Communications Research Centre Canada
An Agency of Industry Canada

Centre de recherches sur les communications Canada
Un organisme d'Industrie Canada

Genetic Algorithms

by Geoff Hayes
RAAT Advanced Antenna Technology

CRC Report No. CRC-RP-2000-12

Genetic Algorithms

Author: Geoff Hayes

Supervisor: Dr. Jafar Shaker

Communications Research Centre Canada

RAAT Advanced Antenna Technology

TABLE OF CONTENTS

1.0 INTRODUCTION

This summer I continued working with the Genetic Algorithm (GA) and the original optimisation package (created last year) in hopes of improving its performance. This binary chromosome GA had proven to be successful, but there still existed several limitations. Convergence reliability, the ability of the genetic algorithm to locate the optimal (or near optimal) solution over many runs was not guaranteed; the GA could become stuck at sub-optimal peaks in the fitness landscape. Convergence time and essentially the GA's "speed" in locating that optimal solution depended greatly upon the initial population. If "good" starting points existed then the solution could be found with relative ease. If not, then the time (the number of iterations) required in locating the solution would increase, and the GA ran the risk of converging prematurely into a local minimum. So the question that should be asked is "what does constitute a good starting point, and how can these points be used to ensure the success of the GA.? "Unfortunately, there is no clear-cut answer to this question. For many of the problems that we wish to solve, we do not know where to begin or even what the final outcome will be. Nevertheless, GA relies on an initial randomness and hopes (!) that through its evolutionary operators it can determine a solution that is both feasible and optimal. Obviously, something more is needed and new directions must be explored - a *hunt* for an ideal optimization package in order to ensure that the genetic algorithm satisfies the convergence to the global minimum with acceptable and practical rate.

Various ideas were considered and implemented. These include different selection schemes, introducing multiple populations into the GA, adjustments to the crossover operator, and the introduction of "parasites" into the population. A hybrid genetic algorithm was constructed that used a second population of "rules" in its search for the optimal solution. This led to the combination of a genetic algorithm and another evolutionary techniques *i.e.* Evolutionary Programming (EP). The latter two implementations were encouraging, but *something* was still lacking. Since EP uses real parameters as its chromosomes, a real parameter GA was then considered and tested against well-known functions. It was this last method that proved to be far more successful than all previously tested methods.

2.0 THE HUNT BEGINS...

2.1 The Selection Operator

The selection operator that was used last year (and is still being maintained as the operator of choice), is linear ranking in nature. Organisms are ranked according to their fitness, and parents are then selected using the stochastic universal selection method. The user controls the selection pressure by entering a number between one and two, through the interface. The larger the value, the higher the selection pressure which in turn means that those organisms with a higher fitness will be selected more so than the rest. A lower selection pressure ensures that those organisms with a lower fitness will have an opportunity to be selected.

An alternative is the *n*-tournament selection [3]. From the population of organisms, a sub-population of size *n* is randomly chosen. Then a tournament or competition among these organisms is held, the winner of which being the one that has the highest fitness; this "champion" becomes a parent. The organisms are then placed back into the population, and another random sample of size *n* is found. This process is repeated until the parent population has been filled. It is clear that as n increases, the probability of obtaining a sub-population with the more fit individuals increases. And as *n* decreases, this probability decreases. So the value assigned to n controls the "selection pressure" of the population, in much the same way as we assign the selection pressure to be some value between 1.0 and 2.0 in the linear ranking method.

Comparisons were made as to which method yielded better results, and unfortunately no definite conclusion could be drawn. In many cases, both methods converged to similar solutions. The binary-tournament (n=2) appeared to outperform linear ranking for smaller population sizes. This was probably due to the fact that organisms of higher fitness were always sampled more frequently in a smaller population as compared to larger populations. However, binary tournament performance will become inferior to linear ranking as the replacement size of the population (at each generation) became too small. This was probably due to the fact that with a slowly changing (evolving) population, the same parents were chosen more frequently, hence no new information was introduced (via the crossover operator) into the population. Since no clear

distinction could be made between the two methods, *n*-toumament method was discarded in favour of the linear ranking method. This method has demonstrated satisfactory results thus far.

2.2 Multi-Population Genetic Algorithms

Parallel Genetic Algorithm (PGA) is another method that is more robust compared to single population GA. In this algorithm, many populations are evolved in parallel, through the use of parallel processors. These populations can exchange and share highly-fit solutions amongst themselves– along the same line of sharing resources and opportunities- through migration of highly fit organisms. This concept can be easily implemented; GA is executed on a multiplicity of machines, and throughout the execution, best organisms are sent periodically into a pool that could be exploited by the other population. Therefore, underprivileged populations with low fitness indices can improve their status and use the gains and "achievements" of other populations. The implementation of this method can create difficulties or "tensions" between the populations such as fast evolution of one population compared to the others, or two populations attempt to access the pool of fit organisms at the same time. However, the whole concept is sound; through evolving individual populations which share resources amongst themselves, premature convergence of populations can be controlled (with the exception of highly improbable situation of convergence of different populations to the same organism). New and highly fit members introduced into populations by the sharing mechanism, can be the "building blocks" necessary in the successful evolution of the optimal solution.

It should be noted that PGA is more CPU intensive compared to single population GA. The higher the number of evolving populations, the longer is the execution time of the algorithm. This is especially true for the computationally complicated objective functions. To avoid such complications, the program can be executed on parallel processors. However, the algorithm was executed on a single processor to investigated whether n populations of size N/n that benefit from the migration of fittest members will outperform a single population of size N.

Going one step further, each population can evolve under different conditions such as, variety of selection pressures, crossover, and mutation probabilities. Thus, one population could remain more diverse with respect to its organisms and conduct a more exploratory search of the solution space, while another population may exploit its organisms, converging to an optimal solution at a quicker rate. An environment was created to "hold" these populations, and the GA was used to during this process.

However, the following question is yet to be answered, "how does migration work? The initial migration technique, or circular-migration, would be as follows: at every generation the best organism in population *i* migrates to population *i+1*, and replaces the best organism in that population only if its fitness were higher than that of the best organism, otherwise it replaces the worst organism in that population. Thus, each population would be guaranteed the introduction of a highly fit organism at each generation. However, if a population changes minimally during the course of execution, it sends almost the same organism to its adjacent population whenever the migration mechanism is activated. This floods the adjacent *i+1* population with almost the same organism which might cause convergence to one and the same organism. Therefore, a domino effect can occur in the sense of convergence to the same organism in all populations. This scenario should be avoided for the success of PGA.

It seems that activation of migration mechanism in every generation is unrealistic. A migration rate would be necessary to control sharing of the "fittest organisms" amongst the populations. This entails the determination of an "optimal" rate, or generation *intervals* at which migration is allowed. Various rates such as 5,9,12,13 and 18 were used as the migration intervals during the course of this project yielding successful results for some runs, and poor results for others. In other words, the search for optimal rate was not conclusive.

"Mass migration" was the next technique to be considered. In this method migration is exercised only from the population with higher fitness, with respect to the best organism in that population. We select an organism randomly from the less fit population and find four other

organisms with highest resemblance to it. These five organisms would be replaced by the best organism from the population with higher fitness and four other organisms that are dissimilar to it in order to maintain diversity in the population with lower fitness. But again, by sending over the best organism, we are at the risk of causing one of other populations to converge prematurely. What kind of information do we have about the organisms that are being migrated to the lower fit population? It is true that their dissimilarity maintain variety, but they will not be selected as parents if they are poorly fit. Therefore, the host population will not benefit from their presence. Furthermore, this method relies on *a priori* knowledge of migration rate which is a problem dependant parameter. This raises serious questions about the effectiveness of this method. A cooperative approach between the populations seems to be a viable alternative. With this in mind, two questions arose:

1. When should a population receive cooperation? (or ask for help), and
2. How much help should a population receive?

2.3 Cooperating Populations

Let us consider an environment composed of two populations only for which the objective function is to be minimized. The first step is to determine which population is the better of the two - with respect to the fitness of the best organism in each population. Using these two parameters, the fitness of the best member of the better fit population is divided by the fitness of the best member of lower fit population. The result of this division *(x)* is an indication of the gap between the two populations. The closer it is to one, the smaller is the distance between the best members of the two populations. And the closer it is to zero, the greater is this distance. The latter case demonstrates the need of the lower fit population for help in order to improve its fitness.

Help is provided on a random basis. A random number is generated between zero and one, its value is compared to x, and help will be offered if it is larger than x. Clearly, the probability of provision of help is higher for smaller values of x. It should be noted that in this scheme the populations are not competing, so they will converge preferably to the "same peak" at the end of

the run. If x is close but not equal to one, then it is quite unlikely for the population with lower fitness to receive any help. A record is kept of the cooperation among populations. The longer they go without cooperation, the greater the probability that cooperation will be activated between the populations. Cooperation will not be given to the same population for two successive generations if it manages to acquire a better organism than the one it had in the previous generation. If no improvement has been made, then the population will be offered help. Thus each generation will keep a "log" or "memory" of its past best solutions. This answers the question about when a population should receive help.

Secondly, considering the population with a higher fitness it will be decided upon the criteria to migrate organisms to the population with lower fitness. The organisms are ranked in the order from the least similar to the most similar to the best organism of better fit population, with respect to the binary chromosomes. The value of x determines the organisms that will be migrated to the population with lower fitness figure. Having an x close to one indicates that there is little difference between the "champions" of the two populations, so the least similar members (compared to the best organism of the better fit population) will be migrated. As x tends to zero, there exists a greater distance between the two populations, and so organisms that are more similar to the best organism of the better fit population will be transferred over. We never transfer over the best organism. Now, what happens the transferred organism is sub-optimal?

Thirdly, it should be decided upon the number of organisms that are migrated *i.e.* the transfer rate. Since time is not a luxury and there is a "deadline" that should be met by most populations in creating highly fit organisms. Also, populations are cooperating and it is desired that they converge to the same peak, the transfer rate must increase as the population ages.

Finally, the survival of organisms in the population with lower fitness is decided by the degree of their similarity to the best organism of the same population. Those that resemble the best organism the most will be replaced with organisms from the other population. We keep the best, just in case it has some "good" building blocks.

However, one problem may arise; both populations may converge to the same sub-optimal peak. To avoid such an event, new organisms could be introduced into the population to replace some of the older organisms if no change occurs after a specified number of generations. It is very much like the game of Scrabble; you have your seven letters but can do nothing with them, so you replace some with new ones. But this might present a further problem that is the new organisms are created randomly but are not guaranteed to be highly fit members. Also, how often should new members be introduced into the population? There are no easy answers to these questions. Therefore, the idea of introducing new members into the population was abandoned.

It must be noted that if three populations are constructed, cooperation occurs only between the best and the worst. The "one in the middle" would be left alone until it needs help or is good enough to offer help. Tests were conducted on a function of two real parameters. A good convergence was observed at the beginning. Out of ten runs (of ten different initial populations) the multi-population GA performed better than the single population. But then 100 runs were considered, and the results were not as favorable. In fact, it was observed that the multi-population GA outperforms single population GA in only 50 of the 100 runs which is not an impressive standing for the amount of extra computational effort. This can be attributed to the fact that the attempt to find a general cooperation scheme was not so successful. Finding a clear cut definition for the amount of sharing and the similarity between organisms was not an easy task. Ranking organisms in terms of fitness (rather than similarity) increases the risk of premature convergence. Taking a break from real parameter functions, the travelling salesman problem (TSP) was considered.

2.4 The Travelling Salesman Problem

The Taveling Salesman Problem (TSP) is as follows: finding the shortest path through n cities that are to be visited only once by a traveling salesman on his was back to the city that he started his journey. This problem can be used as a benchmark in the assessment of optimization algorithms. Unfortunately, results were not supportive of the implementation multi-population

GA. It seemed to be necessary for the populations to have at least one organism for every city in the tour, hence splitting a population in half or into thirds was not helpful.

Better results for TSP were obtained using an adjacency representation for the chromosomes and using a heuristic/greedy crossover [2]. An adjacency representation (for five cities) is as follows:

Chromosome: 5 1 2 3 4.

The i^{th} position of the chromosome with value j, denotes a path from city i to city j. Thus the tour for the above representation is (assuming a start at city 1):

1-5-4-3-2-1.

The crossover worked as follows. A starting city i is randomly chosen. Then the i-j path in both parents is considered. The shorter of the two paths is given to the child. This process continues, always taking the shorter of the two paths, ensuring that no cycle exists. This method proved to be successful. If the tour consisted of n cities, then a population of 2n organisms almost always found the optimal route.

2.5 Crossover Modifications

During the numerical trials, some modifications were made to the crossover operator:

1. Elimination of useless crossover points. Suppose the parents are:

000000010101001, and

000001001010101.

Thus selection of any of the first six crossover points does not yield any new organism. It is noted that the choice of crossover point "beyond" the seventh bit results in organisms different from parents

2. Mates are selected based upon Hamming Distance: one parent is randomly chosen, and its mate is selected to be the one that is most different (opposites attract). This promotes diversity in the creation of children, so as to explore a greater portion of the search.

3. Once the population of children has been created, the children that outlive the older organisms must be selected. We randomly choose one child to add to the population. The next child is added to the population only if it is different from the latest child that was added to the population (the degree of difference can be quantified using the Hamming Distance concept). The third child differs the most from the previous two, etc. This promotes diversity and avoids the creation of similar children in the population.

While these modifications improved the search for the optimal solution, it increased the run time considerably. This was due to the fact that the binary representation of each organism had to be considered and compared against every other organism in the population. Therefore, the minor improvement was overshadowed by the increase in the execution time of the genetic algorithm.

So the search for a more efficient and robust GA continued, leading to the idea of parasites.

3.0 THE HUNT CONTINUES.

3.1 Parasites

The main problem is that the GA might be trapped in sub-optimal solutions. So the question arises as to how divert GA from these traps, *i.e.* force the GA to consider alternative schema (building blocks). We know that as the generations pass, the population is filled up with organisms that are highly fit. Also, these same organisms are converging to the same peak since they have the same building blocks. We know from the Schema Theorem of genetic algorithm [2] that the low-order, short, and above average fitness schema are represented in exponentially increasing numbers in subsequent generations. These are the building blocks that lead to GA solution. These blocks should be avoided if they happen to lead to a sub-optimal solution. A method was devised to this end. Consider a second population that contains parasites rather than organisms. These parasites [61 are those short, low-order schema mentioned earlier. If each member of the initial population contributes to the existence of a y parasite (one for each of the y parameters being optimised), then our parasite population is of size $y*N$, where N is the size of "regular" GA population. The parasites are of the same length as the variables they correspond

to, and each gene of the chromosome belongs to the set (0, 1, *), where "*" is the "don't care" symbol (*i.e.* "*" could be either one or a zero). The bits of the parasite are specified (probabilistically) from the organisms that create it and we ensure that these parasites are of low-order (i.e. very few *1s* and *0s*, and many *s).

The goal of the above is three-fold:

1. To evolve our "regular" genetic algorithm population as it is normally done.
2. To evolve a population of parasites/building blocks that are to be avoided.
3. To evolve a second GA population that will be discouraged from using the building blocks found in the parasite population.

The first goal is straightforward. The logic behind the second goal is to determine the fitness of the parasites. Again by the Schema Theorem, we know that those parasite that encourage the population to converge to a suboptimal solution are "represented in exponentially increasing numbers in subsequent generations". Thus we can assign an integer value to each parasite which is equal to the number of organisms in the first population that this parasite "inhabits". Since we wish to acknowledge or identify those that are represented in increasing quantities, the larger the integer value, the higher the fitness of a parasite. Having assigned fitness values to each parasite, we evolve this population in the normal genetic algorithm fashion. Finally, the third goal must be satisfied. Having identified the parasites -the building blocks- that lead the first population to a sub-optimal solution, we now discourage their use in the third population by decreasing the fitness of those organisms that contain (in their binary representation) these dangerous representations. It has been assumed throughout this section that the first population is converging to a sub-optimal solution. This may not be necessarily true if the population is converging to the optimal solution. If this is the case, we have lost nothing by encouraging the third population to look elsewhere. The algorithm is as follows:

While termination criteria not met do

Evolve first population for one generation.

Assign fitness values to parasite population.

Evolve parasite population for one generation.

Adjust fitness of organisms in third population.

Evolve third population for one generation.

End.

The above procedure can be summarised as follows. By discovering those schema/building blocks that lead one population to a sub-optimal solution, we can *discourage* the use of these schemas in another population. In doing so, we *encourage* this population to explore other directions that (hopefully) lead to the optimal solution.

Testing in many cases did indeed show that the third population was "pushed" away from the peak into which the first population was trapped. Unfortunately, the "push" was not far enough since the population did not converge to some other optimal solution. This was probably due to many factors. The "good" parasites that may have led to the optimal solution were identified as deadly, and so the third population would have no choice but to become stuck at some local peak. Also, not enough parasites were identified. Lastly, after successive generations the population of parasites would no longer contain the short and low-ordered schema as detailed in the Schema Theorem - an unfortunate side effect of crossover and mutation.

Although these results were not encouraging, an idea grew from this study. We began with multi-population GA schemes that led to another type that considered not only our "regular" GA populations, but another type of population as well - the parasites. This led to combining of a GA population with one of the "rules".

3.2 Rules

Suppose that a function is to be optimised by hand. The optimisation starts with a random initial "guess". Then, this guess can be adjusted in a certain fashion which might entail increasing the value of the first parameter, decreasing the value of the second parameter, setting the third

parameter to zero, and leaving the fourth unchanged. The same trend in the variation of parameters can be continued if the initial adjustment happens to move optimisation in the right direction. Now assume that many adjustments can be made to the parameters in different directions and the results of these adjustments can be combined to obtain the best set of adjustments to optimise the function. This set of adjustments, or population of rules, can operate in conjunction with a regular genetic algorithm population-, which is evolved according to the standard GA code. Its algorithm is as follows:

Construct initial population of organisms

Construct initial population of rules

While termination criteria not met

Evolve population of organisms by one generation

Determine best of this population

Apply every rule of rule population to the parameters of this organism

Fitness of organism after rule applied is the fitness assigned to the rule

Evolve population of rules by one generation

If a rule applied to the organism created a better organism, substitute this organism into the organism population

End

Both of the different evolving populations are working towards the common goal of optimising the function at hand, sharing whenever one comes up with a better solution.

A test function of two variables

$$f(x,y) = 100*(x-y)^2 + (1-y)^2 \text{ ([2])}$$

It was very difficult for binary GA to minimise this function. However, the method described above was successfully applied to minimise this function. Numerous runs were conducted with different initial populations and each time the method was capable of converging to optimal or near optimal solutions. In the absence of theoretical evidence on the effectiveness of this method, it was applied to a wide variety of optimisation problems to assess its strengths and limitations.

Keeping the idea of different evolving populations in mind, the concept of Evolutionary Programming was examined.

3.3 Evolutionary Programming

Evolutionary programming (EP) is an evolutionary algorithm which includes traditional GA as its subset [1]. Unlike traditional GA, EP uses real-valued chromosomes (no binary encoding is required). In EP, a population of organisms is evolved by applying a Gaussian mutation operator that is quite similar to traditional GA.

The EP involves a population of N organisms and each chromosome of each organism is mutated according to the following rule:

$$x'_i = x_i + (fitness(organism))^{1/2} N_i(0,1)$$

See [1] for the case where the fitness of an organism is zero. EP assumes that the optimisation task is one of minimization.

In the initial population, we expect large fitness values for the organisms. Thus large mutations are encountered, and the search is quite broad (the search space is explored the as large as possible). However as the fitness of an organism decreases, (as it approaches the optimal solution) search area is narrowed down. This is exactly what the above mutation operator does; it uses the fitness of an organism as its guide to conduct the search.

Subsequent to the application of this operator, the number of organisms in the populations becomes

$2N$ (*i.e.* combination of newly mutated organisms and old organisms). Using a tournament selection scheme (much like the one discussed previously), N of the best organisms are selected to become members of the next population, and the process continues. Connecting this population with the genetic algorithm population proves to be more effective than when either method is being implemented in isolation from the other. This method was successful for the previously mentioned function and performed slightly better for four other test functions (which

will be mentioned later) when compared to the GA population paired with a population of rules. But its performance was not satisfactory.

4.0 THE HUNT ENDS?

4.1 Real-Parameter GA

Evolutionary programming uses real-valued chromosomes. Real-valued GAs are discussed in [4] and have proven to be more successful than its binary chromosome counterpart. Due to its representation, precision is gained compared to the binary genetic algorithm. Also the absence of binary encoding and decoding speeds up the execution of the real valued algorithm. A sample of the operators and their success in various optimisation problems will be discussed in the next section.

4.2 Simple Crossover

As the title implies, this crossover method is easy to describe and understand. A crossover point is randomly chosen. All chromosomes at and beyond this point are swapped between the two parents. Unfortunately, this method does not create any new chromosomes (only rearranges those already generated from the initial population). So if the initial population is poor, the GA becomes stuck and have to rely on mutation as its only source for generating new and different information.

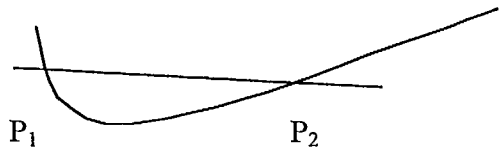
4.3 Arithmetic Crossover

This method is more interesting. For every chromosome of the two parents, the child's chromosome is constructed via:

$$c=a*p_1+(1-a)*p_2$$

where c, p_1, and p_2 are the real values that correspond to the same chromosome for the parents and the newly created child, respectively, a is a random number between zero and one. The

equation "$a*p_1+(1-a)*p_2$" is a line from p_1, to p_2 and the choice of a can determine any point on this line.



Where p_1, and p_2 are points on the curve that intersect the line. Since we are minimising the objective functions, we wish to obtain a value for the child that lies near the "valley" of the curve. This operator eventually finds the optimal variable value between the two parents, and in doing so obtains the optimal solution. This is true provided that the optimal value lies between these two points, if it lies beyond, then this operator will have created a child that is not very useful. It is clear that during the infancy period of the population, our choice of a will most likely cause a wider or broader search around the parent's chromosomes. But as the population matures and the chromosomes tend to converge, the search domain narrows down considerably.

4.4 Heuristic Crossover

This operator uses the objective function of the parents. Suppose that parent two has a higher fitness than parent one. Then the resulting child is constructed as follows:

$$c = r*(p_2 - p_1) + p_2,$$

where c, p_1, and p_2 represent the same parameters as in previous equation, and r is a random number between zero and one. This allows the search to concentrate around the chromosomes of the better parent. Depending on whether $p_2 - p_1$ is positive or negative, values to the left or right of p_2 will be considered. Again, it is clear that as the run matures, exploration becomes more local.

4.5 Mutation

The mutation operator uses the maximum number of generations that the GA is allowed to execute, and the present generation number. The idea is that near the beginning of the run whenever the mutation operator is activated, we wish to explore a greater area around the present

chromosome. As the run approaches its end, we would rather narrow down our search and reduce the effects of a "large" mutation. This is accomplished using the following operator:

$$c' = c + E(t, UB - c) \quad \text{if } m < 0.5, \text{ or}$$
$$c - E(t, c - LB) \quad \text{otherwise,}$$

where m is a random number between zero and one, t is the present generation number, T is the total number of generations, UB and LB are the upper and lower bounds of the parameter. Also defined;

$$E(t,y) = y * (1-r^{5(1-t/T)}),$$

where r is a random number between zero and one. The exponent 5 is chosen as "the degree of dependency on iteration number" [4] and is a standard value. Note that a smaller value weakens the dependency on the iteration (generation) number, and that a larger number strengthens this dependency. It is clear that if $1-t/T$ is close to one (i.e. we are in the early stages of the run), the chromosomes are expected to be perturbed greatly. And as $1-t/T$ tends to zero (i.e. as we approach the end of the run), the chromosome will be perturbed minimally.

4.6 The Test Functions

Five different test functions were considered:

1. $f_1 = 100*(x-y^2)^2 + (1-y)^2$, *with x, y e [0,5].*
2. $f_2 = x^2+y^2+u^2+v^2$, with x, y, u, v $e[-30,30]$. This is the sphere function which is continuous and unimodal.
3. $f_3 = S[x_i+0.5]^2$ for $i= 1,4$, with $x_i e[-30,30]$. The sphere step function which introduces several plateaus at which the GA may become trapped.
4. $f_4 = -20 * exp(0.2 * (S (x_i^2)^{1/2})-exp(1/10 * S cos(p*x_i)) + 20 + e$, for i= 1,3 with $x_i e$ [-20,30].

Known as Ackley's function which inflicts moderate complications to the search. Because a strictly local optimisation algorithm that performs hill climbing would surely get trapped in a local optimum, a search strategy that scans a slightly bigger neighbourhood would be able to jump over valleys towards increasingly better optima [1].

5. $f_5 = S\ (A_i - B_i)^2$ for $i=1,2,3$ and $A_i = S\ (a_{ij} * sin(a_j) + b_{ij} * cos(p_j))$ and $Bi = S\ (a_{ij}^* sin(x_j) + b_{ij}^* cos(x_j))$, for $j = 1, 2, 3$ where $A = (a_{ij})$, $B = (b_{ij})$; $a_{ij}, b_{ij}\ e\ [100,100]$; and x_j, $a_j[-p, p\ ]$. This function was introduced by Fletcher and Powell. It is " a typical representative of nonlinear parameter estimation regression problems" [1].

The results of the two crossover methods varied, but both of them were far superior to those obtained using the binary GA, or the combined GA and EP. The arithmetic crossover obtained the following results for the five test functions. For all functions the optimal value was zero, populations of size 100 was used, of which 50 were replaced every generation; 100 generations were executed, selection pressure was 1.31, crossover probability was 0.7, mutation probability was 0.01, and 100 runs were conducted.

1. The optimal solution was found thirteen times and for 12 runs the objective function was less than 0.01
2. Optimal solution found sixteen times and for 75 runs objective function was less than 0.00005.
3. Optimal solution found in all 100 runs.
4. Optimal solution was found thirteen times, and for 50 runs the value of objective function was less than $5.0e^{-8}$
5. Optimal solution found three times, and for 60 runs it was greater than 1.0.

So it can be concluded that this method performed well on all but the last test function. The heuristic crossover yielded the following results:

1. The optimal solution found 100 times!
2. The value of the objective function landed between 0.00001 and 0.005 for 75 runs.
3. Optimal solution found 94 times; 6 solutions greater than 1.0.
4. The value of the objective function landed between 0.00001 and 0.005 for 91.

5. Optimal solution found twice; only twice it was greater than 1.0; and 58 between $1.0e^{-7}$ and 0.00005.

This method performed much better on functions one and five, unlike the arithmetic crossover. It is obvious that one crossover method works better on those functions that the other crossover method has experienced difficulties. Ideally, we would like to know beforehand which method works better for a given problem. Since this is not the case, combining the two methods seems reasonable.

4.7 The Combined Crossover

Both [4,5] make reference to a successful genetic algorithm that uses many different operators. So combining the heuristic and arithmetic operators is a known concept. At the first generation, each crossover operator has an equal probability of being called upon to work its "magic" on any two parents. Then GA evaluates the child that is created (we ignore the effects of the mutation operator, as it is called with only a small probability). In all the above tests, 50% of the population was replaced at each generation, thus 50% remained untouched. A child is considered "a good one", if its fitness is better than the average of the remaining 50% of the population. After all, we want to create children that are better than those organisms in the population. Each time that a "good" child is created, the crossover method that created it receives a point. After the completion of one generation, the GA reassigns probabilities to these operators, based on the children that each created. If one operator did not create any children at all (neither good nor bad children) then it is assigned a probability of zero, and the other is assigned a value of:

(the number of good children created)/(the total number of children created).

However if both create children, then the both operators are assigned values of:

(number of good children created by that operator)/(total number of good children created)

This is because one operator might have created two children in one generation with one being good, while the other operator might have created five children of which only two are good. Considering these values one may think that the first operator is the more successful, after all

50% of the children it created were good. But it may have performed worse if it had been given the chance to create more children. This reassignment of the probabilities seems more "just". When it comes to select the operator, the one with the larger probability is considered. A random number is created, and if it is less than the probability assigned to an operator, GA will use that operator. Otherwise, the other operator is used.

The results for the combined crossover method on the five test functions are as follows:

1 . Optimal solution found 100 times.

2. Optimal solution found 71 times; worst was less than 0.00000 1.

3. Optimal solution found 100 times.

4. 77 of the runs landed between 1. $0e^{-8}$ and $5.0e^{-7}$.

5. Optimal solution found 46 times, only 1 greater than 1.0, 86 times less than $5.0e^{-8}$

These results surpass those obtained from either of the arithmetic or heuristic operators when used separately. It is clear that the genetic algorithm was able to determine which operator to use in order to maximize its success over the different test functions.

5.0 THE REAL GA

5.1 The Interface

The interface remains much the same as what was developed last year; the only difference being that the real-valued chromosomes are used rather than binary ones. It is menu-driven and is very straightforward. Several points must be considered:

1. The population size must always be even and less than 1000 (can be changed).

2. The number of parameters must always be less than 150 (can be changed).

3. Population replacement refers to the number of organisms that get replaced at each generation. It can be set at most one half of the population size.

4. Penalty depends on how important it is to keep the variables within the specified bounds.

5. The selective pressure determines the extent to which highly successful organisms get selected compared to the lower fit organisms. So the higher the pressure, the more successful organisms

will be selected to be parents. Its value ranges from 1.0 to 2.0. Selection pressure is only necessary for linear ranking.

6. The number of runs is the number of times that GA is executed with different initial populations.

7. Suggested values are given for the selection pressure, crossover and mutation probabilities.

5.2 The Reader

The present GA program arranges the output in a different manner as compared to the routine that was developed last year. A special reader program was developed to post process the binary output. The program is called "readerc" and is a menu driven interface. The following options will be available upon execution of this program:

1. Determine run values.
2. Tabulate the optimal values found.
3. Create own table and tabulate optimal values.
4. Find optimal solution.
5. Determine run values and variable values.
6. Who beats whom?
7. List Runs and optimal values.
8. Determine improvement in run of population.
9. List optimal solutions found.
10. List starting points.
11. Exit.

Descriptions of the options are:

1. Lists the best solution found at each generation for a given run.
2. Creates a "histogram" tabulating the optimal solutions found for every run. Table is preset.
3. Allows the user to create own table and tabulate results.
4. Lists the optimal solution with parameters out of all the runs.
5. Same as 1, except lists the parameters as well.

6. Allows comparison between two output files as to which one was better.

7. Lists the optimal solution and parameters that yield these solutions for every run.

8. Allows user to see how many times the GA found a better solution in the course of a run.

9. Lists optimal solution found in every run; no parameters.

10. Lists the starting points, or initial best, of every run.

5.3 Other Instructions

Below are instructions that have been probably mentioned before but are nevertheless helpful reminders. In the GA program, the following is declared within the set of variable declarations:

extern float func_0,

The function func_0 is the function to be optimised. It is used as follows:

s 1 [I].objective = func-(values),

where values is an array containing the parameters being optimised. Note that if there are three parameters, then they are stored in values[1], values[2], and values[3], leaving values[0] "empty"

The optimisation function can either be written in C or Fortran.

Optimisation of a function written in C

The C function (and all other necessary functions that it may need to call in order to obtain the objective function and fitness for any organism) can be stored in a single file, with the name of the function being **func_**. It must also return a float number. For example:

```
#include <math.h>
#include <string.h>
float ftinc-(float values[4]){
                    function body
                    return (float number);}
```

It is in this function that all calculation related to objective function and fitness of organisms are done. Suppose that all routine required to that end are saved in a single file called function.c. Then setting GA to optimise the function, all that is necessary is to type:

```
tango% cc ga.c function.c -lrn
tango% a.out
```

Note that the output (the best solution at each generation) is sent to a file called "o1" and can be read and interpreted using "reader.c".

Optimisation of a function written in Fortran77

Like the C function, the Fortran function must be written as a function that returns a real value, and must include within itself all other subroutines, functions that would be called to obtain the objective for each organism. It should be written as follows:

```
real function func(values)
real values(0:n)
function body
func= ?
return
end
```

The zero in *values(0:n)* allows the indexing to begin at zero. Note that parameters are still stored in *values(l), values(2),...,* and *values(n)*, with *values(0)* being "empty". This can then be saved in a file called function.f.

To execute the GA and the function to be minimized the following commands must be executed:

```
tango% f77 -c -silent funct.f
tango% cc -c ga.c
tango% f77 funct.o ga.o
tango% a.out
```

Why header files?

Suppose a new selection operator is to be tried in the GA code. It should be copied into the GA program. However, it would be better if we could select the procedure directly from the interface. Without going into the details (yet) on how to incorporate this idea into the interface, the easiest way is to create for each selection procedure *i*, a header file called *selecti.h*. The interface will then "include" this header into the GA program and allow this procedure to be used. The same can be done for crossover and mutation operators as well.

All that is necessary in the header file is the operator itself (see selectl.h, crossl.h and cross2.h for example). Also, the function name should be the one that is used in the GA program (select, crossover, or mutate). It does not matter that various crossover operators have the same function name, since only one will be included in the file. A further restriction is that all functions pertaining to the same operator pass the same parameters.

Modifying the interface to include more operators

All that is necessary is to add the new choice of operators to the menu found under "TYPE" Operator, where TYPE is either selection, crossover or mutation. Add the choice, as is done for the crossover operator, and increment the number of choices (so that while loop is called and it scans for a choice, it selects appropriately). *i.e.* increment the "2" found below :

```
while(status != 1 || choice < 1 || choice > 2){
fflush(stdin);
printf("Invalid entry. Enter choice:");
status = scanf("%d",&choice);};
```

How does the program get set with the proper operators?

Upon exiting the interface, and only if it is saved, a file called header.c is created or updated. The following text is printed to it:

```
#include <stdio.h>
```

```
#include <math.h>
#include "ga.h"
#include "selecti.h"
#include "mutatei.h"
#include "crossi.h"
```

i is determined by the operator selection.

This file is then concatenated with generic5.c to create ga.c. This can be carried out through the startup file:

```
cat headcr.c realGA.c > ga.c
cc -c ga.c
rm ga.c
f77 -c -silent bessel.f
f77 bessel.o ga.o
rm bessel.o
rm ga.o
```

The above shows the setup for the GA to maximise the Bessel function. This is a text file. To transform it into an executable type:

```
chmod +x startup.
```

a.out then executes the GA. So it should be ensured that the proper operator selected and also that operator "*i*" is included in the header file "*i*".

6.0 CONCLUSION

I began this work term with the task of improving upon last summer's GA package. A new selection method has been added to the GA arsenal, even though linear ranking is still the selection method of choice. All that followed led to the eventual declaration that the real-valued genetic algorithm was superior to the standard binary GA. Multi-population GAs led to the consideration of a mixture of regular populations and parasites. Population of parasites led to

populations of rules which led to combining of GA and evolutionary programming. The real chromosomes of EP led to the real-valued GA. Results of five different test functions testified to the power of this new algorithm. And the realisation of a combined crossover method led to a success that surpassed all that was encountered before.

REFERENCES

1. Baeck, Thomas, Evolutionary algorithms in theory and practice; evolution strategies, evolutionary programming, genetic algorithms. Oxford University Press, 1996.

2. Goldberg, David E., Genetic algorithms in search, optimization, and machine learning, New York, Addison-Wesley, 1989.

3. Johnson, J. Michael, and Rahmat-Samii, Yahya, 'Introduction to Genetic Algorithms', IEEE Antennas and Propagation Magazine, Vol. 39, No.4, August 1997, pp. 7-21.

4. Michalewicz, Zbigniew, Genetic algorithms plus data structures equals evolution programs, 3rd revised and extended edition, Berlin, Springer-Verlag, 1996.

5. Sandlin, Brian S., and Terzuoli, Andrew J. 'A comparison of simple and complex genetic algorithms in wire antenna design', Applied Computational Electromagnetics Society, Annual Review of Progress, Monterey CA, 1997, pp. 1080-1086.

6. Sumida, Brian H., and Hamilton, William D., 'Both Wrightian and 'parasite' peak shifts enhance genetic algorithm performance in the travelling salesman problem', Computing With Biological Metaphors, Chapman and Hall, pp. 264-279, 1994.