Experiment No : 5

Aim : Implement Genetics / Hill Climbing in python

Theory :

- **Hill Climbing**

⇒ Hill climbing is simply a combination of depth first with generate and test where a 'feedback' is used here to decide on the direction of motion in the search space.

- Hill climbing technique is used widely in AI, to solve computationally hard problems, which has multiple possible solutions.
- In the DFS, the test function will accept or reject a solution. But in Hill climbing the test function is provided with a heuristic function which provides an estimate of how close a given state is to goal state.
- In hill climbing, each state is provided with the additional information needed to find the solution, i.e. the heuristic value.
- The algorithm is memory efficient since it does not maintain the complete search tree.
- For eg. if you want to find a mall from your current location.
- There are n possible paths with different directions to reach the mall. The heuristic function will just give you the distance of each path which is reaching to the mall,

so that it becomes very simple and time efficient for you to reach to the mall.

- Algorithm :

1) Evaluate the initial state. If it is a goal state, then return and quit; otherwise make it a current state and go to step 2.
2) Loop until a solution is found or there are no new operators left to be applied.
   a) Select and apply new operator.
   b) Evaluate the new state :
      i) If it is a goal state, then return and quit.
      ii) If it is better than current state then make it a new current state.
      iii) If it is not better than the current state then continue the loop, go to step 2.

- Genetics Algorithm

⇒ GAs are adpative heuristic search algorithms based on the evolutionary ideas of natural selection and genetics.

- As they represent an intelligent exploitation of a random search used to solve optimization problems.

- Although randomized, GAs are by no means random, instead they exploit historical information to direct the search

- for better performance within the search space.
- The basic techniques of the GAs are designed to simulate processes in natural systems necessary for evolution, especially those of following the principles of "Survival of the fittest" laid down by Charles Darwain
- For e.g.,

Parents chromosomes

Chromosome A	1 5 3 2 6 4 7 9 8
Chromosome B	8 5 6 7 2 3 1 4 9

- Let's consider the example of pair chromosome encoded using permutation encoding technique and are undergoing the complete process of GA.
- Assuming that they are selected using rank selection method and will be applied, arithmetic crossover and value encoding mutation techniques.
- Child chromosome after arithmetic crossover i.e. adding bits of both chromosomes.

Child chromosome :

Chromosome C	9 0 9 9 8 7 8 3 7

- After apply value encoding mutation i.e. adding or substraction a small value to selected values e.g. ; substraction 1 to 3rd and 4th bit

- Child chromosome

Chromosome C	90 8 -8 8 -7 8 37

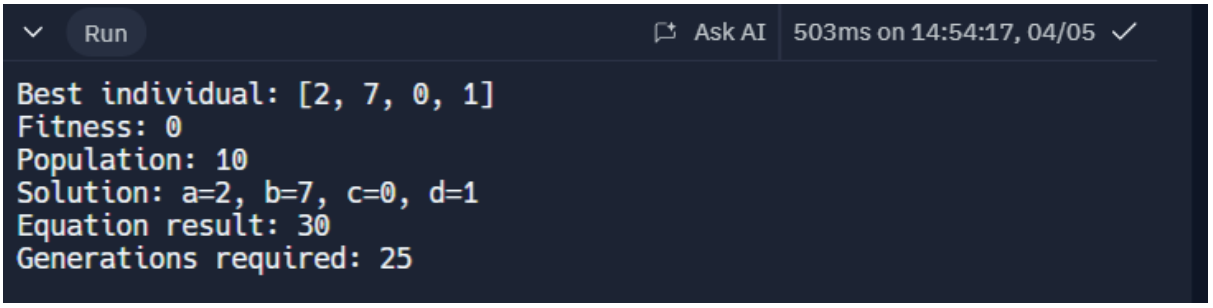
It can be observed that this child produced is much better than both parents.

Code: (Genetics*)

```
import random

def equation(a, b, c, d):
  return a + 2 * b + 17 * c + 14 * d


def eval_equation(individual):
  a, b, c, d = individual
  result = equation(a, b, c, d)
  return abs(result - 30)


def generate_individual(size):
  return [random.randint(0, 10) for _ in range(size)]


def crossover(parent1, parent2, crossover_prob):
  if random.random() < crossover_prob:
    crossover_point = random.randint(0, len(parent1) - 1)
    child = parent1[:crossover_point] + parent2[crossover_point:]
  else:
    child = parent1
  return child


def mutate(individual, mutation_rate):
  mutated_individual = individual[:]
  for i in range(len(mutated_individual)):
    if random.random() < mutation_rate:
      mutated_individual[i] = random.randint(0, 10)
  return mutated_individual


def genetic_algorithm(population_size, mutation_rate, crossover_prob):
  best_fitness = float('inf')
  best_individual = None
  generations = 0

  while True:
    generations += 1
```

```
    population = [generate_individual(4) for _ in
range(population_size)]

    for gen in range(population_size):
      fitnesses = [eval_equation(ind) for ind in population]

      min_fitness = min(fitnesses)
      best_index = fitnesses.index(min_fitness)
      best_individual = population[best_index]

      if min_fitness == 0:
        break

      selected = [random.choice(population) for _ in
range(population_size)]

      offspring = []
      for i in range(0, population_size, 2):
        parent1, parent2 = selected[i], selected[i + 1]
        child = crossover(parent1, parent2, crossover_prob)
        child = mutate(child, mutation_rate)
        offspring.extend([child])

      population[:] = offspring

    if min_fitness == 0 or generations >= 40:
      break

  print("Best individual:", best_individual)
  print("Fitness:", min_fitness)
  a, b, c, d = best_individual
  print("Population:", population_size)
  print("Solution: a={}, b={}, c={}, d={}".format(a, b, c, d))
  print("Equation result:", equation(a, b, c, d))
  print("Generations required:", generations)


if __name__ == "__main__":
  population_size = 10
  mutation_rate = 0.2
  crossover_prob = 0.5
  genetic_algorithm(population_size, mutation_rate, crossover_prob)
```

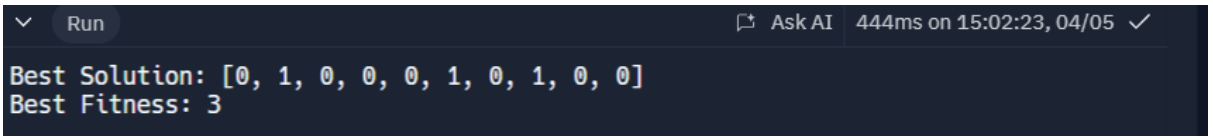
Output:



Code:

```
import random

def objective_function(solution):
  return sum(solution)

def generate_neighbor(current_solution):
  neighbor = current_solution[:]
  index = random.randint(0, len(neighbor) - 1)
  neighbor[index] = 1 - neighbor[index]
  return neighbor

def hill_climbing():
  current_solution = [random.randint(0, 1) for _ in range(10)]
  current_fitness = objective_function(current_solution)

  while True:
    neighbor = generate_neighbor(current_solution)
    neighbor_fitness = objective_function(neighbor)

    if neighbor_fitness >= current_fitness:
      current_solution = neighbor
      current_fitness = neighbor_fitness
    else:
      break

  return current_solution, current_fitness

best_solution, best_fitness = hill_climbing()
print("Best Solution:", best_solution)
print("Best Fitness:", best_fitness)
```

Output:


Run
Ask AI
444ms on 15:02:23, 04/05
Best Solution: [0, 1, 0, 0, 0, 1, 0, 1, 0, 0]
Best Fitness: 3