College of Saint Benedict and Saint John's University
DigitalCommons@CSB/SJU

Honors Theses, 1963-2015 | Honors Program

1997

Genetic Algorithms: A Visual Search

Paul W. Jones
College of Saint Benedict/Saint John's University

Follow this and additional works at: https://digitalcommons.csbsju.edu/honors_theses

 Part of the Biology Commons, Computer Sciences Commons, and the Mathematics Commons

Recommended Citation

Jones, Paul W., "Genetic Algorithms: A Visual Search" (1997). *Honors Theses, 1963-2015*. 584.
https://digitalcommons.csbsju.edu/honors_theses/584

Available by permission of the author. Reproduction or retransmission of this material in any form is prohibited without expressed written permission of the author.

Genetic Algorithms: A Visual Search

Senior Research

by
Paul Jones

Computer Science Department

Saint John's University

Committee:

Mike Gass (chair)
Andy Holey
Noreen Herzfeld

Abstract

Genetic algorithms apply the biological principles of selection, mutation, and crossover to a population set containing individuals representing target solutions to a given problem. Using these principles genetic algorithms attempt to create a migration of the individuals in subsequent generations toward the optimal solution.

This project is an attempt to visually represent the progress of a genetic algorithm. The coordinate fitness program attempts to find the maximum or minimum value of a given function. It visually represents the progress of the algorithm by providing a plot of each individual in each generation in time. It is then possible to view the migration of points toward a know maximum or minimum value. Visual representation is also achieved by a plot of the highest and lowest fitness per generation, as well as average fitness per generation.

The parameters of crossover rate and mutation rate can be altered. This allows experimentation in finding a good combination of these rates for a particular function, and viewing the results. Many involved in the field of genetic algorithms believe that this is an area of the subject that requires further research.

Introduction

This paper will document my senior research in the area of genetic algorithms. The goal of the project is to visually represent the progress of a genetic algorithm on a given function, allowing certain parameters such as mutation and crossover rate to be changed. Advisor for the project is Professor Michael Gass.

I selected the topic of genetic algorithms in May of 1996 after attending a presentation by Saint John's student Mike Criswell on genetic algorithms applied to what is known as the traveling salesman problem. Before this time I had some, but very little knowledge of genetic algorithms.

1.0 A Brief Timeline and The Topic of this Genetic Algorithm

At the beginning of the semester I had initially divided the project into four parts: research (summer-early fall), design (fall-winter), implementation (winter-early spring), and testing (spring). I began research during the summer and continued into the early part of the school year. In October I began meeting with Professor Gass approximately once a cycle. During the first few meetings we determined that the initial problem I had suggested, an academic advisor program based on genetic algorithms, was not feasible in a one year individual project. We discussed how a scaled down version of that problem might be possible, but not as interesting as some possible alternatives. One of the alternatives we discussed is this project. In this paper I will refer to the genetic algorithm based program that I have written as the coordinate fitness program.

After deciding on a topic for the project, the design phase continued into late November, and overlapped with the implementation stage, which began in early November. This was by choice, I decided it would be easier to implement such things as

the initial population, and add selection, mutation, and crossover later since I had an abstract view of how they would work.

The implementation also overlapped with testing, as I tested functions as they were completed. At this point implementation of the project is completed, but I am still in the process of attempting to implement another separate program. This would be to run the coordinate fitness program on itself to determine the best crossover and mutation rates. Due to the nature of the program, this should not be extremely difficult. It will be discussed further in section 5.0.

I got the idea for an academic advisor program mentioned above in my research on a separate project involving the object-oriented paradigm. One of the articles I was researching documented an advisor program using a knowledge base. I had envisioned a similar knowledge base of the CSB/SJU curriculum, with a genetic algorithm available to search the thousands of class combinations for students. Genetic algorithms work well for this type of problem, but a number of difficulties would have had to have been resolved before implementing it. The course catalog, or a subset of it would have to be entered into the knowledge base. Each course or combination of courses would have to be represented as a string. Creating a scheme for this representation would have been difficult.

Rather, this program uses genetic algorithms to maximize or minimize a given function. That in itself does not appear to be extremely challenging but it contains significant experimental value in the area of genetic algorithms. It also lends itself well to allowing the user to view the progress of genetic algorithms. The solution of the problem is known, so we can adjust the parameters of mutation and crossover to

determine what effect they have on how fast a solution is found, if at all, for a given function. Many functions do not contain a maximum or minimum which is easily calculated. Some have misleading peaks and valleys that can fool a conventional algorithm.

2.0 The Basics of Genetic Algorithms

Genetic algorithms use the biological principles of selection, mutation, and crossover applied to a population of individuals. The individuals are codified to represent a particular solution to a problem. "The term chromosome typically refers to a candidate solution to a problem, often encoded as a bit string...the 'genes' are either single bits or short blocks of adjacent bits that encode a particular element of the candidate solution" (Mitchell 6).

Each individual has a fitness value determined by a fitness function associated with a chromosome. The chromosome is comprised of a fixed length string of characters typically in binary or hexadecimal notation. The fitness function examines each chromosome and assigns to it a fitness value based on the arrangement of the characters.

An initial population is generated randomly, and its size is pre-defined. Each individual in the initial population represents a possible solution to the given problem. Each subsequent generation is determined by calculating the fitness value of each individual, selecting a certain number of 'fit' individuals to move on to the next generation and form a percentage of that population, mutating a certain percentage of individuals in the population, and crossing over a certain percentage of individuals in the population.

2.1 Methods of Selection, Mutation, and Crossover

Variations on the selection process are also used. One such process is fitness proportionate selection. Here an individual's expected value, the expected number of times it will be selected to reproduce, is its fitness divided by the average fitness of the population (Mitchell, 166). An individual's expected value divided by the sum of expected values is the probability that individual should be selected into the next generation on any given selection pass. For example:

Individual	Fitness	Expected Val	/4.0
A	0.25	0.5	.125
B	1.05	2.1	.525
C	0.05	0.1	.025
D	0.65	1.3	.325
Total	2.0	4.0	1.0

The total fitness for the population is 2.0, the average fitness is 0.5. The expected value, or number of times an individual is expected to be selected for reproduction is proportionate to its fitness. This method can be envisioned as a roulette wheel with slices for each individual corresponding to the size of their expected value divided by the sum of expected values. The wheel is spun four times in this case, and the individual selected moves on. On a given run we would expect individual D to be selected .325 percent of the time, or 1.3 times in four spins of the roulette wheel.

The Rank selection scheme is to allow a percentage of the current generation to survive in the next generation. This method assigns a fitness value to each individual, sorts the population by fitness, and selects the desired percentage of the population size to move on.

Tournament selection uses a tournament pool (a subset of the entire population) of a defined size to select from. This method (in a single iteration) selects some number k of individuals and selects the best one from this set of k elements into the next generation. This process is repeated pop-size number of times.

Mutation works by randomly selecting a bit (or hex-digit) and flipping it (or changing it to a random hex-digit). "The intuition behind the mutation operator is the introduction of some extra variability into the population" (Michalewicz, 16). Genetic algorithms work well for problems in which the search space is large. Where a conventional algorithm may get stuck on a local maximum, for example, genetic algorithms are able to randomly jump off of that point. Mutation is one of the means that allow for this.

Crossover works by selecting a random point within two individuals, and swapping the contents of their remaining strings. The idea is that crossing over two 'fit' individuals will possibly result in an ever 'fitter' individual.

There are a number of variations of the above processes and where they are applied. For example, two bits could be mutated instead of one. A population can be selected in its entirety for the next generation, and then mutation and crossover applied, or mutation and crossover can occur in the preceding generation and then copied over to the next generation to comprise a percentage of it.

Melanie Mitchell discussed the relation, and importance of mutation and crossover in genetic algorithms:

> A common view in the GA (genetic algorithm) community...is that crossover is the major instrument of variation and innovation in GAs, with mutation insuring the population against permanent fixation at any particular locus...the appreciation of the

> role of mutation is growing as the GA community attempts to understand how GAs solve complex problems...it is not a choice between mutation and crossover or mutation, but rather the balance among crossover, mutation, and selection that is all important. The correct balance also depends on details of the fitness function and encoding. Furthermore, crossover and mutation vary in relative usefulness in the course of a run. Precisely how all this happens still needs to be elucidated. In my opinion, the most promising prospect for producing the right balances over the course of a run is to find ways for the GA to adapt its own mutation and crossover rates during a search (173-174).

A specific crossover or mutation rate does not guarantee success or failure, and they are closely tied together in determining the results of a given run as shall be demonstrated in section 4.0. Section 5.0 will consider the possibility of using the coordinate fitness program to address the second thing Ms. Mitchell alluded to: the relation of a particular mutation rate to a crossover rate in a particular run. Although that section will not discuss the dynamic nature of those rates over the course of a run, it will examine how the coordinate fitness program can be used to suggest good mutation and crossover rates together.

3.0 The Code and Design

The coordinate fitness program uses tournament selection, with tournament size of two as its means of selection. Individuals are codified as bit strings. The individuals are further subdivided into two halves: an x value and a y value representing coordinates. Each half is a binary representation of a number in the range zero to one. The first place in each respective half of the chromosome represents the value 0.5, the second 0.25, etc. The fitness function is then simply a function of two values that we are attempting to maximize or minimize, and the fitness value for an individual is the evaluation of its x

and y coordinates by that function. The code I have written is in C++. The major data structures are a Population and an Individual class.

3.1 Population Class

The code for the Population class can be found in section 7.4.2. The Population header file contains a static definition for the population size named POP_SIZE. Data members include float values for the total and average fitness of the population and for the highest and lowest individual in a generation, as well as an integer value representing the generation number. The mutation and crossover rates are kept as private data members of type double. Two pointers to Individual objects are declared as well, population and population2. The constructor initializes these as arrays of length POP_SIZE.

The job of the population class is to oversee the population in each generation. The containers for the population in each generation are population and population2, representing odd and even generations respectively. Each new population is created as follows: selection occurs in the current generation, the winner of each tournament being copied into the opposite population container (from population into population2 if the current generation is odd, and vice versa). Mutation works on the newly created population. The generation number has not yet changed, so mutation must work on what will be the next generation (population2 in odd generations, and population in even generations). Crossover occurs next, and will be applied to the same population that mutation was.

The order that the above mentioned functions are called is dependent on the current generation. The function IncGen increments the data member generation. The

test program for the coordinate fitness program creates each population by calling the functions select, mutate, crossover, and IncGen in that order.

A number of functions that enable data to be gathered are called before IncGen as well. The function SetBestWorst sets the private data members highest and lowest fitness and appends to a file called 'bestworst' for each generation (note: the values in the file 'bestworst' are relative to the fitness function and whether an attempt is being made to maximize or minimize it). The function PrintPairs writes the x and y value of each individual to a file called 'genXXX' where XXX is the current generation. Another function, PrintAvgFitness appends the average fitness to the file 'avgfit' for each generation.

3.2 Individual Class

The Individual class contains the attributes of one individual in a population. The pertinent data member is a character array called chromosome. It also contains member functions to mutate the chromosome, return an element of the chromosome array (GetDNA), return the entire chromosome (GetChrom), return the fitness value for the chromosome array, set a value for the chromosome array at some position, set the x and y values contained in the chromosome, and return the x and y values. Both the x and y values are kept as private floats. The class also contains a function to assign a fitness value for the chromosome. In the case of the coordinate fitness program, the fitness function is the function we are trying to maximize or minimize. The data members are integer values for the length of the chromosome, and float values for the normalized fitness, adjusted fitness, x value, and y value. The x and y values of a chromosome are

converted to a floating point number between zero and one to allow them to be graphed using the program gnuplot.

3.3 Selection, Mutation, and Crossover

Each of these functions rely heavily on randomly generated numbers. I did not do a great deal of research on random number generators. A more complete project would require information on the effectiveness of the random number generator used. This project uses Rand functions from the rando class library.

The selection process used in this program is tournament selection with a tournament size of two. Two individuals are selected at random from the population. If we are attempting to maximize, the one with a higher fitness value (or lower fitness if minimizing) is selected to move to the next generation. Both individuals are then put back into the population with the possibility of being selected again. This procedure is repeated POP_SIZE number of times, so the next generation is completely filled.

Mutation occurs after selection. Once a new generation has been completed by the selection process, the mutation process examines each individual for the possibility of mutation. At each individual in the population, a random number is generated between zero and one, to three decimal places. If that number is less than the mutation rate, the individual is to be mutated and another random number is generated between zero and the CHROM_LENGTH. The corresponding bit of the individual is flipped.

Crossover uses the same technique of generating a random number at each individual. If that random number is less than the crossover rate, it is crossed with another randomly chosen member of the population. The crossover point is randomly selected and may occur at any point of the chromosome. For example if a crossover point

of seven were selected, with a chromosome length of ten, the two individuals crossing over would swap bits eight and nine. After a chromosome has been crossed, its partner is returned to the population. If that partner has yet to be examined for crossover, it may be crossed again.

3.4 Moving From Generation to Generation

From an implementation standpoint, one of the most important aspects of designing a program such as this is to determine what the containing devices should be for the individuals, and the population. It is also important to create an efficient but effective way of moving from generation to generation.

The coordinate fitness program uses two population arrays, called population and population2. Each contain POP_SIZE Individual objects. The first array, population, contains the entire population for odd numbered generations, while the second contains even numbered generations.

In creating each generation the following procedure is repeated. The selection process uses tournament selection to choose from the current generation. If the current generation mod two is one, selection occurs in population; if it is zero, selection occurs in population2. Each winner (a particular Individual object) is copied into the other population array. Next mutation and crossover are applied to the population that has just been created (current generation +1). The values for each individual must be re-calculated to account for any changes. The re-calculation actually occurs after mutation, and again after crossover. This must be done so that crossover does not write a bit as a one, when it actually should be a zero because it was mutated.

4.0 Experimental Results

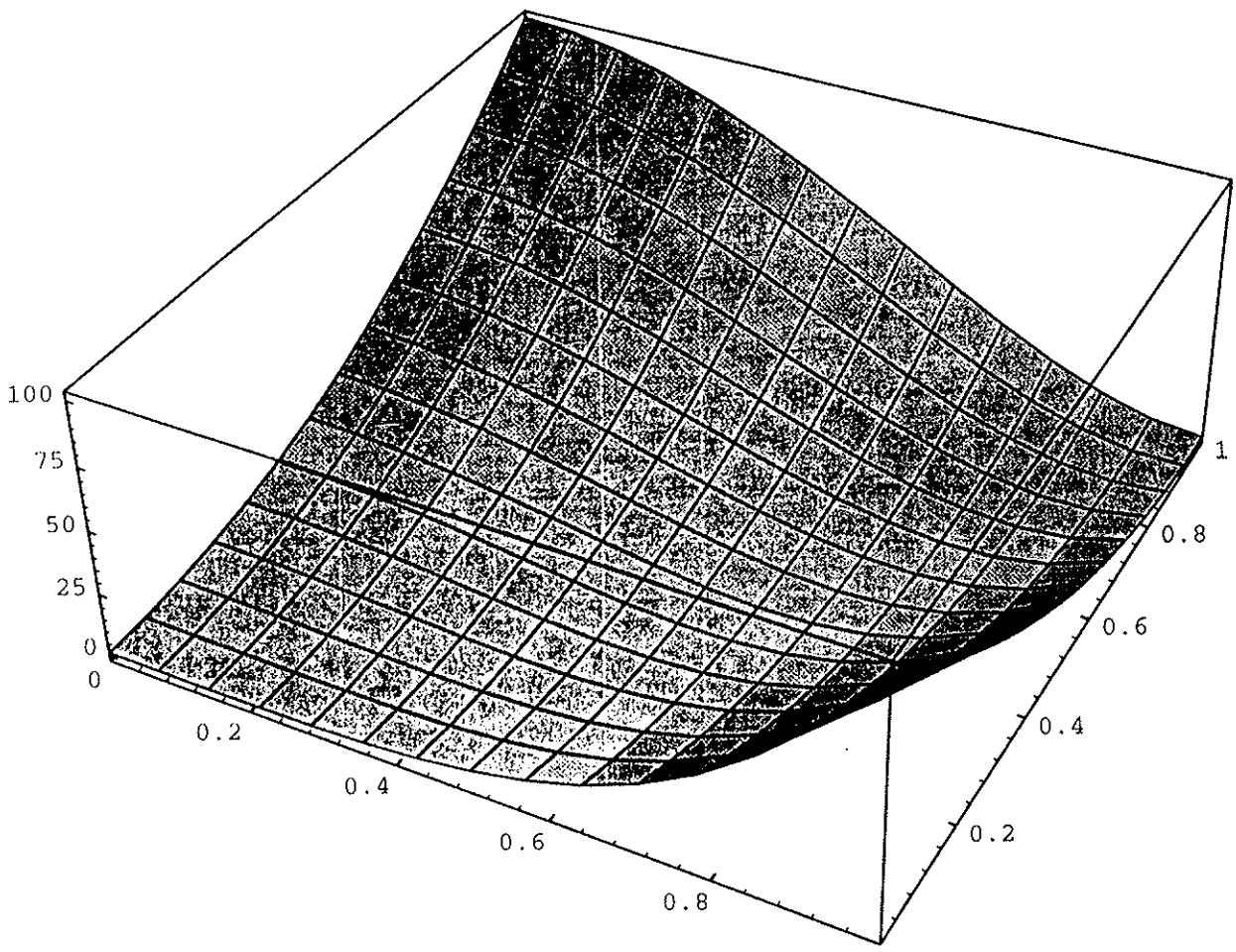
The coordinate fitness program is an attempt to visually represent the progress of a genetic algorithm that is searching for 'fit' points. It is also an attempt to allow experimentation in choosing mutation and crossover rates, and to visually represent the program's progress based on selected rates. The experimental results can be broken into two categories: the visual representation of a genetic algorithm's progress, and the effects of different combinations of mutation and crossover rates. Each of the following sections will contain experimental results in these two areas. The results are obtained from running the coordinate fitness program on minimizing the function 100(y-x^2)^2 +(1-x)^2 and maximizing the function sin(Pi*x)*(1 - fabs(sin(6*Pi*x)))*sin(Pi*y)*(1 - fabs(sin(6*Pi*y))). For each result, the parameters of the run from which they were obtained are listed.

Each test run includes a graph of the average fitness per generation and highest/lowest fitness per generation. A capsule look at each population is provided in the form of a plot of each point in certain generations. For the sake of saving space, every fifth generation is shown up to generation 25. Each graph of a generation, such as the ones which appear next, can be viewed in the order they were generated by the coordinate fitness program. When running the coordinate fitness program (using the gnuplot program), a graph of each generation's population appears every two seconds, allowing the user to view the migration of points.

4.1 Function f(x,y)= 100(y-x^2)^2 +(1-x)^2

The coordinate fitness program attempts to find the minimum which occurs at points (0.0, 0.0) and (1.0, 1.0). Both pairs evaluate to zero, however the chromosomes are incapable of representing an x or y value of 1.0.

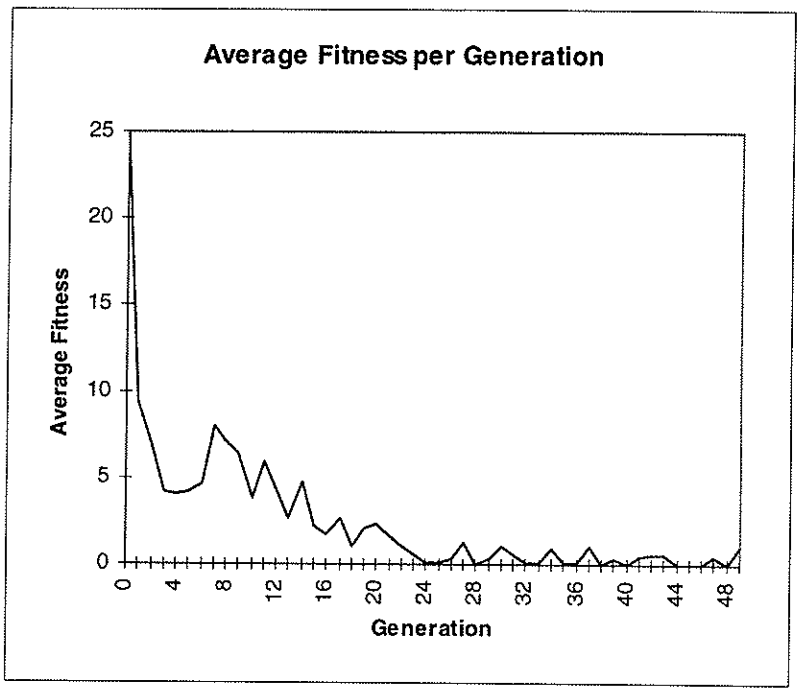
4.1.1 Graph of 100(y-x^2)^2 +(1-x)^2

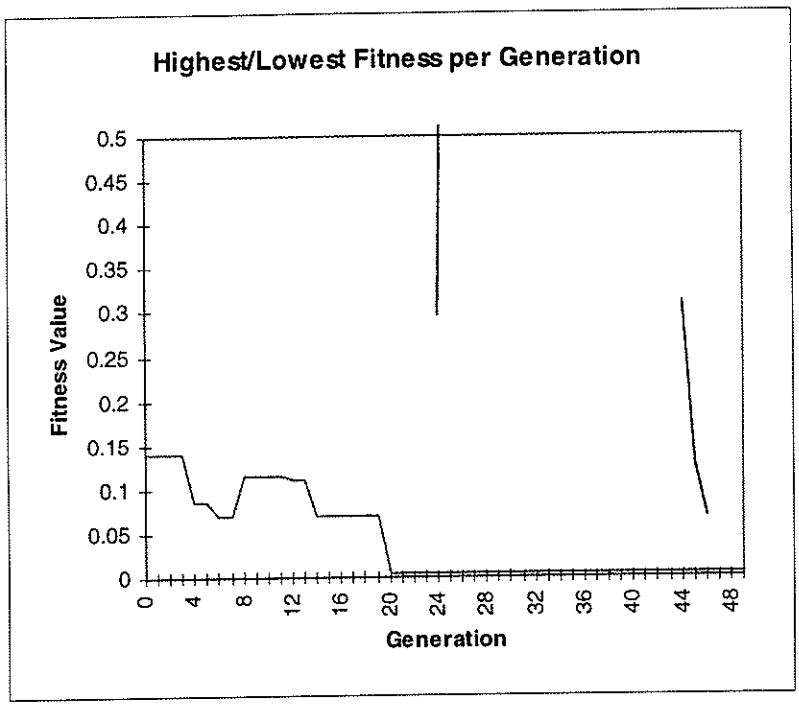


4.1.2 Test Run 1

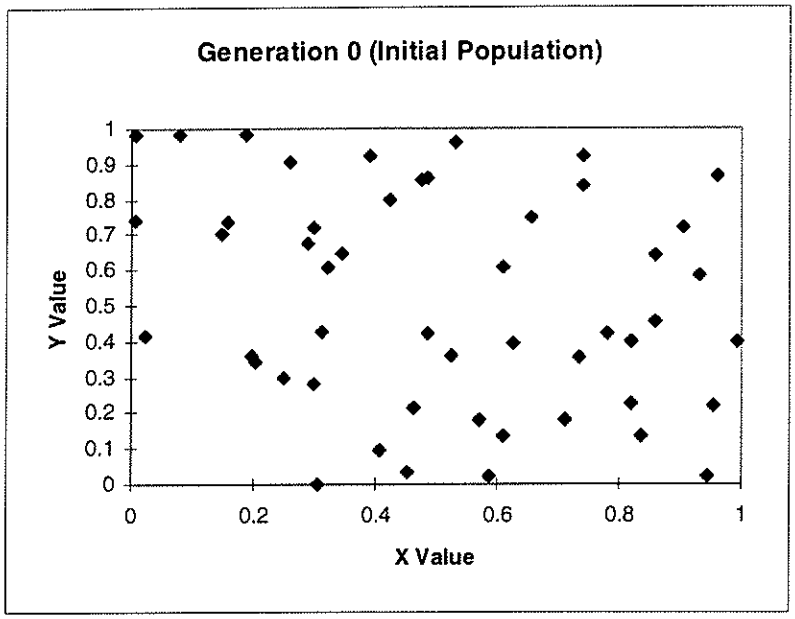
The following graphs are taken from a test run with the parameters: chromosome length = 14, population size = 50, mutation rate = 0.05, crossover rate = 0.2, function to minimize is 100(y-x^2)^2 +(1-x)^2.

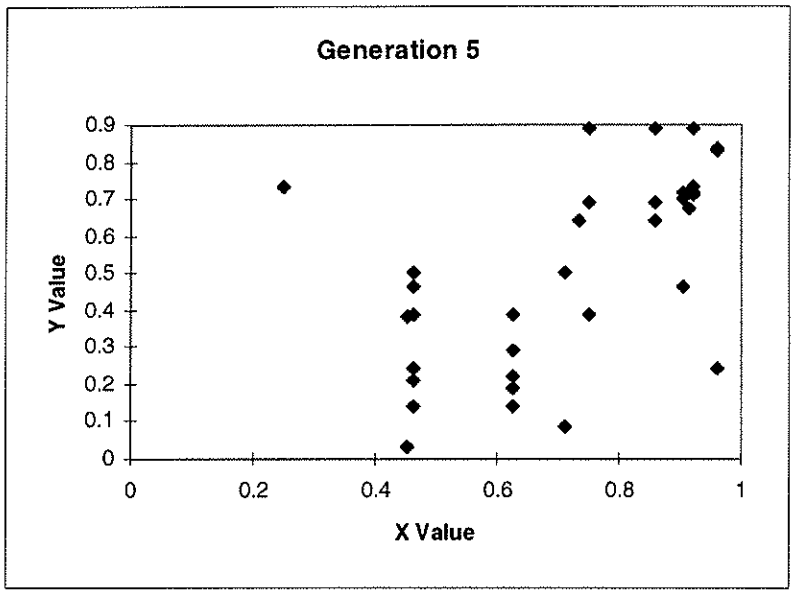
In this run the optimal solution was not found. The graphs appear to indicate that an optimal fitness value of zero was found, however the minimal solution found was 0.00543213 at the point (0.9375, 0.882812).

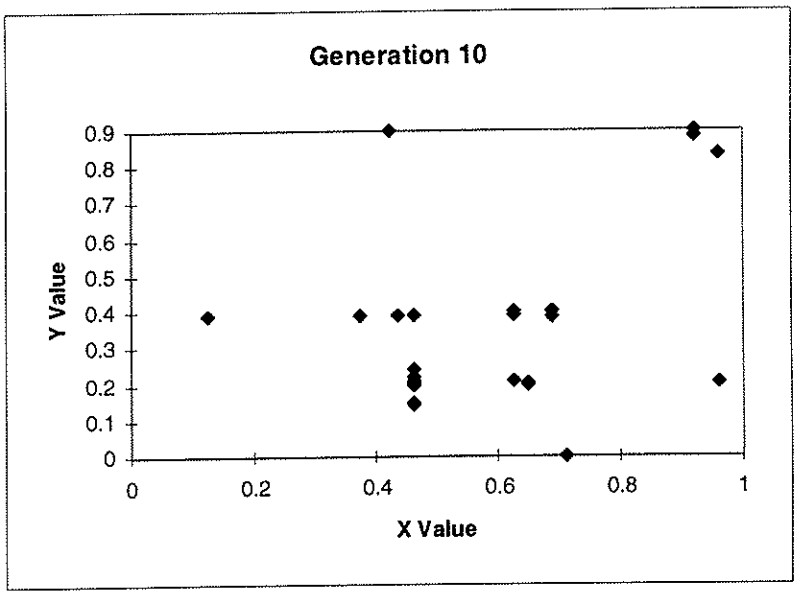


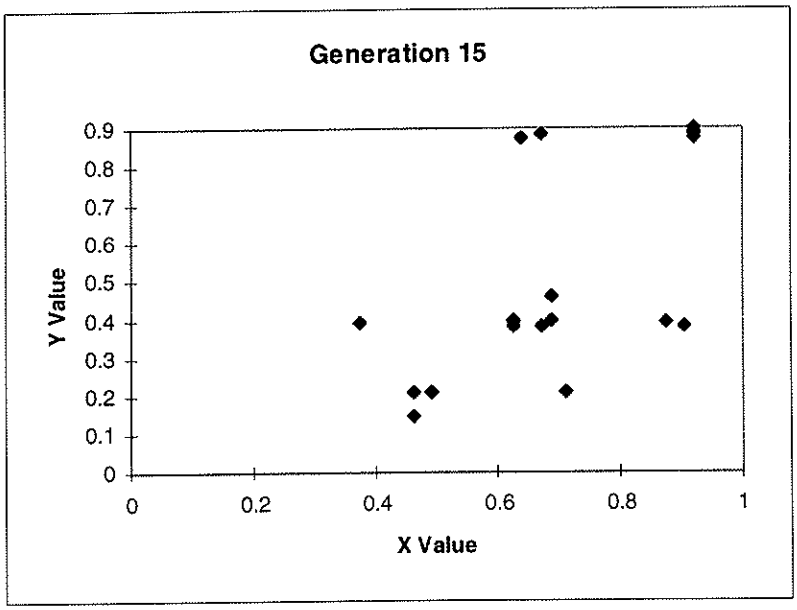


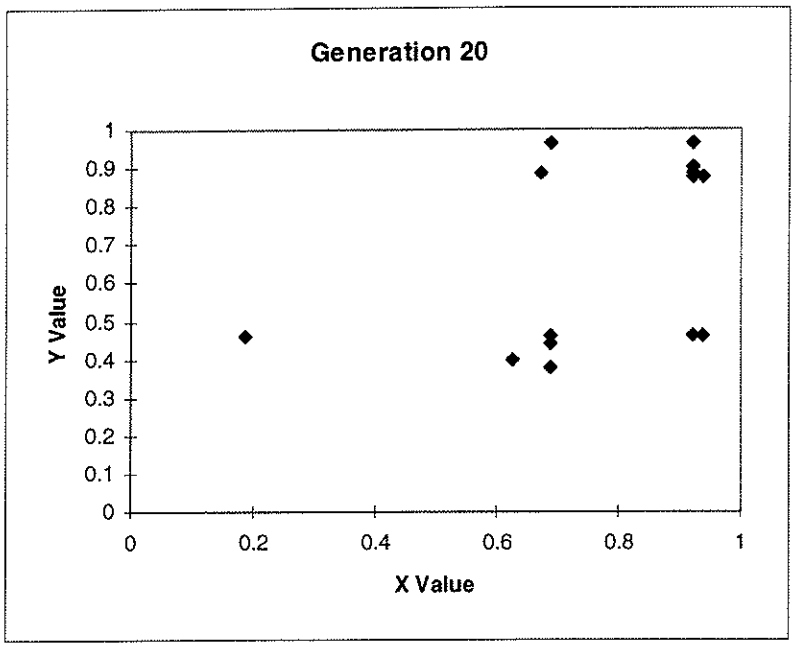
The following graphs are samples of the population at a generation. The population of points migrates to the upper right, rather than the lower left. Either of these patterns of migration would indicate a migration toward an optimal solution. Since the chromosomes cannot represent the value 1.0, but can represent 0.0, a migration toward the lower left would allow the possibility of finding an optimal solution.

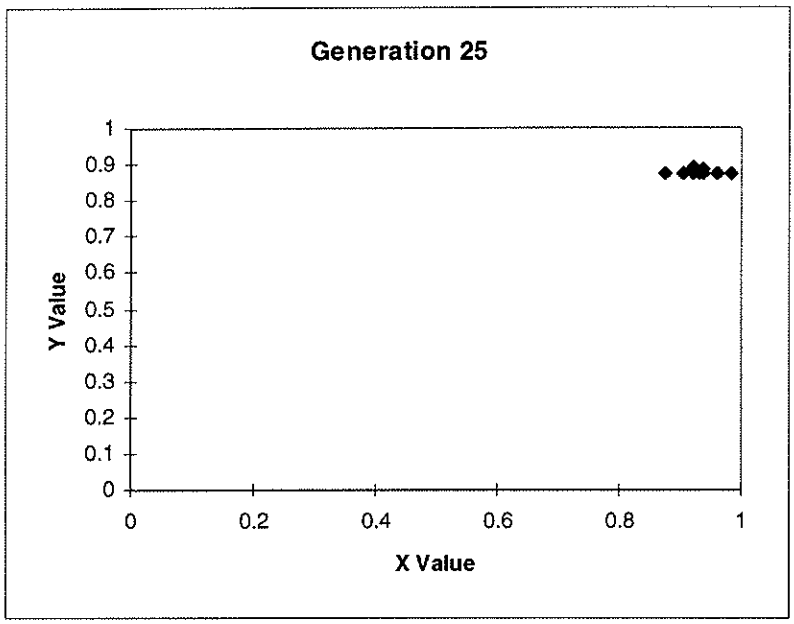

Generation 0 (Initial Population)
Y Value
X Value
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
0
0.2
0.4
0.6
0.8
1


Generation 5
Y Value
X Value
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
0
0.2
0.4
0.6
0.8
1


Generation 10
Y Value
X Value
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1


Generation 15
Y Value
X Value
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1

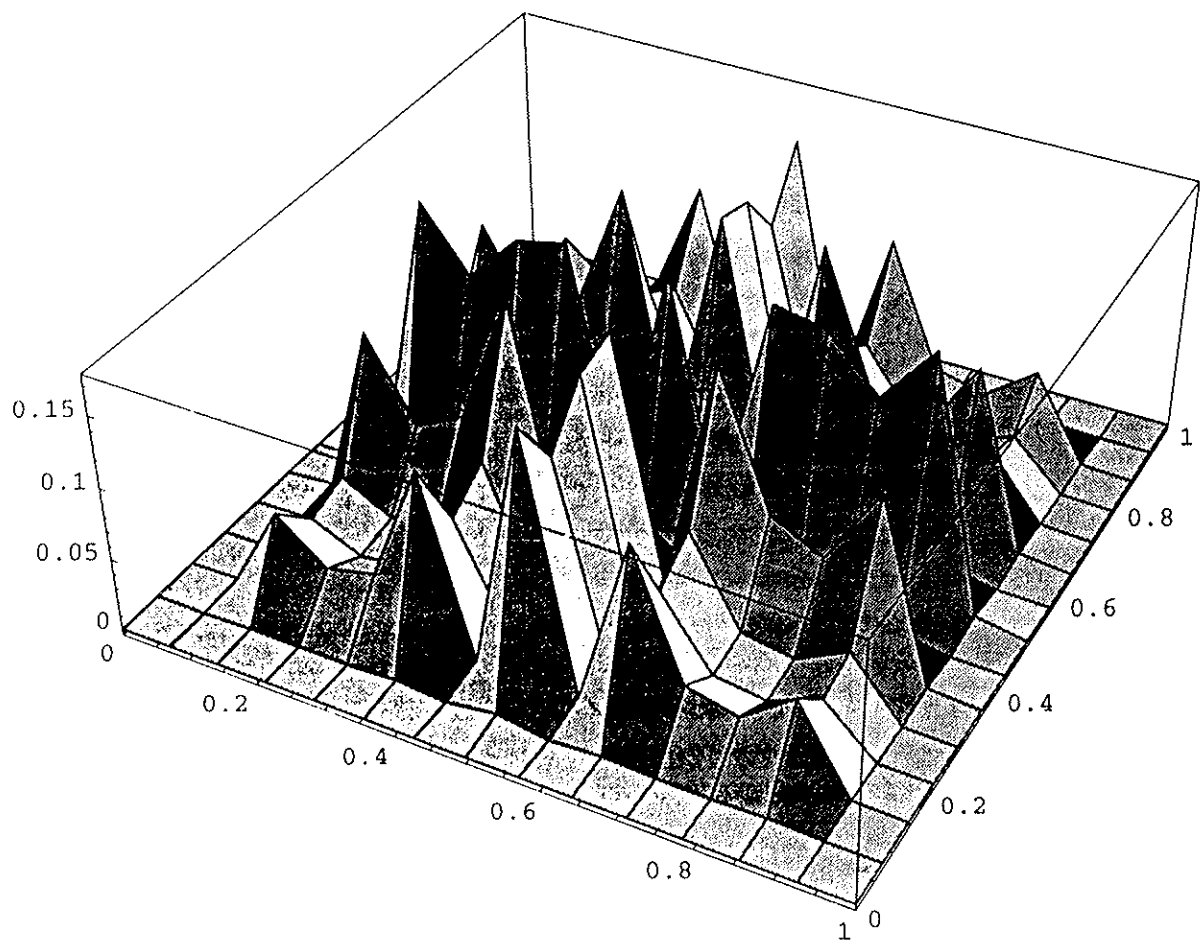

Generation 20
Y Value
X Value
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
0
0.2
0.4
0.6
0.8
1


Generation 25
Y Value
X Value
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
0
0.2
0.4
0.6
0.8
1

4.2 Function sin(Pi*x)*(1 - fabs(sin(6*Pi*x)))*sin(Pi*y)*(1 - fabs(sin(6*Pi*y))).

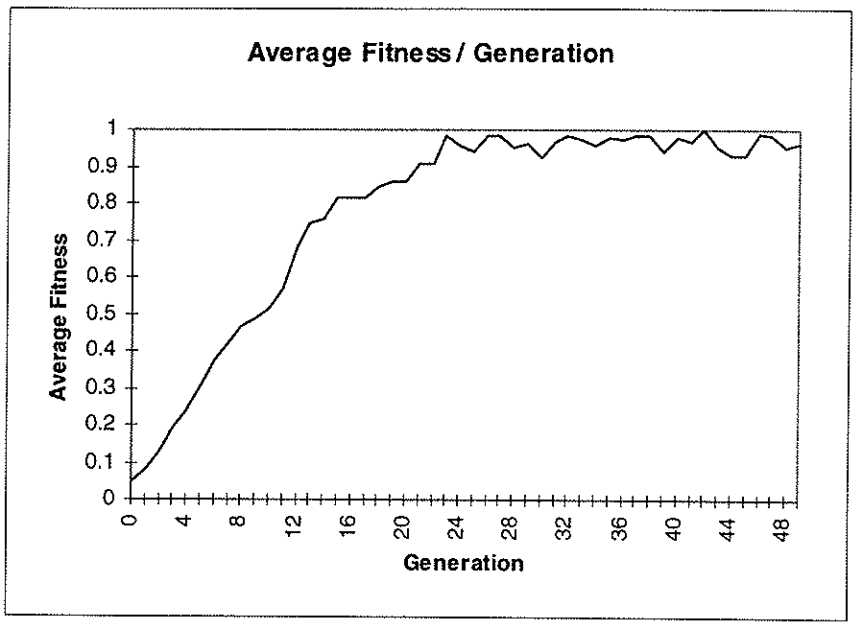
The following test runs include data obtained from varying the parameters of crossover and mutation rate. The coordinate fitness program attempts to find the maximum value of the function, at (0.5, 0.5) which evaluates to 1.0.

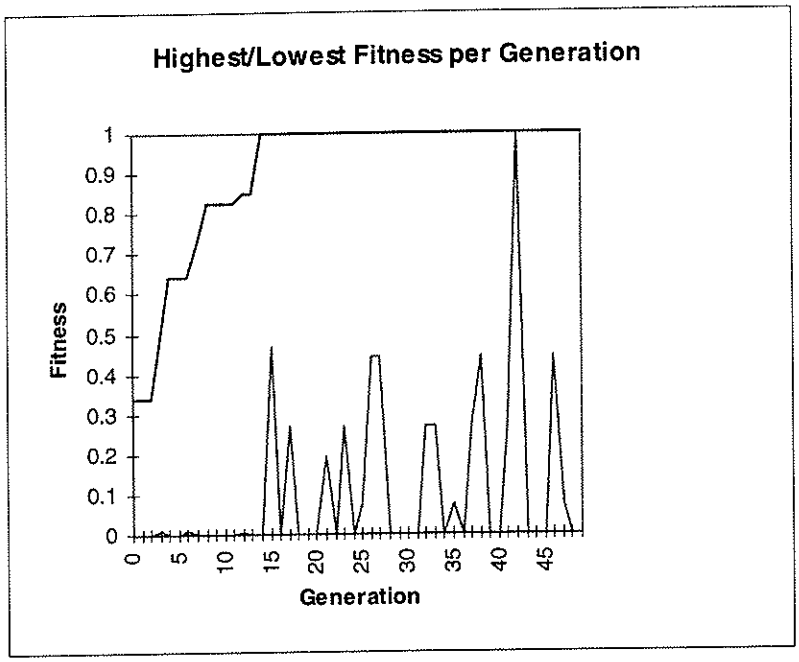
4.2.1 Graph of sin(Pi*x)*(1 - fabs(sin(6*Pi*x)))*sin(Pi*y)*(1 - fabs(sin(6*Pi*y)))



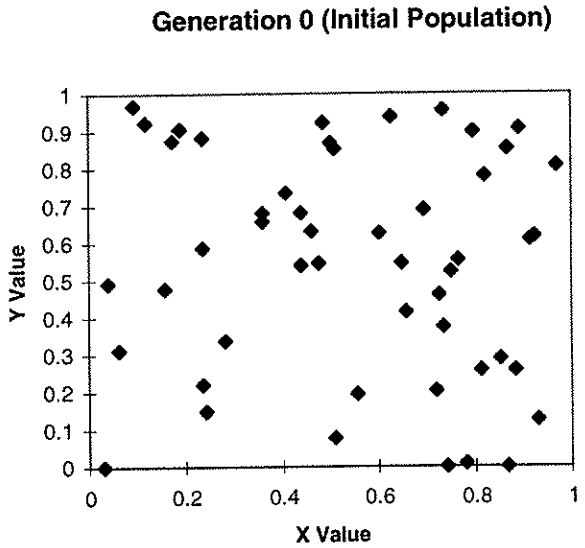
4.2.2 Test Run 1

The following graphs are taken from a test run with the parameters: chromosome length = 14, population size = 50, mutation rate = .2, crossover rate = .05, function to maximize is: sin(Pi*x)*(1 - fabs(sin(6*Pi*x)))*sin(Pi*y)*(1 - fabs(sin(6*Pi*y))). In this run the optimal maximization point was found in generation 14, as indicated by the graph titled highest/lowest fitness per generation. The average fitness of a population indicates the sum total of each individual's fitness divided by the number of individuals in the population. By generation 23 the population began to converge, as indicated by the graph at generation 23 with an average fitness near the optimal.

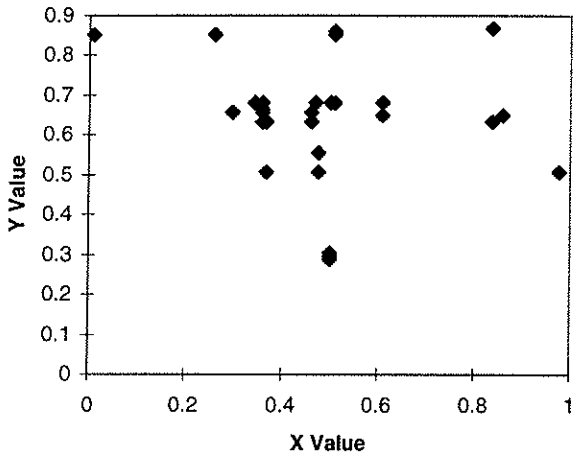




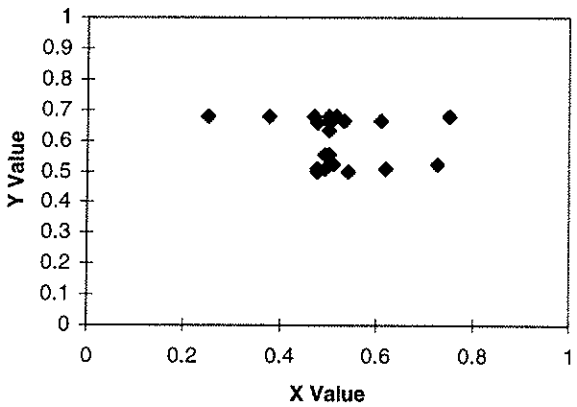
The following graphs will indicate each point in a given generation



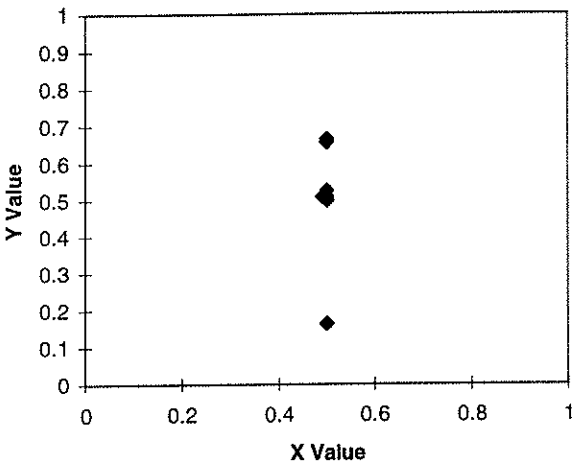
Generation 5



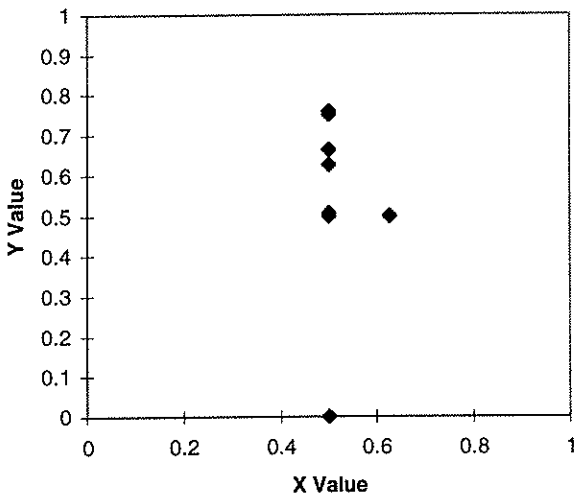
Generation 10

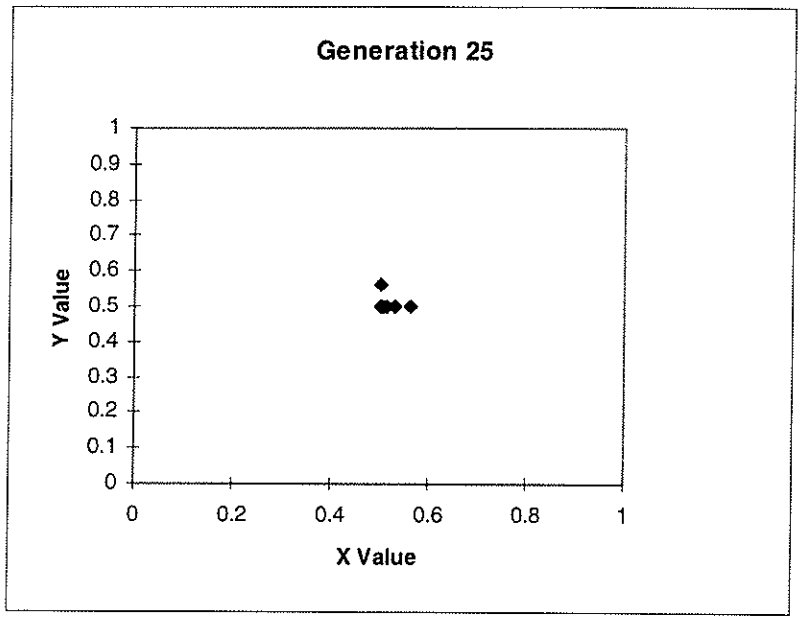


Generation 15



Generation 20





In this run, the population converged on the point (0.5, 0.5). The graph of the population in generation 15 indicates points migrating toward the center. Generation 20 and 25 indicate a greater degree of convergence on the point (0.5, 0.5), with some variation due to mutation and crossover.

4.2.3 Test Run 2

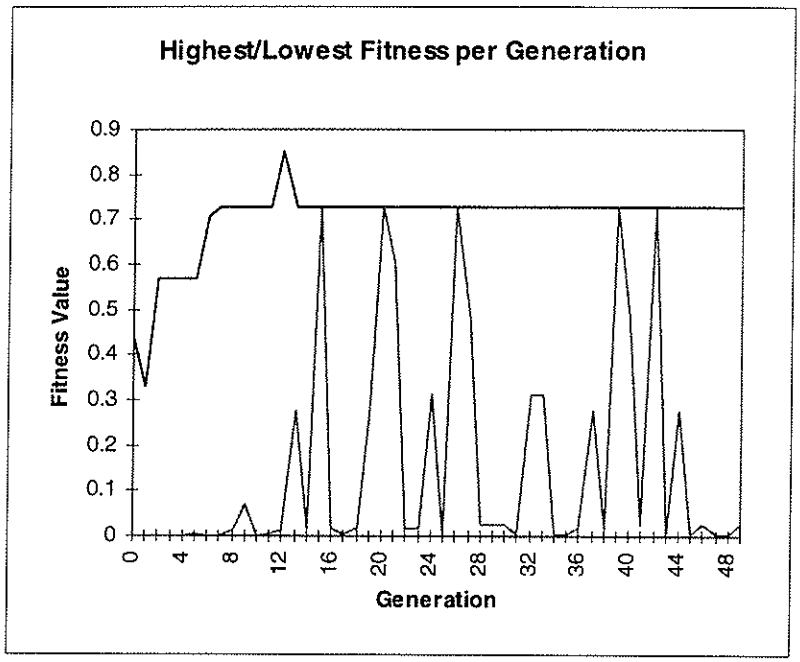
The following graphs are taken from a test run with the same parameters: chromosome length = 14, population size = 50, mutation rate = .2, crossover rate = .05, function to maximize is: sin(Pi*x)*(1 - fabs(sin(6*Pi*x)))*sin(Pi*y)*(1 - fabs(sin(6*Pi*y))).

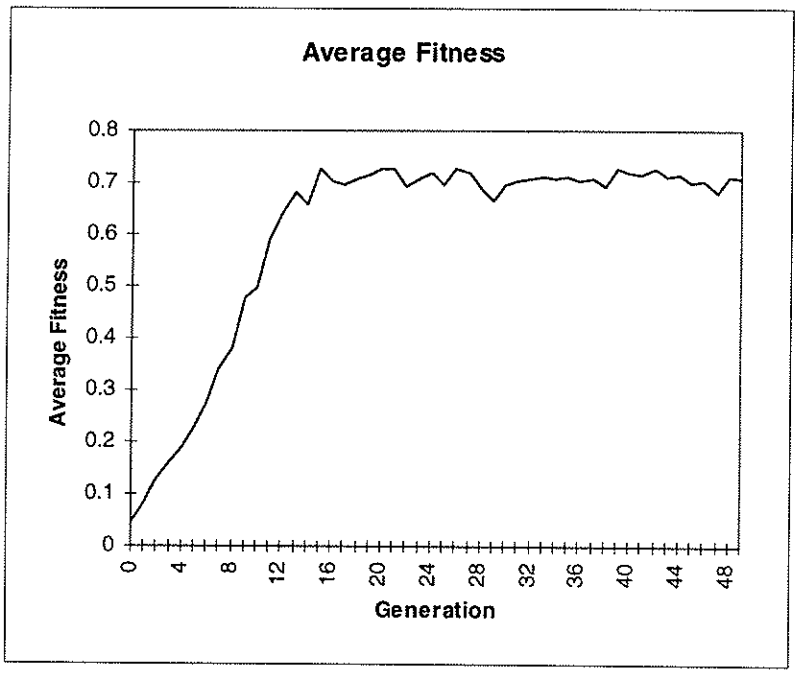
In this case, the population converged on the chromosome 0111111 0111111 which represents x and y values of .492188 and a fitness value of .72763. The optimal solution is represented by a chromosome that looks like 1000000 1000000. The fitness value .72763 is not bad, allowing individuals containing this chromosome to live in many

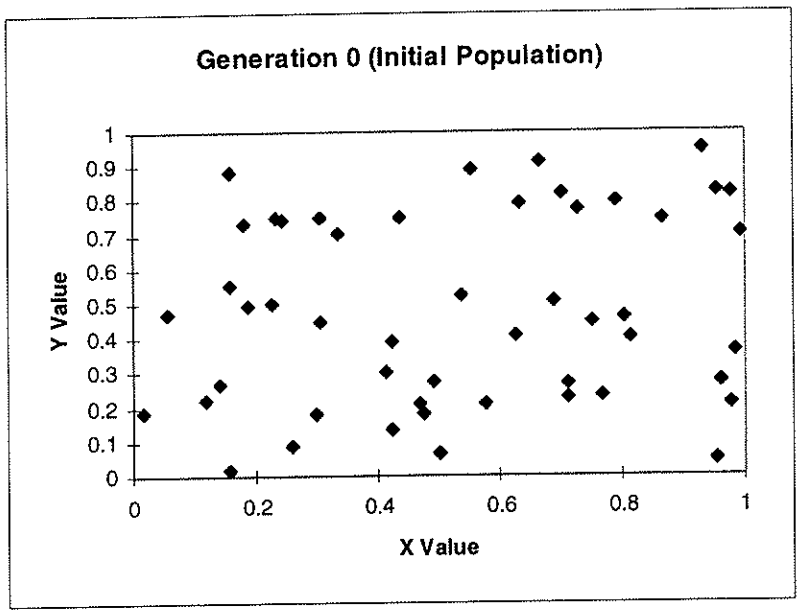
cases. Since it very often survives tournament selection, in order to improve the fitness value crossover and mutation are needed.

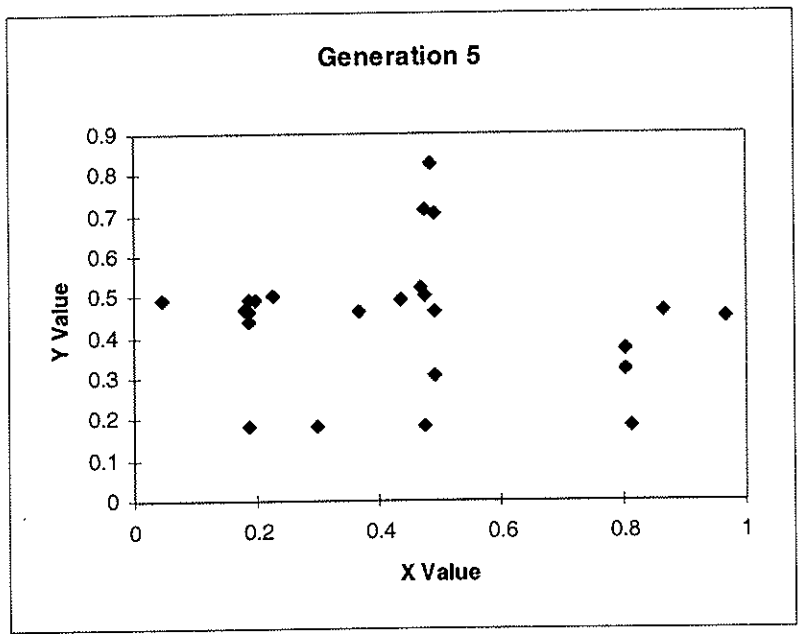
Once the population begins to converge, crossover does nothing, it simply crosses two identical pairs which result in two identical children. Mutation will have a difficult time of changing the chromosome 0111111 0111111 to 1000000 1000000. In the coordinate fitness program mutation only flips one bit. In order for mutation to have a positive effect, the mutated chromosome must have a higher fitness value than before, and then be chosen for the selection process. With the chromosome 0111111 0111111 that becomes difficult. For example, in a generation of the test run two separate chromosomes of this type were chosen for mutation. In the first, the sixth bit was mutated for an x and y value of .476562, .492188 represented by the chromosome 0111101 0111111 and a fitness value of .486979. In the second case, the eleventh bit was mutated for x and y values .492188, .429688 represented by the chromosome 0111111 0110111 and a fitness value of .0249425.

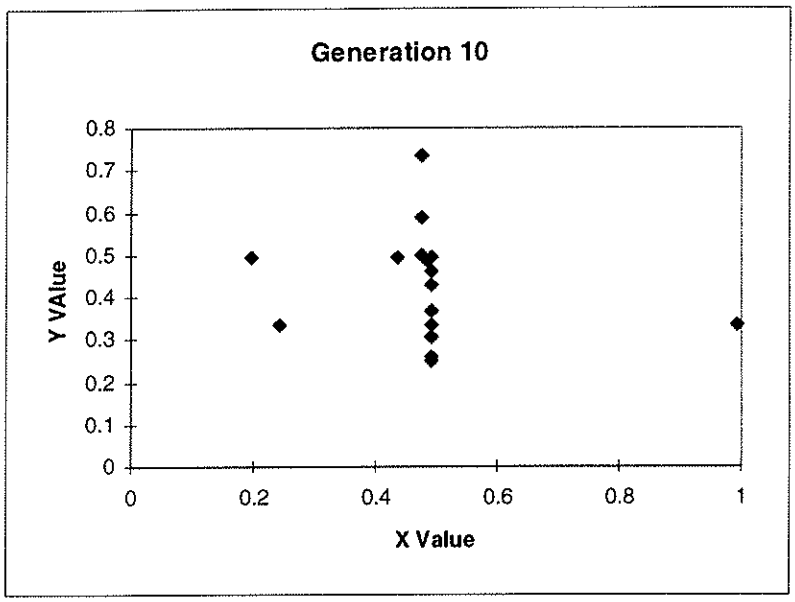
In generation 12 it appears that a mutated chromosome did result in a higher fitness value, as indicated by the graph of highest and lowest fitness per generation. That individual was not selected, however, and did not move on to the next generation.

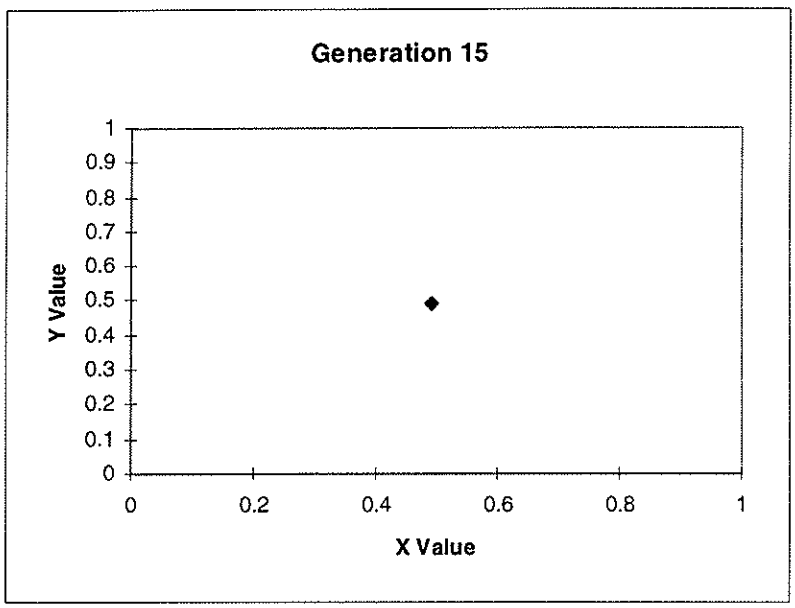

Highest/Lowest Fitness per Generation
Fitness Value
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
4
8
12
16
20
24
28
32
36
40
44
48
Generation


Average Fitness
Average Fitness
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
4
8
12
16
20
24
28
32
36
40
44
48
Generation

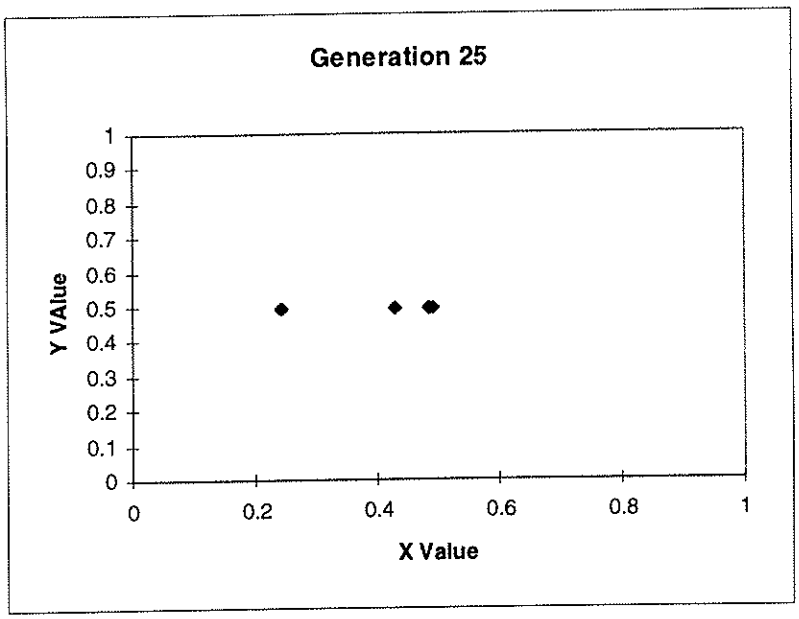

Generation 0 (Initial Population)
Y Value
X Value
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
0
0.2
0.4
0.6
0.8
1


Generation 5
Y Value
X Value
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
0
0.2
0.4
0.6
0.8
1





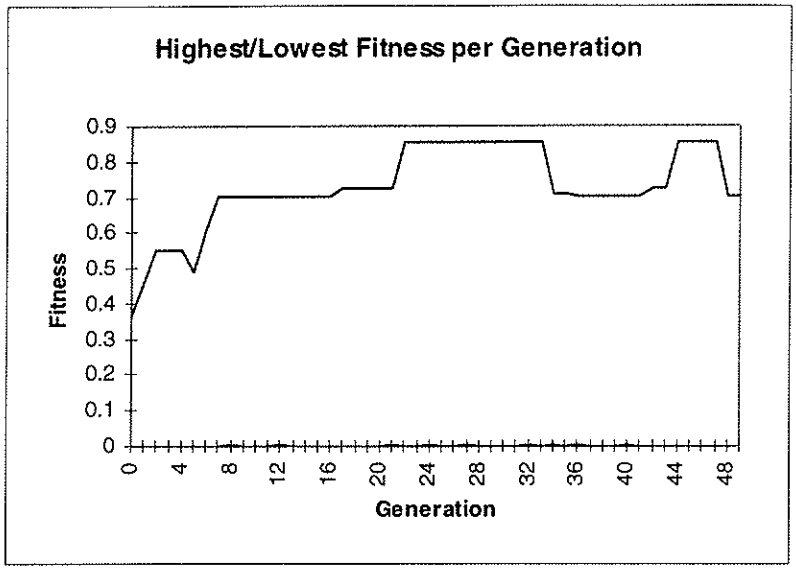
Generation 20 contains exactly the same population as generation 15, indicating that crossover may have occurred, but mutation did not.

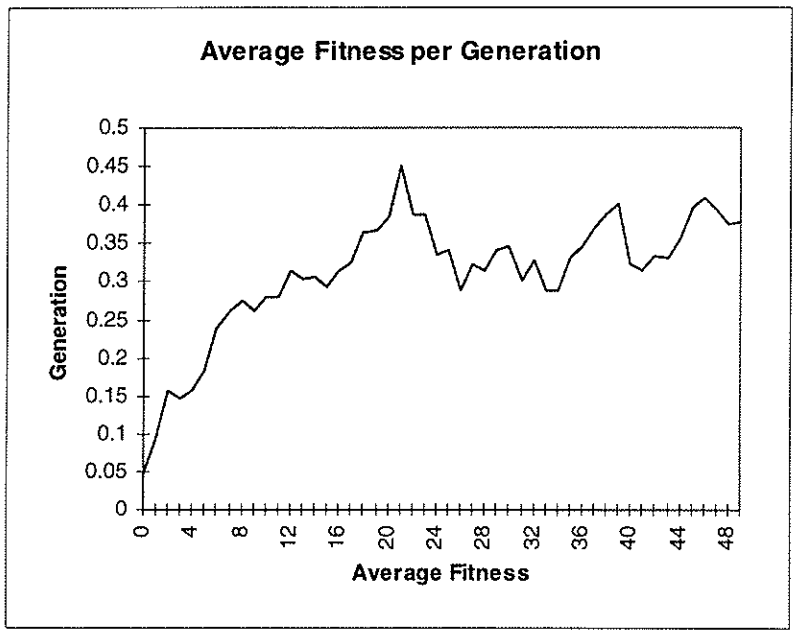


4.2.4 Test Run 3

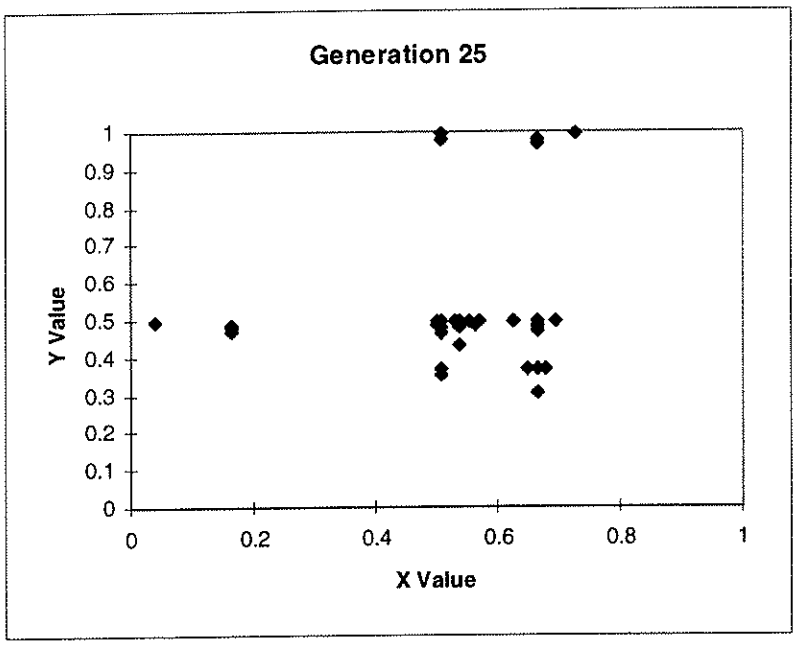
In this test run, the crossover rate is kept the same at 0.2, but mutation is increased to 0.5. The remaining parameters are kept the same: chromosome length = 14, population size = 50 number of generations = 50.

With a high mutation rate the population should remain unstable. It would be difficult for the population to converge on a single point when approximately one half of the chromosomes in any given generation are being mutated.



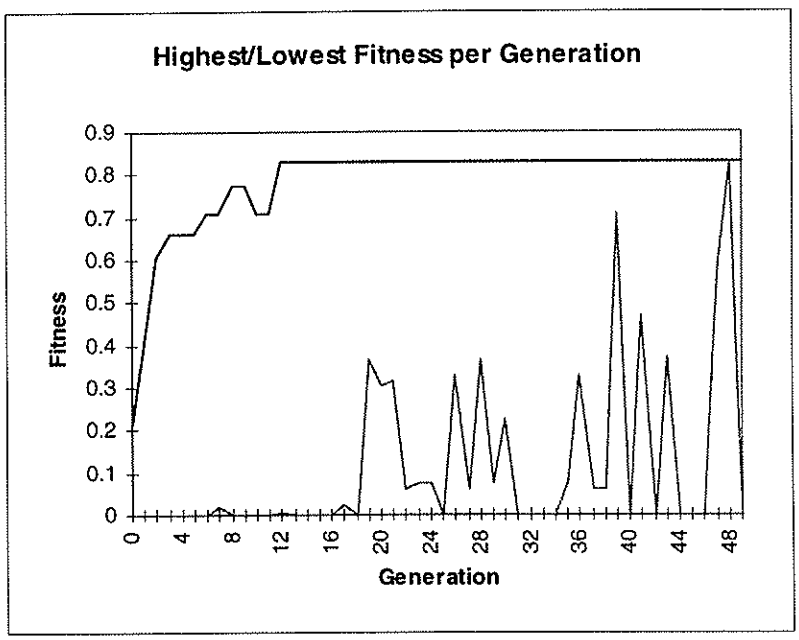


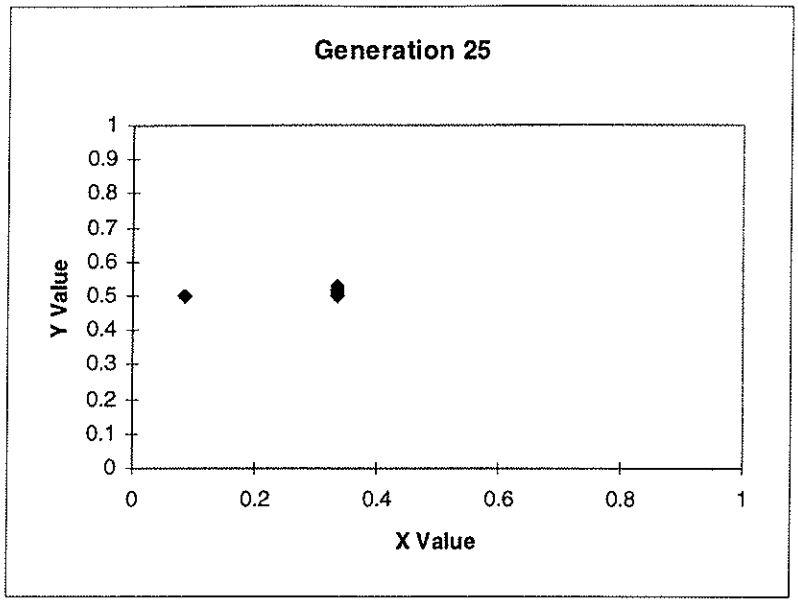
In each of the previous test runs the population began to migrate to one area by generation 25. In this run the points did migrate toward the center, but the convergence was much less drastic.



4.2.5 Test Run 4

In this run mutation is kept at 0.05, but crossover is increased to 0.5. The remaining parameters are kept the same: chromosome length = 14, population size = 50, number of generations = 50.





4.3 Conclusions Based on Test Data

Crossover and mutation together are important parameters in genetic algorithms. Determining precisely the best combination of those rates together, however, is a difficult task. When the coordinate fitness program was applied to both of the functions above with a crossover rate of 0.2 and a mutation rate of 0.05 the optimal solution was found in some cases, but not all. Is this the best we can do, or would a better combination of these rates ensure the optimal was found more frequently? Perhaps the program in section 5.0 could answer that question. With the limited amount of experimentation I was able to perform, I am not prepared to draw a decisive conclusion on the matter.

A mutation rate that is relatively high introduces a high degree of variance to a population. This may be desirable in cases where a population tends to converge very quickly. It may also cause a point to jump off of the optimal solution it has found. A low

mutation rate will allow the population to converge quickly. This may be acceptable in cases where the optimal solution can be found rather easily.

The goal of crossover is to create a more fit individual from two other individuals. In section 4.2.5 crossover was increased while mutation remained at 0.05. In that test run the optimal solution was not found, but in a number of test runs with the same parameters the optimal was found. The reason for this may be the way the crossover function is written. When contemplating how to implement crossover in the coordinate fitness program, Professor Gass and I considered two alternatives. One called for all chromosomes to be crossed at the midway point, thereby allowing the x and y values to remain intact. The logic behind this is that a chromosome is fit because of its x and y pairs. The second alternative allows crossover to occur at any point in the chromosome. The first implementation would not disrupt a chromosome as much as the latter. The version that is currently used allows crossover to occur at any point in the chromosome. This is not entirely unfavorable in that it always allows one half of the chromosome to remain intact, and replaces a portion of the other half with that of a different chromosome.

I feel that the functions tested would have fared better with the other version of crossover. A common point found in optimizing the function used in section 4.2 was an x or y value of 0.5, and corresponding pair close to 0.5. Consider the chromosome 1000000 0111111 with x value 0.5 and y value 0.492188. The current version of crossover might attempt to cross the above mentioned chromosome with 0110111 1000000. It has to get lucky to find the right crossover point. A higher crossover rate increases the chances of drawing a lucky crossover point.

The function in 4.2 happens to have an optimal solution at the point 0.5, 0.5. Whether this or another crossover function would be best suited for other functions requires further experimentation.

It is possible for a genetic algorithm to act as a random search. Without an effective means of mutation and crossover this is likely to occur. This was one of the tests on the implementation of crossover and mutation. If those functions were not working well in the coordinate fitness program, a test run would have a difficult time evolving more fit individuals. The best fit individual in the initial and final populations would look similar.

The program does seem to provide an adequate means for viewing the progress of a genetic algorithm. By viewing the population at each generation the user is able to see the migration of points in time.

5.0 Next Phase: Running the Program on Itself

As mentioned above, one of the questions involving genetic algorithms today is a method for selecting good crossover and mutation rates together. This particular program is set up well for implementing such a method.

I am currently working on the following variation of the coordinate fitness program. The idea is that the genetic algorithm will work on itself. The lower layer genetic algorithm will work on maximizing or minimizing a particular function, with a mutation and crossover rate given to it by the higher layer genetic algorithm. The lower layer GA will contain individuals as x and y coordinates as before. The higher layer will contain individuals with mutation and crossover rates represented as before by a chromosome split into two halves. The value of each will be computed in the same

manner as the x and y values. The higher layer GA will pass to its counterpart a mutation and crossover rate. The receiving GA will run a specified number of generations, record the highest (or lowest) fitness value in the run, and pass that information on to the higher layer. The individual containing mutation and crossover rates that spawned that run will record the fitness value it receives as its fitness value. This will continue for each individual of the higher layer population. That population will then perform selection, mutation, and crossover (re-calculating the values if necessary) for the next generation.

The next generation will carry out the same procedure. Each individual in each generation of the higher layer then represents a run of the lower layer GA. Theoretically, each run should improve with each generation in both the higher and lower GAs. The final run of the higher layer should result in better choices of mutation and crossover rates, and the lower layer should result in increasingly better fitness values per generation. The higher layer should not count on this, however, and should record the best fitness value (or best handful of fitness values) and their corresponding rates.

6.0 Appendices

6.1 Output of an entire program run

The following is an example of output from the coordinate fitness program. Only ten generations appear here, and the population size has been reduced to ten. The crossover rate is set to 0.2 and mutation to 0.1. The fitness function is sin(Pi*x)*(1 - fabs(sin(6*Pi*x)))*sin(Pi*y)*(1 - fabs(sin(6*Pi*y))), and the program is attempting to maximize. Selection is indicated by the number of the new chromosome, followed by the two chromosomes in the previous generation that will compete for that slot. Mutation is indicated by the number of the newly created chromosome chosen for mutation, followed

by the bit to mutate. Pairs of chromosomes selected for crossover are indicated by the two new chromosome numbers followed by the point at which they will cross.

```
Script started on Tue May 13 14:03:45 1997
pwjones@dzogbegan 1% gatest

Initializing population
initial population (generation 0) is:
Chromosome 1: 00001010111111  fitness: 0.0342951 X: 0.0390625 Y: 0.492188
Chromosome 2: 11110011110011  fitness: 0.000446 X: 0.945312 Y: 0.898438
Chromosome 3: 01100111000110  fitness: 0.0124628 X: 0.398438 Y: 0.546875
Chromosome 4: 11010110100101  fitness: 0.0957207 X: 0.835938 Y: 0.289062
Chromosome 5: 10110010010010  fitness: 0.0897841 X: 0.695312 Y: 0.140625
Chromosome 6: 11111110100110  fitness: 0.00614929 X: 0.992188 Y: 0.296875
Chromosome 7: 01100110011100  fitness: 0.00593436 X: 0.398438 Y: 0.21875
Chromosome 8: 11000110110010  fitness: 0.00697204 X: 0.773438 Y: 0.390625
Chromosome 9: 01111111100101  fitness: 0.135949 X: 0.492188 Y: 0.789062
Chromosome 10: 00100000000001  fitness: 0.0023471 X: 0.125 Y: 0.0078125
Total fitness is: 0.39006
Average fitness is: 0.039006
Highest: 0.135949
Lowest: 0.000446
what do you want the crossover rate to be?
.2
What do you want the mutation rate to be?
.1
How many generations do you want to run?
10
1: chosen pairs to fight are: 9, 6
2: chosen pairs to fight are: 10, 6
3: chosen pairs to fight are: 2, 9
4: chosen pairs to fight are: 9, 1
5: chosen pairs to fight are: 3, 1
6: chosen pairs to fight are: 10, 7
7: chosen pairs to fight are: 1, 8
8: chosen pairs to fight are: 3, 5
9: chosen pairs to fight are: 2, 8
10: chosen pairs to fight are: 2, 8
mutating chrom 1 Randomly chosen bit to mutate is 10
6 cross with 4 at 10
3 cross with 7 at 11
after generation 1
Chromosome 1: 01111111110101  fitness: 0.000274048 X: 0.492188 Y: 0.914062
Chromosome 2: 11111110100110  fitness: 0.00614929 X: 0.992188 Y: 0.296875
Chromosome 3: 00001010111101  fitness: 0.0229526 X: 0.0390625 Y: 0.476562
Chromosome 4: 01100110010101  fitness: 0.0260162 X: 0.398438 Y: 0.164062
Chromosome 5: 00001010111111  fitness: 0.0342951 X: 0.0390625 Y: 0.492188
Chromosome 6: 01111111101100  fitness: 0.32366 X: 0.492188 Y: 0.84375
Chromosome 7: 01111111100111  fitness: 0.238659 X: 0.492188 Y: 0.804688
Chromosome 8: 10110010010010  fitness: 0.0897841 X: 0.695312 Y: 0.140625
Chromosome 9: 11000110110010  fitness: 0.00697204 X: 0.773438 Y: 0.390625
Chromosome 10: 11000110110010  fitness: 0.00697204 X: 0.773438 Y: 0.390625
Total fitness is: 0.755734
```

Average fitness is: 0.0755734
Highest: 0.32366
Lowest: 0.000274048
1: chosen pairs to fight are: 5, 8
2: chosen pairs to fight are: 7, 10
3: chosen pairs to fight are: 1, 5
4: chosen pairs to fight are: 6, 8
5: chosen pairs to fight are: 1, 1
6: chosen pairs to fight are: 5, 7
7: chosen pairs to fight are: 5, 5
8: chosen pairs to fight are: 4, 4
9: chosen pairs to fight are: 4, 4
10: chosen pairs to fight are: 6, 4
mutating chrom 4 Randomly chosen bit to mutate is 8
9 cross with 9 at 0
after generation 2
Chromosome 1: 10110010010010 fitness: 0.0897841 X: 0.695312 Y: 0.140625
Chromosome 2: 01111111100111 fitness: 0.238659 X: 0.492188 Y: 0.804688
Chromosome 3: 00001010111111 fitness: 0.0342951 X: 0.0390625 Y: 0.492188
Chromosome 4: 01111110101100 fitness: 0.605525 X: 0.492188 Y: 0.34375
Chromosome 5: 01111111110101 fitness: 0.000274048 X: 0.492188 Y: 0.914062
Chromosome 6: 01111111100111 fitness: 0.238659 X: 0.492188 Y: 0.804688
Chromosome 7: 00001010111111 fitness: 0.0342951 X: 0.0390625 Y: 0.492188
Chromosome 8: 01100110010101 fitness: 0.0260162 X: 0.398438 Y: 0.164062
Chromosome 9: 01100110010101 fitness: 0.0260162 X: 0.398438 Y: 0.164062
Chromosome 10: 01111111101100 fitness: 0.32366 X: 0.492188 Y: 0.84375
Total fitness is: 1.61718
Average fitness is: 0.161718
Highest: 0.605525
Lowest: 0.000274048
1: chosen pairs to fight are: 5, 7
2: chosen pairs to fight are: 7, 3
3: chosen pairs to fight are: 9, 7
4: chosen pairs to fight are: 4, 7
5: chosen pairs to fight are: 4, 1
6: chosen pairs to fight are: 6, 8
7: chosen pairs to fight are: 2, 7
8: chosen pairs to fight are: 6, 6
9: chosen pairs to fight are: 6, 4
10: chosen pairs to fight are: 7, 7
mutating chrom 8 Randomly chosen bit to mutate is 14
after generation 3
Chromosome 1: 00001010111111 fitness: 0.0342951 X: 0.0390625 Y: 0.492188
Chromosome 2: 00001010111111 fitness: 0.0342951 X: 0.0390625 Y: 0.492188
Chromosome 3: 00001010111111 fitness: 0.0342951 X: 0.0390625 Y: 0.492188
Chromosome 4: 01111110101100 fitness: 0.605525 X: 0.492188 Y: 0.34375
Chromosome 5: 01111110101100 fitness: 0.605525 X: 0.492188 Y: 0.34375
Chromosome 6: 01111111100111 fitness: 0.238659 X: 0.492188 Y: 0.804688
Chromosome 7: 01111111100111 fitness: 0.238659 X: 0.492188 Y: 0.804688
Chromosome 8: 01111111100110 fitness: 0.185779 X: 0.492188 Y: 0.796875
Chromosome 9: 01111110101100 fitness: 0.605525 X: 0.492188 Y: 0.34375
Chromosome 10: 00001010111111 fitness: 0.0342951 X: 0.0390625 Y: 0.492188
Total fitness is: 2.61685
Average fitness is: 0.261685
Highest: 0.605525

Lowest: 0.0342951
1: chosen pairs to fight are: 8, 3
2: chosen pairs to fight are: 10, 10
3: chosen pairs to fight are: 3, 9
4: chosen pairs to fight are: 5, 1
5: chosen pairs to fight are: 5, 5
6: chosen pairs to fight are: 6, 7
7: chosen pairs to fight are: 3, 8
8: chosen pairs to fight are: 10, 2
9: chosen pairs to fight are: 8, 7
10: chosen pairs to fight are: 4, 2
mutating chrom 5 Randomly chosen bit to mutate is 1
mutating chrom 10 Randomly chosen bit to mutate is 11
9 cross with 4 at 7
after generation 4
Chromosome 1: 01111111100110 fitness: 0.185779 X: 0.492188 Y: 0.796875
Chromosome 2: 00001010111111 fitness: 0.0342951 X: 0.0390625 Y: 0.492188
Chromosome 3: 01111110101100 fitness: 0.605525 X: 0.492188 Y: 0.34375
Chromosome 4: 01111110101100 fitness: 0.605525 X: 0.492188 Y: 0.34375
Chromosome 5: 11111110101100 fitness: 0.0148648 X: 0.992188 Y: 0.34375
Chromosome 6: 01111111100111 fitness: 0.238659 X: 0.492188 Y: 0.804688
Chromosome 7: 01111111100110 fitness: 0.185779 X: 0.492188 Y: 0.796875
Chromosome 8: 00001010111111 fitness: 0.0342951 X: 0.0390625 Y: 0.492188
Chromosome 9: 01111111100111 fitness: 0.238659 X: 0.492188 Y: 0.804688
Chromosome 10: 01111110100100 fitness: 0.111127 X: 0.492188 Y: 0.28125
Total fitness is: 2.25451
Average fitness is: 0.225451
Highest: 0.605525
Lowest: 0.0148648
1: chosen pairs to fight are: 9, 6
2: chosen pairs to fight are: 6, 1
3: chosen pairs to fight are: 9, 5
4: chosen pairs to fight are: 4, 3
5: chosen pairs to fight are: 7, 9
6: chosen pairs to fight are: 1, 8
7: chosen pairs to fight are: 1, 10
8: chosen pairs to fight are: 4, 4
9: chosen pairs to fight are: 7, 8
10: chosen pairs to fight are: 1, 7
mutating chrom 4 Randomly chosen bit to mutate is 8
8 cross with 2 at 1
after generation 5
Chromosome 1: 01111111100111 fitness: 0.238659 X: 0.492188 Y: 0.804688
Chromosome 2: 01111111100111 fitness: 0.238659 X: 0.492188 Y: 0.804688
Chromosome 3: 01111111100111 fitness: 0.238659 X: 0.492188 Y: 0.804688
Chromosome 4: 01111111101100 fitness: 0.32366 X: 0.492188 Y: 0.84375
Chromosome 5: 01111111100111 fitness: 0.238659 X: 0.492188 Y: 0.804688
Chromosome 6: 01111111100110 fitness: 0.185779 X: 0.492188 Y: 0.796875
Chromosome 7: 01111111100110 fitness: 0.185779 X: 0.492188 Y: 0.796875
Chromosome 8: 01111110101100 fitness: 0.605525 X: 0.492188 Y: 0.34375
Chromosome 9: 01111111100110 fitness: 0.185779 X: 0.492188 Y: 0.796875
Chromosome 10: 01111111100110 fitness: 0.185779 X: 0.492188 Y: 0.796875
Total fitness is: 2.62694
Average fitness is: 0.262694
Highest: 0.605525

Lowest: 0.185779
1: chosen pairs to fight are: 7, 6
2: chosen pairs to fight are: 6, 5
3: chosen pairs to fight are: 7, 8
4: chosen pairs to fight are: 3, 8
5: chosen pairs to fight are: 3, 8
6: chosen pairs to fight are: 3, 10
7: chosen pairs to fight are: 6, 8
8: chosen pairs to fight are: 6, 3
9: chosen pairs to fight are: 3, 1
10: chosen pairs to fight are: 7, 7
mutating chrom 4 Randomly chosen bit to mutate is 9
8 cross with 1 at 1
10 cross with 5 at 4
after generation 6
Chromosome 1: 01111111100110 fitness: 0.185779 X: 0.492188 Y: 0.796875
Chromosome 2: 01111111100111 fitness: 0.238659 X: 0.492188 Y: 0.804688
Chromosome 3: 01111110101100 fitness: 0.605525 X: 0.492188 Y: 0.34375
Chromosome 4: 01111110001100 fitness: 0.00475788 X: 0.492188 Y: 0.09375
Chromosome 5: 01111110101100 fitness: 0.605525 X: 0.492188 Y: 0.34375
Chromosome 6: 01111111100111 fitness: 0.238659 X: 0.492188 Y: 0.804688
Chromosome 7: 01111110101100 fitness: 0.605525 X: 0.492188 Y: 0.34375
Chromosome 8: 01111111100111 fitness: 0.238659 X: 0.492188 Y: 0.804688
Chromosome 9: 01111111100111 fitness: 0.238659 X: 0.492188 Y: 0.804688
Chromosome 10: 01111111100110 fitness: 0.185779 X: 0.492188 Y: 0.796875
Total fitness is: 3.14753
Average fitness is: 0.314753
Highest: 0.605525
Lowest: 0.00475788
1: chosen pairs to fight are: 9, 3
2: chosen pairs to fight are: 9, 8
3: chosen pairs to fight are: 3, 2
4: chosen pairs to fight are: 8, 10
5: chosen pairs to fight are: 7, 2
6: chosen pairs to fight are: 7, 8
7: chosen pairs to fight are: 9, 10
8: chosen pairs to fight are: 5, 2
9: chosen pairs to fight are: 3, 8
10: chosen pairs to fight are: 10, 7
mutating chrom 2 Randomly chosen bit to mutate is 8
mutating chrom 6 Randomly chosen bit to mutate is 7
3 cross with 7 at 2
6 cross with 3 at 8
after generation 7
Chromosome 1: 01111110101100 fitness: 0.605525 X: 0.492188 Y: 0.34375
Chromosome 2: 01111110100111 fitness: 0.33887 X: 0.492188 Y: 0.304688
Chromosome 3: 01111100101100 fitness: 0.503197 X: 0.484375 Y: 0.34375
Chromosome 4: 01111111100111 fitness: 0.238659 X: 0.492188 Y: 0.804688
Chromosome 5: 01111110101100 fitness: 0.605525 X: 0.492188 Y: 0.34375
Chromosome 6: 01111110101100 fitness: 0.605525 X: 0.492188 Y: 0.34375
Chromosome 7: 01111111100111 fitness: 0.238659 X: 0.492188 Y: 0.804688
Chromosome 8: 01111110101100 fitness: 0.605525 X: 0.492188 Y: 0.34375
Chromosome 9: 01111110101100 fitness: 0.605525 X: 0.492188 Y: 0.34375
Chromosome 10: 01111110101100 fitness: 0.605525 X: 0.492188 Y: 0.34375
Total fitness is: 4.95254

Average fitness is: 0.495254
Highest: 0.605525
Lowest: 0.238659
1: chosen pairs to fight are: 8, 1
2: chosen pairs to fight are: 4, 10
3: chosen pairs to fight are: 7, 9
4: chosen pairs to fight are: 5, 4
5: chosen pairs to fight are: 5, 6
6: chosen pairs to fight are: 3, 10
7: chosen pairs to fight are: 10, 1
8: chosen pairs to fight are: 7, 6
9: chosen pairs to fight are: 5, 1
10: chosen pairs to fight are: 6, 2
mutating chrom 8 Randomly chosen bit to mutate is 2
mutating chrom 10 Randomly chosen bit to mutate is 10
3 cross with 4 at 12
2 cross with 3 at 12
after generation 8
Chromosome 1: 01111110101100 fitness: 0.605525 X: 0.492188 Y: 0.34375
Chromosome 2: 01111110101100 fitness: 0.605525 X: 0.492188 Y: 0.34375
Chromosome 3: 01111110101100 fitness: 0.605525 X: 0.492188 Y: 0.34375
Chromosome 4: 01111110101100 fitness: 0.605525 X: 0.492188 Y: 0.34375
Chromosome 5: 01111110101100 fitness: 0.605525 X: 0.492188 Y: 0.34375
Chromosome 6: 01111110101100 fitness: 0.605525 X: 0.492188 Y: 0.34375
Chromosome 7: 01111110101100 fitness: 0.605525 X: 0.492188 Y: 0.34375
Chromosome 8: 00111110101100 fitness: 0.00529791 X: 0.242188 Y: 0.34375
Chromosome 9: 01111110101100 fitness: 0.605525 X: 0.492188 Y: 0.34375
Chromosome 10: 01111110111100 fitness: 0.377279 X: 0.492188 Y: 0.46875
Total fitness is: 5.22678
Average fitness is: 0.522678
Highest: 0.605525
Lowest: 0.00529791
1: chosen pairs to fight are: 5, 9
2: chosen pairs to fight are: 4, 9
3: chosen pairs to fight are: 5, 6
4: chosen pairs to fight are: 7, 9
5: chosen pairs to fight are: 5, 7
6: chosen pairs to fight are: 5, 6
7: chosen pairs to fight are: 4, 4
8: chosen pairs to fight are: 3, 4
9: chosen pairs to fight are: 5, 6
10: chosen pairs to fight are: 5, 1
6 cross with 2 at 2
after generation 9
Chromosome 1: 01111110101100 fitness: 0.605525 X: 0.492188 Y: 0.34375
Chromosome 2: 01111110101100 fitness: 0.605525 X: 0.492188 Y: 0.34375
Chromosome 3: 01111110101100 fitness: 0.605525 X: 0.492188 Y: 0.34375
Chromosome 4: 01111110101100 fitness: 0.605525 X: 0.492188 Y: 0.34375
Chromosome 5: 01111110101100 fitness: 0.605525 X: 0.492188 Y: 0.34375
Chromosome 6: 01111110101100 fitness: 0.605525 X: 0.492188 Y: 0.34375
Chromosome 7: 01111110101100 fitness: 0.605525 X: 0.492188 Y: 0.34375
Chromosome 8: 01111110101100 fitness: 0.605525 X: 0.492188 Y: 0.34375
Chromosome 9: 01111110101100 fitness: 0.605525 X: 0.492188 Y: 0.34375
Chromosome 10: 01111110101100 fitness: 0.605525 X: 0.492188 Y: 0.34375
Total fitness is: 6.05525

```
Average fitness is: 0.605525
Highest: 0.605525
Lowest: 0.605525
pwjones@dzogbegan 2% exit
script done on Tue May 13 14:05:01 1997
```

6.2 Program Code

The code is listed in order of inheritance from the bottom up: Individual class, Population class, test class. The Makefile appears after the source code and contains reference to a test class called Gacommand. This code is not included, but was used as a command interpreter for running genetic algorithms.

6.2.1 Individual Class

```
// Individual.h
// Paul Jones
// This class should contain the attributes of 1 Individual

#include "rando.h"
#include <iostream.h>
#include <math.h>

#ifndef Individual_H
#define Individual_H
#define Pi 3.1415926
#define CHROM_LENGTH 14

class Individual
{
 public:
        //Individual:  constructor, uses random number generator 'rando' to
        //generate each bit of the individual's chromosome.  These are
        //contained in an array called 'chromosome' initialized here to a
        //length of CHROM_LENGTH.  fitness, x_val, and y_val set to 0.0
        Individual();

        //Individual: destructor, deletes chromosome
        ~Individual();

        //Mutate: randomly selects an integer in the range (0, CHROM_LENGTH-1)
        //corresponding bit in the chromosome is flipped
        void Mutate();

        //GetGene: return an individual bit as a charater in the chromosome,
        //with the character '0' added to it.  This is for the purpose of
        //returning a sequence of characters in the display function called
        //GetChrom.
        char GetGene(int n);

        //GetBit: return the integer value of the chromosome at position n.
```

```
        //This function is used for things other than display, such as in the
        //Population class function crossover.
        int GetBit(int n);

        //GetChrom: return a chromosome.  Creates a temporary char* array to
        //write the individual characters to, and return that array.
        char* GetChrom();

        //AssignFitness: FITNESS FUNCTION for the genetic algorithm.  Assign
        //a fitness value to a chromosome based on its x_val (first half of
        //the chromosome), and y_val (second half).  Currently three fitness
        //funtions appear (two should be commented out).
        void AssignFitness();

        //GetFitness: return the fitness value for this individual
        float GetFitness();

        //SetChromVal: allow a bit in the chromosome, parameter i, to be set
        //to a given value, parameter v.
        void SetChromVal(int i, int v);

        //SetXY: calculate the x_val, and y_val for this individual's
        //chromosome.  Each chromosome has an x and y value, first and second
        //half.  Each value is set according to the binary representation of it
        //places are equal to .5, and 1/2 of the previous place value from
        //there.
        void SetXY();

        //GetX: return the private data member x_val
        float GetX();

        //GetY: return the private data member y_val
        float GetY();

        //operator=: copy the data members of one Individual into another
        //Individual object using the = assignment.
        Individual& operator= (const Individual & a);

  private:
        //chromosome data member: length set to CHROM_LENGTH upon object
        //instantiation.  Contains the bits that represent solutions to the
        //function is question.  Constructor randomly chooses bits.
        int * chromosome;

        //private data members containing the state of an individual
        const int length;
        float fitness;
        float x_val;
        float y_val;

        //RandGen object: used in Mutate and constructor
        RandGen rnd;
};

ostream& operator << (ostream& out, Individual as);
#endif
```

```
// Individual.C
// Paul Jones
```

```
#include "Individual.h"
#include <math.h>

Individual :: Individual () : length (CHROM_LENGTH) {
        fitness = 0.0;
        x_val, y_val = 0.0;
        chromosome = new int[CHROM_LENGTH];
        for (int i=0; i < CHROM_LENGTH; i++) {
                chromosome[i] = rnd.RandInt(0,1);  //random()&01;
        }
        SetXY();
        AssignFitness();
}

Individual :: ~Individual(){
        delete [] chromosome;
}

void Individual :: Mutate () {
        int i = rnd.RandInt(0,(CHROM_LENGTH-1) );  //random()&0length;

        //this cout statement is only for reasons of checking the result,
        //it will be used only if this individual is chosen for mutation,
        //can be commented, un-commented as desired.  The Pop_Mutate function
        //will indicate which numbered individual it is.
        //cout <<" Randomly chosen bit to mutate is " << i+1 << endl;

        if (chromosome[i] == 0) {
                chromosome[i] = 1;
        }
        else {//(chromosome[i] == 1) {
                chromosome[i] = 0;
        }
}


char Individual :: GetGene(int n) {
        return (chromosome[n] + '0');
}

int Individual :: GetBit(int n) {
        return (chromosome[n]);
}

char* Individual :: GetChrom() {
        char* temp = new char[CHROM_LENGTH];
        for (int i = 0; i < CHROM_LENGTH; i++) {
                temp[i] = GetGene(i);
        }
        return (temp);
}

void Individual :: AssignFitness() {
        fitness = 0.0;

        // function to minimize is: f(x,y)= 100(y-x^2)^2 +(1-x)^2
        //fitness = 100*((y_val-x_val*x_val)*(y_val-x_val*x_val))
        //+(1-x_val)*(1-x_val);

        //function to maximize is:      (max appears at 0.5, 0.5)
        fitness = sin(Pi*x_val)*(1-fabs(sin(6*Pi*x_val)))*sin(Pi*y_val)
```

```
            *(1-fabs(sin(6*Pi*y_val)));

            //function to maximize is:      (max appears at 0.5, 0.5)
            //fitness = sin(Pi*x_val)*fabs(sin(5*Pi*x_val))*sin(Pi*y_val)
            //*fabs(sin(5*Pi*y_val));
}

float Individual :: GetFitness() {
            return (fitness);
}

void Individual :: SetChromVal(int i, int v) {
            chromosome[i] = v;
}

void Individual :: SetXY() {
            x_val = 0.0;
            y_val = 0.0;
            float power = 0.25;
            float power2 = 0.25;
            int middle = (CHROM_LENGTH/2) - 1;

            for (int l = 0; l <= middle; l++) {
                        x_val = (chromosome[l]*(2*power) ) + x_val;
                        power = power/2;
            }
            for (int r = middle + 1; r < CHROM_LENGTH; r++) {
                        y_val = (chromosome[r]*(2*power2)) + y_val; //(2^-(r-middle))
                        power2 = power2/2;
            }
}

float Individual :: GetX() {
            return (x_val);
}

float Individual :: GetY() {
            return (y_val);
}

Individual & Individual :: operator = (const Individual & a) {
            x_val = a.x_val;
            y_val = a.y_val;
            fitness= a.fitness;

            for (int i = 0; i < CHROM_LENGTH; i++) {
                        chromosome[i] = a.chromosome[i];
            }
            return *this; //Individual;
}

ostream& operator << (ostream& out, Individual as) {
            for (int i = 0; i < CHROM_LENGTH; i++) {
                        out << as.GetGene(i);
            }
            return out;
}
```

6.2.2 Population Class

```
// Population.h
// Paul Jones
// This class should instantiate Chromosome objects to create the population
// It will keep track of the total fitness of the population per generation

#ifndef Population_H
#define Population_H
#define POP_SIZE 50
//#define MUTATION_RATE 0.2
//#define CROSSOVER_RATE 0.1

#include "rando.h"
#include "Individual.h"
#include <iostream.h>
#include <fstream.h>
#include <math.h>


class Population
{
 public:
        //Population constructor:  Initializes the arrays population, and
        //population2 to POP_SIZE.  Each array contains a set of 'Individual'
        //objects.  By initializing the array to POP_SIZE, POP_SIZE objects of
        //type Individual are instantiated.  Set generation = 0,
        //total_fitness and average_fitness = 0.0
        Population();

        //Destructor, does nothing
        ~Population();

        //SetAverageFitness:  sets private data member 'average_fitness' =
        //total_fitness/POP_SIZE
        void SetAverageFitness();

        //GetAverageFitness:  return the float value average_fitness
        float GetAverageFitness();

        //GetTotalFitness:  return total_fitness
        float GetTotalFitness();

        //Select:  generates two random numbers (c1,c2) representing two
        //chromosomes.  c1 and c2 are between 0 and POP_SIZE-1
        //Determines which generation to look in. If the current generation
        //is odd (generation mod 2 == 1) look in 'population' else current
        //generation is even, look in 'population2'.  Select the greater (or
        //lesser) fitness value of c1 and c2, depending on whether trying to
        //maximize or minimize.
        void Select();

        //Pop_Mutate:  different than Mutate as defined in the Individual class
        //This mutate function is applied to the entire population.  For each
        //member of the population, a random number between 0 and .999 is
        //generated.  If that number is less than MUTATION_RATE, the Individual
        //class member function Mutate is called.  The generation number
        //determines which population the function will be applied to.  If
        //the generation is even, the next generation has been selected
        //already, and mutation must occur in 'population' (if current is odd,
        //mutation occurs in 'population2')
        void Pop_Mutate();
```

```
        //Pop_Crossover:  works in the same way as mutation by operating on an
        //entire population, generating a random number at each between 0 and
        //.999 to determine if that individual should be crossed with another.
        //If is to be crossed, another individual is randomly selected.  It is
        //also assumed that crossover will occur after selection, after the
        //next generation has been selected.
        void Pop_Crossover();

        //SetVals: sets the x, y and the fitness value for each member
        //of the population.  This function is also assumes that the current
        //generation is in the process of creating the next generation, so the
        //values are set in for individuals in the next generation.
        void SetVals();

        //SetTotalFitness: assigns the sum of all fitness values in a
        //population to total-fitness
        void SetTotalFitness();

        void SetBestWorst();

        //GetChromosome: return an individual chromosome.
        char* GetChromosome(int ind);

        //Pop_GetX, Pop_GetY, Pop_GetFitness: return the desired value for
        //a particular chromosome.  The prefix Pop_ is added because these
        //functions call the corresponding Individual class functions
        float Pop_GetX(int chrom);
        float Pop_GetY(int chr);
        float Pop_GetFitness(int chrom);

        //IncGen:  Increment the private data member generation.
        //!!!!IMPORTANT THAT THIS FUNCTION IS CALLED IN THE CORRECT PLACE
        //because all other functions depend on the generation number being
        //correct!!!  In the test program a new generation is created by
        //selection, mutation and crossover in that order, and IncGen is called
        //after the next generation has been completely created.  The way the
        //program is set up, selection must occur first, and mutation and
        //crossover can happen in either order, but the next generation must
        //be created.
        void IncGen();

        //SetCrossRate, SetMutationRate: mutation and crossover rates.
        //These values must be between 0 and 1.
        void SetCrossRate(double rate);
        void SetMutationRate(double rate);

        //PrintPairs:  print the contents of a generation (each x,y pair) to
        //a file...still working on a better way
        void PrintAvgFitness();
        void PrintBestWorst();
        void PrintPairs();
        void IncFileCounter();

private:
        double MUTATION_RATE;
        double CROSSOVER_RATE;

        char gen[10];                   //used for naming data files

        RandGen rand;                   //Object of the RandGen class
        int generation;                 //number of the current generation
```

```
        float total_fitness;        //total fitness of all chromosomes in a gen.
        float average_fitness;      //the total fitness divided by POP_SIZE

        //Each of the following arrays will contain Individuals.  The size of
        //both is set in the constructor to POP_SIZE.  'population' is used for
        //odd numbered generations, 'population2' is used for even numbered
        //generations.  The selection function will choose from the current
        //generation, in population or population2, and write to the other.
        //Mutation and crossover must then look ahead one generation to
        //operate correctly, this is also explained in comments before the
        //Mutate and Crossover functions.

        Individual * population;    //array of individuals
        Individual * population2;   //second array used for even # gens.
        float * highest_individual;
        float * lowest_individual;

        ofstream fit_file;
        ofstream best_worst;
};
//ostream& operator = (ostream& out, Population a);
#endif
```

```
// Population.C
// Paul Jones

#include "Population.h"

Population :: Population (){
        population = new Individual[POP_SIZE];
        population2 = new Individual[POP_SIZE];
        highest_individual = new float[POP_SIZE];
        lowest_individual = new float[POP_SIZE];
        generation = 0;
        total_fitness = 0.0;
        average_fitness = 0.0;

        //Array used for naming files genXXX
        gen[0] = 'g';
        gen[1] = 'e';
        gen[2] = 'n';
        gen[3] = '0';
        gen[4] = '0';
        gen[5] = '0';
        gen[6] = '\0';

        fit_file.open("avg_fit");
        best_worst.open("bestworst");
}

Population :: ~Population(){
        fit_file.close();
}

void Population :: SetAverageFitness() {
        average_fitness = total_fitness/POP_SIZE;
}
```

```
float Population :: GetAverageFitness() {
        return (average_fitness);
}

float Population :: GetTotalFitness() {
        return (total_fitness);
}

void Population :: Select() {

for (int i= 0; i < POP_SIZE; i++) {

        int c1 = rand.RandInt(0, POP_SIZE - 1);
        int c2 = rand.RandInt(0, POP_SIZE - 1);

        //The cout statement below can be used to see which two individuals will fight
        //to move on to the next generation
        //cout<<i+1<<": chosen pairs to fight are: "<<c1+1<<", "<<c2+1<<endl;

        if (generation % 2 == 1) {
                if (population[c1].GetFitness() > population[c2].GetFitness() ) {
                        population2[i] = population[c1];
                }
                else {//population[c1].GetFitness()<population[c2].GetFitness()
                                population2[i] = population[c2];
                }

        }
        else { //if generation == 0mod2
                if (population2[c1].GetFitness() > population2[c2].GetFitness() ) {
                        population[i] = population2[c1];
                }
                else {//population2[c1].GetFitness()<population2[c2].GetFitness()
                        population[i] = population2[c2];
                }
        }
}
}

// The following functions operate on the opposite generation as the current
// Mutate: assumes that mutation will come after selection, therefore the
// next generation.
// Crossover: also assumes that crossover will come after the next generation
// has been selected
// SetVals, SetTotalFitness, GetChromosome, GetX, GetY, and GetTotalFitness:
// assume that the next generation's bits have been set, and they must perform
// their operations on those bits, which are in the next generation
// These functions assume a generation will be completed as follows:
// Select, Pop_Mutate, Pop_Crossover, (PrintPairs), IncGen
// The location of the function IncGen is most important because all these
// functions depend on it.

void Population :: Pop_Mutate() {

        for (int i = 0; i < POP_SIZE; i++) {
                //generate random real between 0,1
                double m1 = 0.0;
                int div1 = 1000;
                int rnd1 = rand.RandInt(0,1000);
                m1 = rnd1 % div1;
                m1 = m1/div1;
```

```
                //this cout statement allows user to see what each random real
                //is, to verify whether it should be mutated or not.
                //cout <<"m1 is "<<m1<<endl;

                //The cout statements below indicate the number of the
                //individual that will be mutated.
                if (generation % 2 == 1) {
                        if (MUTATION_RATE > m1) {
                                //cout <<"mutating chrom "<<i+1;
                                population2[i].Mutate();
                                SetVals();
                        }
                }
                else { //generation mod 2 == 0
                        if (MUTATION_RATE > m1) {
                                //cout <<"mutating chrom "<<i+1;
                                population[i].Mutate();
                                SetVals();
                        }
                }
        }
}

void Population :: Pop_Crossover() {
int j, k = 0;
for (int i = 0; i < POP_SIZE; i++) {
        //generate random real between (0,1)
        double cross = 0.0;
        int div = 1000;
        int rnd = rand.RandInt(0,1000);
        cross = rnd % div;
        cross = cross/div;

        int temp_gene = 0;
        int mate1 = rand.RandInt(0, POP_SIZE - 1);
        int mate2 = rand.RandInt(0, POP_SIZE - 1);
        int cross_point = rand.RandInt(0,(CHROM_LENGTH-1));

        //The cout statements below can be used to see which two individuals will cross
        if (CROSSOVER_RATE > cross) {
                if (generation % 2 == 1) {
                        //cout << mate1 + 1<<" cross with "<< mate2 +1 <<" at "<<cross_point<< endl;

                        for (j = 0; j < cross_point; j++) {
                                temp_gene = population2[mate2].GetBit(j);
                                population2[mate2].SetChromVal(j, population2[mate1].GetBit(j));
                                population2[mate1].SetChromVal(j, temp_gene);
                                SetVals();
                        }
                }
                else { // generation mod 2 == 0
                        //cout << mate1 + 1<<" cross with "<< mate2 + 1 <<" at "<<cross_point <<endl;

                        for (k = 0; k < cross_point; k++) {
                                temp_gene = population[mate2].GetBit(k);
                                population[mate2].SetChromVal(k, population[mate1].GetBit(k));
                                population[mate1].SetChromVal(k, temp_gene);
                                SetVals();
                        }
                }
```

```
        }
}
}

void Population :: SetVals() {

        for (int i= 0; i < POP_SIZE; i++) {
                if (generation % 2 == 1) {
                        population2[i].SetXY();
                        population2[i].AssignFitness();
                }
                else {//generation%2 == 0
                        population[i].SetXY();
                        population[i].AssignFitness();
                }
        }
}

void Population :: SetBestWorst() {
        //Set the highest and lowest individual per generation
        if (generation % 2 == 1) {
          highest_individual[generation] = population2[0].GetFitness();
          lowest_individual[generation] = population2[0].GetFitness();

          for (int i = 0; i < POP_SIZE; i++) {
                if (population2[i].GetFitness() > highest_individual[generation]) {
                        highest_individual[generation] = population2[i].GetFitness();
                }
                if (population2[i].GetFitness() < lowest_individual[generation]) {
                        lowest_individual[generation] = population2[i].GetFitness();
                }
           }
        }
        else {  //generation % 2 == 0
         highest_individual[generation] = population[0].GetFitness();
         lowest_individual[generation] = population[0].GetFitness();

         for (int i = 0; i < POP_SIZE; i++) {
                if (population[i].GetFitness() > highest_individual[generation]) {
                        highest_individual[generation] = population[i].GetFitness();
                }
                if (population[i].GetFitness() < lowest_individual[generation]) {
                        lowest_individual[generation] = population[i].GetFitness();
                }
           }
        }

        //The following cout statement can be commented out if desired
        cout <<"Highest: "<< highest_individual[generation]<< endl;
        cout <<"Lowest: "<< lowest_individual[generation]<<endl;
}

void Population :: SetTotalFitness() {

        total_fitness = 0.0;

        if (generation % 2 == 1) {
                for (int i = 0; i < POP_SIZE; i++) {
                total_fitness = population2[i].GetFitness() + total_fitness;
                }
```

```
        }
        else //if (generation % 2 == 0) {
                for (int i = 0; i < POP_SIZE; i++) {
                total_fitness = population[i].GetFitness() + total_fitness;
                }

}

char* Population :: GetChromosome(int ind) {
        char* temp = new char[CHROM_LENGTH];

        if (generation % 2 == 1) {
                for (int j = 0; j < CHROM_LENGTH; j++) {
                        temp[j] = population2[ind].GetGene(j);
                }
                return (temp);
        }
        else { //generation == 0mod2
                for (int j = 0; j < CHROM_LENGTH; j++) {
                        temp[j] = population[ind].GetGene(j);
                }
                return (temp);
        }
}

float Population :: Pop_GetX(int chrom) {
        if (generation % 2 == 1) {
                return (population2[chrom].GetX());
        }
        else { //generation == 2mod2
                return (population[chrom].GetX());
        }
}

float Population :: Pop_GetY(int chr) {
        if (generation % 2 == 1) {
                return (population2[chr].GetY());
        }
        else {//generation == 0mod2
                return ( population[chr].GetY() );
        }
}

float Population :: Pop_GetFitness(int chrom) {
        if (generation % 2 == 1) {
                return (population2[chrom].GetFitness());
        }
        else {//generation == 0mod2
                return (population[chrom].GetFitness());
        }
}

void Population :: IncGen() {
        generation++;
}

void Population :: SetCrossRate(double rate) {
        CROSSOVER_RATE = rate;
}

void Population :: SetMutationRate(double rate) {
```

```
        MUTATION_RATE = rate;
}

void Population :: PrintAvgFitness() {
        fit_file <<generation<<" "<< GetAverageFitness() << endl;
}

void Population :: PrintBestWorst() {
        best_worst<<generation<<" "<<highest_individual[generation]<<" "<<lowest_individual[generation]<<endl;
}

void Population :: PrintPairs() {
 ofstream gen_file (gen);

 if (generation % 2 == 0) { //print from 'population'
        for (int i =0; i < POP_SIZE; i++) {
                gen_file << population[i].GetX() <<" "<< population[i].GetY() << endl;
        }
 }

 else { // (generation % 2 == 1) print from 'population2'
        for (int i =0; i < POP_SIZE; i++) {
                gen_file << population2[i].GetX() <<" "<< population2[i].GetY() << endl;
        }
 }
 gen_file.close();
 IncFileCounter();
}

void Population :: IncFileCounter() {
        if (gen[5] < '9') {
                (gen[5])++;
        }
        else {
                gen[5] = '0';
        }

        if (gen[5] == '0') {
                if (gen[4] < '9') {
                        (gen[4])++;
                }
                else { //gen[4] == '9'
                        gen[4] = '0';
                }
        }
}
```

6.2.3 Test Class

```
// Paul Jones
// File: GAtester.h.

#ifndef GA_TESTER_H
#define GA_TESTER_H

#include "Population.h"

class GAtester

{
```

```
public :

  //GAtester constructor: display initial population created by instatiation
  //of Population object.  The Population class IncGen function is also
  //called
  GAtester ();

  //GAtester destructor: does nothing
  ~GAtester ();

  //RunTester: only function called in main.  Get crossover and mutation
  //rates from user.  Get number of generations to run from the user.  for
  //each generation, execute the Population class functions Select,
  //Pop_Mutate, Pop_Crossover, SetVals, SetTotalFitness, SetAverageFitness.
  //(Optional: DoReturnPop, DoReturnTotFitness, DoReturnAvgFitness,
  //PrintPairs.  can be used to examine the state of each generation).
  //NOT OPTIONAL: IncGen.
  void RunTester ();

 private :

  //DoReturnPop: for the POP_SIZE, return the chromosome number, actual
  //chromosome (bits), fitness value, x value, and y value.
  void DoReturnPop();

  //DoReturnTotalFitness: return the total fitness for population in current
  //generation.
  void DoReturnTotFitness();

  //DoReturnAvgFitness: return the average fitness for population in current
  //generation.
  void DoReturnAvgFitness();

  //Instantiate Population object
  Population initial;
};

#endif
```

```
// gatester.C
// Paul Jones

#include "GAtester.h"
#include "Population.h"
#include <iostream.h>

GAtester :: GAtester() {
        cout << endl;
        cout << "Initializing population" << endl;
        cout <<"initial population (generation 0) is: " << endl;
        initial.SetVals();
        initial.SetTotalFitness();
        initial.SetAverageFitness();
        DoReturnPop();
        DoReturnTotFitness();
        DoReturnAvgFitness();
        initial.SetBestWorst();

        initial.PrintPairs();
```

```
        initial.PrintAvgFitness();
        initial.PrintBestWorst();
        initial.IncGen();
}

GAtester :: ~GAtester () {
}

void GAtester :: RunTester () {
        //loop to return each generation
        //also to select, mutate

        double cross_rate = 0.0;
        double mutation_rate = 0.0;
        cout <<"what do you want the crossover rate to be? " << endl;
        cin >> cross_rate;
        cout <<"What do you want the mutaion rate to be? " << endl;
        cin >> mutation_rate;

        initial.SetCrossRate(cross_rate);
        initial.SetMutationRate(mutation_rate);

        int no_of_generations=0;      //number of generations
        cout <<"How many generations do you want to run?" << endl;
        cin >> no_of_generations;

        for (int gen_number = 1; gen_number < no_of_generations; gen_number++) {
                initial.Select();
                initial.Pop_Mutate();
                initial.Pop_Crossover();
                initial.SetVals();
                initial.SetTotalFitness();
                initial.SetAverageFitness();
                cout << "after generation " <<gen_number<<endl;

                DoReturnPop();
                DoReturnTotFitness();
                DoReturnAvgFitness();
                initial.SetBestWorst();

                initial.PrintPairs();
                initial.PrintAvgFitness();
                initial.PrintBestWorst();
                initial.IncGen();
        }
}

void GAtester :: DoReturnPop() {
        for (int k = 0; k < POP_SIZE; k++) {
                cout <<"Chromosome " <<k+1<<": " << initial.GetChromosome(k)
                <<"  fitness: " << initial.Pop_GetFitness(k)
                <<" X: "<<initial.Pop_GetX(k)<<" Y: "<<initial.Pop_GetY(k)<<endl;
        }
}

void GAtester :: DoReturnTotFitness() {
        cout <<"Total fitness is: "<< initial.GetTotalFitness() << endl;
}

void GAtester :: DoReturnAvgFitness() {
        cout <<"Average fitness is: "<<initial.GetAverageFitness() << endl;
```

}

6.2.4 Makefile

```
all : gatest GAcommand

gatest : gatest.o GAtester.o Population.o Individual.o rando.o
        CC -g -o gatest gatest.o GAtester.o Population.o Individual.o rando.o -lm

gatest.o : gatest.C GAtester.h Population.h
        CC -g -c gatest.C

GAtester.o : GAtester.C GAtester.h Population.h
        CC -g -c GAtester.C

Population.o : Population.C Population.h Individual.h
        CC -g -c Population.C

Individual.o : Individual.C Individual.h rando.h
        CC -g -c Individual.C

rando.o : rando.cc rando.h
        CC -g -c rando.cc


GAcommand : GAcommand.o GAcommandInt.o Population.o Individual.o rando.o
        CC -g -o GAcommand GAcommand.o GAcommandInt.o Population.o Individual.o rando.o -lm

GAcommand.o : GAcommand.C GAcommandInt.h Population.h
        CC -g -c GAcommand.C

GAcommanderInt.o : GAcommandInt.C GAcommandInt.h Population.h
        CC -g -c GAcommandInt.C
```

6.2.5 Gnuplot Script

To run the following script the gnuplot program must be running. Gnuplot contains a number of different terminal settings. On the Saint John's University SGI machines, the terminal type is iris4d, and the gnuplot command is set term iris4d. A .gnuplot file can be added to the user's home directory. That file will contain any number of gnuplot commands to be executed when the program begins. The only command I have in the file is set term iris4d, but it is possible to include the command load 'genplot' which would execute the following script. I have not included it because the initial plot

must be manually resized on the screen, and including that command would execute the script without re-sizing the plot window.

The following gnuplot script plots the data contained in files genXXX, where XXX is the generation number and the filename is generated by the program. Each of these files contain the x and y values each chromosome in a generation. Since it is possible to generate file gen000 through gen999, the script must be modified to plot the desired generations. Currently the script plots the first 50 generations. Only the first few lines of the script appear below.

```
plot [0:1] [0:1] 'gen000'
pause 2
plot [0:1] [0:1] 'gen001'
pause 2
plot [0:1] [0:1] 'gen002'
pause 2
plot [0:1] [0:1] 'gen003'
pause 2
```

Including [0:1] for both x and y endpoints keeps those values from shifting.

works cited

Goldberg, David. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Reading, Massachusetts: Addison-Wesley,1989.

Michalewiczs, Zbigniew. *Genetic Algorithms + Data Structures = Evolution Programs.* New York, New York: Springer, 1996.

Mitchell, Melanie. *An Introduction to Genetic Algorithms.* Cambridge, Massachusetts: The MIT Press, 1996.

Rawlins, Gregory. *Foundations of Genetic Algorithms.* San Mateo, California: M. Kaufmann, 1991.

Ribeiro, Filho, and Treleaven. "Genetic-Algorithm Programming Environments" Computer. June 1994: 28-43.

Srinivas, M. "Genetic Algorithms: A Survey" Computer. June 1994: 17-26.