Genetic Algorithm

B ANIRUDH SRINIVSAN
COE17B019

1. Genetic Algo Operators

1.1 Selection Operator

Selection is the stage of a genetic algorithm in which individual genomes are chosen from a population for later breeding (using the crossover operator).

A generic selection procedure may be implemented as follows:

1. The fitness function is evaluated for each individual, providing fitness values, which are then normalized. Normalization means dividing the fitness value of each individual by the sum of all fitness values, so that the sum of all resulting fitness values equals 1.

2. The population is sorted by ascending fitness values.

3. Accumulated normalized fitness values are computed: the accumulated fitness value of an individual is the sum of its own fitness value plus the fitness values of all the previous individuals; the accumulated fitness of the last individual should be 1, otherwise something went wrong in the normalization step.

4. A random number R between 0 and 1 is chosen.

5. The selected individual is the first one whose accumulated normalized value is greater than or equal to R.

There are other selection algorithms that do not consider all individuals for selection, but only those with a fitness value that is higher than a given (arbitrary) constant. Other algorithms select from a restricted pool where only a certain percentage of the individuals are allowed, based on fitness value.

Retaining the best individuals in a generation unchanged in the next generation, is called elitism or elitist selection. It is a successful (slight) variant of the general process of constructing a new population.

1.1.1 Roulette Wheel Selection

Parents are selected according to their fitness. The better the chromosomes are, the more chances to be selected they have. Imagine a roulette wheel where are placed all chromosomes in the population, every has its place big accordingly to its fitness function, like on the following picture.

Then a marble is thrown there and selects the chromosome. Chromosome with bigger fitness will be selected more times.

$$p(i) = \frac{fitness(i)}{\sum_{j \in genes} fitness(j)} \tag{1}$$

where,

- p(i) is the probability of selecting the i^{th} gnome.
- fitness(i) is the fitness value of i^{th} gnome.

For example, for fitness values (100, 25, 50, 25), the probability of selecting them are (0.5, 0.125, 0.25, 0.125). With these population prob we choose the N genes for crossover, where N is the population size.

So randomly selected gnomes are (100, 100, 50, 25).

1.1.2 **Rank Selection**

Rank selection first ranks the population and then every chromosome receives fitness from this ranking. The worst will have fitness 1, second worst 2 etc. and the best will have fitness N (number of chromosomes in population).

You can see in following picture, how the situation changes after changing fitness to order number.

After this all the chromosomes have a chance to be selected. But this method can lead to slower convergence, because the best chromosomes do not differ so much from other ones.

The rank n is accorded to the best individual whilst the worst individual gets the rank 1. Thus, based on its rank, each individual i has the probability of being selected given by the expression

$$p(i) = \frac{rank(i)}{n(n-1)} \tag{2}$$

where,

- p(i), is the probability of i^{th} chromosome.

For example, for fitness score of (100, 25, 50, 25), we rank them as (4, 2, 3, 1), and get the probabilities as (0.33, 0.16, 0.25, 0.083).

So randomly selected gnomes are (100, 50, 25, 100).

1.1.3 **Tournament Selection**

Tournament selection is a variant of rank-based selection methods. Its principle consists in randomly selecting a set of k individuals. These individuals are then ranked according to their relative fitness and the fittest individual is selected for reproduction. The whole process is repeated n times for the entire population. Hence, the probability of each individual to be selected is given by the expression:

$$p(i) = \begin{cases} \frac{C_{n-1}^{k-1}}{C_n^k} & \text{if } i \in [1, n-k-1] \\ \\ 0 & \text{if } i \in [n-k, n] \end{cases}$$

For example, for fitness score of (100, 25, 50, 25, 75), lets take k as 2 and get the The sorted fitness score is (100, 75, 50, 25, 25) probabilities as (0.4,0.4, 0, 0, 0)

Another way of doing this would be to select 2 random gnome and select the best one out of them, until we reach a desired number of gnomes.

For our example,

Table 1: Initial Generation.

Iteration	Chosen Candidates	best
1	[75, 50]	75
2	[50, 25]	50
3	[100, 50]	100
4	[75, 100]	100
5	[25, 75]	75

Therefore, the final gnomes would be (75, 50, 100, 100, 75).

1.2 Boltzmann Selection

In Boltzmann selection, the rate of selection is controlled by a continuously varying temperature. Initially the temperature is high and selection pressure is inversely proportional to temperature. So selection pressure is low initially. The temperature is decreased gradually which increases the selection pressure. This results in narrowing of search space along with maintaining the diversity in population. The selection of an individual is done with Boltzmann probability which is given by,

$$p(i) = e^{-\frac{fmax - f(X_i)}{T}} \tag{3}$$

where,

- fmax is the maximum fitness of that population.
- $f(X_i)$ is the fitness of that chromosome.
- T = $T_o(1-\alpha)^k$, k = $(1 + 100 * \frac{g}{G})$
- α a constant.
- g, current generation number.
- G, maximum generation number.

In Boltzmann selection, the probability of selecting best string for mating is very high. Execution time of this technique is also very less. However by using this technique, certain information may be lost during mutation stage. But this can be prevented through elitism.

For our example, let the current generation be (100, 25, 50, 25) and initial temperature = 100. Let the $\alpha = 1.009$ and k = 11 (initial value).

The probabilities are (1, 0.47, 0.60, 0.47).

So randomly selected gnomes are (100, 100, 50, 25). $T_{new} = 100 * (1 - 1.009)^{11} = 90$

1.3 Cross-over Operator

In genetic algorithms and evolutionary computation, crossover, also called recombination, is a genetic operator used to combine the genetic information of two parents to generate new offspring. It is one way to stochastically generate new solutions from an existing population, and analogous to the crossover that happens during sexual reproduction in biology. Solutions can also be generated by cloning an existing solution, which is analogous to asexual reproduction. Newly generated solutions are typically mutated before being added to the population.

Different algorithms in evolutionary computation may use different data structures to store genetic information, and each genetic representation can be recombined with different crossover operators. Typical data structures that can be recombined with crossover are bit arrays, vectors of real numbers, or trees.

1.3.1 Single point crossover

One crossover point is selected, binary string from beginning of chromosome to the crossover point is copied from one parent, the rest is copied from the second parent For example, **1100**1011+11011**111** = **11001111**

Figure 1: Illustration of single crossover method.

1.3.2 Two point crossover

Two crossover point are selected, binary string from beginning of chromosome to the first crossover point is copied from one parent, the part from the first to the second crossover point is copied from the second parent and the rest is copied from the first parent For example, **11**001011 + 11**011**111 = **11011111**

Figure 2: Illustration of double-point crossover method.

1.3.3 Uniform crossover

Bits are randomly copied from the first or from the second parent For example, 1**1**0010**1**1 + **1**1011**1**01 = **11011111**

Figure 3: Illustration of uniform crossover method.

1.4 Arithmetic crossover

Some arithmetic operation is performed to make a new offspring. For example, 11001011 + 11011111 = 11001001 (AND)

1.5 Mutation

The mutation operator can be applied to either function or terminal nodes. This operator can modify one or many nodes. Given a selected individual, the mutation operator first randomly selects a node in the tree representation of the individual. Then, if the selected node is a terminal (source or target metamodel element), it is replaced by another terminal (another metamodel element).

If the selected node is a function (e.g., AND operator), it is replaced by a new function (i.e., AND becomes OR). If a tree mutation is to be carried out, the node and its subtrees are replaced by a new randomly generated subtree.

1.5.1 Bit Flip Mutation

In this bit flip mutation, we select one or more random bits and flip them. This is used for binary encoded GAs.

For example, if out gene is "1001" and if we flip 3^{rd} position, we get "1011".

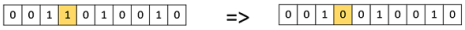

Figure 4: Illustration of bitflip mutation method.

1.5.2 Scramble Mutation

Scramble mutation is also popular with permutation representations. In this, from the entire chromosome, a subset of genes is chosen and their values are scrambled or shuffled randomly.

For example, let the gnome be "5857394753", if we choose a subset "739" and scramble it randomly into "973", the mutated gene would be "5859734753". Note that, for this example, each bit in the gnome can take values from 0 to 9.

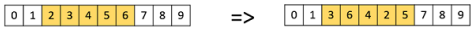

Figure 5: Illustration of scramble mutation method.

1.5.3 **Inversion Mutation**

In inversion mutation, we select a subset of genes like in scramble mutation, but instead of shuffling the subset, we merely invert the entire string in the subset.

For example, let the gnome be "5857394753", if we choose a subset "739" and the inverted string of the subset is "937", the mutated gene would be "5859374753". Note that, for this example, each bit in the gnome can take values from 0 to 9.

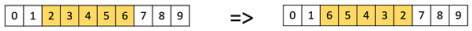

Figure 6: Illustration of inversion mutation method.

1.6 Swap Mutation

In swap mutation, we select two positions on the chromosome at random, and interchange the values. This is common in permutation based encodings.

For our example, let the gnome be "029321" and randomly chosen position to swap is, 4 and 1, the mutated string is "329021".

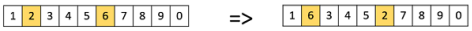

Figure 7: Illustration of swap mutation method.

2. Travelling Salesman Using GA

2.1 Problem Statement

Given a set of cities and distance between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point.

2.2 Algo

1. Initialize the population randomly.

2. Determine the fitness of the chromosome.

3. Until done repeat:

   (a) Select parents.

   (b) Perform crossover and mutation.

   (c) Calculate the fitness of the new population.

   (d) Append it to the gene pool.

2.3 Example

For our example, Lets take,

Table 2: Example dataset, note that, this is a distance matrix.

	0	1	2	3	4
0	0	2	∞	12	5
1	2	0	4	8	∞
2	∞	4	0	3	3
3	12	8	3	0	10
4	5	∞	3	10	0

Let the encoding be in such a way that 012340 means he is travel travelled from 0 to 1 to 2 to 3 to 4 to 0. And the cost of this path will be fitness.

Let the Population size be 10. Initial Temperature be 10000.

And before mutating each generation, lets sort based on fitness in ascending order.

We are gonna run this algo for 5 generations.

The New temperature = 0.9*(Old temperature)

Table 3: Initial Generation.

Gnome	Fitness
043210	24
023410	∞
031420	∞
034210	31
043210	24
023140	∞
032410	∞
012340	24
012340	24
032410	∞

For our mutation, lets take a 2 random position of a gene in the current population and swap them.

This new gene will only get accepted if its fitness is less than its parents, or if its boltzmann probability is greater than 0.5.

So, our current gnome is 043210 with fitness score of 24. The first mutated gene is 043120 with fitness score of ∞.

Clearly, the fitness of the child is greater than the parent, therefore we need to calculate the probability. By formula, we get prob = 0.

Therefore, we reject this child and calculate another,

The next mutated child gnome is 013240 with fitness score of 21. As the child is fitter than the parent, we send them to the next gen.

Similarly do for the rest, To show the use of probability, So, our current gnome is 012340 with fitness score of 24. The first mutated gene is 012430 with fitness score of 31.

Clearly, the fitness of the child is greater than the parent, therefore we need to calculate the probability. By formula, we get prob = 0.999305.

Therefore, we accept this child into the next generation.

This way, we get,

Table 4: Gen 1

Gnome	Fitness
031240	32
043210	24
012340	24
042130	32
043210	24
012340	24
034210	31
014320	∞
014320	∞
023140	∞

$T_{new} = 9000$

Table 5: Gen 2

Gnome	Fitness
013240	21
013240	21
012430	31
012430	31
031240	32
024310	∞
013420	∞
032140	∞
034210	31
012430	31

$T_{new} = 8100$

Table 6: Gen 3

Gnome	Fitness
013240	21
042310	21
013240	21
013240	21
031240	32
013240	21
012430	31
034120	∞
041320	∞
043120	∞

$T_{new} = 7290$

Table 7: Gen 4

Gnome	Fitness
031240	32
043210	24
043210	24
043210	24
012340	24
042130	32
013240	21
014320	∞
021340	∞
043210	24

$T_{new} = 6561$

Table 8: Gen 5

Gnome	Fitness
043210	24
042310	21
042310	21
013240	21
042310	21
034210	31
013240	21
042310	21
024310	∞
024310	∞

Its clear from the 5^{th} generation that the minimum distance is 21 and 034210 can be one of the possible solution.

3. Schema Theorem for Genetic Algorithm

3.1 Introduction

We have seen that Genetic Algorithms are one of the most successful engineering tools ever developed. They are easy to implement. A simple GA and GAs are surprisingly robust against lack of skill. GAs tend to be used when nothing else works. But still questions like why does GA work exists. To answer that, we need to reason rigorously about the operation of GAs. This proof for working of GA is given by schema theorem.

3.2 Some Terms

- **Schema**
  A schema is a pattern comprising 0s, 1s, and *s could be either 0 or 1. Schemas capture particular subsets of strings. * stands for "I don't care" or, if you like, a "0" or a "1". Some examples of schemas are
  - S1 = 0****0 (only beginning and final bits are defined, the rest middle 4 positions may be filled by 16, z^4 = possible strings)
  - S2 = 011 (Every bit is defined, so only one possible string).
- **Instance of a Schema**
  An instance of a schema is a is a bitstring that satisfy the schema. . Thus 010101 is an instance of 010101. 010101 is an instance of 01****. 010101 is an instance of 01**01. 010101 is an instance of ******. 010101 is not an instance of 11****. A string x can be an instance of many schemata.

  And a bitstring can belong to more than one schema. 0 is an instance of S = 0 and also an instance of s = *.
- **Length and order of a Schema**
  The defining length d(S) of a schema S is the distance between the first defined bit and the last defined bit i.e. it is the distance between first and last non * gene is schema S.
  - d(101010) = 5
  - d(**10**) = 1
  - d(**1*1*) = 2.

  The order o(s) of a schema s is the number of defined bits it contains, i.e. order is the number of non * genes in schema S.
  - o(101010) = 6
  - o(**10**) = 2
  - o(**1*1*) = 2.

3.3 Operators in Schema theorem

1. **Selection Operator**
   The probability of an individual k x , samples schema S i.e.p(x_k). For this we assume that

   (a) The probability of x_k is proportional to the number of instances of schema S in the population

   (b) The probability of k x is proportional to the average fitness of schema S relative to the average fitness of all individuals in the population.

   Let us consider the number of instances of S at time t, which is denoted by N(S,t) and the expected number of instances of S at time t is denoted by E(N(S,t)). The observed average fitness of S at time t (i.e. the average fitness of instances of S in the population) is denoted by u(S,t). Then the probability of selecting x_k under proportional selection is :

$$p_k = \frac{F(x_k)}{\sum_{j=1}^{N} F(x_j)} \tag{4}$$

   where,

   - F(x_i) is the fitness of x_i.

   Now, the expected number of instances of schema S at time t is given by,

$$E(N(S,t) = \frac{F(x_k)}{F(t)} \tag{5}$$

   where,

   - F(t) is the average fitness of all strings in population at time t.

   This E(N (s, t)) implies that the schemas with fitness greater (lower) than the average population fitness are likely to account for proportionally more (less) of the population at the next generation. Also it can be concluded that for the accurate estimates of expectation and probability, the population size should be infinite.

2. **Crossover**
   The fitness of individuals in a population cannot be improved by reproduction, therefore let us apply single point crossover to modify the distribution of schema in the population.

   Now consider crossover. Let P_c be the probability that single point cross over will be applied to a string. Instance of schema S is picked as a parent. Schema S survives if one of the offspring is also an instance of schema S.

   Lower bound on probability Sc(s) of survival of schema S is given by :

$$S_c(S) = (1 - p_c) + pr(Survive|cross) * p_c \tag{6}$$

Any of the d(S) points between the first and last defined bits is potentially disruptive. So the probability of randomly choosing a potentially destructive crossing point is for a Schema with total length as L is :

$$p(destructivecrossover) = \frac{d(S)}{L-1} \tag{7}$$

The probability of actually causing disruption must be less than this. And so the probability of surviving crossover given that you do crossover must be greater than

$$pr(Survive|cross) \geq (1 - \frac{d(S)}{L-1})$$

Therefore,

$$S_c(S) \geq (1 - p_c) + (1 - \frac{d(S)}{L-1}) * p_c \tag{8}$$

3. **Mutatuion**

In the special worst case lower bound $p_c = 1$.

Now we consider the disruptive effects of mutation. Mutation is applied gene by gene. Let pm be the probability of flipping any bit in a string. A schema s is disrupted when any defined bit is flipped. There are o(s) bits defined, and so the probability they all non * genes given by survive is,

$$S_m(S) = (1 - p_m)^{o(S)} \tag{9}$$

where,

- p_m is the probability of mutation.

Note that : If a bit is in a position corresponding to a * in the schema template it does matter whether or not it is flipped.

3.4 Schema Theorem

The expected number of schema S at generation t + 1 with proportional selection, single point crossover and genewise mutation is given by,

$$E(N, S, t+1) \geq \frac{u(\hat{S}, t)}{F(t)} N(S, t)(1 - \frac{d(S)}{L-1}) * p_c)(1 - p_m)^{o(S)} \tag{10}$$

3.4.1 Conservatism

Note that the schema theorem is conservative.

1. Only deals with destructive effects.
2. Evolution may also be constructive effects.
3. Non-s schema instances may combine to give s-schema instances.
4. Similar properties are valid for mutation.

3.4.2 Implications of Schema theorem

Schemas, like families, need nourishment and encouragement and careful protective management

1. The more bits in your building block family the more likely one is to go off the rails (causing great frustration and heartache).
2. Genes living far apart are prone to breaking up.
3. Conducting a constructive relationship at a distance is hard.
4. Best results achieved by the family unit huddled together in consecutive positions.

3.4.3 Applications of Schema Theorem

The schema theorem is more applicable at the early stages of a search rather than at the end. Schema theorem indicates that fitter than average schemas are rewarded. The fitter the schema the more it is rewarded.

1. Reward is immediate: you see it in the next generation.
2. Should alert us to one danger immediately.
3. Premature converge of the population.

3.5 Trace for optimisation using Schema Theorem

The optimisation function is,

$$x^3 - 2x^2 + x$$

The constraint on x is :

$$x \in [0, 31]$$

Step 1: Decide the encoding

As x maximum value is 31, we are using 5 bit binary representation of x.

Step 2: Generate Schema

The randomly generated schemas are : ['01*0*', '000*0', '0*1**', '*0*00', '10*11', '01**1']

Step 3: Calculate the average fitness of each schema

The average fitness of a schema is the average of all the instances of that schema.

In our case, the fitness of each schema are : [1073.0, 2147.0, 1680.25, 6074.5, 20793.0, 12018.5]

Step 4: Select schema for crossover

Schemas are seleced in russian roulette fasion with probabilities of getting selected being defined as :

$$p(x_i) = \frac{F(x_i)}{\sum_{j=1}^{N} F(x_j)} \tag{11}$$

In our case, the probability of each schema getting selected are : [0.025, 0.049, 0.038, 0.139, 0.475, 0.274]

And the schema chosen for crossover are : ['10*11' '000*0' '10*11' '10*11' '10*11' '10*11']

Step 5: Crossover

Randomly pair up the schemas selected for crossover: In our case:
10*11 and 10*11
000*0 and 10*11
10*11 and 10*11

We are using single point crossover and the offsprings generated are : ['01*0*', '00*11', '10*11', '100*0', '10*11', '01**1']

Step 6: Mutation

For mutation, we define a probability of mutation, p_m and we go through each gene of the chromosome and mutate them.

In our case, $p_m = 0.02$.

For schema bit with '*', mutation makes no difference, so we may not see any mutation.

After mutation, our schemas are : ['01*0*', '00*11', '10*11', '100*0', '10*11', '01**1']

Step 7: Repeat

Repeat Selection, Crossover and Mutation on the schemas until there is no improvement in the average fitness of the generation.

In our case, that is : ['11**1', '01*11', '10**1', '10*11', '00*11', '01**1']

Step 8: Perform GA on the schema population

Now that we have the best schemas, create a population with all the instances of all the schemas in the final generation and apply normal ga to this population.

In our case, after applying ga to the best schema instance population the optimal solution is : '11111'