots, x_n^2) \qquad\qquad \boldsymbol{y}^2 = \beta\boldsymbol{x}^1 + (1-\beta)\boldsymbol{x}^2$$

Parents **Children**

$$\alpha \sim U(0,1)$$

$$\beta = 1 - \alpha$$

Local Arithmetic Crossover (LAC)

$$\boldsymbol{x}^1 = (x_1^1, x_2^1, \dots, x_n^1) \qquad \Rightarrow \qquad \boldsymbol{y^1} = \boldsymbol{\alpha} \cdot \boldsymbol{x^1} + (\boldsymbol{1} - \boldsymbol{\alpha}) \cdot \boldsymbol{x}^2$$

$$\boldsymbol{x}^2 = (x_1^2, x_2^2, \dots, x_n^2) \qquad \qquad \boldsymbol{y^2} = \boldsymbol{\beta} \cdot \boldsymbol{x^1} + (\boldsymbol{1} - \boldsymbol{\beta}) \cdot \boldsymbol{x}^2$$

Parents **Children**

$$\boldsymbol{\alpha} = (\alpha_1, \alpha_2, \dots, \alpha_n), \qquad \alpha_i \sim U(0,1)$$

$$\boldsymbol{\beta} = (\beta_1, \beta_2, \dots, \beta_n), \qquad \beta_i = 1 - \alpha_i$$

While this does increasing the exploration mechanism as far as introducing new variables into the GA, the range of the population will not be increased beyond the extreme individuals represented.

Linear Crossover (LC)

Linear crossover is a method that was first introduced by Wright as a way of increasing the range of the standard arithmetic crossover method. Using this technique, three children are generated, but only the strongest two (those with the highest quality) are kept. In the event that the value of either y^2_i or y^3_i is generated outside the bounds of search space, the value of y^1_i is used instead. In the example shown below, a factor of 0.5 is used, but this factor can be modified to a more appropriate value when considering the search space.

$$\boldsymbol{y}^1 = (y_1^1, y_2^1, \dots, y_n^1)$$

$$\boldsymbol{x}^1 = (x_1^1, x_2^1, \dots, x_n^1) \qquad \Rightarrow \qquad \boldsymbol{y}^2 = (y_1^2, y_2^2, \dots, y_n^2)$$

$$\boldsymbol{x}^2 = (x_1^2, x_2^2, \dots, x_n^2)$$

Parents

$$\boldsymbol{y}^3 = (y_1^3, y_2^3, \dots, y_n^3)$$

Children

$$y_i^1 = 0.5x_i^1 + 0.5x_i^2$$

$$y_i^2 = 1.5x_i^1 - 0.5x_i^2$$

$$y_i^3 = -0.5x_i^1 + 1.5x_i^2$$

Heuristic Crossover (HC)

The heuristic crossover is a variation used to expand the range of the standard arithmetic crossover method. The child genes generated using this technique will lie somewhere along the line connecting the corresponding parent genes or slightly outside of it and will have preference towards the parent with the higher fitness. In the event that a gene is generated outside of the bounds of the search space, a new β will be generated and the crossover will be attempted again.

$$\boldsymbol{x}^1 = (x_1^1, x_2^1, \dots, x_n^1)$$

$$\boldsymbol{x}^2 = (x_1^2, x_2^2, \dots, x_n^2)$$

Parents

$$\Rightarrow$$

$$for\ fitness(\boldsymbol{x}^1) > fitness(\boldsymbol{x}^2)$$
$$y_i = x_i^2 + \beta(x_i^1 - x_i^2)$$

$$for\ fitness(\boldsymbol{x}^2) \geq fitness(\boldsymbol{x}^1)$$
$$y_i = x_i^1 + \beta(x_i^2 - x_i^1)$$

Children

$$\beta \sim U(0.8, 1.2)$$

Blend Crossover (BC)

Blend crossover is another method that seeks to expand the range of the arithmetic crossover by means of extrapolation. Whereas the arithmetic crossover will only create new genes in children that lay along the line connecting the corresponding genes of the parents, the blend crossover increases the diversity of the offspring while still preserving the statistics of the parents. The blend crossover operation is performed at the gene level. α is a variable that controls the range of the expansion of the crossover. It has been reported that a value of 0.5 is usually a good choice for this variable.

$$\boldsymbol{x}^1 = (x_1^1, x_2^1, \dots, x_n^1)$$

$$\boldsymbol{x}^2 = (x_1^2, x_2^2, \dots, x_n^2)$$

Parents

$$\Rightarrow$$

$$\boldsymbol{y} = (y_{\boldsymbol{1}}, y_{\boldsymbol{2}}, \dots, y_{\boldsymbol{n}})$$

Child

$$for\ x_i^1 < x_i^2, \quad y_i \sim U\left(\left(x_i^1 - \alpha(x_i^2 - x_i^1)\right), \left(x_i^2 + \alpha(x_i^2 - x_i^1)\right)\right)$$

Analysis

For the analysis used in this report, a simple genetic algorithm was implemented in an effort to compare the performance of the several types of variation mentioned above. The source code can be viewed online at: http://github.com/bjcrawford/SGAVariationAnalysis/.

GA Implementation and Setup

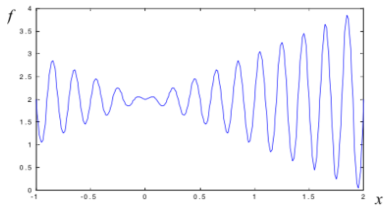
The implementation of the GA is based largely on the SGA described in chapter 2.2 of the text "Introduction to Evolutionary Algorithms" by Xinjie Yu. Upon initialization, a population of 20 individuals is uniformly generated across the search space described by the test function. The GA has a limit of 20 generations. With each generation, a mating pool is populated using a roulette wheel selection. This method uses the relative fitness values to determine the size of each individual's "slice" of the roulette wheel. Selection is done with replacement, meaning an individual can be selected from the population into the mating pool multiple times.

Once the mating pool has been populated, the reproduce method mates adjacent individuals within the mating pool list (i.e., 0 and 1, 2 and 3, etc…). After performing the crossover operation, each child individual undergoes the mutation operation. The probability for crossover has been set at 0.8 and the probability of mutation, at the gene level for the binary representations and at the variable level for the continuous representations, has been set at 0.01. Each run (i.e., a test using a particular representation, test function, and variation method), consists of 100 independent trials. For each run the best and worst individuals are recorded as well as the mean objective values produced.

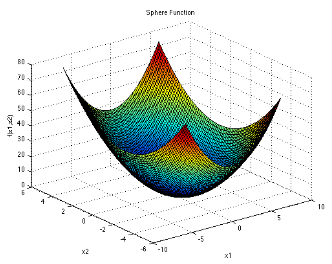
Test Functions

The following test functions were used for this analysis. They represent some of the more common benchmark functions used in comparison of GA performance.

- Function 1: $\max f(x) = x\sin(10\pi x) + 2.0, \ s.t. \ -1 < x < 2$
  - A simple sinusoidal function with many local minima and maxima. This is a maximum problem with a global maxima at f(1.85) = 3.85.



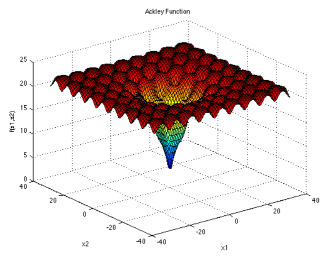
- Function 2: $\min f(x) = \sum_{i=1}^{n} x_i^2, \ s.t. \ -5 < x < 5 \ and \ n = 5$
  - The sphere model function (De Jong's Function 1) is convex, continuous, and unimodal. This is a minimum problem with a global minima at f(0) = 0.0.



- Function 3: $\min f(x) = -20e^{-0.2\sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2}} - e^{\frac{1}{n}\sum_{i=1}^{n}\cos(2\pi x_i)} + 20 + e,$

$$s.t. -20 < x < 30 \ and \ n = 2$$

  - The Ackley function offers many local minima on the outer edges with a deep global minima located at the center. This is a minimum problem with a global minima at f(0) = 0.0.



Experimental Results

Binary Representation Results

	SPC	DPC	SPCRS	DPCRS	RC	UC	SC	SCRS	TPC
Func1									
Best:	3.849834	3.800835	3.832992	3.849834	3.818516	3.639529	3.848416	3.299756	3.475067
Worst:	0.276081	0.052819	0.065000	0.056280	0.050993	0.053120	0.049901	0.252710	1.214753
Mean:	2.630076	2.558270	2.590681	2.604430	2.041554	2.521279	2.631119	2.551577	2.612739
Func2									
Best:	8.223149	12.84400	8.534937	12.28211	6.884248	4.381790	1.854469	5.913441	14.04026
Worst:	88.75400	75.05564	86.85380	89.32472	88.12869	77.13687	88.25831	87.51461	72.35113
Mean:	28.92989	30.12281	28.37166	28.66065	36.31976	31.93706	31.11571	30.57386	32.63222
Func3									
Best:	11.55117	5.635655	10.83425	11.84558	6.634444	6.032643	11.09433	5.697629	14.95531
Worst:	21.93011	21.87508	21.73663	22.01513	22.21997	22.10998	21.83411	22.09076	21.99965
Mean:	14.01581	13.93320	14.17935	13.64442	18.69616	13.51210	13.59522	13.33998	15.35078

Gray Representation Results

	SPC	DPC	SPCRS	DPCRS	RC	UC	SC	SCRS	TPC
Func1									
Best:	3.622670	3.845704	3.649894	3.848208	3.850272	3.838306	3.742275	3.248937	2.773600
Worst:	0.657038	0.107185	0.065000	0.503408	0.265100	0.086363	0.283522	0.059914	0.457942
Mean:	2.644264	2.553607	2.626598	2.525538	2.046237	2.428903	2.477820	2.407746	2.547334
Func2									
Best:	3.866322	1.868844	5.965534	4.380225	9.347806	9.406166	8.822122	1.787980	9.052771
Worst:	85.80446	72.57595	77.01564	78.02373	85.06991	81.77403	85.54622	73.67473	80.69628
Mean:	28.98835	28.05611	28.34636	27.49247	37.75561	28.41386	28.92575	28.69795	32.41914
Func3									
Best:	8.830922	8.440495	10.182341	8.297264	9.111006	6.994958	7.713238	2.783320	8.225912
Worst:	22.16091	22.00986	21.977657	21.97826	22.23471	21.91077	21.82708	21.98960	22.14135
Mean:	14.71300	14.14835	14.86216	14.67138	19.06880	14.11378	13.94120	13.75339	16.46617

Continuous Representation Results

	WAC	LAC	LC	HC	BC
Func1					
Best:	3.443246	3.650207	3.625999	3.650306	3.645560
Worst:	0.309041	0.940608	0.320326	0.344280	0.255581
Mean:	2.168394	2.204762	2.142460	3.009605	2.338584
Func2					
Best:	3.004641	0.757786	1.570550	7.194151	6.377517
Worst:	68.32616	77.62269	79.40512	70.24675	87.21973
Mean:	11.06570	11.28763	9.025025	12.96918	49.85977
Func3					
Best:	1.086268	0.376553	3.915845	1.365332	4.488860
Worst:	22.12246	22.17808	21.68338	22.18832	21.79650
Mean:	9.406477	9.811599	10.54843	9.072875	15.67351

Conclusions

Within this report, several methods for performing variation in use with genetic algorithms have been explained and analyzed. For discrete representations, both binary and gray code encoding schemes were analyzed; for continuous representation an encoding scheme using floating point numbers was analyzed. Several benchmark test functions common to genetic algorithm testing were used in the comparative analysis. The results of the best performing combination for each test function are listed below:

- Function 1:
    - Binary representation
    - Tie, single point crossover and dual point crossover
    - 0.000166 from optimal solution
- Function 2:
    - Continuous representation
    - Local arithmetic crossover
    - 0.757786 from optimal solution
- Function 3:
    - Continuous representation
    - Local arithmetic crossover
    - 0.376533 from optimal solution

For the discrete representations, the gray code showed slightly better performance than the binary code, however this may be a result of the stochastic nature of the experiments. The binary shuffle crossover has the best performance for function 2 in the discrete representations. For function 3, the gray shuffle crossover with reduced surrogate showed the best results.

In general, the continuous representations performed much better on the test functions with larger search spaces (functions 2 and 3), but alternatively they did not perform as well as the discrete representations on the simplest test function (function 1). This would appear to be due to the fact that while a smaller search space reduces the possible solutions in a discrete representation, it does not have the same effect for continuous representations.

It is also worth noting that this implementation used for these experiments made use of the java.util.Random class for random number generation. The general consensus regarding the performance of this class is that it should mostly be used for informal random number generation. Unfortunately, due to time constraints, I was unable to locate an implementation that provided both fast performance and quality random number generation.

The results from this experiment have shown on some level that there is no clear winner in variation with regards to performance. Each crossover is generally designed to exploit a certain type of problem or solution landscape. With these considerations in mind, it is clear that in designing and implementing a genetic algorithm, great care must be taken to understand the problem that is being explored.

References

Bierwirth, C., Mattfeld, D. C., & Kopfer, H. (1996). On permutation representations for scheduling problems. *Parallel Problem Solving from Nature—PPSN IV* (pp. 310-318). Springer Berlin Heidelberg.

Deb, K., & Beyer, H. G. (1999). Self-Adaptive Genetic Algorithms with Simulated Binary Crossover.

Eldos, T. (2013, June). Mutative Genetic Algorithms. *Journal of Computation & Modelling.* 3(2), 111-124.

Galan, S., Mengshoel, O., & Pinter, R. (2013). A Novel Mating Approach for Genetic Algorithms. *Evolutionary Computation,* 21(2), 197-229.

Haupt. R., & Haupt, S. (2004). *Practical Genetic Algorithms.* New York: Wiley.

Mitchell, M. (1996). *An Introduction to Genetic Algorithms*. Cambridge, Mass.: MIT Press.

Reeves, C. (2003). *Handbook of Metaheuristics.* US: Springer.

Sivanandam, S., & Deepa, S. (2007). *Introduction to Genetic Algorithms.* Berlin: Springer.

Yu, X., & Gen, M. (2010). *Introduction to Evolutionary Algorithms*. London: Springer.