Genetic Algorithms: Theory and Application

Dennis I. Merino, Edgar N. Reyes/Carl Steidley
Southeastern Louisiana University/Texas A&M University - Corpus Christi
Hammond, LA 70402/ Corpus Christi, TX 78412

1 Introduction

Genetic algorithms, a class of robust and efficient search techniques that can be randomly sample large spaces, have applications in the field of optimization and in a wide range of computer science problems in pattern recognition, search, scheduling, and machine learning. Genetic algorithms are motivated by characteristics found in natural population genetics, among them robustness and efficiency. Features of biological systems found in genetic algorithms include reproduction, self-guidance, self-repair, the nature of survival of the fittest, and variation through mutation. Genetic algorithms were developed by John Holland of the University of Michigan in the 1970's. Many of the essential properties of genetic algorithms discussed in this paper can be found in [1, 2].

When a genetic algorithm is used to find an optimal solution in the space of all feasible solutions, the algorithm maintains a population (or set) of feasible solutions which evolve through random process based on principles found in the mechanics of natural selection and genetics. Each time this set of solution evolves (or as we say iterated), a new set of solutions is generated. The iterations are repeated several times as necessary. An important goal from the standpoint of genetic algorithms is to improve the values of the objective function in an efficient, practical, or quick method. Ideally, as it is true for many optimization techniques, one would like to generate after several iterations a set of solutions that includes an optimal global solution; but a genetic algorithm has its priority set on tying improvement of an objective function with performance-practicality of the method. In the literature, one finds several applications of genetic algorithms in routing and scheduling problems, machine learning, setting weights in neural nets, and testing expert systems that control complex systems, amongst others [2]. In the next section, we focus on the algorithm itself by solving an elementary optimization problem.

2 Genetic Algorithm: An example

We present the following to illustrate the effectiveness of genetic algorithms. We note that although the problem can be solved easily using a different method, it is the process involved with genetic algorithms that is put to focus.

We wish to

$$\text{Minimize } f(x_1, \ldots, x_{12}) = \sum_{i=1}^{12} (x_i - \mu)^2 \text{ subject to } L \le x_i \le M,$$

where μ is a given real number. We note that this is essentially the same problem that [3] solved. In this case, the initial state is given, say $(a_1, \ldots, a_{12})$. We take L to be the minimum of these numbers, M to be the maximum, and μ is the average of these numbers, that is, $\mu = \frac{1}{12} \sum_{i=1}^{12} a_i$.

In particular, we look at the problem where the initial state is given by

(2, 24, 4, 22, 6, 20, 8, 18, 10, 16, 12, 14),

so that $\mu = 13$ and that our problem becomes

$$\text{Minimize } f(x_1, \ldots, x_{12}) = \sum_{i=1}^{12} (x_i - 13)^2 \text{ subject to } 2 \le x_i \le 24.$$

The first step in applying a genetic algorithm to our problem is to change the values of the variables x_i into digits consisting of 1 and 0, such as the binary expansion of the numbers. The length of each string depends greatly on the accuracy that is needed. In our particular problem, the initial state was rigged so that we would only use integer values between 0 and 31 for each variable x_i. Each solution will then have a string of 60 characters (0 or 1), five for each variable x_i. The first five of these characters is the binary expansion of x_1, the second five is that of x_2, and so on. Notice that the values of x_i. Notice that the values of x_i in this case turns out to be between 0 and 31. This should not pose a great problem because if a solution happens to satisfy our constraints, then it is also a solution of our original problem. If a solution does not satisfy our constraints, then we can simply disregard it.

The next step is to randomly generate an initial set of solutions. This can be done in a number of ways, one of which is to generate each possible solutions manually. For each digit of the solution, x, we can flip a coin. If the coin lands head, then that particular digit of that solution is 1, and if the coin lands tail, then that digit is 0. In this manner, we have to flip 60 coins for every solution. Since this seems tedious for this particular problem, we can simply write a program that will generate such a string of characters.

Note that there are 2^{60} (more than $1.15 * 10^{18}$) such solutions. If we choose for our initial set one that contains a small number of solutions, then we expect our genetic algorithms to run for a long time. Meanwhile, if we choose to have an initial set that contains a large number of solutions, then the probability that the strong genes mate might lessen considerably. It would be interesting to find out if this apparent dilemna has a solution.

For simplicity, we look at the case when we have four solutions in our initial set. In our particular example, say we get the set whose strings have the following decimal representations.

$$y^1 = (\ 1\ \ 2\ \ 3\ \ 4\ \ 5\ \ 6\ \ 7\ \ 8\ \ 9\ \ 10\ 11\ 12\ )$$

$$y^2 = (\ 21\ 10\ 12\ 13\ 12\ 11\ 16\ 13\ 1\ 3\ 5\ 15\ )$$

y^3 = (2 27 31 19 1 29 3 4 28 10 11 13)

y^4 = (19 23 12 14 14 13 22 11 23 17 19 11).

To each possible solution $y = (x_1, \ldots, x_{12})$, we associate a *fitness* value. Notice that our objective function, **f**, is nonnegative, so that the better fit solutions must be the ones where **f** has value close to 0. The fitness function

$$\mathbf{g}(y) = \frac{1}{f(y)}$$

does the trick. The fitness value of a solution is used for the operation called *reproduction*. Reproduction is the process in which individual strings are copied according to their fitness values. One easy way to attain our goal is to calculate the percentage of total fitness, which simply is the fitness of the string divided by the sum of the fitness of the group. This percentage can be thought of as the probability of picking that particular value of x in that group. This clearly illustrates the notion of survival of the fittest, since the greater fitness an element has, the greater is its chance of being used for reproduction. The following table shows the fiyness and percentage of total fitness for our set of initial solutions.

y	**f**(y)	Fitness	Percentage of Total Fitness	Expected Value
y^1	650	0.001538	22.90%	0.9160
y^2	400	0.002500	37.22%	1.4888
y^3	1496	0.000067	1.00%	0.0400
y^4	383	0.002611	38.88%	1.5552
Total		0.006716	100.00%	4.0000

The last column in the table gives the expected number of copies of the solution y^i that will be reproduced. Looking at the table, we expect 1 copy of y^1 , and depending on our criterion on choosing which solution is better, we will either have 2 copies of y^2 and 1 copy of y^4, or 2 copies of y^4 and 1 copy of y^2.

For our particular problem, say we reproduce 2 copies of y^4 since it has the highest expected value. This means that we have 1 copy each of y^1 and y^2, and 2 copies of y^4.

After reproduction, simple *crossover* may proceed in two steps. The members of the newly reproduced strings in the mating pool are paired at random. In our example, say one copy of y^1 is paired with y^4; and y^2 is paired with the other copy of y^4. For each of the pairs, one selects an integer between (and including) 1 and *n* - 1, where *n* is the length of the string (in our case, *n* = 60). This is where crossover will take place. Notice that there are *n* - 1 spaces between the first element of the string and the last element of the string; that is, the space between the first element and the second element, between the second and third elements, etc. In our case, we will take the space number as increasing from left to right. As an example, suppose we are mating string y^1 with string y^4, and we have chosen crossover to take place at space *l*. Then every digit of y^1 to the right of space *l* (where the space numbers are larger than *l*) will be placed in y^4 and all the digits of y^4 to the right of space *l* will be placed in y^1. This is the process called crossover.

One checks that if the crossover happens at a space number that is divisible by 5, then the crossover can be done using the decimal representation of the digits.

For the pair y^1 and y^4, say we have randomly chosen the thirty-fifth space.

$$y^1 = (\ 1\ \ 2\ \ 3\ \ 4\ \ 5\ \ 6\ \ 7\ |\ 8\ \ 9\ \ 10\ 11\ 12\ )$$

$$y^4 = (\ 19\ 23\ 12\ 14\ 15\ 13\ 22\ |\ 11\ 23\ 17\ 19\ 11\ )$$

Crossover yields the following offsprings.

$$z^1 = (\ 1\ \ 2\ \ 3\ \ 4\ \ 5\ \ 6\ \ 7\ \ 11\ 23\ 17\ 19\ 11\ )$$

$$z^2 = (\ 19\ 23\ 12\ 14\ 15\ 13\ 22\ \ 8\ \ 9\ 10\ 11\ 12\ )$$

Suppose that when mating y^2 and y^4, the crossover occurred on the forty second space. The following are the offsprings.

$$z^3 = (\ 21\ 10\ 12\ 13\ 12\ 11\ 16\ 13\ \ 6\ 17\ 19\ 11\ )$$

$$z^4 = (\ 19\ 23\ 12\ 14\ 15\ 13\ 22\ 11\ 17\ \ 3\ \ 5\ 15\ )$$

The following table summarizes the result of crossover for our mating pool.

Mating Pool	Mate	Crossover Site	New Population	Fitness
y^1	y^4	35	z^1	0.001324
y^4	y^1	35	z^2	0.003597
y^2	y^4	42	z^3	0.005181
y^4	y^2	42	z^4	0.002433
Total				0.12535

One checks that the total fitness of this solution set has almost doubled, from our original solution set with a total fitness of 0.006716 to 0.012535. This illustrates that our solution set is improving. The maximum fitness for this group, 0.005181, is also almost twice as that of the initial group (0.002611). In the case that the maximum fitness in the current group is not as big as that of our initial group, we can flag the string that gives a current maximum value for our objective function.

There are a number of ways to terminate a genetic algorithm. One way is to stop when the string that gives the maximum value for our objective function has not changed for a certain number of iteration (say, 100). Another way is if the total fitness in our current solution set ceases to increase.

Genetic algorithms have another operator that plays a secondary role to reproduction according to fitness combined with crossover. It is called *mutation*. Although there is much confusion as to the role of mutation in natural and artificial genetics, it is needed because reproduction and crossover may occasionally become overzealous and may potentially lose the 1's and 0's in places where they are needed. Also, the initial gene pool may be very weak. Mutation is done by randomly choosing a location in the staring, and changing the digit in that particular string. For example, a mutation on x = (0 0 0 0 1) that takes place on the third digit would make it x' = (0 0 1 0 1) . To obtain good results in empirical genetic algorithm studies, [1] suggests an order of one mutation per thousand bit (position) transfer.

3 Summary

Genetic algorithm is a robust and efficient search technique, motivated by the mechanics of natural population genetics, with direct applications in the field of optimization. The algorithm provides for a practical method for improving the values of an objective function that is being optimized.

A genetic algorithm, as we have illustrated with an elementary optimization problem, maintains a population or set of feasible solutions which evolve through three basic operators, namely reproduction, crossover, and mutation; operators that are inspired by features commonly found in natural genetic systems. When a population evolves, a new population or set of solutions is generated. It would be ideal if a new population would contain a global optimal solution. But the class of search techniques, genetics algorithms, has its priority set on tying improvement of an objective function with practicality and efficiency.

The particular optimization problem considered in the second section belongs to a class of optimization problems of the form

$$\text{Maximize } \mathbf{f}(x_1,\ldots,x_n) \text{ subject to } L_i \le x_i \le M_i$$

where **f** is a positive-valued objective function, x_i are integers, and L_i, M_i are fixed constants for $i = 1,\ldots, n$. To implement a genetic algorithm, one must represent or code the set of all possible solutions $(x_1,\ldots,x_n)$. In our particular example, we represented each x_i by its binary representation. Thus, $(x_1,\ldots,x_{12})$ was represented by a string of sixty digits consisting of 0's and 1's.

After the coding of the set of all possible solutions, we randomly chose an initial set with four solutions s^1,..., s^4, where s^i is a sequence of sixty digits consisting of 0's and 1's. (In general, one would choose an initial set with an even number of solutions so that mating and reproduction can follow.) The fitness $\mathbf{f}(s^i)$ of each initial solution s^i was found. Then this was followed by reproduction, i.e., a random selection of four solutions t^i; the stochastic nature of the selection was based on the relative fitness of the four initial solutions. For example, as illustrated in our example, a solution with a relatively high fitness when compared to those of the other solutions has a strong probability of being selected and reproduced. This illustrates the principle of survival of the fittest - a feature found in genetic systems. If $\{ t^1,...,t^4 \}$ is the set of solutions that has been reproduced from the initial set, then the t^i 's were randomly paired. Given a pair, mating proceeded and this is described in general by an exchange of information between the mates in the pair. The exchange in information, called crossover, is guided by stochastic means; thus generating a set of solution $\{ u^1,...,u^4 \}$. After the crossovers, a third operator called mutation may be applied to the u^i's and which can change the solution u^i to another solution v^i. Mutation, also a feature of genetic systems, is useful in optimization in that it can help identify false global optimum points (which can happen if an optimization problem has local optimum points which are not global optimum points).

After choosing an initial set of four solution s^i, as in our example, we applied three operators, namely reproduction, crossover, and mutation to produce another set of four solutions v^i. This process is iterated several times, i.e., now the v^i's form the new initial set of solutions and the three operators are applied to this new initial set to produce a third set of solutions.

Since genetic algorithms are characterized by robust random searches and inspired by principles of natural genetic systems, they find many applications in the field of combinatorial optimization and in computer science problems related to pattern recognition, search, and machine learning.

References

[1] Goldberg, D.E. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, Reading, MA, 1989.

[2] Grefenstette, J.J. *Genetic Algorithms*. IEEE Expert. October, 1993.

[3] Reyes, E.N. and Steidley, C. *A GAP Approach to the Simulated Annealing Algorithm*. Computers in Education Journal of the American Society for Engineering Education. Vol. 7, No. 3 (1997) 43-47.