

A Short Tour of Genetic Algorithms

Mike Siley, Case Western Reserve University, July 24th 2007

What is a Genetic Algorithm (GA)

A Genetic Algorithm is a probabilistic algorithm that emulates the process of evolution from the natural world in order to solve NP-hard problems. In general, there is a population of individuals that is represented at the genetic level with the concepts of genes and chromosomes. The population undergoes a process of selection, similar to natural selection, wherein the more fit individuals prosper and multiply and the less fit individuals die off. With each generation a population undergoes certain transformations such as genetic mutations and genetic crossover (simulating mating) to form the next population. Over multiple generations the chromosomes/solutions converge and it is hoped that the best individual represents a near optimum solution.

Classic GA Components, Terminology and Representation

All GAs have the following components:

- 1) A genetic representation for potential solutions to the problem.
- 2) A way to create an initial population of potential solutions.
- 3) An evaluation function that rates solutions in terms of fitness.
- 4) Genetic operators that alter the composition of the children.
- 5) Values for various parameters such as:
 - a. Population Size
 - b. Crossover Probability
 - c. Mutation Probability
 - d. Number of Generations

The classic genetic algorithm uses a bit (1 or 0) to represent a gene and a string of bits to represent a chromosome. What the chromosome represents is determined by the evaluation function. In biological terminology a chromosome is the genotype or potential solution and the meaning of the chromosome is the phenotype.

The simplest case binary string representation of a chromosome would be the transformation of the binary string to its natural integer representation, such as 01010101 -> 85. A more complicated case would be that of a chromosome that represents a floating-point number. To determine the necessary length of the chromosome necessary to represent a floating-point number we can use the following formulation:

Precision n of the real number

Range of the function $R = [a, b]$

Let m be the smallest integer such that:

$$(b - a) * 10^n \leq 2^m - 1$$

m will be the length of the bit string necessary to represent the floating point number.

Example:

$$n = 4$$

$$R = [4.1, 5.8]$$

$$(5.8 - 4.1) * 10^4 = 17000$$

$$2^{14} < 17000 \leq 2^{15}$$

Therefore: $m = 15$

To decode a binary string such as 111110010100010

```
x = a + decimal(bitstring) * (b-a)/(2^15 - 1)
decimal(111110010100010) = 70352
x = 4.1 + 70352 * (5.8-4.1)/(2^15-1)
x = 5.755330
```

To represent a function with multiple parameters such as $\sin(x) + y$ the binary string would be segmented upon evaluation. So if 10 bits would be needed to represent a floating point number for this particular problem then the bit string would have a length of 20. The first 10 bits would be the value of x and the second 10 the value of y .

Basic Structure of a GA

Given time period t with population P the classic genetic algorithm is described in pseudo-code below.

```
procedure genetic algorithm
begin
    t <- 0
    1) initialize P(t)
    2) evaluate P(t)
    while (not termination-condition) do
    begin
    3)     t <- t + 1
    4)     select P(t) from P(t-1)
    5)     crossover P(t)
    6)     mutate P(t)
    7)     evaluate P(t)
    end
end
```

We will step through the above algorithm and demonstrate the basic functions of a GA. For simplicity we will assume that the evaluation function converts a bit string directly into its integer form.

Parameters:

```
pop_size = 5           // Population size
ngenes = 5             // Length of each chromosome
mutate_prob = 0.01     // probability of mutation
cross_prob = 0.45      // probability of crossover
generations = 1,000
```

1) initialize P(t)

Randomly create 5 chromosomes of length 5

```
[1](10001), [2](11100), [3](01010), [4](01101), [5](11010)
```

2) evaluate P(t)

Run each chromosome against the evaluation function, determine the best solution so far and save the best fit chromosome (in bold).

[1](17.0000), [2](**28.0000**), [3](10.0000), [4](13.0000),
[5](26.0000)

3) t <- t + 1

Increment the time/generational parameter

4) select P(t) from P(t-1)

Selection Process:

This is where the next generation of chromosomes is decided upon based on the fitness of each chromosome. The more fit the chromosome the more likely it is to make it to the next generation. Also, highly fit individuals may be represented in the next generation multiple times whereas less fit individuals may not make it at all.

Notes:

v is the population vector.

v[0] would be the first chromosome in the population.

p[i] is the probability of the ith chromosome.

f[i] is the fitness of the ith chromosome.

F is the fitness of the entire population.

q[i] is the cumulative probability at the ith chromosome.

r is a randomly generated number between 0 and 1.

1) Calculate the fitness value for each chromosome

f[i] = eval(v[i])

Population Fitness:

[1](17.0000), [2](28.0000), [3](10.0000), [4](13.0000),
[5](26.0000)

2) Find the total fitness of the population

F = Sum(v), Total Fitness = 94

2) Calculate the probability of selection by relating an individual's fitness against the population's.

p[i] = f[i]/F

Probability of Selection:

[1](0.1809), [2](0.2979), [3](0.1064), [4](0.1383),
[5](0.2766)

3) Calculate the cumulative probability across the population.

q[i] = Sum(p[j], i)

Cumulative Probability:

[1](0.1809), [2](0.4787), [3](0.5851), [4](0.7234),
[5](1.0000)

Now that we have constructed the cumulative probability we can begin the chromosome selection process for the next generation.

- 5) Generate a Random r float between 0 and 1.
- 6) if $r < q[0]$ select $v[0]$ else select i th chromosome such that $q[i-1] < r \leq q[i]$. Do this until the original the population size is reached.

Resulting in the selected population.

v(1)' r = 0.341205464898 [1](11100)
v(2)' r = 0.30034236493 [1](11100)
v(3)' r = 0.612695926359 [3](01101)
v(4)' r = 0.156713283731 [0](10001)
v(5)' r = 0.204714142854 [1](11100)

5) crossover P(t)

Now we mate the chromosomes, first we need to determine which chromosomes are to be crossed over using the *cross_prob* parameter then we determine which part of the chromosome is crossed over.

For each chromosome

- 1) Generate a random float r from 0 to 1
- 2) if $r < \text{cross_prob}$ select chromosome for crossover.
- 3) mate selected chromosomes randomly.

If we end up with an odd number of chromosomes we "flip a coin" to determine whether to add or delete a chromosome to the crossover list. If we are to add a chromosome we randomly select a chromosome from the left over population. If we are to remove a chromosome, that is determined randomly also.

For our example we have a single crossover of two chromosomes as follows:

Chromosome #4 (C1) and #5 (C2) was chosen for crossover

Crossover at Position 3

C1 = v(4) (10001)

C2 = v(5) (11100)

X1: (10000)

^

X2: (11101)

^

The new chromosomes X1 and X2 replace C1 and C2 respectively.

Currently our population is as follows with the two crossed over chromosomes in bold.

v(1) [1](11100)

v(2) [1](11100)
v(3) [3](01101)
v(4) [0](10000)
v(5) [1](11101)

6) mutate P(t)

For our example mutation is the random changing of a bit (either from 0 to 1 or 1 to 0). Each gene of each chromosome has an equal probability of being mutated. The probability of a bit being flipped is determined by the `mutate_prob` parameter.

For each bit

- 1) generate a random float from 0 to 1
- 2) if $r < \text{mutate_prob}$ then mutate that bit

For our example we have $5 * 5 = 25$ bits.

Assume $r < \text{mutate_prob}$ for bit #11.

Bit number 11 is the 1st bit in chromosome #3, which is 01101 so, that means we flip the 1st bit from 0 to 1 resulting in 11101.

After mutation the resulting population is as follows with the mutated bit shown in bold.

v(1) (11100)
v(2) (11100)
v(3) (**1**1101)
v(4) (10000)
v(5) (11101)

7) evaluate P(t)

Just like step #2 we evaluate the population against the evaluation function and then start at step #3 and repeat the process for the specified number of generations.

After 1,000 generations, for this problem, we will hopefully end up with an optimal or near optimal solution. For this problem the obvious optimal solution is 11111 giving us the largest decimal number possible with this bit string representation.

GAs: Why Do They Work?

There are no mathematical proofs that show why GAs work. There is however much empirical evidence which has resulted in the Schema Theorem and the Building Block Hypothesis, which are stated below.

Schema Theorem

Short, low-order, above average schemata receive exponentially increasing trials in subsequent generations of a GA.

Building Block Hypothesis

A GA seeks near-optimal performance through the juxtaposition of short, low-order, high-performance schemata, called the building blocks.

Schema

A schema is a pattern that matches one to many chromosomes. In a classical GA a schema is made of a 1 or 0 bit or a '*' which represents either a 1 or a 0 bit. If r is the number of '*' in a schema then the schema will match 2^r schemas. Examples of schemas are below:

(100*0****1) this would match 32 (2^5) chromosomes
(10000*1111) matches 2 chromosomes
(100*0**110) matches 8 (2^3) chromosomes
(1000101010) matches one chromosome

Order and Length

The order of a schema is the number of fixed positions, which is simply the number of static bits.

S1 = (**001*110) Order(S1) = 6
S2 = (****00**0*) Order(S2) = 3
S3 = (11101**001) Order(S3) = 8

The length of a schema is the distance between the first and the last fixed string positions.

S1 = (**001*110)
 ^ ^
Length(S1) = 10 - 4 = 6

S2 = (****00**0*)
 ^ ^
Length(S2) = 5 - 9 = 4

S3 = (11101**001)
 ^ ^
Length(S3) = 10 - 1 = 9

Schema Performance

We find that not only do above average schema increase during the selection process but they do so exponentially, whereas below average schema decrease. We can understand the exponential manner that schema may increase by examining what is called the Reproductive Schema Growth Equation.

Reproductive Schema Growth Equation

Given a schema S and time t :

Let $e(S,t)$ = number of chromosomes a schema matches in a population at time t .

The fitness at time t of the schema S is (let's assume S matches 3 chromosomes.)

$eval(S, t) = f(x1) + f(x2) + f(x3) / 3$

We can then find the number of schema that match schema S for t+1 with the following equation:

$$e(S, t+1) = e(S, t) * \text{eval}(S, t) / (F(t) / \text{pop_size})$$

where $F(t)$ = total fitness of population at time t.

Now with an example:

Assume:

$$e(S, t) = 3$$

$$\text{pop_size} = 20$$

$$\text{eval}(S, t) = 27.0814$$

$$F(t)/\text{pop_size} = 38.7768/20 = 19.3888$$

Then $t = 0$

$$\text{eval}(S, 0)/(F(0)/\text{pop_size}) = 1.396751$$

Given the information at time 0 we can find how many the schema S is expected to match at time t+1.

With the information above we can find potential matching numbers with the general formula:

$$e(S, t+1) = e(S, t) * [\text{eval}(S, 0)/(F(0)/\text{pop_size})]^2$$

Examples:

$$e(S, 0+1) = (3 * 1.396751) = 4.19 \text{ (will match 4-5 strings)}$$

$$e(S, 0+2) = (3 * 1.386751^2) = 5.85 \text{ (will most likely match 6 strings)}$$

$$e(S, 0+3) = (3 * 1.386751^3) = 8.00$$

$$e(S, 0+4) = (3 * 1.386751^4) = 11.09$$

Effects of Crossover

Using order and length we can determine the effects of crossover on a given schema.

Probability of Destruction (Pd) of a Schema at Crossover

$$Pd(S) = \text{length}(S)/(m-1) = \text{where } m \text{ is length of chromosome.}$$

Probability of Survival at Crossover

$$Ps(S) = 1 - Pd(S)$$

Probability of Crossover (Pc) Survival of a Schema is

$$Ps(S) \geq 1 - Pc (\text{length}(S) / (m-1))$$

Now the Reproductive Schema Growth Equation is becomes:

$$e(S, t+1) \geq e(S, t) * \text{eval}(S, t) / \text{Avg}(F(t)) [1 - Pc (\text{length}(S) / (m-1))]$$

The Effects of Mutation

P_m = probability of mutation

$$Ps(S) = 1 - \text{order}(S) * P_m$$

Now Reproductive Schema Growth Equation becomes:

$$e(S, t+1) \geq e(S, t) * \text{eval}(S,t)/\text{Avg}(F(t)) [1-P_c (\text{length}(S) / (m-1)) - \text{order}(S) * P_m]$$

Using the Reproductive Schema Growth Equation we can show the effects of selection, crossover, and mutation of a schema and determine the number of strings it is likely to match (thus determining it's fitness) in future generations. Extrapolating from this information you can see that short, low-order above average fitness schemas will do better probabilistically than the opposite.

The Traveling Salesman Problem: An example of a GA solving a commonly researched problem.

There are three main ways to encode a chromosome for solving the Traveling Salesman Problem (TSP): 1) Binary String, 2) Vector, and 3) Matrix data structures. Using a binary string representation a section of bits in the binary string would represent a city. The adjacent bits would be the next city in the tour. For example if there are four cities in the tour then you would need 2 bits for each city which would result in a bit string with 8 bits as follows:

- 00 City 1
- 01 City 2
- 10 City 3
- 11 City 4

Therefore, using the traditional format of a binary string you could solve the TSP. The downside of this format is that with the transforming operations of mutation and crossover it is very likely that there will be an invalid tour. For instance, say after mutation you end up with a binary string such as 11110110 which is invalid since the tour decodes to 4-4-2-3 resulting in a redundant node and an incomplete tour. To compensate for this fact a separate operation would be needed to repair the tour. There is also the issue with rather large strings resulting from a large number of nodes. For a tour with 255 nodes you would need a binary string that is 2,040 bits long. Operations performed on these strings would be computationally expensive, resulting in a possible solution but one that takes a considerable amount of time compared to other algorithms available.

The vector representation allows us to efficiently encode a TSP tour without running into the repair and length issues associated with binary strings. There are three common ways to encode a vector: Adjacency, Ordinal, and Path. The Path representation is seemingly the most practical and is used throughout the rest of the paper.

Path Representation

This is the straightforward encoding of the three, which is as follows.

vector (1-5-4-6) represents tour 1-5-4-6-1

The matrix representation has been used effectively in solving the TSP using a GA but is beyond the scope of this paper.

There are many ways to perform the crossover operation all of which lead to creating offspring that have genes from both parents and are valid tours. The OX crossover algorithm due to its straightforward nature and seemingly identical performance with other methods is described below. Other methods that were researched, such as PMX, seemed to be missing information in the described process since it did not always resolve to a valid tour.

OX Crossover

- 1) Choose 2 random cut points.
- 2) Copy each section between the cut points to the offspring.

Parent 1 = (1 2 3 | 4 5 6 7 | 8 9)

Parent 2 = (4 5 2 | 1 8 7 6 | 9 3)

Child 1 = (x x x | 4 5 6 7 | x x)

Child 2 = (x x x | 1 8 7 6 | x x)

- 4) Starting with the opposite parent after the cut point, collect the non-duplicate genes and enter them into the child.

For Child 1 = (9 3 2 1 8)

For Child 2 = (9 2 3 4 5)

Resulting in the completed Offspring

Child 1 = (2 1 8 4 5 6 7 9 3)

Child 2 = (3 4 5 1 8 7 6 9 2)

Mutation for the TSP problem involves generating a random number r for each node and if $r < \text{mutate_prob}$ then swap this node with another, also chosen randomly.

Improving the TSP GA: Classical versus Modified Selection

Below is a modified selection algorithm that improves upon the classical method of selection.

Modified GA Selection Process

The modified version of the selection process attempts to utilize higher fitness chromosomes for genetic operations explicitly and hopes that their offspring will be more viable than lower ranking chromosomes.

- 1) Sort Chromosomes by Fitness (higher to lower)
- 2) Separate chromosomes for genetic operations that are at or above specified cutoff point, c .
- 3) For each chromosome above c generate a random number r and if $r < \text{mutate_prob}$ then mark it for mutation, otherwise mark it for crossover. If there are an uneven number of crossover chromosomes add one from the mutation list.

- 4) Copy the parents back into the population.
- 5) Perform the genetic operations against the marked chromosomes and copy their offspring into the population.
- 6) Copy $\text{pop_size} - c * 2$ chromosomes starting at index $c * 2$ into the population. This results in the original population size.

TSP Implementations and Performance Comparisons

Using the Path representation for a TSP tour and the OX crossover discussed previously, two GA programs were developed. The first uses the classical selection process and the second the modified selection process. Appendix A shows tables of trial runs of both methods, here are the summary results.

Test Tour of 10 cities, optimal value = 10

Standard GA

P(Crossover): 65%
P(Mutation): 1.5%
Max Generations 5,000

10 Trials for each Population

<u>Population</u>	<u>Avg. Generations</u>	<u>Avg. Best Fit</u>
50	4,673	13.9
100	4,729	13.3
250	3,765	12.3

Modified GA

Cutoff: 30%
P(Mutation): 1.5%
Max Generations 5,000

10 Trials for each Population

<u>Population</u>	<u>Avg. Generations</u>	<u>Avg. Best Fit</u>
50	4,003	13.7
100	2,007	11.2
250	509	10.1

As shown above the Modified GA selection process vastly improves upon the performance of the classical process across various population levels. Larger than normal populations were chosen because after some research they seemed to increase the performance dramatically with the

Modified GA. Also, there has been much study on GAs with varying populations, which perform better than static GAs. My conjecture is that they perform better because they allow for a larger gene pool.

Using the Modified GA I took a TSP data set found on the site below that represented real cities and their distances from one another.

Website with various TSP tours: <http://www.tsp.gatech.edu/world/countries.html>

29 locations in Western Sahara

Derived from National Imagery and Mapping Agency data

Data set: <http://www.tsp.gatech.edu/world/wi29.tsp>

Modified Parameters are in bold.

Optimal Tour: 27,603

Algorithm: Modified GA
 Population: **250**
 P(Mutation): **1.05%**
 Cutoff 30.00%

Trial	Time	ConvGen*	BestFit	% from Opt.**	
1		13	71	38,530	39.59%
2		23	129	36,268	31.39%
3		18	103	38,522	39.56%
4		19	111	38,081	37.96%
5		13	74	42,982	55.72%
Avg.		17.2	97.6	38,877	40.84%

Algorithm: Modified GA
 Population: **500**
 P(Mutation): **40.00%**
 Cutoff 40.00%

Trial	Time	ConvGen*	BestFit	% from Opt.**	
1		48	131	34,644	25.51%
2		82	203	34,663	25.58%
3		61	138	33,749	22.27%
4		37	96	34,756	25.92%
5		38	89	35,890	30.02%
Avg.		53.2	131.4	34,741	25.86%

*Converging Generation (est.)
**Percent from Optimal
Time is in seconds.

Increasing the population and the mutation probability seems to have improved the overall performance of the algorithm, getting within 26% of the optimal tour on average. It was hoped that the increased mutation probability slowed down the solution convergence as the optimal tour was approached, thus allowing for a solution nearer the optimal.