An Overview of Genetic Algorithms

Joseph Carnahan
Undergraduate Modelling, Simulation, and Analysis
February 16, 2000

Based primarily on Chapter 1 of the book *Genetic Algorithms in Search, Optimization, and Machine Learning*, by David E. Goldberg, ©1989 by Addison-Wesley Publishing Company, Inc.

Outline

- Review of the algorithm
- Step-by-step walk through a simple example
- Look at a few more sophisticated variations
- Look at the limitations of genetic algorithms

The GA Algorithm

Given a problem:

1. Devise a means for representing a solution to the problem.
2. Devise a heuristic for evaluating the fitness of any particular solution.
3. Generate a population of possible solutions, either randomly or with malice of forethought.

4. Evaluate the fitness of each member of the population.
5. Produce a new generation of solutions by picking from the existing pool of solutions with a preference for solutions which are better-suited than others.
6. Randomly pair off members of the new generation.
7. Within each pair, swap parts of the members' solutions to create offspring which are a mixture of the parents.
8. Randomly mutate a *very* small fraction of genes in the population.
9. Go back to step 4.

Some Obvious Questions

- Does this algorithm converge? Will it always be producing better and better solutions?
- Is this algorithm even remotely efficient?
- Does this algorithm leave any part of the search space unexplored?
- How easy is it to fool the algorithm?
- Bottom Line: *Does it work?*

A Simple Example

As a trivial example, let us attempt to maximize the function

$$f(x) = x^2$$

over the range of integers from $0 \ldots 31$.

This function *could* be solved by a variety of traditional methods, such as a hill-climbing algorithm which uses the derivative of $f(x)$:

1. Start from any integer x in the domain of f.
2. Evaluate the derivative at that point, $f'(x)$.
3. Observing that the derivative is positive, we pick a new x which is a small distance in the positive direction from our current x.
4. Repeat until $x = 31$.

However, in many real-life situations, derivative information may be either unavailable or unhelpful.

So, let's see how a genetic algorithm would approach this problem...

1. *Devise a means to represent a solution to the problem.*

   We will represent x with five-digit unsigned binary integers.

2. *Devise a heuristic for evaluating the fitness of any particular solution.*

   Since the function which we are trying to maximize is easy to evaluate, we will use the $f(x)$ value itself to rate the fitness of a solution. If $f(x)$ were more complicated, we might consider a simpler heuristic that would still point us in more or less the same direction.

Why binary?

Genetic algorithms often process binary represenations of solutions. In part, this works well because crossover and mutation can be clearly defined for binary solutions:

crossover: Pick a random point inside the string. Then, create the two offspring from the two parents by swapping the substrings that come after the break point. Given two strings 00000 and 11111 and a break point after the second digit, the crossover is

$$\begin{array}{ccccc} 00000 & \Rightarrow & 00|000 & \Rightarrow & 00111 \\ 11111 & \Rightarrow & 11|000 & \Rightarrow & 11000 \end{array}$$

mutation: With predetermined (and usually very low) probability, flip an individual bit. 01011 could turn into 01001, for example.

3. *Randomly generate a set of solutions.*

   To make things easier, we will only consider a population of four solutions. Obviously, larger populations are used in real applications to explore a larger part of the search space more quickly.

   Our four solutions:

   01101
   11000
   01000
   10011

4. *Evaluate the fitness of each member of the population.*

   Here are is the evaluation of the initial population:

Chromosome	*Value of x*	*Value of $f(x)$*	*Fraction of Total*
01101	13	169	0.144
11000	24	576	0.492
01000	8	64	0.055
10011	19	361	0.309
Total		1170	1.000

   We look at the *total* fitness because each solutions' fraction of the total will determine its likelihood of reproduction.

5. *Produce a new generation of solutions by picking from the existing pool of solutions with a preference for solutions which are better-suited than others.*

   We divide the range 0.0 . . . 1.0 into four bins, sized according to the relative fitness of the solutions which they represent:

String	*Associated Bin*
01101	0.0 . . . 0.14
11000	0.14 . . . 0.63
01000	0.69 . . . 0.69
10011	0.69 . . . 1.00

By generating 4 $Uniform(0, 1)$ random variate values and seeing which bin they fall into, we pick the four strings that will form the basis for the next generation:

Random Number	*Falls Into . . .*	*Chosen String*
0.08	0.0 . . . 0.14	01101
0.24	0.14 . . . 0.63	11000
0.52	0.14 . . . 0.63	11000
0.87	0.69 . . . 1.00	10011

Notice that the least-fit solution, 01000 ($x = 8$), has already “died out”, since no random numbers were picked from its bin.

6. *Randomly pair off members of the new generation.*

   We decide (or rather, "Our random number generator decides for us") to mate the first two strings together and the second two strings together.

7. *Within each pair, swap parts of the members' solutions to create offspring which are a mixture of the parents.*

   For the first pair of strings,

   $$01101,\ 11000$$

   we randomly select the crossover site to be after the fourth digit. Crossing these two strings at that point yields

   $$\begin{array}{ccccc} 01101 & \Rightarrow & 0110|1 & \Rightarrow & 01100 \\ 11000 & \Rightarrow & 1100|0 & \Rightarrow & 11001 \end{array}$$

For the second pair of strings,

$$11000,\ 10011$$

we randomly select the crossover site to be after the second digit. Crossing these two strings at that point yields

$$\begin{array}{ccccc} 11000 & \Rightarrow & 11|000 & \Rightarrow & 11011 \\ 10011 & \Rightarrow & 10|011 & \Rightarrow & 10000 \end{array}$$

8. *Randomly mutate a* very *small fraction of genes in the population.*

   With a typical mutation probability of 0.001 per bit, it happens that none of the bits in our population are mutated.

9. *Go back and reevaluate the fitness of the population.*

   This would be the first step in generating a new generation of solutions. However, it is also useful in showing the way that a single iteration of the genetic algorithm has improved this sample.

Chromosome	*Value of x*	*Value of $f(x)$*	*Fraction of Total*
01100	12	144	0.082
11001	25	625	0.356
11011	27	729	0.416
10000	16	256	0.146
Total		1754	1.000

   Observe that the total fitness has gone from 1170 to 1754 in a single generation.

- Also, it is interesting to note that the algorithm has already come up with the string 11011 ($x = 27$) as a possible solution.
- How did this happen?
  - *In part:* Random chance.
  - *More important:* The algorithm favors decent solutions under the assumption (which holds in this case) that they can be used to build even better solutions.
    Even in our trivial example, the decision to produce two copies of the best string (11000) and zero copies of the worst string (01000) made a difference in generating 11011.
  - *The "Building-Block Hypothesis":* Genetic algorithms work well as long as better solutions can be built by combining parts of other solutions.

More Sophsisticated Examples

(Most of the information in this section comes from the book *Practical Genetic Algorithms* by Randy L. Haupt and Sue Ellen Haupt, ©1998 by John Wiley & Sons, Inc.)

- Continuous vs. Binary representation schemes
  - How can the algorithm do crossovers?
  - How can the algorithm do mutations?
- Efficiency Considerations
  - How can the algorithm avoid re-evaluating expensive functions?
- Alternative Swap Schemes

Continuous Representations

Often, binary numbers do not accurately reflect the parameters of the function which we are trying to optimize. Consider, for instance, a question of finding the highest point on a physical landscape. The height function $f(X, Y)$ maps a lattitude and a longitude to a height H:

$$f : X \times Y \to H.$$

Since it is unlikely that the mountain peak which we are seeking lies directly on an integer-valued lattitude or longitude line, X and Y are naturally floating-point numbers.

More generally, when a genetic algorithm is processing a problem with heuristic function f and f takes n floating-point parameters, then the GA usually looks at solutions as lists of the form

$$S = (s_1, s_2, \ldots s_n).$$

So, in our xy-coordinate example, if we were trying to find locations in between 75° and 80° west longitude and between 40° and 45° north lattitude, we could generate an initial population of possible solutions which would look like

$$\begin{gathered}(72.54234,\ 44.98703),\\(73.21234,\ 42.08763),\\(70.62311,\ 41.93537),\\(73.82468,\ 44.23617)\ldots\end{gathered}$$

and so forth.

Continuous Crossovers

Problem:

How does the genetic algorithm "mate" the potential solutions?

Possible Solution:

Pick a point in the parameter list $s_1, s_2, \ldots s_{n-1}$ and swap the solutions' values after that point.

In the xy-coordinate example, that would mean that ordered pairs would swap their y-coordinates when mating:

$$\begin{aligned}(72.54234,\ 44.98703) &\Rightarrow (72.54234,\ 42.08763)\\(73.21234,\ 42.08763) &\Rightarrow (73.21234,\ 44.98703)\end{aligned}$$

However, this adds very little variety to the sample.

A Better Solution:

In mating, combine chromosomes element-by-element with a $Uniform(0,1)$ weighting β. This means that the offspring of

$$\begin{aligned} S_{dad} &= (dad_1, dad_2, \ldots dad_n) \\ S_{mom} &= (mom_1, mom_2, \ldots mom_n) \end{aligned}$$

would be

$$S_{child} = (child_1, child_2, \ldots child_n)$$

where

$$child_i = \beta \cdot dad_i + (1-\beta) \cdot mom_i.$$

Another child could be produced in the same way by switching dad_i and mom_i with each other.

However, this does not explore outward. A combination of the two points (4.0, 4.0) and (6.0, 1.0) will always produce offspring with $4.0 < x < 6.0$ and $1.0 < y < 4.0$. This leaves parts of the search space near the edges unexplored.

For this reason, more complicated offspring generation schemes exist. Some schemes produce three offspring per parental pair, and some involve combining parents using weights outside of the range $0.0 < \beta < 1.0$. For instance, a child's parameters could be produced by the following equation:

$$\begin{aligned} \beta &= -0.5 \\ child_i &= -0.5 \cdot dad_i + 1.5 \cdot mom_i \end{aligned}$$

Continuous Mutations

Mutating continuous solutions in a genetic algorithm's population is relatively simple. With some small probability, replace one of the parameters of a solution with a totally new random number which is within the range of valid parameters.

So, the continuous solution

$$(0.12345, 0.67890, 0.09876, 0.54321)$$

(with all parameters constrained to the range $0.0 \ldots 1.0$) could mutate to

$$(0.12345, 0.24682, 0.09876, 0.54321)$$

where the second parameter happens to be replaced by a new floating-point number from the range $0.0 \ldots 1.0$.

Efficiency Concerns

One of the major problems with the efficiency of genetic algorithms is that they may re-evaluate the heuristic function value of a particular solution many times. If a string occurs multiple times in the inital population, or if multiple copies of a string survive reproduction (such as when a string is crossed with a copy of itself), then the GA will evaluate the function more than it needs to.

- One simple trick is to make sure that there are no duplicates in the initial population. For instance, if the solutions are binary strings and there will be a population of 8 solutions, just make sure that the first three digits of each solution differ:

000...	001...	010...	011...
100...	101...	110...	111...

Ensuring diversity in the initial population will reduce, if not eliminate, the number of unneeded function evaluations.

- A much more complicated trick is to keep track of all of the strings that have been evaluated so far. This list would not be part of the current population, but whenever the need arose to evaluate a member of the population, the algorithm would consult this list to make sure that this member had not already been evaluated.

  Of course, this solution would *only* be used when the heuristic function is very computationally expensive. Since building and searching the list would take a large amount of time, and since the list itself would take up an enormous amount of memory, this would only be done in the most extreme situations.

Alternative Binary Swapping Schemes

Swapping the ends of binary strings is not always the most effective way to combine them. A more sophisticated or thorough method of mixing strings can often lead to faster convergence to an optimum solution. Two alternative swapping methods are worth considering:

Two-point Swapping: Instead of swapping all of the bits in a string after a certain point, pick two points at random within the string and swap all of the bits *between* the two points.

So, the following swap could occur:

$$\begin{array}{ccccc} 00000000 & \Rightarrow & 000|000|00 & \Rightarrow & 00011100 \\ 11111111 & \Rightarrow & 111|111|11 & \Rightarrow & 11100011 \end{array}$$

Uniform Swapping: Before combining two strings, construct a binary string called a *mask* which is of the same length as the solution strings. Then, use the following rules to cross string $parent_1$ with $parent_2$:

1. If the ith bit of the mask is 0, then the ith bit of $child_1$ is the ith bit of $parent_1$ and the ith bit of $child_2$ is the ith bit of $parent_2$.
2. If the ith bit of the mask is 1, then the ith bit of $child_1$ is the ith bit of $parent_2$ and the ith bit of $child_2$ is the ith bit of $parent_1$.

So, offspring take some of the bits from of their parents, as with one-point and two-point swapping. However, with uniform swapping, the bits taken are considerably more random and the mixing of the parents is more complete.

Schemata and the Limitations of GAs

A *schema* (plural *schemata*) is a string over the three-character alphabet $\Sigma = \{0, 1, *\}$. The 0 and 1 correspond to the two binary digits, and the * signifies a "don't care".

A schema is said to *match* a binary string if all of the 1's and 0's in the schema match corresponding 1's and 0's in the same positions in the string. So, the schema

1**10

matches the strings

10010, 10110, 11010, and 11110,

because they all have a 1 in the first and fourth places and a zero in the fifth place.

Schemata are useful for discussing the progress made by a genetic algorithm. A GA is said to be *processing* a schema if a string which matches the schema is currently in the population. So, if the string

1100

is in the population, then all of the following schemata are being processed:

1*** *1** **0* ***0 11** 1*0* 1**0 *10*
1*00 **00 *100 *1*0 11*0 110* 1100 ****

The *fitness* of a schema is the average fitness of all of the strings which match the schema.

The *order* of a schema H, denoted $o(H)$, is the number of positions in the schema which have a 0 or a 1 for their value. So, the schema

0**110

has $o(H) = 4$. Meanwhile, the very non-descriptive schema

*1****

has only $o(H) = 1$. There is even the schema

******

which has $o(H) = 0$. ****** matches all six-character strings.

The order of a schema is important, because high-order schemata are likely to be broken up in a crossover. Low-order schemata are likely to be passed on from generation to generation, gradually being favored or disfavored based on whether they have above-average or below-average fitness.

Fundamental Theorem of Genetic Algorithms

The good news:

The Fundamental Theorem of Genetic Algorithms states that the number of low-order above-average schemata represented in the population will grow at an exponential rate as the algorithm iterates. So, as long as the building-block hypothesis holds, genetic algorithms will perform very well.

The bad news:

Genetic algorithms are *not* guaranteed to converge. There are a number of relatively simple situations (called “minimal deceptive problems”) that can be engineered to confuse a genetic algorithm. When the building-block hypothesis does not hold and the combinations of good parts lead to bad results, then genetic algorithms fail miserably.