# *Genetic Algorithms*

Genetic Algorithms (GAs) are  algorithms that belong to the larger part of evolutionary algorithms. Genetic algorithms are based on the ideas of natural selection and genetics. These are intelligent exploitation of random search provided with historical data to direct the search into the region of better performance in solution space. **They are commonly used to generate high-quality solutions for optimization problems and search problems.**
**Genetic algorithms simulate the process of natural selection** which means those species who can adapt to changes in their environment are able to survive and reproduce and go to next generation. In simple words, they simulate “survival of the fittest” among individual of consecutive generation for solving a problem. **Each generation consist of a population of individuals** and each individual represents a point in search space and possible solution. Each individual is represented as a string of character/integer/float/bits. This string is analogous to the Chromosome.

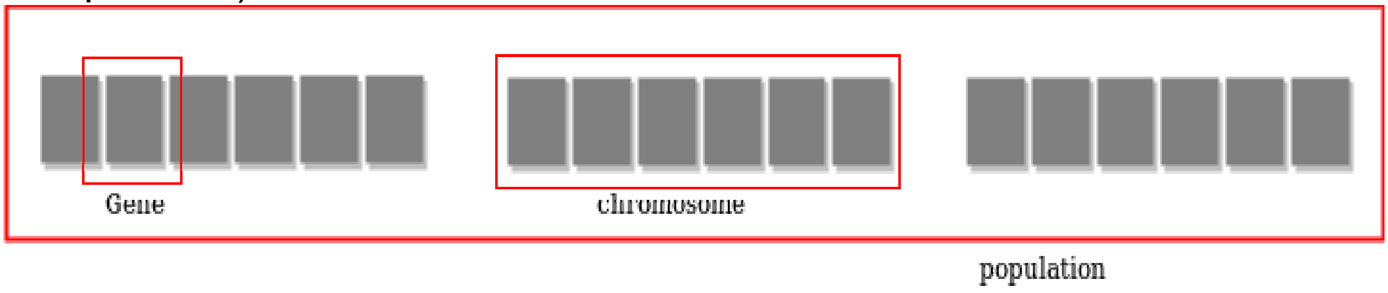
## Foundation of Genetic Algorithms

Genetic algorithms are based on an analogy with genetic structure and behaviour of chromosome of the population. Following is the foundation of GAs based on this analogy –

1. Individual in population compete for resources and mate
2. Those individuals who are successful (fittest) then mate to create more offspring than others
3. Genes from “fittest” parent propagate throughout the generation that is sometimes parents create offspring which is better than either parent.
4. Thus each successive generation is more suited for their environment.

## Search space

The population of individuals are maintained within search space. Each individual represent a solution in search space for given problem. Each individual is coded as a finite length vector (analogous to chromosome) of components. These variable components are analogous to Genes. Thus a chromosome (individual) is composed of several genes (variable components).

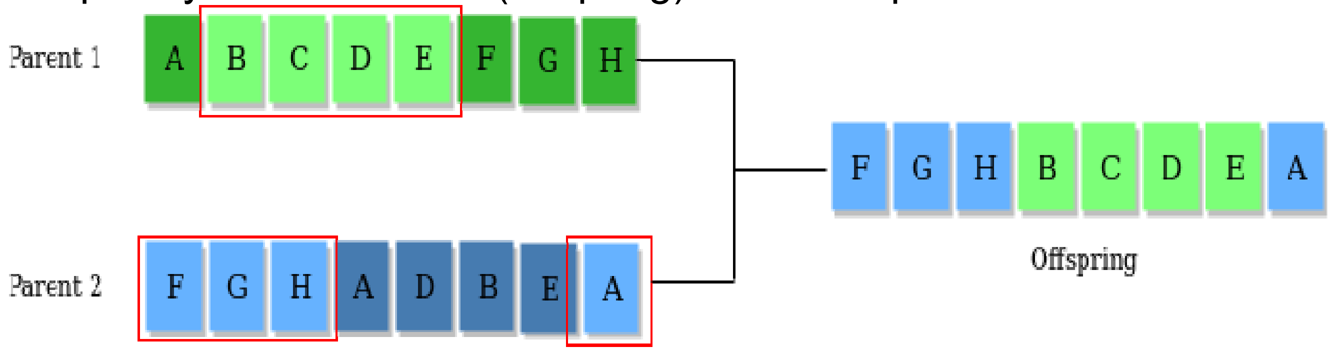


## Fitness Score

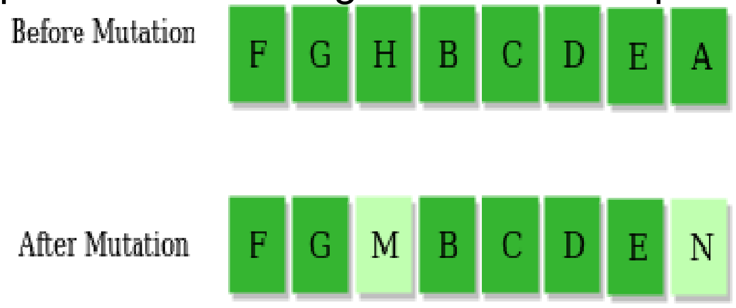
A Fitness Score is given to each individual which **shows the ability of an individual to "compete"**. The individual having optimal fitness score (or near optimal) are sought.
The GAs maintains the population of n individuals (chromosome/solutions) along with their fitness scores. The individuals having better fitness scores are given more chance to reproduce than others. The individuals with better fitness scores are selected who mate and prodadaptive heuristic searchuce **better offspring** by combining chromosomes of parents. The population size is static so the room has to be created for new arrivals. So, some individuals die and get replaced by new arrivals eventually creating new generation when all the mating opportunity of the old population is exhausted. It is hoped that over successive generations better solutions will arrive while least fit die.
Each new generation has on average more "better genes" than the individual (solution) of previous generations. Thus each new generations have better **"partial solutions"** than previous generations. Once the offspring's produced having no significant difference than offspring produced by previous populations, the population is converged. The algorithm is said to be converged to a set of solutions for the problem.

## Operators of Genetic Algorithms

Once the initial generation is created, the algorithm evolve the generation using following operators –
**1) Selection Operator:** The idea is to give preference to the individuals with good fitness scores and allow them to pass there genes to the successive generations.
**2) Crossover Operator:** This represents mating between individuals. Two individuals are selected using selection operator and crossover sites are chosen randomly. Then the genes at these crossover sites are exchanged thus creating a completely new individual (offspring). For example –



**3) Mutation Operator:** The key idea is to insert random genes in offspring to maintain the diversity in population to avoid the premature convergence. For example –



The whole algorithm can be summarized as –

```
1) Randomly initialize population’s p
2) Determine fitness of population
3) Until convergence repeat:
      a) Select parents from population
      b) Crossover and generate new population
      c) Perform mutation on new population
      d) Calculate fitness for new population
```

**Example problem and solution using Genetic Algorithms**

Given a target string, the goal is to produce target string starting from a random string of the same length. In the following implementation, following analogies are made –

- Characters A-Z, a-z, 0-9 and other special symbols are considered as genes
- A string generated by these character is considered as chromosome/solution/Individual

**Fitness score** is the number of characters which differ from characters in target string at a particular index. So individual having lower fitness value is given more preference.

```
// C++ program to create target string, 
starting from
// random string using Genetic Algorithm

#include <bits/stdc++.h>
using namespace std;

// Number of individuals in each generation
#define POPULATION_SIZE 100

// Valid Genes
const string GENES = 
"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOP"\
"QRSTUVWXYZ 1234567890, .-
;:_!\"#%&/()=?@${[]}";

// Target string to be generated
const string TARGET = "I love GeeksforGeeks";

// Function to generate random numbers in given 
range
int random_num(int start, int end)
{
    int range = (end-start)+1;
    int random_int = start+(rand()%range);
    return random_int;
}

// Create random genes for mutation
char mutated_genes()
{
    int len = GENES.size();
    int r = random_num(0, len-1);
    return GENES[r];
}

// create chromosome or string of genes
string create_gnome()
{
    int len = TARGET.size();
    string gnome = "";
    for(int i = 0;i<len;i++)
        gnome += mutated_genes();
    return gnome;
}

// Class representing individual in population
class Individual
{
public:
    string chromosome;
    int fitness;
    Individual(string chromosome);
    Individual mate(Individual parent2);
    int cal_fitness();
};

Individual::Individual(string chromosome)
{
    this->chromosome = chromosome;
    fitness = cal_fitness();
};
```

```
// Perform mating and produce new offspring
Individual Individual::mate(Individual par2)
{
    // chromosome for offspring
    string child_chromosome = "";

    int len = chromosome.size();
    for(int i = 0;i<len;i++)
    {
        // random probability
        float p = random_num(0, 100)/100;

        // if prob is less than 0.45, insert
gene
        // from parent 1
        if(p < 0.45)
            child_chromosome += chromosome[i];

        // if prob is between 0.45 and 0.90,
insert
        // gene from parent 2
        else if(p < 0.90)
            child_chromosome +=
par2.chromosome[i];

        // otherwise insert random
gene(mutate),
        // for maintaining diversity
        else
            child_chromosome +=
mutated_genes();
    }

    // create new Individual(offspring) using
    // generated chromosome for offspring
    return Individual(child_chromosome);
};
```

```
// Calculate fitness score, it is the number of
// characters in string which differ from
target
// string.
int Individual::cal_fitness()
{
    int len = TARGET.size();
    int fitness = 0;
    for(int i = 0;i<len;i++)
    {
        if(chromosome[i] != TARGET[i])
            fitness++;
    }
    return fitness;
};

// Overloading < operator
bool operator<(const Individual &ind1, const
Individual &ind2)
{
    return ind1.fitness < ind2.fitness;
}

// Driver code
int main()
{
    srand((unsigned)(time(0)));

    // current generation
    int generation = 0;

    vector<Individual> population;
    bool found = false;

    // create initial population
    for(int i = 0;i<POPULATION_SIZE;i++)
    {
        string gnome = create_gnome();
        population.push_back(Individual(gnome))
;
    }

    while(! found)
    {
        // sort the population in increasing
order of fitness score
        sort(population.begin(),
population.end());

        // if the individual having lowest
fitness score ie.
        // 0 then we know that we have reached
to the target
        // and break the loop
        if(population[0].fitness <= 0)
        {
            found = true;
            break;
        }

        // Otherwise generate new offsprings
for new generation
        vector<Individual> new_generation;

        // Perform Elitism, that mean 10% of
fittest population
        // goes to the next generation
        int s = (10*POPULATION_SIZE)/100;
        for(int i = 0;i<s;i++)
            new_generation.push_back(population
[i]);

        // From 50% of fittest population,
Individuals
        // will mate to produce offspring
        s = (90*POPULATION_SIZE)/100;
        for(int i = 0;i<s;i++)
        {
            int len = population.size();
            int r = random_num(0, 50);
            Individual parent1 = population[r];
            r = random_num(0, 50);
            Individual parent2 = population[r];
            Individual offspring =
parent1.mate(parent2);
            new_generation.push_back(offspring)
;
        }
        population = new_generation;
        cout<< "Generation: " << generation <<
"\t";
        cout<< "String: "<<
population[0].chromosome <<"\t";
        cout<< "Fitness: "<<
population[0].fitness << "\n";

        generation++;
     }
     cout<< "Generation: " << generation <<
"\t";
    cout<< "String: "<<
population[0].chromosome <<"\t";
    cout<< "Fitness: "<< population[0].fitness
<< "\n";
}
```

Output:

```
Generation: 1    String: tO{"-?=jH[k8=B4]Oe@}
Fitness: 18
```

```
Generation: 2    String: tO{"-?=jH[k8=B4]Oe@}
Fitness: 18
Generation: 3    String: .#lRWf9k_Ifslw #O$k_
Fitness: 17
Generation: 4    String: .-1Rq?9mHqk3Wo]3rek_
Fitness: 16
Generation: 5    String: .-1Rq?9mHqk3Wo]3rek_
Fitness: 16
Generation: 6    String: A#ldW) #lIkslw cVek)
Fitness: 14
Generation: 7    String: A#ldW) #lIkslw cVek)
Fitness: 14
Generation: 8    String: (, o x _x%Rs=, 6Peek3
Fitness: 13
                           .
                           .
                           .
Generation: 29    String: I lope Geeks#o, Geeks
Fitness: 3
Generation: 30    String: I loMe GeeksfoBGeeks
Fitness: 2
Generation: 31    String: I love Geeksfo0Geeks
Fitness: 1
Generation: 32    String: I love Geeksfo0Geeks
Fitness: 1
Generation: 33    String: I love Geeksfo0Geeks
Fitness: 1
Generation: 34    String: I love GeeksforGeeks
Fitness: 0
```

**Note:** Every time algorithm start with random strings, so output may differ
As we can see from the output, our algorithm sometimes stuck at a local optimum solution, this can be further improved by updating fitness score calculation algorithm or by tweaking mutation and crossover operators.

## Why use Genetic Algorithms

- They are Robust
- Provide optimisation over large space state.
- Unlike traditional AI, they do not break on slight change in input or presence of noise

## Application of Genetic Algorithms

Genetic algorithms have many applications, some of them are –

- Recurrent Neural Network
- Mutation testing
- Code breaking
- Filtering and signal processing
- Learning fuzzy rule base etc.

## *Mutation Algorithms for String Manipulation (GA)*

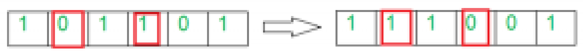
Genetic Algorithms (GAs) are adaptive heuristic search algorithms that belong to the larger part of evolutionary algorithms. In each generation chromosomes (our solution candidates) undergo mutation and crossover and then selection to produce a better population whose candidates are nearer to our desired solution. Mutation Operator is a unary operator and it needs only one parent to work on. It does so by selecting a few genes from our selected chromosome and apply the desired algorithm.

In this article, I will be talking five Mutation Algorithms for string manipulation –
1) Bit Flip Mutation
2) Random Resetting Mutation

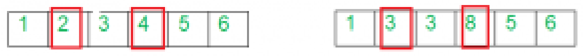
3) Swap Mutation
4) Scramble Mutation
5) Inversion Mutation

Bit Flip Mutation is mainly used for bit string manipulation while others can be used for any
kind of strings. Here our chromosome will be represented as an array and each index will represent one gene. Strings can be represented as an array of characters which in turn is an array of ASCII or numeric values.

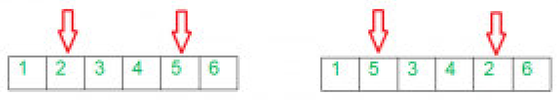
**Bit Flip Mutation —**
In bit flip mutation, we select one or more genes (array indices) and flip their values i.e. we change 1s to 0s and vice versa. It is better explained using the given diagram.



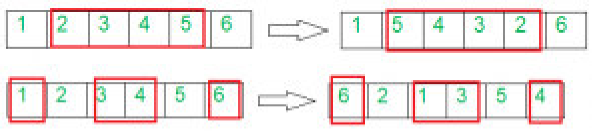
**Random Resetting Mutation —**
In random resetting mutation, we select one or more genes (array indices) and replace their values with another random value from their given ranges. Let's say a[i] (an array index / gene) ranges from [1, 6] then random resetting mutation will select one value from [1, 6] and replace a[i]'s value with it.



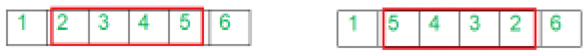
**Swap Mutation —**
In Swap Mutation we select two genes from our chromosome and interchange their values.



**Scramble Mutation —**
In Scramble Mutation we select a subset of our genes and scramble their value. The selected genes may not be contiguous (see the second diagram).



**Inversion Mutation —**
In Inversion Mutation we select a subset of our genes and reverse their order. The genes have to be contiguous in this case (see the diagram).



## *Mutation Algorithms for Real-Valued Parameters (GA)*

Genetic Algorithms (GAs) are adaptive heuristic search algorithms that belong to the larger part of evolutionary algorithms. In each generation chromosomes (our solution candidates) undergo mutation and crossover and selection to produce a better population whose chromosomes are nearer to our desired solution. Mutation Operator is a unary operator and it needs only one parent to work on. It does so by selecting a few genes from our selected chromosome (parent) and then by applying the desired mutation operator on them.
In this article, I will be talking about four Mutation Algorithms for real-valued parameters –
1) Uniform Mutation
2) Non-Uniform
3) Boundary Mutation
4) Gaussian Mutation

Here, we are considering a chromosome with n real numbers (which are our genes) and $x_i$ represents a gene and i belongs to [1,n].
**Uniform Mutation –**
In uniform mutation we select a random gene from our chromosome, let's say $x_i$ and assign a uniform random value to

it.
Let $x_i$ be within the range $[a_i,b_i]$ then we assign $U(a_i,b_i)$ to $x_i$
$U(a_i,b_i)$ denotes a uniform random number from within the range $[a_i,b_i]$.

```
Algorithm –
1.    Select a random integer number i from [1,n]
2.    Set $x_i$ to $U(a_i,b_i)$.
```

**Boundary Mutation –**
In boundary mutation we select a random gene from our chromosome , let's say $x_i$ and assign the upper bound or the lower bound of $x_i$ to it.
Let $x_i$ be within the range $[a_i,b_i]$ then we assign either $a_i$ or $b_i$ to $x_i$.
We also select a variable r= U(0,1) ( r is a number between 0 and 1).
If r is greater than or equal to 0.5 , assign $b_i$ to $x_i$ else assign $a_i$ to $x_i$.

```
Algorithm –
1.    select a random integer number i form [1,n]
2.    select a random real value r from (0,1).
3.    If(r >= 0.5)
            Set $x_i$ to $b_i$
       else
            Set $x_i$ to $a_i$
```

**Non-Uniform Mutation –**
In non-uniform mutation we select a random gene from our chromosome, let's say $x_i$ and assign a non-uniform random value to it.
Let $x_i$ be within the range $[a_i,b_i]$ then we assign a non-uniform random value to it.

We use a function,
f(G)=(r2*(1-G/Gmax))b ,
where r2 = a uniform random number between (0,1)
G = the current generation number
Gmax = the maximum number of generations
b = a shape parameter
Here we select a uniform random number r1 between (0,1).
If r greater than or equal to 0.5 we assign $(b_i-x_i)$ * f(G) to $x_i$ else we assign $(a_i+ x_i)$ * f(G).

```
Algorithm –

1.    Select a random integer i within [1,n]

2.    select two random real values r1 ,r2 from
(0,1).

3.    If(r1 >= 0.5)

              Set xi to $(b_i-x_i)$ * f(G)
       else
              Set xi to $(a_i+ x_i)$ * f(G)
```

**Gaussian Mutation –**

Gaussian Mutation makes use of the Gauss error function. It is far more efficient in converging than the previously mentioned algorithms. We select a random gene let's say $x_i$ which belongs to the range $[a_i,b_i]$. Let the mutated off spring be $x'_i$. Every variable has a mutation strength operator $(\sigma_i)$. We use $\sigma= \sigma_i/(b_i-a_i)$ as a fixed non-dimensional zed parameter for all n variables; Thus the offspring $x'_i$ is given by —

$$x'_i= x_i + \sqrt{2} * \sigma * (b_i-a_i)\mathrm{erf}^{-1}(u'_i)$$

Here erf() denotes the Gaussian error function.

$$\mathrm{erf}(y)={}^{2}/_{\sqrt{\pi}} \int^{y}_{0} e^{-t^2}\, dt$$

For calculation $u_i'$ we first select a random value $u_i$ from within the range (0,1) and then use the following formula

```
if($u_i$>=0.5)
    $u'_i$=2*$u_L$*(1-2*$u_i$)
else
    $u'_i$=2*$u_R$*(2*$u_i$-1)
```

Again $u_L$ and $u_R$ are given by the formula

$u_L=0.5(\mathrm{erf}(\ {}^{(a}{}_{i}-x_i)/_{(\sqrt{2}(bi}-a_i)\sigma)\ )+1)$

$u_R=0.5(\mathrm{erf}(\ {}^{(b}{}_{i}-x_i)/_{(\sqrt{2}(bi}-a_i)\sigma)\ )+1)$