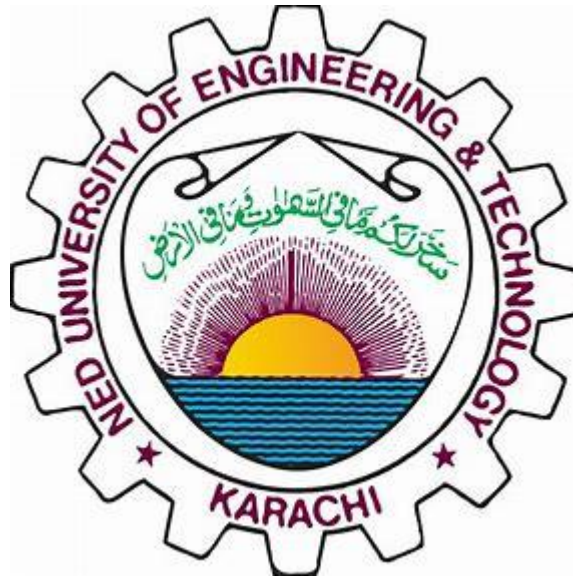


# OPEN-ENDED LAB REPORT

## Artificial Intelligence

Third Year-Computer and Information Systems Engineering

Batch: 2022



### Group Members

Muhammad Owais Alam ..... (CS-22070)

Muhammad Rohan Khan ..... (CS-22072)

Hussain Raza ..... (CS-22082)

Submitted to: Miss Hameeza Ahmed

Submission Date: 26/11/24

## **TABLE OF CONTENT:**

Introduction/Problem Statement.....	3
Genetic Algorithm.....	4-5
Source Code.....	6
Explanation of the code.....	7
Output and Conclusion.....	8

## PROBLEM STATEMENT:

A genetic algorithm (GA) is a metaheuristic inspired by the process of natural selection that belongs to the larger class of evolutionary algorithms (EA). Genetic algorithms are commonly used to generate high-quality solutions to optimization and search problems via biologically inspired operators such as selection, crossover, and mutation. Some examples of GA applications include optimizing decision trees for better performance, solving sudoku puzzles, hyperparameter optimization, and causal inference.

- **Simulate:** The genetic algorithm using a simple example to understand how it works.
- **Produce:** The Python code of the genetic algorithm.
- **Discover:** a suitable computing problem and apply the coded genetic algorithm to solve it.

## GENETIC ALGORITHM:

**Genetic algorithms simulate the process of natural selection** which means those species that can adapt to changes in their environment can survive and reproduce and go to the next generation. In simple words, they simulate “survival of the fittest” among individuals of consecutive generations to solve a problem. **Each generation consists 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.

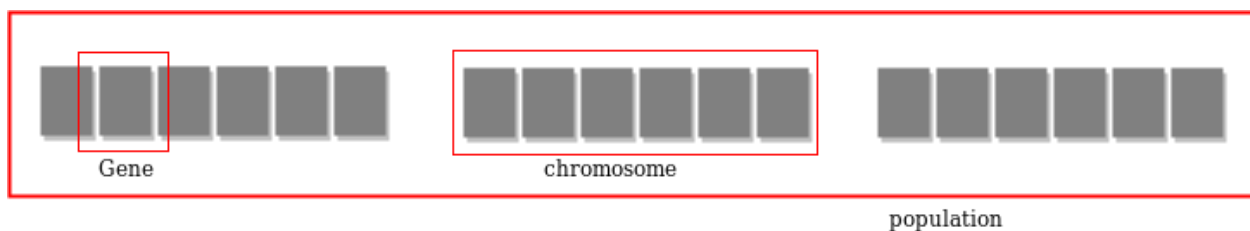
### Foundation of Genetic Algorithms:

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

1. Individuals in the population compete for resources and mate
2. Those individuals who are successful (fittest) then mate to create more offspring than others
3. Genes from the “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 represents 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).



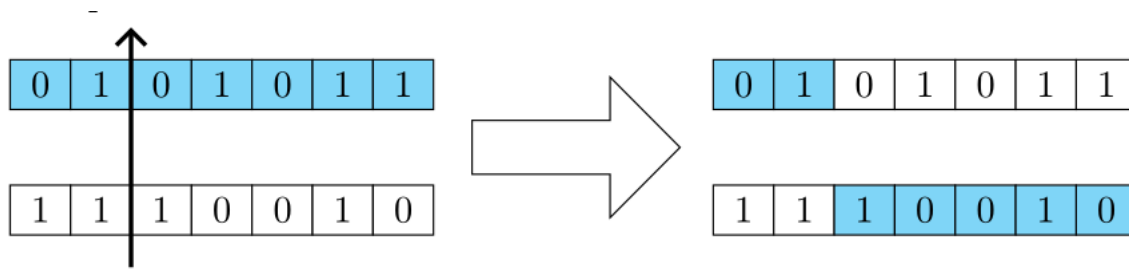
### Operators of Genetic Algorithms

Once the initial generation is created, the algorithm evolves the generation using the following operators.

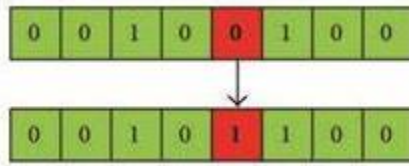
**1) Selection Operator:** The idea is to give preference to the individuals with good fitness scores and allow them to pass their genes to successive generations.

**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 the population to avoid premature convergence. For example



## SOURCE CODE:

```
1  import random
2  POP_SIZE = 6
3  CHROMOSOME_LENGTH = 5
4  GENERATIONS = 10
5  MUTATION_RATE = 0.1
6  def fitness_function(x):
7      return x ** 2
8  def decode_chromosome(chromosome):
9      return int(''.join(map(str, chromosome)), 2)
10 def generate_population():
11     return [[random.randint(0, 1) for _ in range(CHROMOSOME_LENGTH)] for _ in range(POP_SIZE)]
12 def select(population, fitnesses):
13     total_fitness = sum(fitnesses)
14     pick = random.uniform(0, total_fitness)
15     current = 0
16     for i, fitness in enumerate(fitnesses):
17         current += fitness
18         if current > pick:
19             return population[i]
20 def crossover(parent1, parent2):
21     point = random.randint(1, CHROMOSOME_LENGTH - 1)
22     return parent1[:point] + parent2[point:], parent2[:point] + parent1[point:]
23 def mutate(chromosome):
24     return [bit if random.random() > MUTATION_RATE else 1 - bit for bit in chromosome]

25 def genetic_algorithm():
26     population = generate_population()
27     for generation in range(GENERATIONS):
28         fitnesses = [fitness_function(decode_chromosome(individual)) for individual in population]
29         best_individual = max(population, key=lambda ind: fitness_function(decode_chromosome(ind)))
30         best_fitness = fitness_function(decode_chromosome(best_individual))
31         print(f"Generation {generation}: Best Fitness = {best_fitness}, Best Individual = {decode_chromosome}")
32         new_population = []
33         for _ in range(POP_SIZE // 2):
34             parent1 = select(population, fitnesses)
35             parent2 = select(population, fitnesses)
36             offspring1, offspring2 = crossover(parent1, parent2)
37             new_population.extend([mutate(offspring1), mutate(offspring2)])
38         population = new_population
39     genetic_algorithm()
```

## Explanation of Code:

**Initialization:** The generate population function creates a population of random binary chromosomes.

**Fitness Calculation:** Each chromosome's fitness is calculated using the fitness function, which squares the integer representation of the chromosome.

**Selection:** The select function implements roulette wheel selection, where more fit individuals are more likely to be selected.

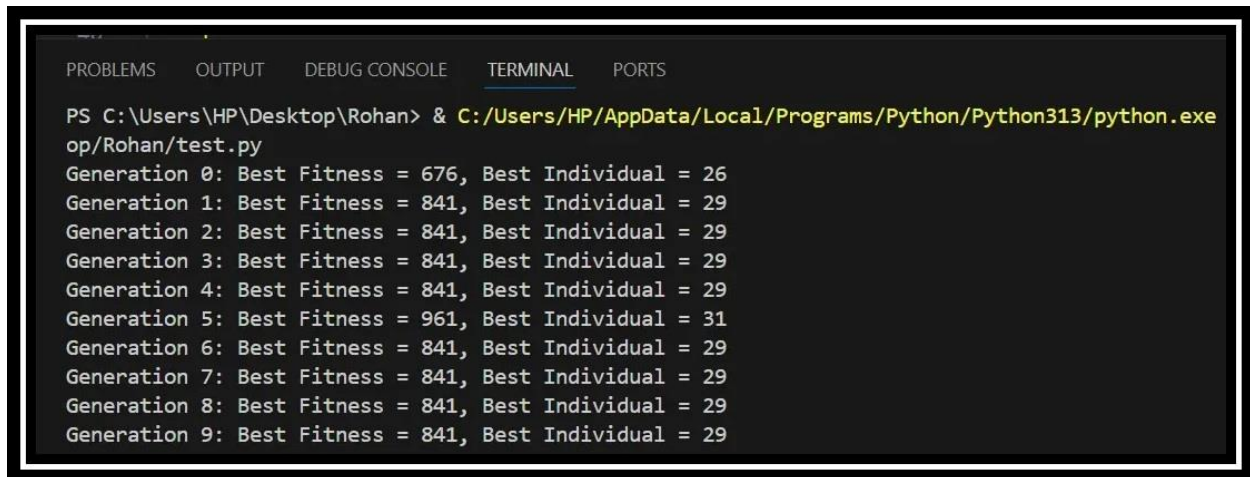
**Crossover:** The crossover function performs a single-point crossover on two parent chromosomes, generating two offspring.

**Mutation:** The mutate function flips bits in a chromosome with a given mutation rate.

**Genetic Algorithm Loop:** The GA's main loop runs for a specified number of generations, performing selection, crossover, and mutation on the population at each step.

**Results:** The algorithm is designed to maximize the fitness function  $f(x)=x^2$ . The best individual (chromosome) is printed in each generation along with its fitness score. Over time, the population evolves to produce better solutions.

## OUTPUT:



```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS C:\Users\HP\Desktop\Rohan> & C:/Users/HP/AppData/Local/Programs/Python/Python313/python.exe
op/Rohan/test.py
Generation 0: Best Fitness = 676, Best Individual = 26
Generation 1: Best Fitness = 841, Best Individual = 29
Generation 2: Best Fitness = 841, Best Individual = 29
Generation 3: Best Fitness = 841, Best Individual = 29
Generation 4: Best Fitness = 841, Best Individual = 29
Generation 5: Best Fitness = 961, Best Individual = 31
Generation 6: Best Fitness = 841, Best Individual = 29
Generation 7: Best Fitness = 841, Best Individual = 29
Generation 8: Best Fitness = 841, Best Individual = 29
Generation 9: Best Fitness = 841, Best Individual = 29
```

## CONCLUSION:

This genetic algorithm implementation demonstrates how evolutionary principles can be applied to optimization problems. The algorithm successfully maximizes the fitness function over generations, showing the power of GAs in solving complex problems. This basic framework can be adapted to other real-world optimization problems such as the knapsack problem, traveling salesman problem, and more.