Assignment No. 2

Title: Implementation Genetic Application – Match Word Finding.

Problem Statement:

Implement genetic algorithm for benchmark function (eg. Square, Rosenbrock function etc). Initialize the population from the Standard Normal Distribution. Evaluate the fitness of all its individuals. Then you will do multiple generation of a genetic algorithm. A generation consists of applying selection, crossover, mutation, and replacement. Use:

Tournament selection without replacement with tournament size s
One point crossover with probability Pc
bit-flip mutation with probability Pm
Use full replacement strategy

Objectives:

1. To learn Genetic Algorithm.
2. To learn about optimization algorithm.

Software Requirements:

- Ubuntu 16.04

Hardware Requirements:

- Pentium IV system with latest configuration

Theory:

Genetic Algorithms are a class of stochastic, population based optimization algorithms inspired by the biological evolution process using the concepts of **"**Natural Selection**"** and **"**Genetic Inheritance**"** (Darwin 1859) and originally developed by Holland.

GAs are now used in engineering and business optimization applications, where the search space is large and/or too complex (non-smooth) for analytic treatment. This algorithm reflects the process of natural selection where the fittest individuals are selected for reproduction in order to produce offspring of the next generation.

The process of natural selection starts with the selection of fittest individuals from a population. They produce offspring which inherit the characteristics of the parents and will be added to the next generation. If parents have better fitness, their offspring will be better than parents and have a better chance at surviving. This process keeps on iterating and at the end, a generation with the fittest individuals will be found. This notion can be applied for a search problem. We consider a set of solutions for a problem and select the set of best ones out of them.

Five phases are considered in a genetic algorithm.

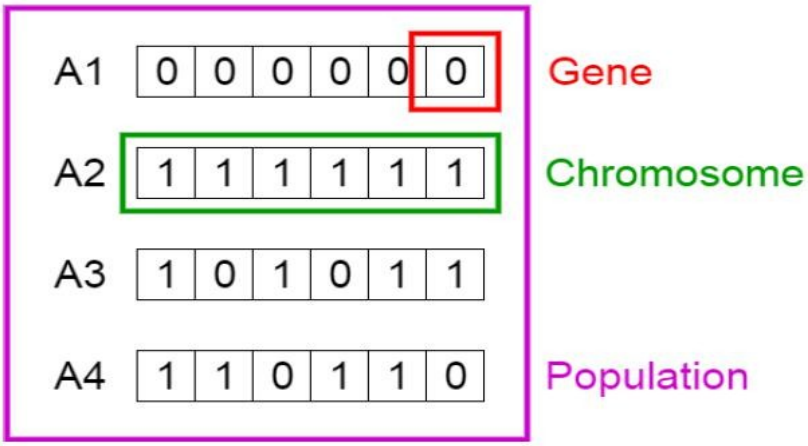
1. Initial population
2. Fitness function
3. Selection

4. Crossover
5. Mutation

Initial Population

The process begins with a set of individuals which is called a **Population**. Each individual is a solution to the problem you want to solve. An individual is characterized by a set of parameters (variables) known as **Genes**. Genes are joined into a string to form a **Chromosome** (solution).

In a genetic algorithm, the set of genes of an individual is represented using a string, in terms of an alphabet. Usually, binary values are used (string of 1s and 0s). We say that we encode the genes in a chromosome.



Fitness Function

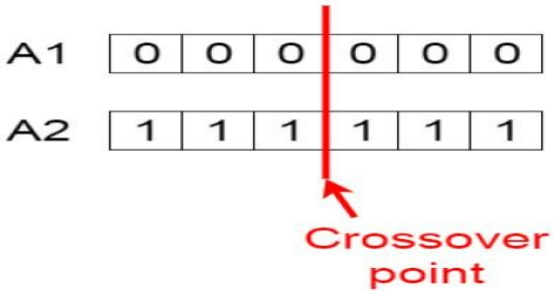
The **fitness function** determines how fit an individual is (the ability of an individual to compete with other individuals). It gives a **fitness score** to each individual. The probability that an individual will be selected for reproduction is based on its fitness score.

Selection

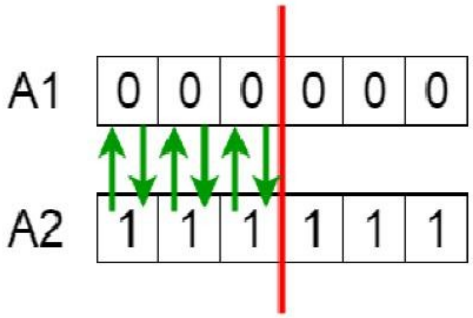
The idea of **selection** phase is to select the fittest individuals and let them pass their genes to the next generation. Two pairs of individuals (**parents**) are selected based on their fitness scores. Individuals with high fitness have more chance to be selected for reproduction.

Crossover

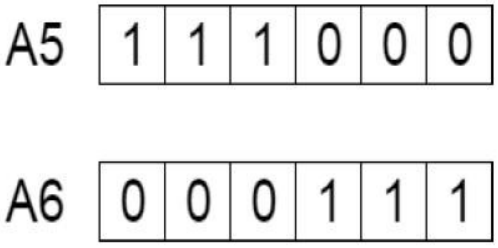
Crossover is the most significant phase in a genetic algorithm. For each pair of parents to be mated, a **crossover point** is chosen at random from within the genes. For example, consider the crossover point to be 3 as shown below.



Offsprings are created by exchanging the genes of parents among themselves until the crossover point is reached.

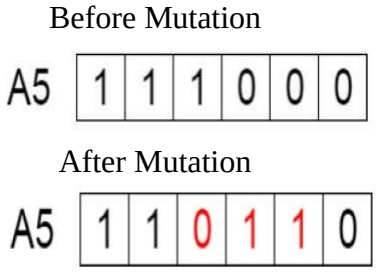


The new offspring are added to the population.



Mutation

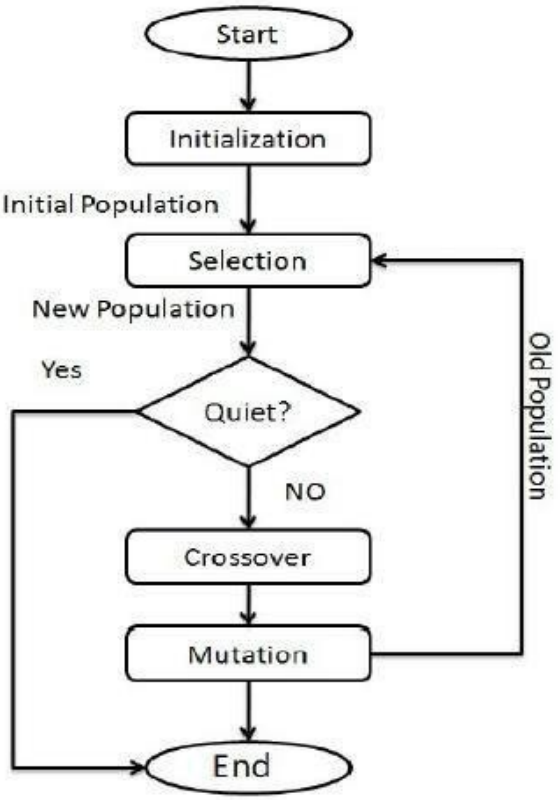
In certain new offspring formed, some of their random probability. This implies that some of the bits in the bit string can be flipped.



Mutation occurs to maintain diversity within the population and prevent premature convergence.

Termination

The algorithm terminates if the population has converged (does not produce offspring which are significantly different from the previous generation). Then it is said that the genetic algorithm has provided a set of solutions to our problem.



Comments

The population has a fixed size. As new generations are formed, individuals with least fitness die, providing space for new offspring. The sequence of phases is repeated to produce individuals in each new generation which are better than the previous generation.

GA Software: Python, Weka , Mendal Soft ,Matlab

Conclusion

Thus we learnt implementation of genetic algorithm for benchmark function.

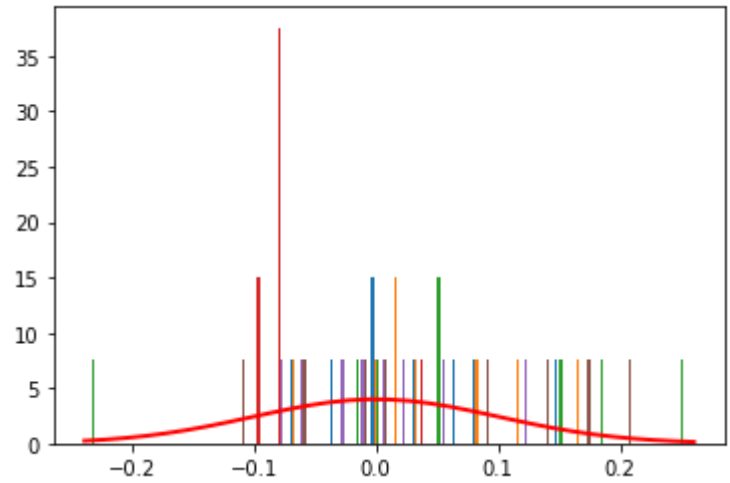
```
import numpy as np
from matplotlib import pyplot as plt
```

```
def cal_pop_fitness(equation_inputs, pop):
    fitness = np.sum(pop*pop, axis=1)
    return fitness

def select_mating_pool(pop, fitness, num_parents):
    parents = np.empty((num_parents, pop.shape[1]))
    for parent_num in range(num_parents):
        max_fitness_idx = np.where(fitness == np.max(fitness))
        max_fitness_idx = max_fitness_idx[0][0]
        parents[parent_num, :] = pop[max_fitness_idx, :]
        fitness[max_fitness_idx] = -99999999999
    return parents

def crossover(parents, offspring_size):
    offspring = np.empty(offspring_size)
    crossover_point = np.uint8(offspring_size[1]/2)

    for k in range(offspring_size[0]):
        parent1_idx = k%parents.shape[0]
        parent2_idx = (k+1)%parents.shape[0]
        offspring[k, 0:crossover_point] = parents[parent1_idx, 0:crossover_point]
        offspring[k, crossover_point:] = parents[parent2_idx, crossover_point:]
    return offspring

def mutation(offspring_crossover):
    for idx in range(offspring_crossover.shape[0]):
        random_value = np.random.uniform(-1.0, 1.0, 1)
        offspring_crossover[idx, 4] = offspring_crossover[idx, 4] + random_value
    return offspring_crossover
```

```
equation_inputs = [4,-2,3.5,5,-11,-4.7]
num_weights = 6
sol_per_pop = 8
num_parents_mating = 4
pop_size = (sol_per_pop,num_weights)

mu,sigma = 0,0.1
new_population = np.random.normal(mu,sigma,pop_size)


print(new_population)
```

```
[[-0.03586291  0.01126042  0.05126282  0.0407957  -0.00621555  0.08680624]
 [ 0.15715148  0.1198302  -0.00179962 -0.09171131  0.1266114  -0.01609103]
 [ 0.07550347  0.00556282  0.26044257 -0.10368689 -0.03395319  0.1395789 ]
 [ 0.00755859  0.03330221 -0.23864191 -0.07349935  0.01347028  0.01075146]
 [ 0.03627713  0.08338747  0.15951852 -0.0737605  -0.08265722 -0.06690229]
 [ 0.08431102  0.16200461  0.04606559 -0.08513495 -0.07185449  0.20143857]
 [-0.00179832 -0.05720219  0.18124986 -0.08758745  0.05783212 -0.1148326 ]
 [-0.07108077  0.01782279 -0.00775485 -0.07594953  0.00450742  0.16955477]]
```

```
abs(mu - np.mean(new_population)) < 0.01


abs(sigma - np.std(new_population, ddof=1)) < 0.01

count, bins, ignored = plt.hist(new_population, 30, density=True)
plt.plot(bins, 1/(sigma * np.sqrt(2 * np.pi)) *
                np.exp( - (bins - mu)**2 / (2 * sigma**2) ),linewidth=2, color='r')
plt.show()
```




```
num_generations = 40
best_outputs=[]
for generation in range(num_generations):
    print("Generation : ", generation)
    # Measing the fitness of each chromosome in the population.
    fitness = cal_pop_fitness(equation_inputs, new_population)

    # Selecting the best parents in the population for mating.
    parents = select_mating_pool(new_population, fitness,
                                       num_parents_mating)

    # Generating next generation using crossover.
    offspring_crossover = crossover(parents,
                                        offspring_size=(pop_size[0]-parents.shape[0]

    # Adding some variations to the offsrping using mutation.
    offspring_mutation = mutation(offspring_crossover)

    # Creating the new population based on the parents and offspring.
    new_population[0:parents.shape[0], :] = parents
    new_population[parents.shape[0]:, :] = offspring_mutation

    # The best result in the current iteration.
    print("Best result : ", np.max(np.sum(new_population*equation_inputs, axis=1)))
    best_outputs.append( np.max(np.sum(new_population*equation_inputs, axis=1)))

# Getting the best solution after iterating finishing all generations.
```

```
#At first, the fitness is calculated for each solution in the final generation.
fitness = cal_pop_fitness(equation_inputs, new_population)
# Then return the index of that solution corresponding to the best fitness.
best_match_idx = np.where(fitness == np.max(fitness))

print("Best solution : ", new_population[best_match_idx, :])
print("Best solution fitness : ", fitness[best_match_idx])

plt.plot(best_outputs)
plt.show()
```

```
Best result :  -85.95947685772074
Generation :  20
Best result :  -85.99624035037029
Generation :  21
Best result :  -97.45085945595908
Generation :  22
Best result :  -100.78165959538188
Generation :  23
Best result :  -96.90104791580174
Generation :  24
Best result :  -106.68097106646367
Generation :  25
Best result :  -104.93118454023586
Generation :  26
Best result :  -106.69337744072658
Generation :  27
Best result :  -115.78633019966979
Generation :  28
Best result :  -113.25604607444747
Generation :  29
Best result :  -114.71749987409464
Generation :  30
Best result :  -135.47058283518427
Generation :  31
Best result :  -135.39872687046963
Generation :  32
Best result :  -142.9283597833293
Generation :  33
Best result :  -141.2983895448177
Generation :  34
Best result :  -140.2049725586676
Generation :  35
Best result :  -150.83854986981447
Generation :  36
Best result :  -156.29647777730938
Generation :  37
Best result :  -155.84770961696194
Generation :  38
Best result :  -156.38923927322932
Generation :  39
Best result :  -166.80954917405734
Best solution :  [[[ 7.55858940e-03  3.33022115e-02 -2.38641914e-01 -8.513494
    1.71380219e+01  2.01438574e-01]]]
Best solution fitness :  [293.81773727]
```