LAB-G: WRITE A PROGRAM TO IMPLEMENT GENETIC ALGORITHM.

OBJECTIVE: To implement genetic algorithm in python.

THEORY:

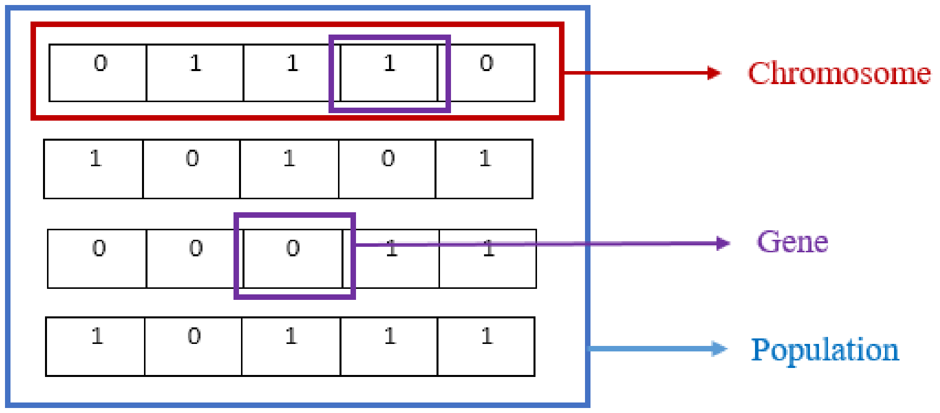
The genetic algorithm is a search-based optimization technique. It is frequently used to find the optimal or nearest optimal solution. It was introduced by John Holland. It is based on Darwins Natural Selection Theory. Before explaining how the genetic algorithm works let me first explain Darwin's theory on natural selection. In his theory, he defined natural selection as the "principle by which each slight variation [of a trait], if useful, is preserved". The concept was simple but powerful: individuals best adapted to their environments are more likely to survive and reproduce. Sometimes this theory is described as survival of the fittest". Those who are fittest than others have the chance to survive in this evolution. The genetic algorithm is all about this. It mimics the process of natural selection to find the best solution.

In genetics we will use some biological terms such as population, chromosome, gene, selection, crossover, mutation.

Population, Chromosome, Gene

At the beginning of this process, we need to initialize some possible solutions to this problem. The population is a subset of all possible solutions to the given problem. In another way, we can say that the population is a set of chromosomes. A chromosome is one of the solutions to that current problem. And each chromosome is a set of genes.

For simplicity, we can describe a chromosome as a string. So, we can say that a population is a collection of some string (each character is a binary value, either 0 or 1). And each character of the string is a gene.



For starting the process of the genetic algorithm, we first need to initialize the population. We can initialize the population in two ways. The first one is random and the second one is heuristical. It is always better to start the algorithm with some random population.

Fitness Function

After initializing the population, we need to calculate the fitness value of these chromosomes. Now the question is what this fitness function is and how it calculates the fitness value.

As an example, let consider that we have an equation, $f(x) = -x^2 + 5$. We need the solution for which it has the maximum value and the constraint is $0 \leq x \leq 31$.

Now, let consider that we have a random population of four chromosomes like below. The length of our chromosome is 5 as $2^5=32$ and $0 \leq x \leq 31$.

0	1	1	1	0

1	0	1	0	1

0	0	0	1	1

1	0	1	1	1

Our fitness function will calculate the functional value of each chromosome as stated in the problem statement:

For the 1st chromosome, 01110 means 14 in integer. So, f(x) = -(14*14) + 5 = -191.

For the 2nd chromosome, 10101 means 21 in integer. So, f(x) = -(21*21) + 5 = -436.

For the 3rd chromosome, 00011 means 3 in integer. So, f(x) = -(3*3) + 5 = -4.

For the 4th chromosome, 10111 means 23 in integer. So, f(x) = -(23*23) + 5 = -524.

Parent Selection

Parent selection is done by using the fitness values of the chromosomes calculated by the fitness function. Based on these fitness values we need to select a pair chromosome with the highest fitness value.

There are many ways for fitness calculation like Roulette wheel selection, rank selection.

In Roulette wheel selection, the chromosome with the highest fitness value has the maximum possibility to be selected as a parent. But in this selection process, a lower can be selected.

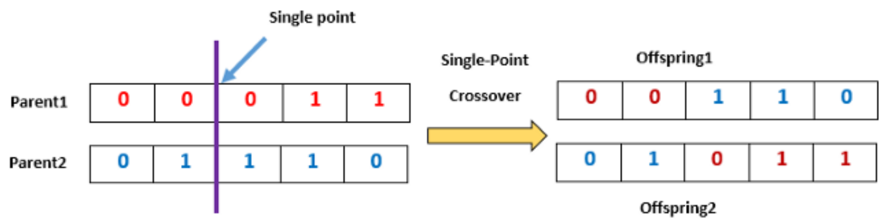
In rank selection, chromosomes are ranked based on their fitness values from higher to lower. As an example, according to those fitness values calculated above, we can rank those chromosomes from higher to lower like 3rd>1st>2nd>4th. So, in the selection phase, 3rd and 1st chromosomes will be selected based on the fitness valued calculated from the fitness function.

Crossover

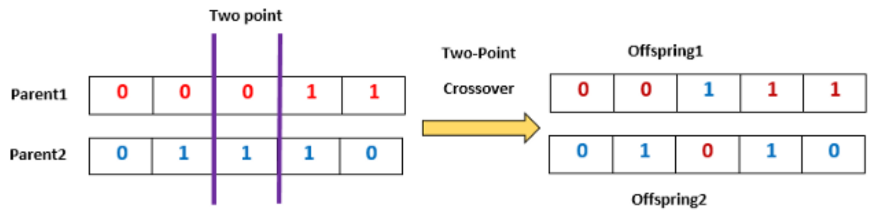
Crossover is used to vary the programming of the chromosomes from one generation to another by creating children or offspring's. Parent chromosomes are used to create these offspring's(generated chromosomes).

To create offspring's, there are some ways like a single-point crossover, two or multi-point crossover.

For a single point crossover, first, we need to select a point and then exchange these portions divided by this point between parent chromosomes to create offspring's. You can use the color combination for easy understanding.



For a two-point crossover, we need to select two points and then exchange the bits.

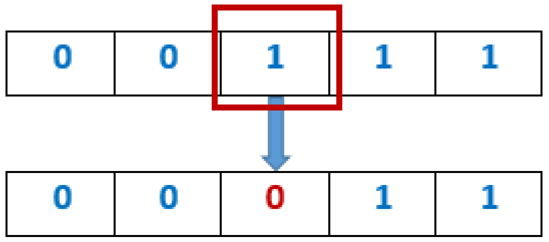


Finally, these new offsprings are added to the population.

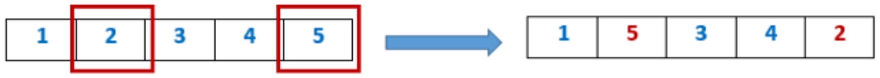
Mutation

Mutation brings diversity to the population. There are different kinds of mutations like Bit Flip mutation, Swap mutation, Inversion mutation, etc. These are so so simple.

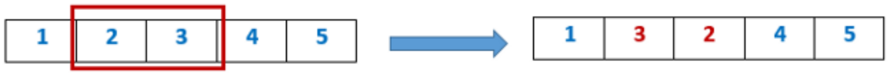
In Bit Flip mutation, just select one or more bits and then flip them. If the selected bit is 0 then turn it to 1 and if the selected bit is 1 then turn it to 0.



In Swap Bit mutation, select two bits and just swap them.



In inverse mutation, just inverse the bits.



Algorithm:

```
1) Randomly initialize populations 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
```

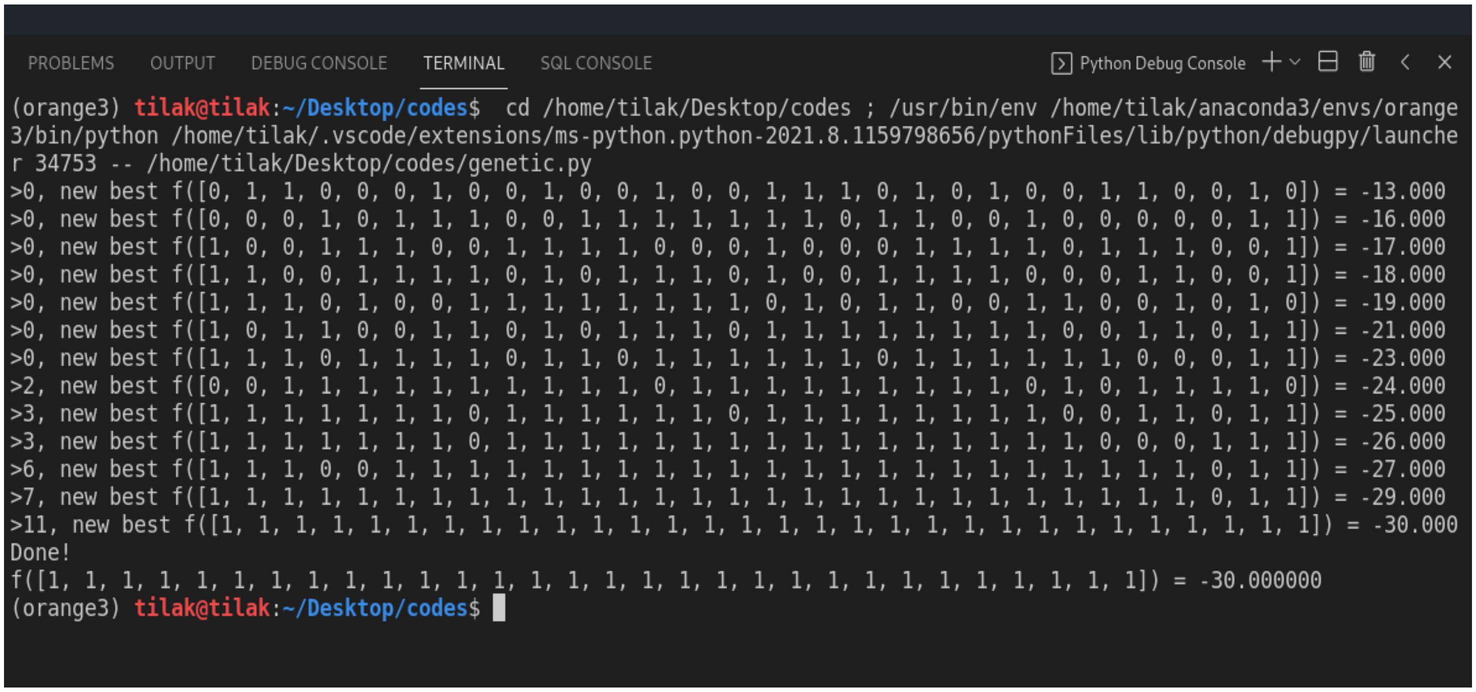
PROGRAM:

```
# genetic algorithm
from numpy.random import randint
from numpy.random import rand

# objective function
def onemax(x):
```

```
return -sum(x)

# tournament selection
def selection(pop, scores, k=3):
# first random selection
selection_ix = randint(len(pop))
for ix in randint(0, len(pop), k-1):
# check if better (e.g. perform a tournament)
if scores[ix] < scores[selection_ix]:
selection_ix = ix
return pop[selection_ix]

# crossover two parents to create two children
def crossover(p1, p2, r_cross):
# children are copies of parents by default
c1, c2 = p1.copy(), p2.copy()
# check for recombination
if rand() < r_cross:
# select crossover point that is not on the end of the string
pt = randint(1, len(p1)-2)
# perform crossover
c1 = p1[:pt] + p2[pt:]
c2 = p2[:pt] + p1[pt:]
return [c1, c2]

# mutation operator
def mutation(bitstring, r_mut):
for i in range(len(bitstring)):
# check for a mutation
if rand() < r_mut:
# flip the bit
bitstring[i] = 1 - bitstring[i]

# genetic algorithm
def genetic_algorithm(objective, n_bits, n_iter, n_pop, r_cross, r_mut):
# initial population of random bitstring
pop = [randint(0, 2, n_bits).tolist() for _ in range(n_pop)]
# keep track of best solution
best, best_eval = 0, objective(pop[0])
# enumerate generations
for gen in range(n_iter):
# evaluate all candidates in the population
scores = [objective(c) for c in pop]
# check for new best solution
for i in range(n_pop):
if scores[i] < best_eval:
best, best_eval = pop[i], scores[i]
print(">%d, new best f(%s) = %.3f" % (gen, pop[i], scores[i]))
# select parents
selected = [selection(pop, scores) for _ in range(n_pop)]
# create the next generation
children = list()
for i in range(0, n_pop, 2):
# get selected parents in pairs
```

```
p1, p2 = selected[i], selected[i+1]
# crossover and mutation
for c in crossover(p1, p2, r_cross):
# mutation
mutation(c, r_mut)
# store for next generation
children.append(c)
# replace population
pop = children
return [best, best_eval]


# define the total iterations
n_iter = 100
# bits
n_bits =30
# define the population size
n_pop = 200
# crossover rate
r_cross = 0.95
# mutation rate
r_mut = 1.0 / float(n_bits)
# perform the genetic algorithm search
best, score = genetic_algorithm(onemax, n_bits, n_iter, n_pop, r_cross, r_mut)
print('Done!')
print('f(%s) = %f' % (best, score))
```

OUTPUT:



```
(orange3) tilak@tilak:~/Desktop/codes$  cd /home/tilak/Desktop/codes ; /usr/bin/env /home/tilak/anaconda3/envs/orange
3/bin/python /home/tilak/.vscode/extensions/ms-python.python-2021.8.1159798656/pythonFiles/lib/python/debugpy/launche
r 34753 -- /home/tilak/Desktop/codes/genetic.py
>0, new best f([0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 1, 1, 0, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0]) = -13.000
>0, new best f([0, 0, 0, 1, 0, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1]) = -16.000
>0, new best f([1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 0, 1, 1, 1, 0, 0, 1]) = -17.000
>0, new best f([1, 1, 0, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 0, 1, 0, 0, 1, 1, 1, 1, 0, 0, 0, 1, 1, 0, 0, 1]) = -18.000
>0, new best f([1, 1, 1, 0, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 0, 1, 0]) = -19.000
>0, new best f([1, 0, 1, 1, 0, 0, 1, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 0, 1, 1]) = -21.000
>0, new best f([1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1]) = -23.000
>2, new best f([0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0]) = -24.000
>3, new best f([1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 0, 1, 1]) = -25.000
>3, new best f([1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 1]) = -26.000
>6, new best f([1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1]) = -27.000
>7, new best f([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1]) = -29.000
>11, new best f([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]) = -30.000
Done!
f([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]) = -30.000000
(orange3) tilak@tilak:~/Desktop/codes$
```

CONCLUSION:

And finally, we implemented genetic algorithm using python.