AN INTRODUCTION TO GENETIC ALGORITHMS

Scott M. Thede
DePauw University
Greencastle, IN 46135
sthede@depauw.edu

ABSTRACT:

A genetic algorithm is one of a class of algorithms that searches a solution space for the optimal solution to a problem. This search is done in a fashion that mimics the operation of evolution – a "population" of possible solutions is formed, and new solutions are formed by "breeding" the best solutions from the population's members to form a new generation. The population evolves for many generations; when the algorithm finishes the best solution is returned. Genetic algorithms are particularly useful for problems where it is extremely difficult or impossible to get an exact solution, or for difficult problems where an exact solution may not be required. They offer an interesting alternative to the typical algorithmic solution methods, and are highly customizable, which make them an interesting challenge for students.

1. AN OVERVIEW OF GENETIC ALGORITHMS

A genetic algorithm is a type of searching algorithm. It searches a solution space for an optimal solution to a problem. The key characteristic of the genetic algorithm is how the searching is done. The algorithm creates a "population" of possible solutions to the problem and lets them "evolve" over multiple generations to find better and better solutions. The generic form of the genetic algorithm is found in Figure 1. The items in bold in the algorithm are defined here.

```
1. Create a population of random candidate solutions named pop.
2. Until the algorithm termination conditions are met, do the following (each
   iteration is called a generation):
   (a) Create an empty population named new-pop.
   (b) While new-pop is not full, do the following:
       i.  Select two individuals at random from pop so that individuals which
           are more fit are more likely to be selected.
       ii. Cross-over the two individuals to produce two new individuals.
   (c) Let each individual in new-pop have a random chance to mutate.
   (d) Replace pop with new-pop.
3. Select the individual from pop with the highest fitness as the solution to the
   problem.
```

Figure 1: The Genetic Algorithm

The **population** is the collection of candidate solutions that we are considering during the course of the algorithm. Over the generations of the algorithm, new members are "born" into the population, while others "die" out of the population. A single solution in the population is referred to as an **individual**. The **fitness** of an individual is a measure of how "good" the solution represented by the individual is. The better the solution, the higher the fitness – obviously, this is dependent on the problem to be solved.

The **selection** process is analogous to the survival of the fittest in the natural world. Individuals are selected for "breeding" (or **cross-over**) based upon their fitness values – the fitter the individual, the more likely that individual will be able to reproduce. The cross-over occurs by mingling the two solutions together to produce two new individuals. During each generation, there is a small chance for each individual to **mutate**, which will change the individual in some small way.

To use a genetic algorithm, there are several questions that need to be answered:

- How is an individual represented?
- How is an individual's fitness calculated?
- How are individuals selected for breeding?
- How are individuals crossed-over?
- How are individuals mutated?
- How big should the population be?
- What are the "termination conditions"?

Most of these questions have problem-specific answers. The last two, however, can be discussed in a more general way.

The size of the population is highly variable. The larger the population, the more possible solutions there are, which means that there is more variation in the population. Variation means that it is more likely that good solutions will be created. Therefore, the population should be as large as possible. The limiting factor is, of course, the running time of the algorithm. The larger the population, the longer the algorithm takes to run.

The algorithm in Figure 1 has a very vague end point – the meaning of "until the termination conditions are met" is not immediately obvious. The reason for this is that there is no one way to end the algorithm. The simplest approach is to run the search for a set number of generations – the longer you can run it, the better. Another approach is to end the algorithm after a certain number of generations pass with no improvement in the fitness of the best individual in the population. There are other possibilities as well.

Since most of the other questions are dependent upon the search problem, we will look at two example problems that can be solved using genetic algorithms: finding a mathematical function's maximum and the traveling salesman problem.

2. FUNCTION MAXIMIZATION

One application for a genetic algorithm is to find values for a collection of variables that will maximize a particular function of those variables. While this type of problem could be solved in other ways, it is useful as an example of the operation of genetic algorithms as the application of the algorithm to the problem is fairly straightforward. For this example, let's assume that we are trying to determine the variables that produce the maximum value for this function:

$$f(w, x, y, z) = w^3 + x^2 - y^2 - z^2 + 2yz - 3wx + wz - xy + 2$$

This could probably be solved using multi-variable calculus (although this author's skills in that area are pretty rusty!), but it is a good simple example of the use of genetic algorithms. To use the genetic algorithm, we need to answer the questions listed in the previous section.

2.1. How is an individual represented?

What information is needed to have a "solution" to the maximization problem we are working on? Hopefully it is clear that all we need is to have values for w, x, y, and z. Assuming that we have values (any values) for these four variables, we have a candidate solution for our problem.

The question is how to represent these four values. A simple way to do this is to simply have an array of four values (integers or floating point numbers, either way). However, for genetic algorithms it is usually best to have a larger individual – this way, variations can be done in a more subtle way. The research shows [Beasley, Bull, and Martin 1993; Holland 1975] that using individuals represented by bit strings offers the best performance. So, we can simply choose a size in bits for each variable, and then concatenate the four values together into a single bit string.

For example, we will choose to represent each variable as a four-bit integer, making our entire individual a 16-bit string. This is obviously very simplistic, and in reality we would probably want to have the variables represented as floating point values with more precision, but for an example it should work. Thus, an individual such as

```
1101 0110 0111 1100
```

represents a solution where $w = 13$, $x = 6$, $y = 7$, and $z = 12$.

2.2. How is an individual's fitness calculated?

Next we consider how to determine the fitness of each individual. There is generally a differentiation between the ***fitness*** and ***evaluation*** functions. The evaluation function is a function that returns an absolute measure of the individual. The fitness function is a function that measures the value of the individual relative to the rest of the population.

In our example, an obvious evaluation function would be to simply calculate the value of f for the given variables. For example, assume we have a population of 4[1] individuals:

```
1010 1110 1000 0011
0110 1001 1111 0110
0111 0110 1110 1011
0001 0110 1000 0000
```

[1] In practical genetic algorithms, the population should be much higher.

The first individual represents $w = 10$, $x = 14$, $y = 8$, and $z = 3$, for an f value of 671. The values for the entire population can be seen in the following table:

Individual	***w***	***x***	***y***	***z***	***f***
1010111010000011	10	14	8	3	671
0110100111110110	6	9	15	6	-43
0111011011101011	7	6	14	11	239
0001011010000000	1	6	8	0	-91

The fitness function can be chosen from many options. For example, the individuals could be listed in order from lowest to highest evaluation function values, and an ordinal ranking applied. Or, the fitness function could be the individual's evaluation value divided by the average evaluation value[2]. Looking at both of these approaches would give us something like this:

Individual	***evaluation***	***ordinal***	***averaging***[3]
1010111010000011	671	4	2.62
0110100111110110	-43	2	0.19
0111011011101011	239	3	0.81
0001011010000000	-91	1	0.03

The key is that the fitness of an individual should represent the value of the individual relative to the rest of the population, so that the best individual has the highest fitness.

2.3. How are individuals selected for breeding?

The key to the selection process is that it should be probabilistically weighted so that higher fitness individuals have a higher probability of being selected. Other than these specifications, the method of selection is open to interpretation.

One possibility is to use the ordinal method for the fitness function, then calculate a probability of selection that is equal to the individual's fitness value divided by the total fitness of all the individuals. In the example above, that would give the first individual a 40% chance of being selected, the second a 20% chance, the third a 30% chance, and the fourth a 10% chance. Clearly this is giving the better individuals more chances to be selected.

A similar approach could be used with the average fitness calculations. This would give the first individual a 72% chance, the second a 5% chance, the third a 22% chance, and the fourth a 1% chance. This method makes the probability more dependent on the relative evaluation functions of each individual.

2.4. How are individuals crossed-over?

[2] Care should be taken with this method when dealing with negative numbers. It is a bad idea to have negative fitness values, so we should adjust the numbers to make sure that no fitness values are negative.

[3] Each evaluation value had 100 points (an arbitrary number) added before the calculations were done.

Once we have selected a pair of individuals, they are "bred" – or in genetic algorithm language, they are *crossed-over*. Typically two children are created from each set of parents. One method for performing the cross-over is described here, but there are other approaches. Two locations are randomly chosen within the individual. These define corresponding substrings in each individual. The substrings are swapped between the two parent individuals, creating two new children. For example, let's look at our four individuals again:

```
1010 1110 1000 0011
0110 1001 1111 0110
0111 0110 1110 1011
0001 0110 1000 0000
```

Let's assume that the first and third individuals are chosen for cross-over (making sense, as these are the two top individuals). Keep in mind that the selection process is random, however. The fourth and fourteenth bits are randomly selected to define the substring to be swapped, so the cross-over looks like this:

`1010111010000011`   101***01110100000***11   `1011011011101011`

→   →

`0111011011101011`   011***10110111010***11   `0110111010000011`

Thus, two new individuals are created. We should keep creating new individuals until we have created enough to replace the entire population – in our example, we need one more cross-over. Assume that the first and fourth individuals are selected this time. Note that an individual may be selected multiple times for breeding, while other individuals might never be selected. Further assume that the eleventh and sixteenth bits are randomly selected for the cross-over point. We would see a second cross-over like this:

`1010111010000011`   1010111010***000011***   `1010111010000000`

→   →

`0001011010000000`   0001011010***000000***   `0001011010000011`

So, our second generation population is:

```
1011 0110 1110 1011
0110 1110 1000 0011
1010 1110 1000 0000
0001 0110 1000 0011
```

2.5. How are individuals mutated?

Finally, we need to allow individuals to mutate. When using bit strings, the easiest way to implement the mutation is to allow every single bit in every individual a chance to mutate. This chance should be very small, since we don't want to have individuals changing dramatically due to mutation. Setting the percentage so that roughly one bit per individual will change on average is probably a reasonably good number.

The mutation will consist of having the bit "flip" – a 1 changes to a 0 and a 0 changes to a 1. In our example, assume that the bold and italicized bits have been chosen for mutation:

1011011011101011 → 1011011011101011
01101***1***1010000011 → 01101***0***1010000011
10101110100***0***0000 → 10101110100***1***0000
0***0***010110100000***1***1 → 0***1***010110100000***0***1

2.6. Wrapping Up

Finally, let's look at the new population:

Individual	*w*	*x*	*y*	*z*	*f*
1011011011101011	11	6	14	11	1,045
0110101010000011	6	10	8	3	51
1010111010010000	10	14	9	0	571
0101011010000001	5	6	8	1	-19

The average evaluation value here is 412, versus an average of 194 for the previous generation. Clearly, this is a constructed example, but the exciting thing about genetic algorithms is that this sort of improvement actually does occur in practice, although the amount of improvement tends to level off as the generations continue.

3. THE TRAVELLING SALESMAN PROBLEM

Now let's look at a less contrived example. Genetic algorithms can be used to solve the traveling salesman problem (TSP). For those who are unfamiliar with this problem, it can be stated in two ways. Informally, there is a traveling salesman who services some number of cities, including his home city. He needs to travel on a trip such that he starts in his home city, visits every other city exactly once, and returns home. He wants to set up the trip so that it costs him the least amount of money possible. The more formal way of stating the problem casts it as a graph problem. Given a weighted graph with N vertices, find the lowest cost path from some city v that visits every other node exactly once and returns to v. For a more thorough discussion of TSP, see [Garey and Johnson, 1979].

The problem with TSP is that it is an NP-complete problem. The only known way to find the answer is to list every possible route and find the one with the lowest cost. Since there are a total of $(N - 1)!$ routes, this quickly becomes intractable for large N. There are approximation algorithms that run in a reasonable time and produce reasonable results – a genetic algorithm is one of them.

3.1. Individual Representation and Fitness

Our first step is to decide on a representation for an individual candidate solution, or **tour**. The bit string model is not very useful, as the cross-overs and mutations can easily produce a tour that is invalid – remember that every city must occur in the tour exactly once except the home city.

The only real choice for representing the individual is a vector or array of cities, most likely stored as integers. The costs of travel between cities should be provided – when assigning this as a project to our students, we use a text file containing a cost matrix. Using a vector of integers causes some problems with cross-over and mutation, as we'll see in the next section.

For example, the following file defines a TSP with four cities:

```
4
0 2 6 3
2 0 9 7
6 9 0 8
3 7 8 0
```

This file shows that the cost of travel from city 0 to city 2 is 6, while the cost from city 3 to city 1 is 7. Then we could represent an individual as a vector of five cities:

`[0 2 1 3 0]`[4]

We also need to be careful when calculating fitness. The clear choice for the evaluation function is the cost of the tour, but remember that a good tour is one whose cost is low, so we need to calculate fitness so that a low cost tour is a high fitness individual. One way to do this is to find the largest cost edge in the graph (say it has cost K), then set the fitness equal to N * K – (path cost), where N is the number of cities. The makes a tour with a low path cost have a high fitness value.

3.2. Cross-over and Mutation

When performing cross-over and mutation, we need to make sure that we produce valid tours. This means modifying the cross-over and mutation process. We can still use the same basic idea, however – choose a substring of the vector of cities at random for cross-over, and choose a single point in the vector at random for mutation. The mechanics are a bit different, though.

For cross-over, rather than simply swapping the substrings (which could easily result in an invalid tour), we will instead keep the same cities, but change their order to match the other parent of the cross-over. For example, assume we have the following two individuals, with the third and sixth cities chosen for the cross-over substring[5]

7 3 ***6 1 0 2*** 5 4

5 4 ***1 7 2 3*** 6 0

[4] Practically speaking, the last city can be omitted, since we can assume that the salesman returns to the home city.
[5] Note that the return to the home city is implicitly done after the last city in the vector. Also note that there is no particular home city provided – in reality, the tour is just a cycle through the cities, and any city could be considered as the home city without changing the problem.

Rather than swapping the cities, which would result in duplications of cities within each tour, we keep the cities in each tour the same, but we re-order the cities to match their order in the other parent. In the above example, we would replace the “6 1 0 2” section in the first parent with “1 2 6 0”, because that is the order in which those four cities appear in the second parent. Similarly, the “1 7 2 3” section in the second parent is replaced with “7 3 1 2”, and we get offspring

7 3 ***1 2 6 0*** 5 4

5 4 ***7 3 1 2*** 6 0

This allows the concept of cross-over to remain – that each parent contributes to the construction of the new individuals – while guaranteeing that a valid tour is created.

There are a number of approaches for mutation. The simplest is that whenever a city is chosen as the location of a mutation, it is simply swapped with the next city in the tour. For example, if the 6 is chosen for mutation in this tour:

7 3 1 2 ***6 0*** 5 4

we would get this tour after the mutation occurs:

7 3 1 2 ***0 6*** 5 4

Other options for mutation include selecting two cities at random and swapping them, or selecting an entire substring at random and reversing it, but the concept remains the same – making a relatively small change to an individual.

4. SUMMARY

Genetic algorithms are very useful, particularly in cases where the problem cannot be solved in more traditional ways. They are interesting to show in class, because it does not seem obvious that they will work; but they do!

We particularly enjoy assigning the TSP as a genetic algorithm because the assignment is so open. There are many ways to do things, and there are many variables to choose. How large should the population be? How many generations should it run? How do we do the cross-over? How do we do the mutations? Additionally, there are tweaks that are not “pure” genetic algorithms, but can improve performance. For example, you could write the algorithm so the fittest individual survives into the next generation, or so mutation is only allowed if it is beneficial, or so the cross-over is performed differently, and so on. It gives the students a chance to solve a problem without a lot of guidance, allowing them to be creative.

We present genetic algorithms in our upper-level Artificial Intelligence course. However, they would be appropriate in any class where the basic search methods have already been demonstrated. Practically speaking, we wouldn’t want to cover them before covering some graph theory and algorithms, which would mean that they would fit best at the end of a second course or during a third course in computer science.

5. BIBLIOGRAPHY

"An Overview of Genetic Algorithms: Part 1, Fundamentals", by David Beasley, David R. Bull, and Ralph R. Martin. *University Computing*, volume 15(2), pages 58-69, 1993.

Computers and Intractability: A Guide to the Theory of NP-Completeness, by Michael R. Garey and David S. Johnson. W.H. Freeman and Company, 1979.

Adaptation in Natural and Artificial Systems, by J.H. Holland. MIT Press, 1975.

Artificial Intelligence: A Modern Approach, Second Edition, by Stuart Russell and Peter Norvig. Prentice Hall, 2003.