Experiment :8

Problem statement:
Implement an algorithm to demonstrate the significance of genetic algorithm
Aim: to Implement an algorithm to demonstrate the significance of genetic algorithm
ALGORITHM:

Step1: Start
Step2: Individual in population compete for resources and mate
Step3: Those individuals who are successful (fittest) then mate to create more offspring than others
Step 4: Genes from "fittest" parent propagate throughout the generation, that is sometimes parents create offspring which is better than either parent.
Step 5: Thus each successive generation is more suited for their environment.
Step 6: Stop.

PROGRAM:

```
# genetic algorithm search of the one max optimization problem
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 = 20
# define the population size
n_pop = 100
# crossover rate
r_cross = 0.9
# 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:

Google Colab notebook output:
>99, new best f([0, 1, 0, 0, 0, 1, 0, 1, 1, 0, 0, 1, 0, 1, 0, 1, 1, 0, 1, 1]) = -10.000
Done!
f([1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 0]) = -16.000000

Jupiter note book output
>99, new best f([1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 0, 1, 1, 0, 1, 1, 0, 1, 0, 0]) = -13.000
Done!
f([1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1]) = -16.000000

Result: The program has been executed successfully and the genetic algorithm is implemented.