Beni-Suif University
College of Computers and AI
Department of Computer Science

Lab Manual

Genetic Algorithms

Preparing the scientific material

A.Prof. Hamdi Mahmoud

T.A. Ahmed Sultan

Contents

Beni-Suif University
College of Computers and AI
Department of Computer Science

Lab Manual

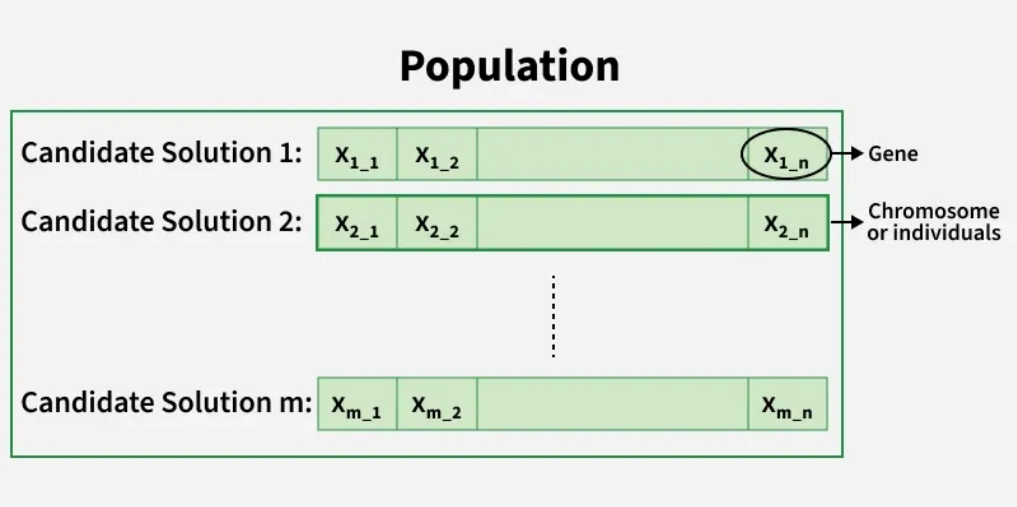
Introduction to Artificial Intelligence

Lab - 1
Introduction to Genetic Algorithms

A Genetic Algorithm (GA) is a population-based evolutionary optimization technique inspired by the principles of natural selection and genetics. It works by iteratively evolving a population of candidate solutions using biologically motivated operators such as selection, crossover and mutation to find optimal or near-optimal solutions to complex problems where traditional optimization techniques are ineffective.

Core Components:-

1. **Population**



A population is a collection of candidate solutions (individuals) that exist at a particular stage (generation) of the genetic algorithm. Instead of working with a single solution, GAs simultaneously evaluate and evolve multiple solutions which helps maintain diversity and reduces the risk of getting trapped in local optima.

- **Population size significantly affects convergence behavior**
- **Larger populations increase exploration but raise computational cost**
- **Smaller populations converge faster but risk premature convergence**

2. **Chromosome**

A chromosome represents a complete candidate solution to the problem. It is a structured collection of genes that encodes all decision variables required to evaluate a solution using the fitness function.

3. **Gene**

A gene is the smallest unit of information in a chromosome and represents a single variable, parameteror trait of the solution. The collective behavior of all genes determines the quality of the chromosome.

4. **Encoding Methods**

Encoding refers to the way candidate solutions are represented inside chromosomes. Choosing an appropriate encoding is critical because it directly impacts the effectiveness of genetic operators.

a. Binary Encoding

- **Uses binary strings (0 and 1)**
- **Simple and easy to implement**
- **Commonly used in theoretical GA models**

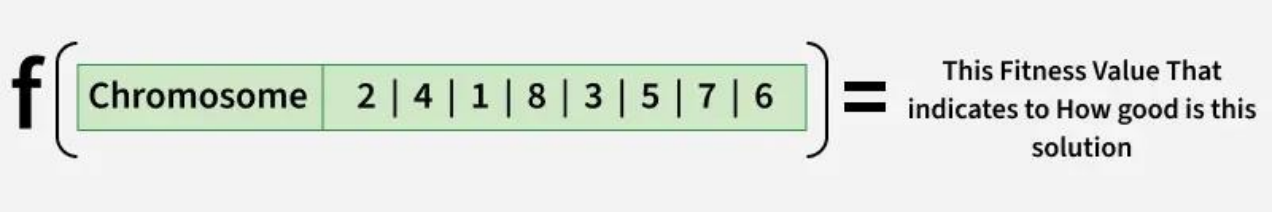
b. Real-Valued Encoding

- **Genes are real numbers**
- **Suitable for continuous optimization problems**
- **Faster convergence and higher precision**

c. Permutation Encoding

- **Chromosomes represent ordered sequences**
- **Used in routing, scheduling and sequencing problems**
- **Requires specialized crossover and mutation operators**

5. Fitness Function



The fitness function is a mathematical formulation that evaluates how well a chromosome solves the given problem. It acts as the guiding force of the genetic algorithm by determining which individuals are more likely to reproduce.

- **Higher fitness implies better solution quality**

- **Fitness function is problem-specific**
- **Can be designed for maximization or minimization**

6. Termination Criteria

Termination criteria define the conditions under which the genetic algorithm stops executing. Proper termination prevents unnecessary computation while ensuring solution quality. Common termination conditions:

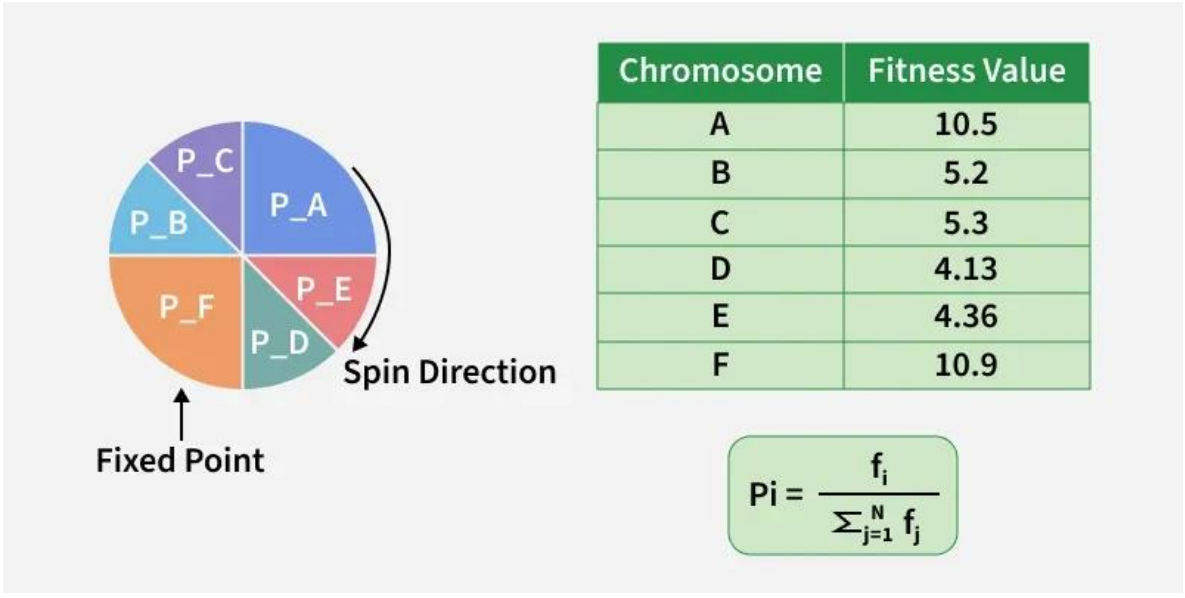
- **Maximum number of generations reached**
- **Desired or threshold fitness achieved**
- **No improvement in fitness for several generations**
- **Computational time limit exceeded**

==========

7. Selection

Selection is the process of choosing chromosomes from the current population to act as parents for the next generation. The goal is to give preference to fitter individuals while still maintaining population diversity. Types of solutions include:

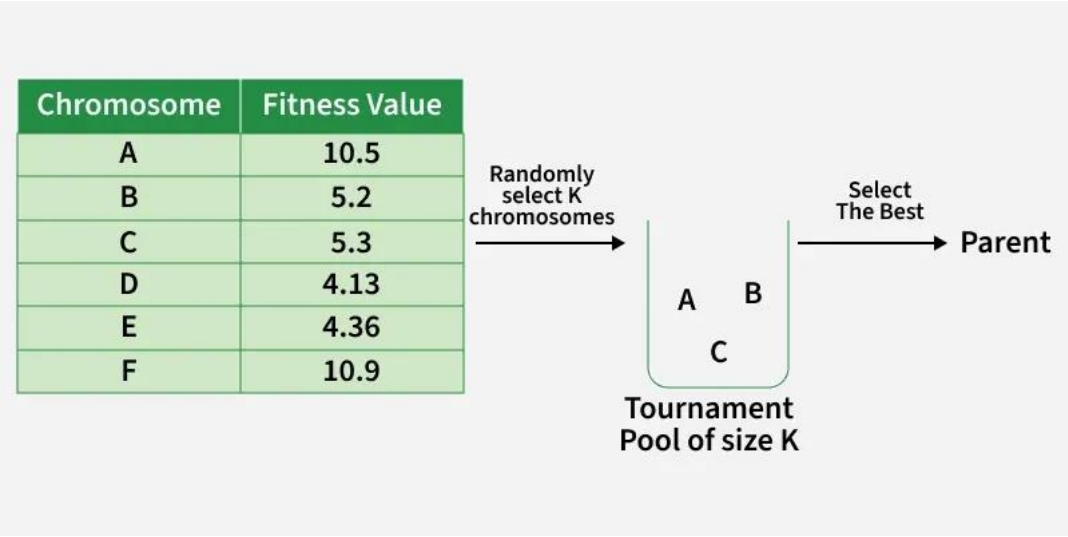
a. Roulette Wheel Selection: It is a fitness-proportionate selection technique where each individual's probability of being selected is directly proportional to its fitness value. Individuals with higher fitness occupy larger segments of the roulette wheel, making them more likely to be chosen.



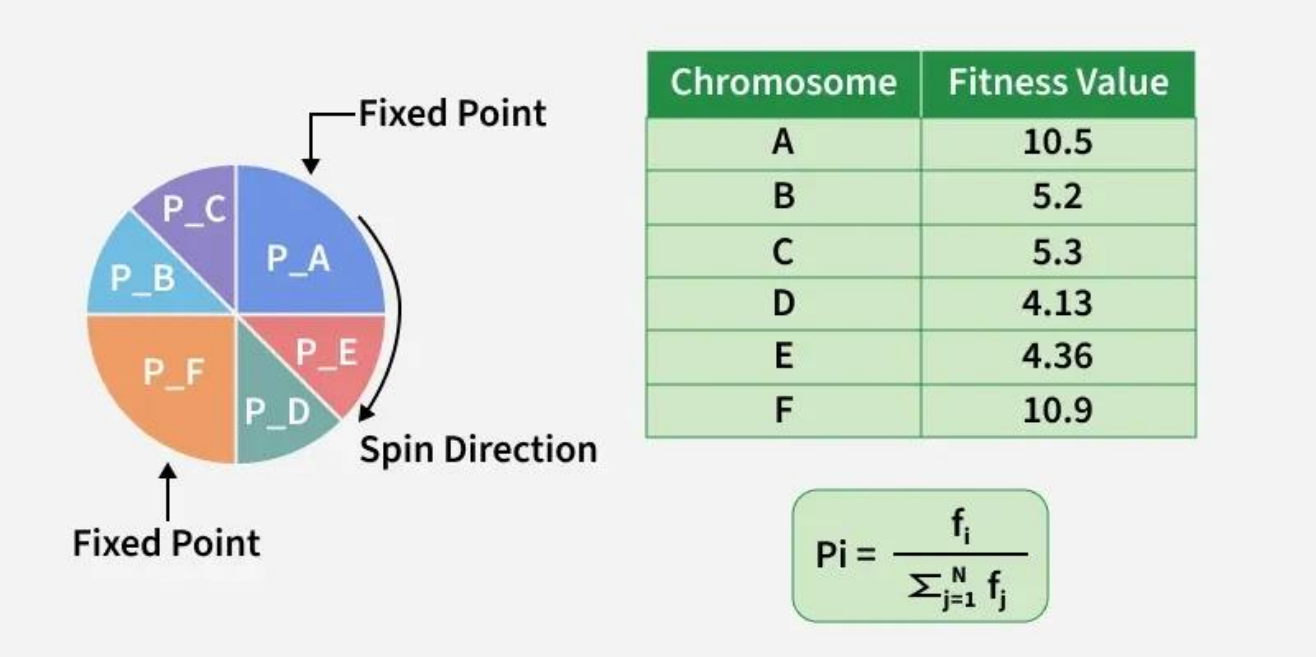
Chromosome	Fitness Value
A	10.5
B	5.2
C	5.3
D	4.13
E	4.36
F	10.9

$$Pi = \frac{f_i}{\sum_{j=1}^{N} f_j}$$

b. Tournament Selection: It randomly selects a small group of individuals from the population and chooses the fittest among them as a parent. This process is repeated until the required number of parents is selected.



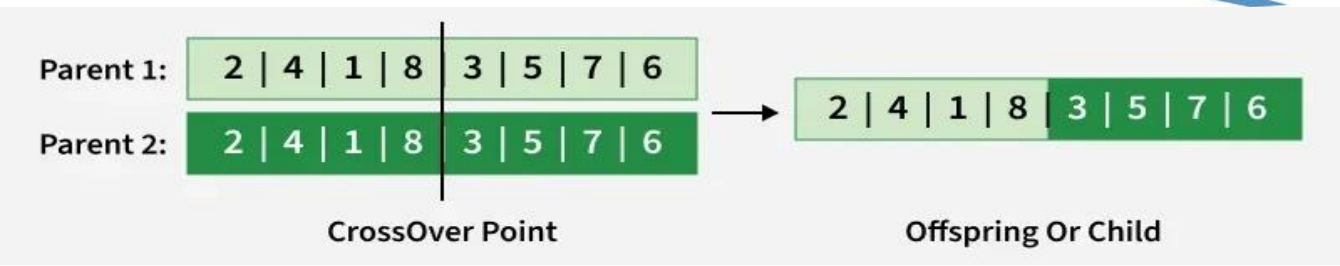
c. Stochastic Universal Sampling (SUS Selection): It is an improved version of fitness-proportionate selection designed to reduce the randomness and sampling bias present in standard roulette wheel selection. Instead of using a single random pointer, SUS uses multiple equally spaced pointers to select individuals from the population.



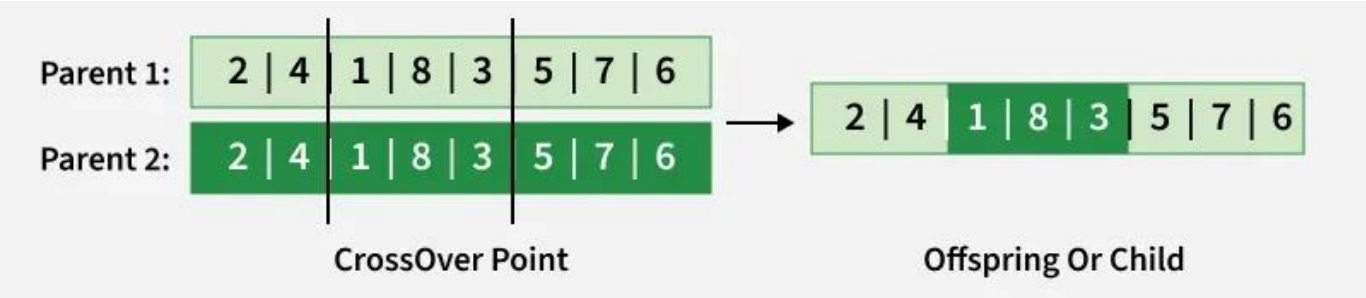
8. CrossOver

Crossover is a genetic operator that combines genetic material from two parent chromosomes to generate new offspring. It enables the algorithm to exploit existing high-quality building blocks. Types of crossover are:

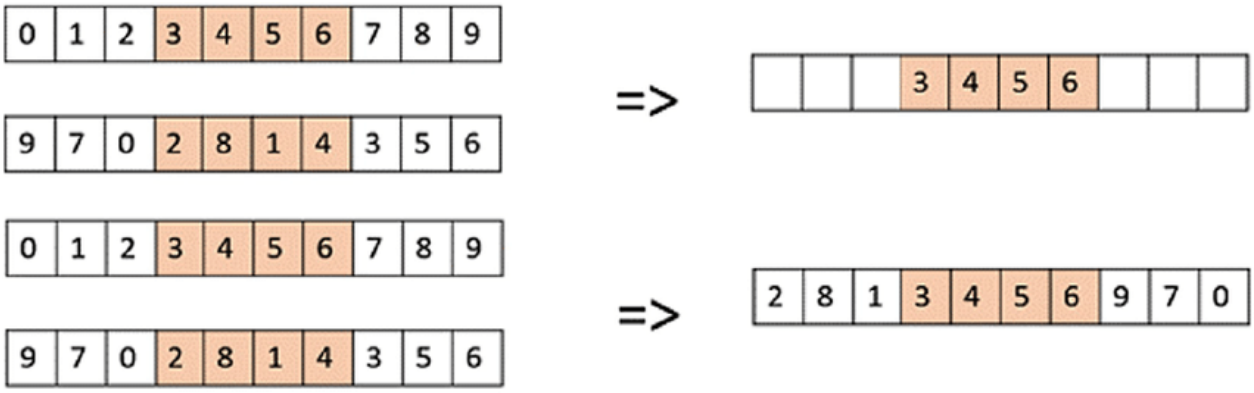
a. One Point Crossover: A random Point is chosen to be The CrossOver Point , then we fill the child with genes from both parents.



b. Multi Point Crossover: A random two Points are chosen to be The CrossOver Points , then we fill the child with genes from both parents.

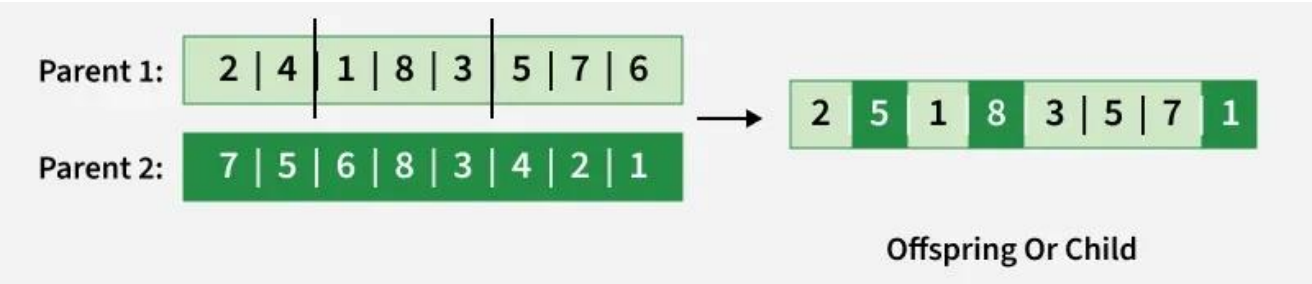


c. Davis Order Crossover (OX1): We Choose two random crossover points in the first parent and we copy that segment into the Child, then we fill the rest of genes in our child with the genes from the second Parent.



Repeat the same procedure to get the second child

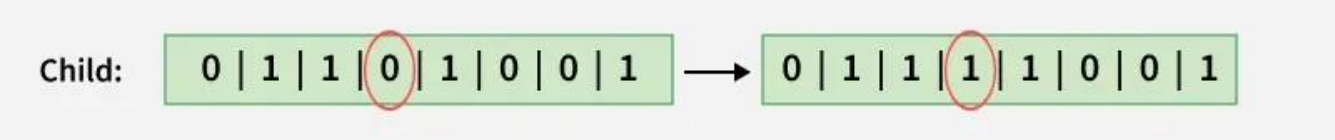
d. Uniform CrossOver: We flip a coin for each genes in our two parents to decide whether or not it'll be included in the off-spring (Child).



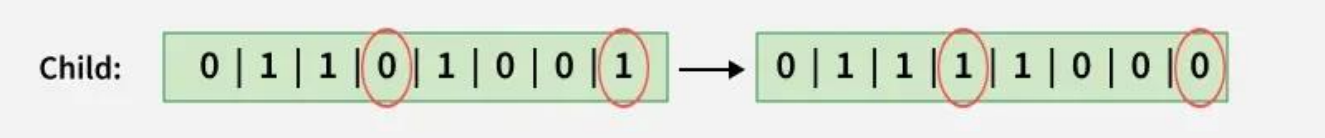
9. Mutation

Mutation introduces random changes in genes to maintain genetic diversity within the population. It helps prevent premature convergence and enables exploration of new solutions. Types of mutation are,

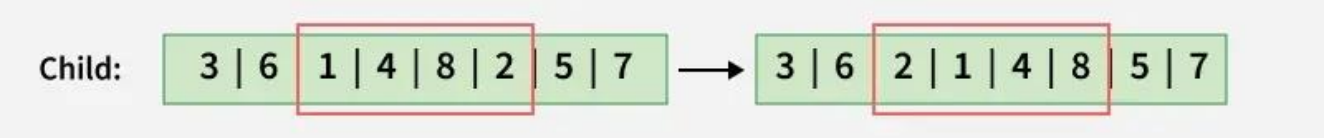
a. Bit flip Mutation: We select one or more random points (Bits) and flip them. This is used for binary encoded Genetic Algorithms.



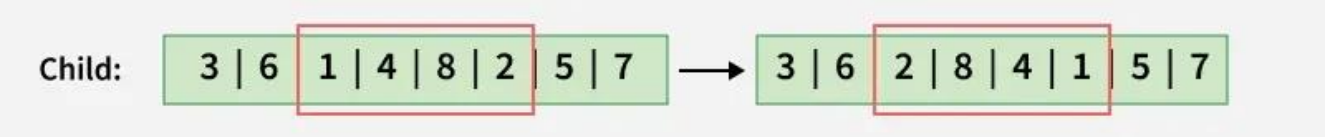
b. Swap Mutation: We Choose two Point and we switch them.



c. Scramble Mutation: We choose a random segment in The Current Chromosome and we interchange the values.



d. Inversion Mutation: We choose a random segment in The Current Chromosome and we reverse The Order of the values.



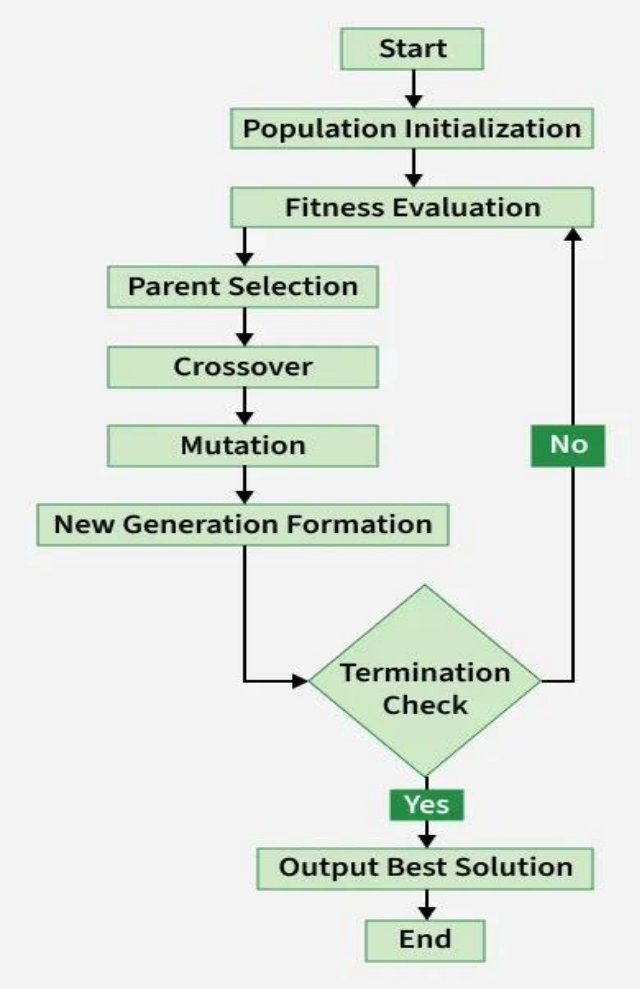
Working of Genetic Algorithms

Let's understand the working of Genetic algorithms:

Working

- **Population Initialization: Generate an initial population of chromosomes randomly within the problem constraints.**
- **Fitness Evaluation: Evaluate each chromosome using the fitness function to measure solution quality.**
- **Parent Selection: Select parent chromosomes based on fitness using methods such as Roulette, Tournamentor SUS selection.**

- **Crossover: Combine genetic material from selected parents to produce offspring.**
- **Mutation: Apply random changes to offspring genes to maintain diversity.**
- **New Generation Formation: Replace the old population with newly generated offspring.**
- **Termination Check: Stop the algorithm if termination criteria are satisfied.**
- **Output Solution: Return the best chromosome obtained during evolution.**



Implementation

Step 1: Import Libraries and Define Fitness Function

- **NumPy is used for numerical computation**
- **Matplotlib for visualization.**
- **Fitness function is multimodal**
- **Demonstrates GA's ability to avoid local optima**

$$f(x) = x \sin(10\pi x) + 1$$

This function is multimodal which means it has a lot of local optima, And this tests the GA's ability not to fall into a Local Optimum and to find the Global Optimum.

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


def fitness_function(x):
    return x * np.sin(10 * np.pi * x) + 1
```

Step 2: Parameter definition and population initialization

- **Population size controls diversity**
- **Mutation and crossover probabilities regulate exploration**
- **Random seed ensures reproducibility**

```
POP_SIZE = 40   //عدد الافراد
GENERATIONS = 100       // عدد الاجيال
X_MIN, X_MAX = -1.0, 2.0  // القيم اللي هيبحث فيها
CROSSOVER_PROB = 0.9
MUTATION_PROB = 0.2
MUTATION_STD = 0.1
```

- احتمال التزاوج = ٩٠ %
- احتمال الطفرة = ٢٠ %
- مقدار الانحراف في الطفرة = ٠,١

```
np.random.seed(42) // عشان كل مرة البرنامج يدي نفس النتيجة (Reproducibility)
population = np.random.uniform(X_MIN, X_MAX, POP_SIZE)// هيولد ٤٠ رقم عشوائي وكل رقم بيمثل فرد
```

Step 3: Genetic operators

- **Tournament selection improves robustness**
- **Arithmetic crossover blends real values**
- **Mutation maintains diversity**

```
//هيختار تلات اقراد عشوائيا وهيختار الافضل بينهم
def tournament_selection(pop, fitness, k=3):
    selected = []
   for _ in range(len(pop)):
      idx = np.random.choice(len(pop), k, replace=False)
      selected.append(pop[idx[np.argmax(fitness[idx])]])
   return np.array(selected)
```

```
def arithmetic_crossover(p1, p2):
  alpha = np.random.rand()
  return alpha * p1 + (1 - alpha) * p2, alpha * p2 + (1 - alpha) * p1

def mutate(x):
  if np.random.rand() < MUTATION_PROB:
    x += np.random.normal(0, MUTATION_STD)
  return np.clip(x, X_MIN, X_MAX)
```

لو حصلت طفرة:

- نضيف Noise Gaussian
- نمنع القيمة تطلع بره المجال

الطفرة:

- بتمنع الانحباس في Local Optimum
- بتحافظ على التنوع

Step 4: Evolution loop

- Tracks convergence behavior
- Applies full GA cycle
- Replaces old population

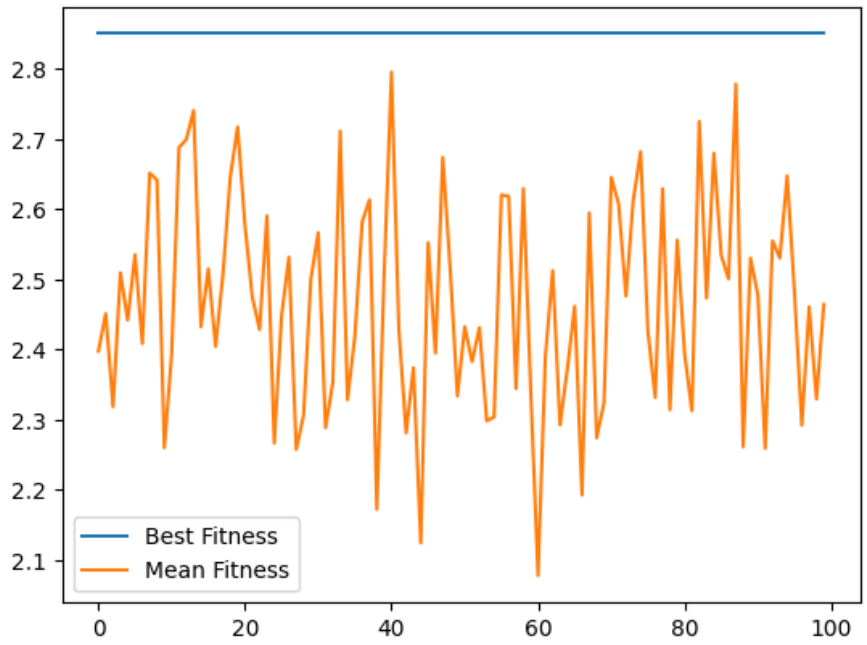
```
best_history = []
mean_history = []
for _ in range(GENERATIONS):
  fitness = fitness_function(population)
  best_history.append(np.max(fitness))
  mean_history.append(np.mean(fitness))
```

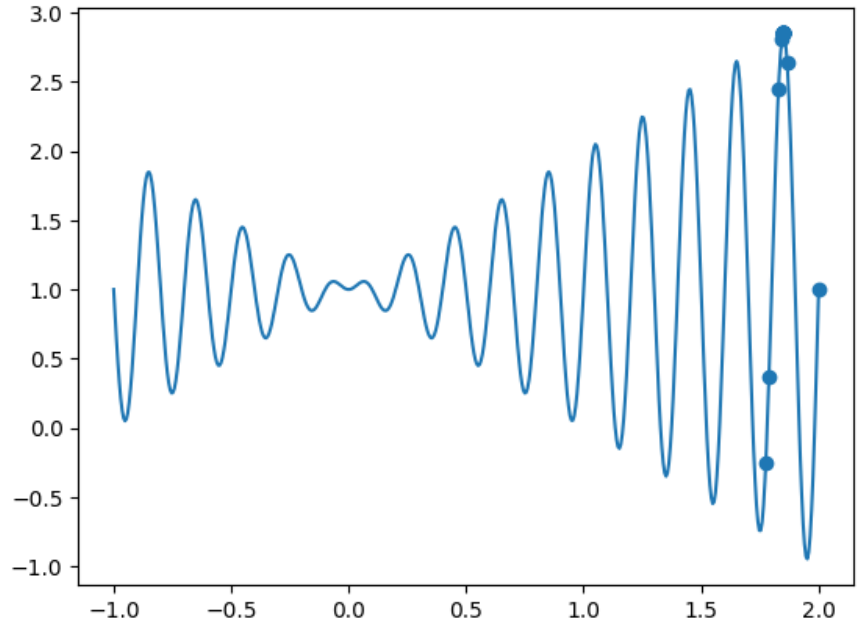
```
    parents = tournament_selection(population, fitness)
    offspring = []
    np.random.shuffle(parents)
    for i in range(0, POP_SIZE, 2):
        if np.random.rand() < CROSSOVER_PROB:
            c1, c2 = arithmetic_crossover(parents[i], parents[i + 1])
        else:
            c1, c2 = parents[i], parents[i + 1]
        offspring.extend([mutate(c1), mutate(c2)])
    population = np.array(offspring)
```

Step 5: Visualization

- **Fitness curve shows convergence**
- **Scatter plot shows population distribution**

```
x = np.linspace(X_MIN, X_MAX, 500)

plt.figure()
plt.plot(best_history, label="Best Fitness")
plt.plot(mean_history, label="Mean Fitness")
plt.legend()
plt.show()

plt.figure()
plt.plot(x, fitness_function(x))
plt.scatter(population, fitness_function(population))
plt.show()
```





Full Code :-

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

def fitness_function(x):
    return x * np.sin(10 * np.pi * x) + 1

POP_SIZE = 40
GENERATIONS = 100
X_MIN, X_MAX = -1.0, 2.0
```

```
CROSSOVER_PROB = 0.9
MUTATION_PROB = 0.2
MUTATION_STD = 0.1

np.random.seed(42)
population = np.random.uniform(X_MIN, X_MAX, POP_SIZE)


def tournament_selection(pop, fitness, k=3):
    selected = []
    for _ in range(len(pop)):
        idx = np.random.choice(len(pop), k, replace=False)
        selected.append(pop[idx[np.argmax(fitness[idx])]])
    return np.array(selected)

def arithmetic_crossover(p1, p2):
    alpha = np.random.rand()
    return alpha * p1 + (1 - alpha) * p2, alpha * p2 + (1 - alpha) * p1

def mutate(x):
    if np.random.rand() < MUTATION_PROB:
        x += np.random.normal(0, MUTATION_STD)
    return np.clip(x, X_MIN, X_MAX)


best_history = []
mean_history = []
for _ in range(GENERATIONS):
    fitness = fitness_function(population)
    best_history.append(np.max(fitness))
    mean_history.append(np.mean(fitness))
    parents = tournament_selection(population, fitness)
    offspring = []
    np.random.shuffle(parents)
    for i in range(0, POP_SIZE, 2):
        if np.random.rand() < CROSSOVER_PROB:
            c1, c2 = arithmetic_crossover(parents[i], parents[i + 1])
        else:
            c1, c2 = parents[i], parents[i + 1]
        offspring.extend([mutate(c1), mutate(c2)])
    population = np.array(offspring)
x = np.linspace(X_MIN, X_MAX, 500)

plt.figure()
plt.plot(best_history, label="Best Fitness")
plt.plot(mean_history, label="Mean Fitness")
plt.legend()
plt.show()
```

```
plt.figure()
plt.plot(x, fitness_function(x))
plt.scatter(population, fitness_function(population))
plt.show()
```

Beni-Suif University
College of Computers and AI
Department of Computer Science

Lab Manual

Knapsack Problem

Lab - 2
Knapsack Problem

Lab - 2
Knapsack Problem

Knapsack Problem using Genetic Algorithm

1. **Problem Definition**

The **Knapsack Problem** is an optimization problem where we must choose a set of items to place in a knapsack so that the **total value is maximized** while the **total weight does not exceed the knapsack capacity**.

Knapsack capacity: **110 kg**

Item	Value	Weight
A	50	30
B	30	50
C	20	40
D	30	20
E	50	60

Objective

Select a combination of items from **A, B, C, D, and E** such that:

1. The **total weight ≤ 110 kg**
2. The **total value is maximized**

Mathematical Representation

Let:

xi = 1 → item is included

xi = 0 → item is not included

Objective Function (Maximize Value)

Maximize:

$$Z = 50x1 + 30x2 + 20x3 + 30x4 + 50x5$$

2. Chromosome Representation

Binary representation: 1 = item selected, 0 = item not selected.

Example chromosome: 1 0 1 1 0 → Items A, C, D selected.

A B C D E

3. Initial Population

In a **Genetic Algorithm**, the first step after defining the problem is to create an **initial population**. The population consists of several **chromosomes**, where each chromosome represents a **possible solution** to the knapsack problem.

Each chromosome represents a **candidate solution**, and in the next step the **fitness function will evaluate these solutions** based on their **total weight and total value**.

Chromosome	A	B	C	D	E
S1	1(gene)	0(gene)	1(gene)	0(gene)	1(gene)
S2	1	1	0	1	0
S3	0	1	1	1	0
S4	1	1	1	0	0

4. Fitness Calculation

After generating the **initial population**, the next step in the Genetic Algorithm is to evaluate each chromosome using a **fitness function**. The fitness function determines how good each solution is for the knapsack problem.

In this problem, the **fitness value is the total value of the selected items**, provided that the **total weight does not exceed the knapsack capacity (110 kg)**. If the total weight of the selected items exceeds the capacity, the solution is considered **invalid**, and its fitness value is set to **0**.

Solution	Chromosome	Weight	Value	Fitness
S1	10101	130	120	0
S2	11010	100	110	110
S3	01110	110	80	80
S4	11100	120	100	0

5. Selection (Roulette Wheel)

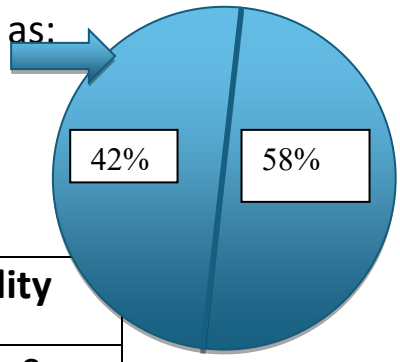
After calculating the **fitness values** of all chromosomes in the population, the next step in the Genetic Algorithm is **selection**. The purpose of selection is to choose chromosomes that will act as **parents** for producing the next generation.

One common method used is **Roulette Wheel Selection**. In this method, the probability of selecting a chromosome is **proportional to its fitness value**. This means that chromosomes with **higher fitness values have a greater chance of being selected**, but weaker solutions may still have a small chance to maintain diversity in the population.

The probability of selecting each chromosome is calculated as:

Total fitness = S2+S3 = 110+80 = 190

$$Probability = \frac{Fitness}{Total\ Fitness}$$



Solution	Chromosome	Fitness	Probability
S1	10101	0	0 / 190 = 0
S2	11010	110	110 / 190 = 0.58

S3	01110	80	80 / 190 = 0.42
S4	11100	0	0 / 190 = 0

Selection probabilities:

S2 = 110/190 ≈ 0.58

S3 = 80/190 ≈ 0.42

Selected parents: S2 and S3

6. Crossover

Crossover is a genetic operator used to **combine the genes of two parent chromosomes** to produce **new offspring chromosomes**. It simulates the biological reproduction process.

Step 1: Choose Crossover Point

Assume the crossover point is **after gene 3**.

Parent 1 : 110 | 10

Parent 2 : 011 | 10

Step 2: Exchange Genes

Parent 1 : 110 | 10

Parent 2 : 011 | 10

↓

Child 1 : 110 | 10

Child 2 : 011 | 10

Resulting children:

Child	Chromosome
Child 1 :	11010
Child 2 :	01110

Parent1 = 11010

Parent2 = 01110

Crossover after gene 3 → children remain similar in this example.

7. Mutation

Example mutation: 11010 → 11011

8. New Population

Chromosome	Items	Weight	Value	Fitness
The best arent 1 11010	A,B,D	100	110	110
Parent 2 01110	B,C,D	110	80	80
Mutated 11011	A,B,D,E	160	160	0
Cross over 10110	A,C,D	90	100	100

9. Final Best Solution

Chromosome: 11010

Selected Items: A, B, D

Total Weight = 100 kg

Maximum Value = 110

Lap to implement:-

```
import random
import numpy as np
import matplotlib.pyplot as plt
# Items A-E
items = ['A','B','C','D','E','F','G','H']

# Values of items
values = [50, 30, 20, 30, 50,60,80,90]

# Weights of items
weights = [30, 50, 40,20, 60,100,50,120]

# Maximum capacity of knapsack
capacity = 200

num_items = len(items)

population_size = 10
generations = 100
mutation_rate = 0.01


def create_population():

    population = []

    for _ in range(population_size):
        chromosome = [random.randint(0,1) for _ in range(num_items)]
        population.append(chromosome)

    return population



def fitness(chromosome):

    total_value = 0
    total_weight = 0

    for i in range(num_items):

        if chromosome[i] == 1:
            total_value += values[i]
            total_weight += weights[i]

    if total_weight > capacity:
```

```
        return 0

    return total_value


def selection(population):

    fitness_values = [fitness(ch) for ch in population]
    total_fitness = sum(fitness_values)

    if total_fitness == 0:
        return random.choice(population)

    probabilities = [f/total_fitness for f in fitness_values]

    selected = random.choices(population, probabilities)[0]

    return selected


def crossover(parent1, parent2):

    point = random.randint(1, num_items-1)

    child = parent1[:point] + parent2[point:]

    return child
def mutation(chromosome):

    for i in range(num_items):

        if random.random() < mutation_rate:
            chromosome[i] = 1 - chromosome[i]

    return chromosome


best_fitness_history = []

for generation in range(generations):

    new_population = []

    for _ in range(population_size):

        parent1 = selection(population)
        parent2 = selection(population)
```

```
            child = crossover(parent1, parent2)

            child = mutation(child)

            new_population.append(child)

        population = new_population

        best = max(population, key=fitness)
        best_fitness_history.append(fitness(best))

best_solution = max(population, key=fitness)


print("Best Chromosome:", best_solution)

selected_items = [items[i] for i in range(num_items) if best_solution[i] == 1]

print("Selected Items:", selected_items)
print("Best Fitness (Total Value):", fitness(best_solution))

total_weight = sum(weights[i] for i in range(num_items) if best_solution[i]==1)

print("Total Weight:", total_weight)




plt.figure()

plt.plot(best_fitness_history)

plt.title("Fitness Improvement Over Generations")

plt.xlabel("Generation")

plt.ylabel("Best Fitness")

plt.show()



selected = [best_solution[i] for i in range(num_items)]

plt.figure()

plt.bar(items, selected)
```

```
plt.title("Items Selected in Final Solution")

plt.xlabel("Items")

plt.ylabel("Selected (1=yes, 0=no)")

plt.show()
```

Beni-Suef University
College of Computers and AI
Department of Computer Science

Lab Manual

PhenoTypes & GenoTypes

Lab - 3
PhenoTypes & GenoTypes

Lab - 3
PhenoTypes & GenoTypes

Phenotype and Genotype are two terms used in Genetics. Genetics is a branch of science that deals with genes, heredity, and genetic variations. These terms are given by Wilhelm Johannsen in 1909. The concepts of Phenotype and Genotype are further included in The Genotype Conception of Heredity in 1911. *The major difference between Genotype and Phenotype is that Genotype is the genetic makeup of an organism while the Phenotype is the external physical appearance of an organism.*

Features	Phenotype	Genotype
Definition	The external appearance of an organism	Genetic composition of an organism
Inheritance	Not inherited from parents to offspring	Inherited from parents to offspring
Determination	Determined by observing the organism	Determined by biological methods like PCR
Effect of Environment	Affected by environmental conditions	Affected by genes
Visibility	Visible	Not visible
Appearance	Physical appearance outside the body	Genetic material inside the body
Change over time	Changes with time	Does not change with time

Relationship	**It's not necessary that organisms that have the same Phenotype will also have the same Genotype**	**One Genotype produces only one Phenotype.**

Example:-

Concept	Example
Genotype	Student ID number
Decoder	University database
Phenotype	Student information
Fitness	GPA

Genotype

The genotype is the encoded representation of a candidate solution used internally by the algorithm. It belongs to the genotype search space:

G={g1,g2,...,gn}

Characteristics

Property	Description
Internal form	Used by algorithm
Manipulated by GA	Yes
Human readable	Not required
Subject to mutation	Yes
Subject to crossover	Yes

Example:-

Genotype = 101101 (This is only information encoding — not the real solution.)

Phenotype

Definition

The phenotype is the decoded and interpretable solution derived from the genotype.

It exists in the problem domain.

P=decode(G)

Characteristics

Property	Description
Real solution	Yes
Evaluated by fitness	Yes
Manipulated directly	No
Meaningful to problem	Yes

Example

Genotype: 101101

Phenotype: x = 45

Fitness: f(45)

Types of Genotype Encoding

Binary Encoding (Classic GA)

Most traditional representation.

Genotype

10110

Phenotype conversion

$$x = \sum b_i 2^i$$

Example:

$10110_2 = 22_{10}$

Integer Encoding .٢

Used in scheduling and ordering problems.

Genotype:

[3,1,4,2]

Phenotype:

visit order of cities

<u>How to Convert Between Genotype and Phenotype?</u>

Binary → Real Value Conversion .١

Suppose:

$$x \in [a, b]$$

Binary string length = L

Step 1 — Convert Binary to Integer

$$I = int(binary)$$

Step 2 — Normalize

$$x = a + \frac{I}{2^L - 1}(b - a)$$

Example

Binary:

GenoType = 10110

Decimal:

$$22$$

Range:

$$[0, 10]$$

Then:

$$x = 0 + \frac{22}{31}(10) = 7.09$$

Phenotype = 7.09

Real → Binary (Encoding) .٢

Reverse mapping:

$$I = \frac{(x-a)(2^L-1)}{b-a}$$

Convert integer → binary.

Example

$$x = 7.09$$

Compute integer ≈ 22

Binary:

10110

Lab to implement :-

```python
# Binary genotype → integer phenotype
def binary_to_integer(genotype: str) -> int:
    return int(genotype, 2)

# Integer phenotype → binary genotype
def integer_to_binary(value: int, bits: int) -> str:
    return format(value, f'0{bits}b')

# Example
g = "10110"
p = binary_to_integer(g)
print("Genotype:", g)
print("Phenotype:", p)
print("Back to genotype:", integer_to_binary(p, len(g)))
```

```python
# Binary genotype → real phenotype
def binary_to_real(genotype, a, b):
    L = len(genotype) #10110
```

```
    integer = int(genotype, 2)
    return a + integer * (b - a) / (2**L - 1)

# Real phenotype → binary genotype
def real_to_binary(x, a, b, bits):
    integer = round((x - a) * (2**bits - 1) / (b - a))
    return format(integer, f'0{bits}b')

# Example
g = "10110"
real_value = binary_to_real(g, 0, 10)
print("Real phenotype:", real_value)

g_back = real_to_binary(real_value, 0, 10, len(g))
print("Recovered genotype:", g_back)
```

```
# Binary → Gray
def binary_to_gray(binary):
    gray = binary[0]
    for i in range(1, len(binary)):
        gray += str(int(binary[i-1]) ^ int(binary[i]))
    return gray

# Gray → Binary
def gray_to_binary(gray):
    binary = gray[0]
    for i in range(1, len(gray)):
        binary += str(int(binary[i-1]) ^ int(gray[i]))
    return binary

# Example
b = "1011"
g = binary_to_gray(b)
print("Gray code:", g)
print("Recovered binary:", gray_to_binary(g))
```

```
# Real-coded representation (genotype == phenotype)
genotype = [2.3, -1.5, 4.8]

def evaluate(solution):
    return sum(x**3+1 for x in solution)
```

```
print("Genotype:", genotype)
print("Phenotype:", genotype)
print("Fitness:", evaluate(genotype))
```

```
import random

# Genotype: permutation
genotype = [3, 1, 4, 2]

# Phenotype interpretation (example path length)
def path_length(order):
    # dummy distance function
    return sum(abs(order[i] - order[i+1]) for i in range(len(order)-1))

print("Genotype:", genotype)
print("Phenotype (route interpretation):", genotype)
print("Fitness (path length):", path_length(genotype))

# Mutation example (swap mutation)
def swap_mutation(chromosome):
    a, b = random.sample(range(len(chromosome)), 2)
    chromosome[a], chromosome[b] = chromosome[b], chromosome[a]
    return chromosome
new_geno=swap_mutation(genotype.copy())
print("Mutated genotype:", new_geno)


print("New Genotype:", new_geno)
print("Phenotype (route interpretation):", new_geno)
print("Fitness (path length):", path_length(new_geno))
```

Beni-Suef University
College of Computers and AI
Department of Computer Science

Lab Manual

Traveler Saleman Problem TSP

Lab - 4

Informed Search & Optimization

A*, Greedy, Hill-Climbing, Genetic Algorithms

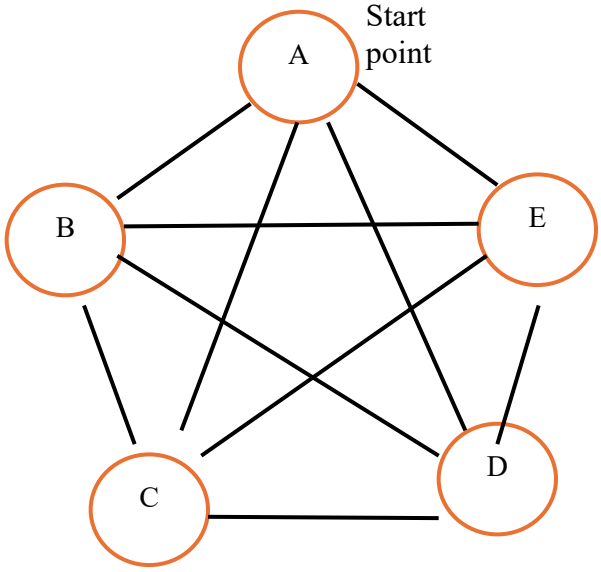
Lab - 4
Traveler Sales-Man Problem (TSP)

TSP Imagine that you manage a **small fulfillment center** and need to deliver packages to a list of customers using a single vehicle. What is the best route for the vehicle to take so that you visit all your customers and then return to the starting point? This is an example of the classic TSP.

The TSP dates back to 1930, and since then has been and is one of the most thoroughly studied problems in **optimization**. It is often used to benchmark optimization algorithms. The problem has many variants, but it was originally based on a traveling salesman that needs to take a trip covering several cities:

"Given a list of cities and the distances between each pair of the cities, find the shortest possible path that goes through all the cities, and returns to the starting city.

Note :-Using combinatorics, you could find that when given n cities, the number of possible paths that go through all cities is (n − 1)!/2



# of Cites = (5-1)! / 2= 12

#	Tour (Cycle)
1	A → B → C → D → E → A
2	A → B → C → E → D → A
3	A → B → D → C → E → A
4	A → B → D → E → C → A
5	A → B → E → C → D → A
6	A → B → E → D → C → A
7	A → C → B → D → E → A
8	A → C → B → E → D → A
9	A → C → D → B → E → A
10	A → C → E → B → D → A
11	A → D → B → C → E → A
12	A → D → C → B → E → A

The following screenshot represents the shortest path for The traveling salesperson problem that covers the 15 largest cities in Germany:

As in this case n=15, the number of possible routes is (14!)/2, which calculates to the staggering number of **43,589,145,600**.

Note :- You can visualize how to solve the problem from this Travelling Salesman | Visualize It

TSP using genoType and phenotype:-

- **Genotype** (the "DNA" of the solution): A chromosome = a list of city indices in visit order (a permutation). Because it's a cycle, we normalize it to always start at city 0 (A). Example: [0, 1, 2, 3] = visit A → B → C → D → A.

- **Phenotype** (the decoded real-world solution): The actual tour with city labels + total distance cost. Fitness = 1 / cost (shorter tour → higher fitness → more likely to survive).

GENOTYPE (DNA)

↓ encoding

[0,2,1,4,3]

↓ decoding

PHENOTYPE (behavior)

A → C → B → E → D → A

↓ evaluation

FITNESS

Total distance = 24.6

How GA Solves TSP (Step-by-Step Process)

1. Create a random **population** of genotypes (random tours).
2. Evaluate each genotype → compute its **phenotype** (tour + cost + fitness).
3. **Selection**: Pick good parents using tournament selection.
4. **Crossover** (Order Crossover - OX): Combine parents while preserving valid permutations.
5. **Mutation** (swap two cities): Add small random changes.
6. Replace old population with new children → repeat for many generations.
7. The best phenotype (lowest cost tour) evolves over time.

Implementation :-

```
"""
============================================================
GENETIC ALGORITHM FOR TSP
Genotype-Phenotype Illustration Version
============================================================

GENOTYPE  : permutation of city indices
PHENOTYPE : decoded travel route
FITNESS   : inverse tour cost (scaled)

Pipeline:
Initialize → Decode → Evaluate → Select → Crossover → Mutate
→ Elitism → Repeat → Optimal Tour
============================================================
"""

import numpy as np
import matplotlib.pyplot as plt
import random

# ------------------------------------------------------------
# 1 REPRODUCIBILITY
# Fix randomness so experiments are repeatable
# ------------------------------------------------------------
np.random.seed(42)
random.seed(42)

# ------------------------------------------------------------
# 2 PROBLEM DEFINITION (TSP WORLD)
# Cities = environment where phenotype exists
# ------------------------------------------------------------
cities_coords = np.array([
    [0, 0],   # A
```

```
    [1, 3],   # B
    [4, 2],   # C
    [3, 0]    # D
])

city_labels = ['A', 'B', 'C', 'D']
n_cities = len(cities_coords)

# Distance matrix (phenotype evaluation space)
dist_matrix = np.linalg.norm(
    cities_coords[:, None, :] - cities_coords[None, :, :],
    axis=2
)

# ---------------------------------------------------------------
# 3GA PARAMETERS
# Evolution control variables
# ---------------------------------------------------------------
population_size = 30
generations = 40
mutation_rate = 0.25
elite_size = 1


# ---------------------------------------------------------------
# 4GENOTYPE CREATION
# Chromosome = permutation of cities
# Normalize so city A (0) always first
# ---------------------------------------------------------------
def normalize(chrom):
    idx = chrom.index(0)
    return chrom[idx:] + chrom[:idx]


def create_random_chromosome():
    chrom = list(range(n_cities))
    random.shuffle(chrom)
    return normalize(chrom)


# ---------------------------------------------------------------
# 5 GENOTYPE → PHENOTYPE DECODING
# Converts genetic encoding into real tour
# ---------------------------------------------------------------
def decode(chrom):
    return [city_labels[i] for i in chrom] + [city_labels[chrom[0]]]
```

```python
# ------------------------------------------------------------
# 6 PHENOTYPE EVALUATION (COST)
# Computes physical tour length
# ------------------------------------------------------------
def calculate_cost(chrom):
    cost = 0
    for i in range(n_cities):
        cost += dist_matrix[chrom[i],
                            chrom[(i + 1) % n_cities]]
    return cost


# ------------------------------------------------------------
# 7 FITNESS SCALING
# Stable alternative to 1/cost
# Prevents selection explosion
# ------------------------------------------------------------
def calculate_fitness(costs):
    max_cost = max(costs)
    return [max_cost - c + 1e-6 for c in costs]


# ------------------------------------------------------------
# 8 SELECTION (Tournament)
# Survival of the fittest
# ------------------------------------------------------------
def tournament_selection(population, fitnesses, k=3):
    selected = random.sample(range(len(population)), k)
    best = max(selected, key=lambda i: fitnesses[i])
    return population[best][:]


# ------------------------------------------------------------
# 9 ORDER CROSSOVER (OX)
# Permutation-safe recombination
# Preserves city order information
# ------------------------------------------------------------
def order_crossover(p1, p2):
    size = len(p1)
    start, end = sorted(random.sample(range(size), 2))

    child = [None] * size
    child[start:end] = p1[start:end]

    idx = end % size
    for gene in p2:
        if gene not in child:
            while child[idx] is not None:
```

```
                idx = (idx + 1) % size
            child[idx] = gene
            idx = (idx + 1) % size

    return normalize(child)


# ------------------------------------------------------------
# 10 MUTATION (Swap Mutation)
# Introduces diversity into genotype space
# ------------------------------------------------------------
def mutate(chrom):
    if random.random() < mutation_rate:
        i, j = random.sample(range(n_cities), 2)
        chrom[i], chrom[j] = chrom[j], chrom[i]
    return normalize(chrom)


# ------------------------------------------------------------
# 11 VISUALIZATION
# Shows phenotype (actual route)
# ------------------------------------------------------------
def plot_tsp(tour, title):
    plt.figure(figsize=(7,5))
    plt.title(title)

    plt.scatter(cities_coords[:,0], cities_coords[:,1], s=150)

    for i, label in enumerate(city_labels):
        plt.text(cities_coords[i,0]+0.1,
                 cities_coords[i,1]+0.1,
                 label, fontsize=12)

    route = tour + [tour[0]]

    for i in range(len(route)-1):
        p1 = cities_coords[route[i]]
        p2 = cities_coords[route[i+1]]
        plt.plot([p1[0], p2[0]],
                 [p1[1], p2[1]], linewidth=2)

    plt.grid(True)
    plt.show()


# ------------------------------------------------------------
# 12 INITIAL POPULATION
# Random solutions (genotype space)
```

```
# ------------------------------------------------------------
population = [create_random_chromosome()
              for _ in range(population_size)]

best_cost_overall = float('inf')
best_chrom_overall = None
history = []


# ------------------------------------------------------------
# 13 EVOLUTION LOOP
# Core GA lifecycle
# ------------------------------------------------------------
for gen in range(generations + 1):

    costs = [calculate_cost(ch) for ch in population]
    fitnesses = calculate_fitness(costs)

    best_idx = np.argmax(fitnesses)
    best_cost = costs[best_idx]
    best_chrom = population[best_idx]

    # Save global best (ELITISM MEMORY)
    if best_cost < best_cost_overall:
        best_cost_overall = best_cost
        best_chrom_overall = best_chrom[:]

    history.append(best_cost_overall)

    print(f"\n===== GENERATION {gen} =====")
    for i, ch in enumerate(population):
        print(
            f"Ind {i+1}: "
            f"Genotype {ch} → "
            f"Phenotype {'→'.join(decode(ch))} "
            f"| Cost {costs[i]:.2f}"
        )

    if gen == generations:
        break

    # ---------------- Evolution ----------------
    new_population = []

    # ✅ ELITISM (keep best solution)
    new_population.append(best_chrom[:])

    while len(new_population) < population_size:
```

```
        p1 = tournament_selection(population, fitnesses)
        p2 = tournament_selection(population, fitnesses)

        child = order_crossover(p1, p2)
        child = mutate(child)

        new_population.append(child)

    population = new_population


# ------------------------------------------------------------
# 14 RESULTS
# Best phenotype discovered
# ------------------------------------------------------------
print("\n========== FINAL RESULT ==========")
print("Optimal Tour:",
      " → ".join(decode(best_chrom_overall)))
print("Minimum Cost:", round(best_cost_overall, 3))

plot_tsp(best_chrom_overall,
         f"Optimal Tour (Cost {best_cost_overall:.2f})")


# ------------------------------------------------------------
# 15 CONVERGENCE VISUALIZATION
# Shows evolution learning behaviour
# ------------------------------------------------------------
plt.figure()
plt.plot(history)
plt.title("GA Convergence Curve")
plt.xlabel("Generation")
plt.ylabel("Best Cost")
plt.grid(True)
plt.show()
```

Beni-Suef University
College of Computers and AI
Department of Computer Science

Lab Manual

Vehicle Routing Problem (VRP)

Lab - 5

Vehicle Routing Problem (VRP)

Lab - 5
Vehicle Routing Problem (VRP)

Imagine that you now manage a larger fulfillment center. You still need to deliver packages to a list of customers, but now you have a fleet of several vehicles at your disposal. What is the best way to deliver the packages to the customers using these vehicles?
This is an example of VRP, a generalization of the TSP described in the previous section. The basic VRP consists of the following three components:

- **The list of locations that need to be visited.**
- **The number of vehicles.**
- **The location of the depot**, which is used as the **starting** and **ending** point for each one of the vehicles.

Note :- Instead of **one vehicle**, we have **multiple vehicles** delivering to customers.

Goal

Minimize:

- total distance
- or total cost
- or delivery time

Real Scenario

You have:

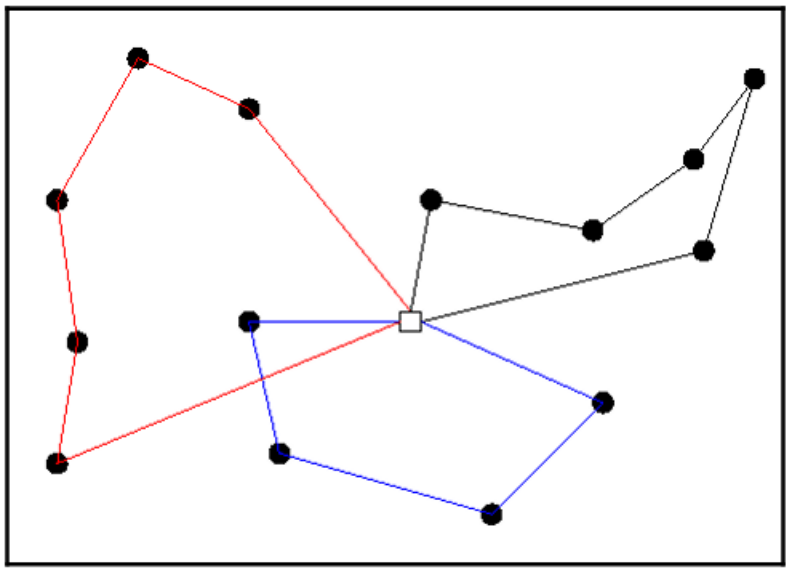
- A warehouse (depot)
- Many customers
- Multiple delivery vehicles

Each vehicle:

- starts from depot
- visits some customers

- returns to depot

An illustration of a VRP with three vehicles is shown here. The cities are marked with dark circles and the depot location is marked with an empty square, while the routes of the three vehicles are marked with three different colors:



Example VRP with three vehicles
Source: https://commons.wikimedia.org/wiki/File:Figure_illustrating_the_vehicle_routing_problem.png
Image by PierreSelim. Released to public domain

Problem Components:-

1. **Customers (Locations)**

   Set of nodes:
   C={1,2,3,...,n}C = \{1,2,3,...,n\}C={1,2,3,...,n}

   Each has coordinates:

```
Customer 1 → (x1, y1)
Customer 2 → (x2, y2)
```

2. *Depot*

   Single node:

```
Depot → (x0, y0)
```

3. *Vehicles*

   k=number of vehicles (Each vehicle handles a subset of customers).

Example :-

Suppose:

- Depot: (0,0)
- Customers:

  1 → (2,3)

  2 → (5,4)

  3 → (1,7)

  4 → (6,8)

  5 → (3,6)

- Vehicles = 2

Possible Solution:

Vehicle 1:

Depot → 1 → 3 → 5 → **Depot**

Vehicle 2:

Depot → 2 → 4 → **Depot**

4. *Objective Function (Fitness)*

$$Total\ Distance = \sum distances\ of\ all\ routes$$

Distance between two points:

$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

We minimize:

5. *Genetic Algorithm Representation*

Genotype (Chromosome)

We encode:

```
[1, 3, 5 | 2, 4]
```

- "|" separates vehicles

Phenotype

Actual routes:

```
Route1: Depot → 1 → 3 → 5 → Depot
```

```
Route2: Depot → 2 → 4 → Depot
```

6. *Fitness Function*

(we minimize distance → maximize fitness)

$$Fitness = \frac{1}{Total\ Distance}$$

Lab to implement:-

```
"""
============================================================
GENETIC ALGORITHM FOR TSP
Genotype-Phenotype Illustration Version
============================================================

GENOTYPE  : permutation of city indices
PHENOTYPE : decoded travel route
FITNESS   : inverse tour cost (scaled)

Pipeline:
Initialize → Decode → Evaluate → Select → Crossover → Mutate
→ Elitism → Repeat → Optimal Tour
============================================================
"""
```

```python
import numpy as np
import matplotlib.pyplot as plt
import random

# ------------------------------------------------------------
# [1] REPRODUCIBILITY
# Fix randomness so experiments are repeatable
# ------------------------------------------------------------
np.random.seed(42)
random.seed(42)

# ------------------------------------------------------------
# [2] PROBLEM DEFINITION (TSP WORLD)
# Cities = environment where phenotype exists
# ------------------------------------------------------------
cities_coords = np.array([
    [0, 0],   # A
    [1, 3],   # B
    [4, 2],   # C
    [3, 0],   # D
    [1, 4]    # E

 ])

city_labels = ['A', 'B', 'C', 'D','E']
n_cities = len(cities_coords)

# Distance matrix (phenotype evaluation space)
dist_matrix = np.linalg.norm(
    cities_coords[:, None, :] - cities_coords[None, :, :],
    axis=2
)




# ------------------------------------------------------------
# [3] GA PARAMETERS
# Evolution control variables
# ------------------------------------------------------------
population_size = 30
generations = 40
mutation_rate = 0.25
elite_size = 1


# ------------------------------------------------------------
```

```python
# [4] GENOTYPE CREATION
# Chromosome = permutation of cities
# Normalize so city A (0) always first
# ------------------------------------------------------------
def normalize(chrom):
    idx = chrom.index(0)
    return chrom[idx:] + chrom[:idx]


def create_random_chromosome():
    chrom = list(range(n_cities))
    random.shuffle(chrom)
    return normalize(chrom)




# ------------------------------------------------------------
# [5] GENOTYPE → PHENOTYPE DECODING
# Converts genetic encoding into real tour
# ------------------------------------------------------------
def decode(chrom):
    return [city_labels[i] for i in chrom] + [city_labels[chrom[0]]]


# ------------------------------------------------------------
# [6] PHENOTYPE EVALUATION (COST)
# Computes physical tour length
# ------------------------------------------------------------
def calculate_cost(chrom):
    cost = 0
    for i in range(n_cities):
        cost += dist_matrix[chrom[i],
                            chrom[(i + 1) % n_cities]]
    return cost




# ------------------------------------------------------------
# [7] FITNESS SCALING
# Stable alternative to 1/cost
# Prevents selection explosion
# ------------------------------------------------------------
def calculate_fitness(costs):
```

```
    max_cost = max(costs)
    return [max_cost - c + 1e-6 for c in costs]


# ------------------------------------------------------------
# 8 SELECTION (Tournament)
# Survival of the fittest
# ------------------------------------------------------------
def tournament_selection(population, fitnesses, k=3):
    selected = random.sample(range(len(population)), k)
    best = max(selected, key=lambda i: fitnesses[i])
    return population[best][:]


# ------------------------------------------------------------
# 9 ORDER CROSSOVER (OX)
# Permutation-safe recombination
# Preserves city order information
# ------------------------------------------------------------
def order_crossover(p1, p2):
    size = len(p1)
    start, end = sorted(random.sample(range(size), 2))

    child = [None] * size
    child[start:end] = p1[start:end]

    idx = end % size
    for gene in p2:
        if gene not in child:
            while child[idx] is not None:
                idx = (idx + 1) % size
            child[idx] = gene
            idx = (idx + 1) % size

    return normalize(child)


# ------------------------------------------------------------
# 10 MUTATION (Swap Mutation)
# Introduces diversity into genotype space
# ------------------------------------------------------------
def mutate(chrom):
    if random.random() < mutation_rate:
        i, j = random.sample(range(n_cities), 2)
        chrom[i], chrom[j] = chrom[j], chrom[i]
    return normalize(chrom)
```

```
# ------------------------------------------------------------
# 1️⃣1️⃣ VISUALIZATION
# Shows phenotype (actual route)
# ------------------------------------------------------------
def plot_tsp(tour, title):
    plt.figure(figsize=(7,5))
    plt.title(title)

    plt.scatter(cities_coords[:,0], cities_coords[:,1], s=150)

    for i, label in enumerate(city_labels):
        plt.text(cities_coords[i,0]+0.1,
                 cities_coords[i,1]+0.1,
                 label, fontsize=12)

    route = tour + [tour[0]]

    for i in range(len(route)-1):
        p1 = cities_coords[route[i]]
        p2 = cities_coords[route[i+1]]
        plt.plot([p1[0], p2[0]],
                 [p1[1], p2[1]], linewidth=2)

    plt.grid(True)
    plt.show()
```

```
# ------------------------------------------------------------
# 1️⃣2️⃣ INITIAL POPULATION
# Random solutions (genotype space)
# ------------------------------------------------------------
population = [create_random_chromosome()
              for _ in range(population_size)]

best_cost_overall = float('inf')
best_chrom_overall = None
history = []
```

```python
# ------------------------------------------------------------
# 1️⃣3️⃣ EVOLUTION LOOP
# Core GA lifecycle
# ------------------------------------------------------------
for gen in range(generations + 1):

    costs = [calculate_cost(ch) for ch in population]
    fitnesses = calculate_fitness(costs)

    best_idx = np.argmax(fitnesses)
    best_cost = costs[best_idx]
    best_chrom = population[best_idx]

    # Save global best (ELITISM MEMORY)
    if best_cost < best_cost_overall:
        best_cost_overall = best_cost
        best_chrom_overall = best_chrom[:]

    history.append(best_cost_overall)

    print(f"\n===== GENERATION {gen} =====")
    for i, ch in enumerate(population):
        print(
            f"Ind {i+1}: "
            f"Genotype {ch} → "
            f"Phenotype {'→'.join(decode(ch))} "
            f"| Cost {costs[i]:.2f}"
        )

    if gen == generations:
        break

    # ---------------- Evolution ----------------
    new_population = []

    # ✅ ELITISM (keep best solution)
    new_population.append(best_chrom[:])

    while len(new_population) < population_size:
        p1 = tournament_selection(population, fitnesses)
        p2 = tournament_selection(population, fitnesses)

        child = order_crossover(p1, p2)
        child = mutate(child)

        new_population.append(child)
```

```
    population = new_population




# ------------------------------------------------------------
# 1️⃣4️⃣ RESULTS
# Best phenotype discovered
# ------------------------------------------------------------
print("\n========== FINAL RESULT ==========")
print("Optimal Tour:",
      " → ".join(decode(best_chrom_overall)))
print("Minimum Cost:", round(best_cost_overall, 3))

plot_tsp(best_chrom_overall,
         f"Optimal Tour (Cost {best_cost_overall:.2f})")


# ------------------------------------------------------------
# 1️⃣5️⃣ CONVERGENCE VISUALIZATION
# Shows evolution learning behaviour
# ------------------------------------------------------------
plt.figure()
plt.plot(history)
plt.title("GA Convergence Curve")
plt.xlabel("Generation")
plt.ylabel("Best Cost")
plt.grid(True)
plt.show()
```



Beni-Suef University
College of Computers and AI
Department of Computer Science

Lab Manual

NQueen Problem

Lab - 6
NQueen Problem

Lab - 6

NQueen Problem

Constraint Satisfaction Problem (CSP)

It is a problem which has variables, values and constraints, which you are asked to find solution based on all constraints.

Optimization Problem

We search for the **best possible solution**.

Search Problem

We search for a correct solution only

Role of Genetic Algorithms

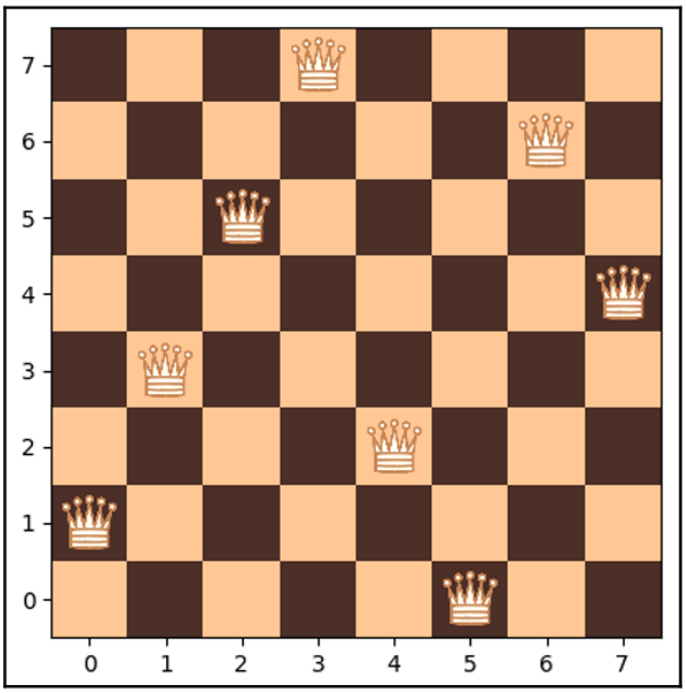
Instead of using traditional search methods:

- We use a **population of solutions**
- We **generate multiple candidate solutions**
- We evaluate them based on:
  - **Solution quality**
  - **Degree of constraint satisfaction**

N-QUEEN Problem

Originally known as the **eight-queen puzzle**, the classic N-Queens problem originated from the game of chess, and the 8x8 chessboard was its early playground. The task was to place eight chess queens on the board without any two of them threatening each other. In other words, no two queens can share the same row, same column, or same diagonal. The N-Queens problem is similar, using an N×N chessboard and N chess queens.

The problem is known to have a solution for any natural number, n, except for the cases of n=2 and n=3. For the original eight-queen case, there are 92 solutions, or 12 unique solutions if we consider symmetrical solutions to be identical. One of the solutions is as follows:



One of the 92 possible solutions for the eight-queen puzzle

By applying combinatorics, the count of all possible ways to place eight pieces on the 8×8 board yields 4,426,165,368 combinations. However, if we can create our candidate solutions in a way that ensures that no two queens will be placed on the same row or the same column, the number of possible combinations is dramatically reduced to 8! (factorial of 8), which amounts to 40,320.

Intuition : Constraint-aware encoding dramatically reduces the search space, making genetic algorithms much faster and more effective.

Example for conflicts:-

```
individual = [0,2,4,6,1,3,5,7]
```

Row	Column
0	0
1	2
2	4
3	6
4	1
5	3
6	5
7	7

Q . . . . (0,0)

. . Q . . . . . (1,2)

. . . . Q . . . (2,4)

. . . . . . Q . (3,6)

. Q . . . . . . (4,1)

. . . Q . . . . (5,3)

. . . . . Q . . (6,5)

. . . . Q (7,7)

Our Rule : |row1 - row2| = |col1 - col2| → Conflict

For example :

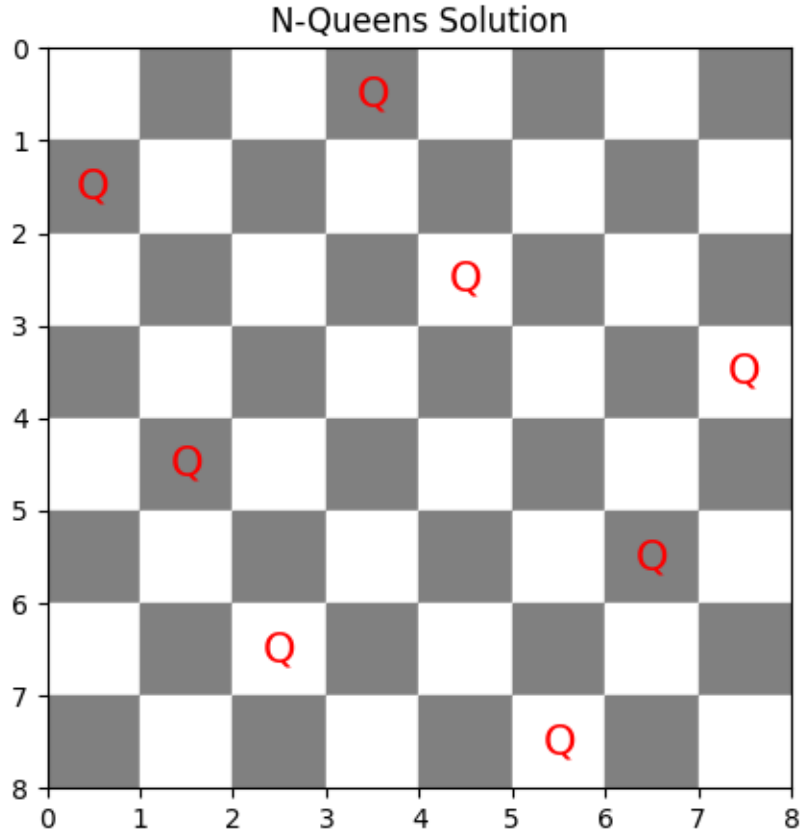
- row diff = |0 - 1| = 1 , col diff = |0 - 2| = 2 → no conflict
- |0 - 2| = 2 ,|0 - 4| = 4 ,→ no conflict
- |1 - 4| = 3 ,|2 - 1| = 1 ,→ no conflict
- |0 - 7| = 7 ,|0 - 7| = 7 ,→ conflict

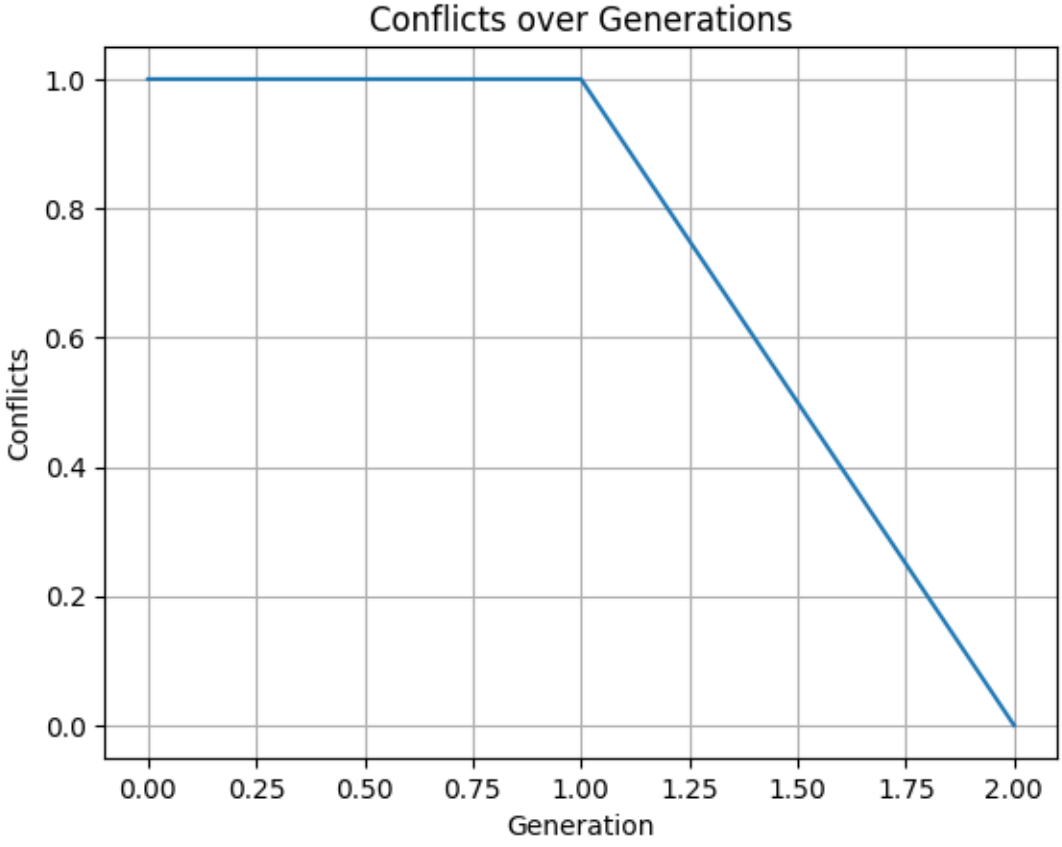
Lab to implement:-

```
import random
import matplotlib.pyplot as plt
N = 8  # number of queens
POP_SIZE = 100
GENERATIONS = 200
MUTATION_RATE = 0.01


def create_individual():
    individual = list(range(N))
    random.shuffle(individual)
    return individual


def create_population():
    return [create_individual() for _ in range(POP_SIZE)]


def fitness(individual):
    conflicts = 0

    for i in range(N):
        for j in range(i + 1, N):
            # diagonal conflict
            if abs(individual[i] - individual[j]) == abs(i - j):
                conflicts += 1

    return -conflicts  # maximize fitness (less conflicts)


def selection(population, k=3):
    selected = random.sample(population, k)
    return max(selected, key=fitness)


def crossover(p1, p2):
    size = len(p1)
    start, end = sorted(random.sample(range(size), 2))
```

```
    child = [None] * size
    child[start:end] = p1[start:end]

    fill = [x for x in p2 if x not in child]

    j = 0
    for i in range(size):
        if child[i] is None:
            child[i] = fill[j]
            j += 1

    return child


def mutate(individual):
    if random.random() < MUTATION_RATE:
        i, j = random.sample(range(N), 2)
        individual[i], individual[j] = individual[j], individual[i]
    return individual


def genetic_algorithm():
    population = create_population()
    best_history = []

    for gen in range(GENERATIONS):
        new_population = []

        # Elitism (keep best)
        best = max(population, key=fitness)
        new_population.append(best)

        while len(new_population) < POP_SIZE:
            p1 = selection(population)
            p2 = selection(population)

            child = crossover(p1, p2)
            child = mutate(child)

            new_population.append(child)

        population = new_population

        best = max(population, key=fitness)
        best_conflicts = -fitness(best)
        best_history.append(best_conflicts)
```

```
        print(f"Gen {gen}: Conflicts = {best_conflicts}")

        if best_conflicts == 0:
            break

    return best, best_history


def plot_board(solution):
    plt.figure(figsize=(5,5))

    for i in range(N):
        for j in range(N):
            color = 'white' if (i + j) % 2 == 0 else 'gray'
            plt.gca().add_patch(plt.Rectangle((j, i), 1, 1, color=color))

    for row in range(N):
        col = solution[row]
        plt.text(col + 0.5, row + 0.5, 'Q',
                 ha='center', va='center', fontsize=16, color='red')

    plt.xlim(0, N)
    plt.ylim(0, N)
    plt.gca().invert_yaxis()
    plt.title("N-Queens Solution")
    plt.show()


def plot_fitness(history):
    plt.plot(history)
    plt.title("Conflicts over Generations")
    plt.xlabel("Generation")
    plt.ylabel("Conflicts")
    plt.grid()
    plt.show()


best_solution, history = genetic_algorithm()

print("\nBest Solution:", best_solution)
print("Final Conflicts:", -fitness(best_solution))

plot_board(best_solution)
plot_fitness(history)
```


N-Queens Solution
Q
Q
Q
Q
Q
Q
Q
Q
0
1
2
3
4
5
6
7
8
0
1
2
3
4
5
6
7
8


Conflicts over Generations
Conflicts
1.0
0.8
0.6
0.4
0.2
0.0
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00
Generation

Beni-Suef University
College of Computers and AI
Department of Computer Science

Lab Manual

Nurse Scheduling Problem

Lab - 7
Nurse Scheduling Problem

Lab - 7

Nurse Scedluing Problem

Constraint Satisfaction Problem (CSP)

Nurse Scheduling Problem(NSP)

Imagine you are responsible for scheduling the shifts for the nurses in your hospital department for this week. There are three shifts in a day – morning, afternoon, and night and for each shift, you need to assign one or more of the eight nurses that work in your department. If this sounds like a simple task, take a look at the list of relevant hospital rules:

- A nurse is not allowed to work two consecutive shifts.
- A nurse is not allowed to work more than five shifts per week.
- The number of nurses per shift in your department should fall within the following limits:
  1. Morning shift: 2–3 nurses
  2. Afternoon shift: 2–4 nurses
  3. Night shift: 1–2 nurses

In addition, each nurse can have shift preferences. For example, one nurse prefers to only work morning shifts, another nurse prefers to not work afternoon shifts, and so on.

This task is an example of the nurse scheduling problem (NSP), which can have many variants. Possible variations may include different specialties for different nurses, the ability to work on cover shifts (overtime), or even different types of shifts – such as 8-hour shifts and 12-hour shifts

By now, it probably looks like a good idea to write a program that will do the scheduling for you. Why not apply our knowledge of genetic algorithms to implement such a program? As usual, we will start by representing the solution to the problem.

Solution representation:-

For solving the nurse scheduling problem, we decided to use a binary list (or array) to represent the schedule as it will be intuitive for us to interpret, and we've seen that genetic algorithms can naturally handle this representation.

For each nurse, we can have a binary string representing the 21 shifts of the week. A value of 1 represents a shift that the nurse is scheduled to work on. For example, take a look at the following binary list:

(0, 1, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0)

This can be broken into the following groups of three values, representing the shifts this nurse will be working each day of the week:

Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
(0, 1, 0)	(1, 0, 1)	(0, 1, 1)	(0, 0, 0)	(0, 0, 1)	(1, 0, 0)	(0, 1, 0)
afternoon	morning and night	afternoon and night	(none)	night	morning	afternoon

Note :- The schedules of all nurses can be then concatenated together to create one long binary list representing the entire solution.

Evaluating a solution:-
this long list can be broken down into the schedules of the individual nurses, and violations of the constraints can be checked for:-

- The preceding sample nurse schedule, for instance, contains two occurrences of consecutive 1 values that represent consecutive shifts being worked (afternoon followed by night, and night followed by morning).
- The number of weekly shifts for that same nurse can be calculated by totaling the binary values of the list, which results in 8 shifts.

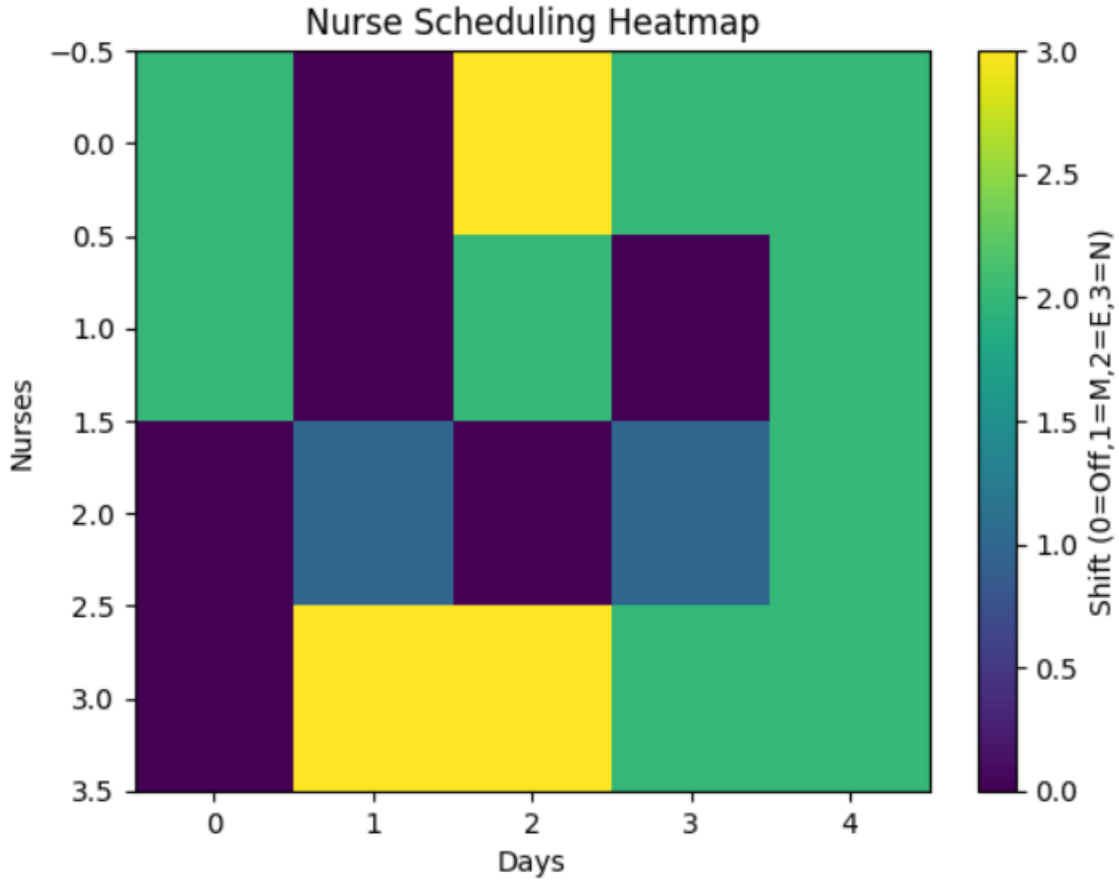
Difference between hard & soft constraints:-
In the case of the N-Queens problem, all the constraints – row, column, and diagonal – were hard constraints. Had we not found a solution where the number of violations was zero, we would not have a valid solution for the problem. Here, on the other hand, while the hospital rules are considered hard constraints, the nurses' preferences are soft constraints. So, we are actually looking for a solution that will not violate any of the hospital rules while minimizing the number of breaches to the nurses' preferences.

NSP Implementation:-

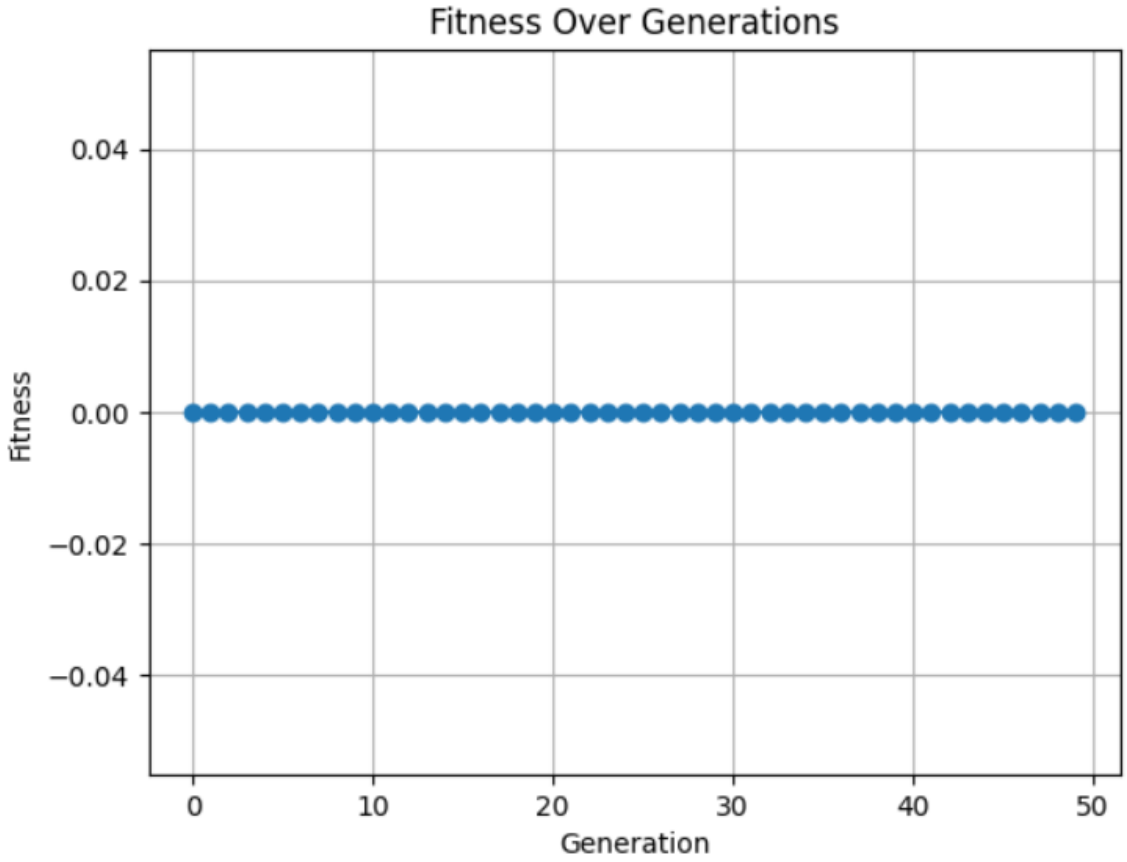
```
import random
import numpy as np
import matplotlib.pyplot as plt


NUM_NURSES = 4
DAYS = 5

# 0 = Off, 1 = Morning, 2 = Evening, 3 = Night
SHIFTS = [0, 1, 2, 3]
```

```
MAX_SHIFTS = 4
POP_SIZE = 20
GENERATIONS = 50


def create_individual():
    return np.random.choice(SHIFTS, size=(NUM_NURSES, DAYS))


def fitness(individual):
    penalty = 0

    # Constraint 1: max shifts per nurse
    for nurse in individual:
        work_days = np.count_nonzero(nurse)
        if work_days > MAX_SHIFTS:
            penalty += (work_days - MAX_SHIFTS) * 2

    # Constraint 2: at least one working nurse per day
    for day in range(DAYS):
        if np.all(individual[:, day] == 0):
            penalty += 5

    return -penalty


def select(population):
    return max(random.sample(population, 3), key=fitness)



def crossover(parent1, parent2):
    point = random.randint(1, DAYS - 1)
    child = np.hstack((parent1[:, :point], parent2[:, point:]))
    return child


def mutate(individual):
    if random.random() < 0.3:
        nurse = random.randint(0, NUM_NURSES - 1)
        day = random.randint(0, DAYS - 1)
        individual[nurse][day] = random.choice(SHIFTS)
    return individual


def run_ga():
    population = [create_individual() for _ in range(POP_SIZE)]
```

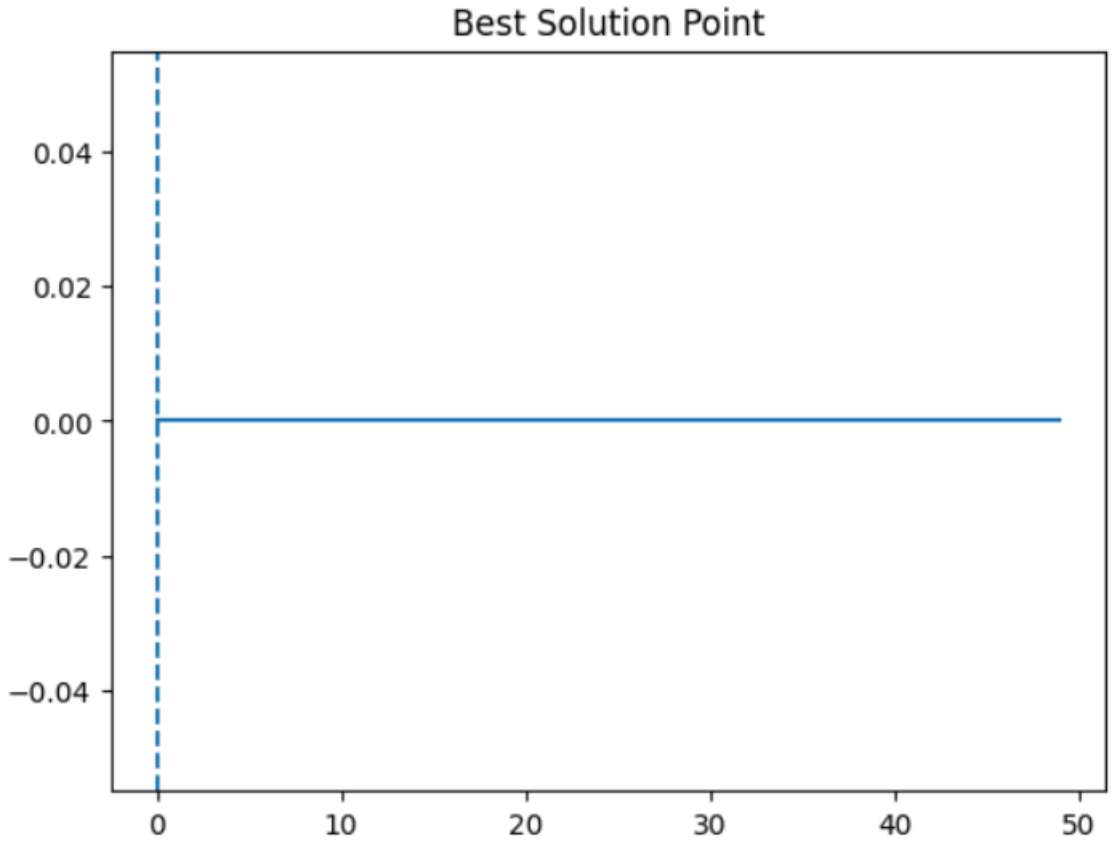
```
    fitness_history = []

    for gen in range(GENERATIONS):
        new_population = []

        for _ in range(POP_SIZE):
            p1 = select(population)
            p2 = select(population)

            child = crossover(p1, p2)
            child = mutate(child)

            new_population.append(child)

        population = new_population

        best = max(population, key=fitness)
        best_fit = fitness(best)
        fitness_history.append(best_fit)

        print(f"Generation {gen}: Fitness = {best_fit}")

    return best, fitness_history


def visualize_schedule(schedule):
    plt.imshow(schedule, aspect='auto')
    plt.title("Nurse Scheduling Heatmap")
    plt.xlabel("Days")
    plt.ylabel("Nurses")
    plt.colorbar(label="Shift (0=Off,1=M,2=E,3=N)")
    plt.show()

visualize_schedule(best_solution)
```




```
plt.plot(history, marker='o')
plt.title("Fitness Over Generations")
plt.xlabel("Generation")
plt.ylabel("Fitness")
plt.grid()
plt.show()
```




```
best_gen = history.index(max(history))
print("Best solution found at generation:", best_gen)
plt.plot(history)
plt.axvline(x=best_gen, linestyle='--')
plt.title("Best Solution Point")
plt.show()
```


Best Solution Point
0.04
0.02
0.00
−0.02
−0.04
0
10
20
30
40
50

Beni-Suef University
College of Computers and AI
Department of Computer Science

Lab Manual

Graph Coloring Problem

Lab - 8
Graph Coloring Problem

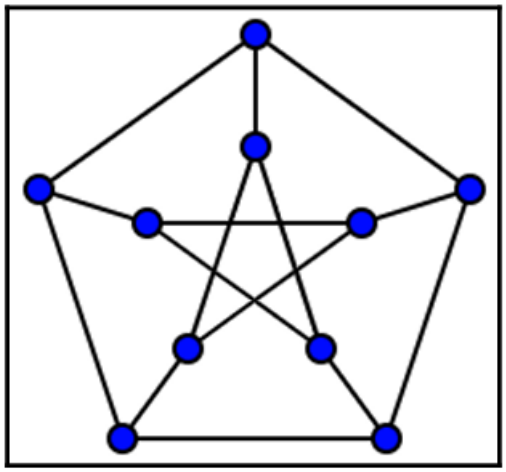
Lab - 8

Graph Coloring Problem

Constraint Satisfaction Problem (CSP)

Graph Coloring Problem (GCP)

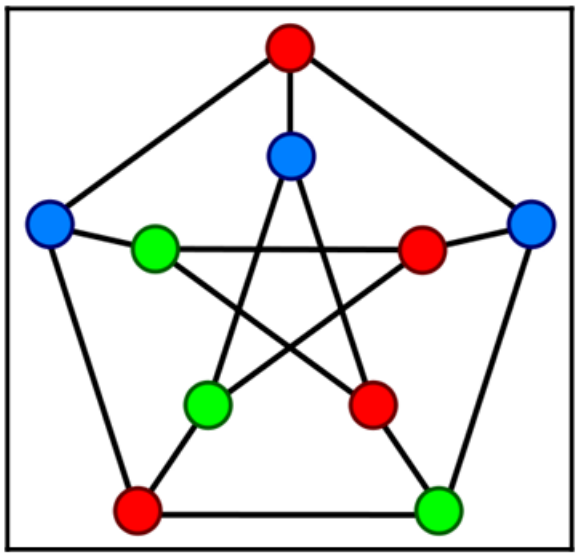
In the mathematical branch of graph theory, a graph is a structured collection of objects that represents the relationships between pairs of these objects. The objects appear as vertices (or nodes) in the graph, while the relation between a pair of objects is represented using an edge. A common way of illustrating a graph is by drawing the vertices as circles and the edges as connecting lines, as depicted in the following diagram of the Petersen graph, named after the Danish mathematician Julius Petersen:



Graphs are remarkably useful objects as they can represent and help us research an overwhelming variety of real-life structures, patterns, and relationships, such as social networks, power grid layouts, website structures, linguistic compositions, computer networks, atomic structures, migration patterns, and more.

The graph coloring task is used to assign a color for every node in the graph in such a way that no pair of connected (adjacent) nodes will share the same color. This is also known as the **proper coloring** of the graph.

The following diagram shows the same Petersen graph, but this time colored properly:



The color assignment is often accompanied by an optimization requirement – use the **minimum possible number of colors**. For example, the Peterson graph can be properly colored using **three** colors, as demonstrated in the preceding diagram. But it would be impossible to properly color it using only two colors. In graph theory terms, this means that the chromatic number of this graph is three.

Solution representation:-

Expanding on the commonly used binary list (or array) representation, we can employ a list of integers, where **each integer represents a unique color**, while each element of the list matches one of the graph's nodes.

For example, since the Petersen graph has 10 nodes, we can assign each node an index between 0 and 9. Then, we can represent the node coloring for that graph using a list of 10 elements.

For example, let's have a look at what we have in this particular representation:

(0, 2, 1, 3, 1, 2, 0, 3, 3, 0)

Let's talk about what we have here in detail: Four colors are used, represented by the integers 0, 1, 2, 3.

- The first, seventh, and tenth nodes of the graph are colored with the first color (0).
- The third and fifth nodes are colored with the second color (1).
- The second and sixth nodes are colored with the third color (2).
- The fourth, eighth, and ninth nodes are colored with the fourth color (3).

To **evaluate** the solution, we need to iterate over each pair of **adjacent nodes** and check if they share **the same color**. If they do, this is **a coloring violation**, and we seek to minimize the number of violations to **zero** to achieve **the proper coloring** of the graph.

Using hard and soft constraints for the graph coloring problem:-

When solving the nurse scheduling problem earlier in this chapter, we noted the difference between hard constraints – those we have to adhere to for the solution to be considered valid – and soft constraints – those we strive to minimize to get the best solution. In the graph coloring problem, the color assignment requirement – where no two adjacent nodes can have the same color – is a hard constraint. We have to minimize the number of violations of this constraint to zero to achieve a valid solution.

Minimizing the number of colors used, however, can be introduced as a soft constraint. We would like to minimize this number, but not at the expense of violating the hard constraint.

However, you may recall that we also seek to minimize the number of colors that are used. If we happen to already know this number, we can just use as many integer values as the known number of colors. But what if we don't? One way to go about this is to start with an estimate (or just a guess) for the number of colors used. If we find a proper solution using this number, we can reduce the number and try again. If no solution was found, we can increase the number and try again until we have the smallest number we could find a solution with. However, we may be able to get to this number faster by using soft and hard constraints, as described in the next subsection.

Implementation:-

Cell 1: Imports

```
import random
import networkx as nx
import matplotlib.pyplot as plt
```

Cell 2: Define Graph

```
G = nx.Graph()
```

```
edges = [
    (0,1),(0,2),(1,2),(1,3),
    (2,3),(3,4),(4,5),(5,0)
]

G.add_edges_from(edges)

NUM_NODES = G.number_of_nodes()
NUM_COLORS = 3
```

Cell 3: Create Individual (Genotype)

```
def create_individual():
    return [random.randint(0, NUM_COLORS-1) for _ in range(NUM_NODES)]
```

Cell 4: Fitness Function

```
def fitness(individual):
    conflicts = 0

    for u, v in G.edges():
        if individual[u] == individual[v]:
            conflicts += 1

    return -conflicts  # maximize
```

Cell 5: Selection

```
def select(pop):
    return max(random.sample(pop, 3), key=fitness)
```

Cell 6: Crossover

```
def crossover(p1, p2):
    point = random.randint(1, NUM_NODES-1)
    return p1[:point] + p2[point:]
```

Cell 7: Mutation

```
def mutate(ind):
    if random.random() < 0.3:
```

```
        i = random.randint(0, NUM_NODES-1)
        ind[i] = random.randint(0, NUM_COLORS-1)
    return ind
```

Cell 8: GA Loop

```
def run_ga():
    pop = [create_individual() for _ in range(20)]
    history = []

    for gen in range(50):
        new_pop = []

        for _ in range(len(pop)):
            p1 = select(pop)
            p2 = select(pop)

            child = crossover(p1, p2)
            child = mutate(child)

            new_pop.append(child)

        pop = new_pop

        best = max(pop, key=fitness)
        best_fit = fitness(best)
        history.append(best_fit)

        print(f"Gen {gen}, Conflicts = {-best_fit}")

    return best, history
```

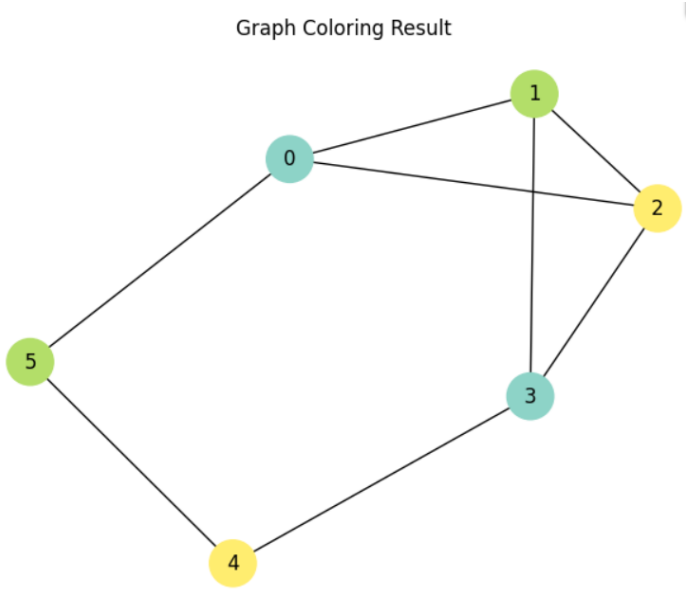
Cell 9: Run GA

```
best_solution, history = run_ga()
print("\nBest Coloring:", best_solution)
```

Cell 10: Visualize Graph Coloring

```
def visualize_graph(solution):
    color_map = solution
    pos = nx.spring_layout(G)

    nx.draw(G, pos,
```

```
            node_color=color_map,
            with_labels=True,
            cmap=plt.cm.Set3,
            node_size=800)

    plt.title("Graph Coloring Result")
    plt.show()

visualize_graph(best_solution)
```



Beni-Suef University
College of Computers and AI
Department of Computer Science

Lab Manual

Feature Selection With GA

Feature Selection With GA

Lab - 9

Feature Selection With GA

Feature Selection using Genetic Algorithms in Machine Learning

Objective

In this lab, students will:

- Understand **supervised learning tasks**:
  - Regression
  - Classification
- Learn the importance of **Feature Selection**
- Apply **Genetic Algorithms (GA)** to improve model performance
- Evaluate models before and after feature selection

Background

Machine learning models often suffer when:

- Too many irrelevant features exist
- Features add noise instead of useful information

This leads to:

- Lower accuracy
- Overfitting
- Slower training

Solution: Feature Selection

We aim to:

```
Select the most useful subset of features
```

Instead of:

```
Using all features blindly ❌
```

Why Genetic Algorithm?

Because:

- Feature selection = **combinatorial problem**
- GA can efficiently search large spaces

Representation (Genotype)

Each chromosome is:

```
[1, 0, 1, 1, 0, 0, 1]
```

Where:

- 1 → feature selected
- 0 → feature ignored

Fitness Function

We evaluate:

```
Fitness = Model Performance (Accuracy or R²)
```

Part 1: Regression (Friedman-1 Dataset)

Step 1: Import Libraries

```
import numpy as np
import random
from sklearn.datasets import make_friedman1
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
```

```
from sklearn.metrics import r2_score
```

Step 2: Generate Dataset

```
X, y = make_friedman1(n_samples=200, n_features=10, noise=0.1)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)
NUM_FEATURES = X.shape[1]
```

Step 3: Create Individual

```
def create_individual():
    return [random.randint(0, 1) for _ in range(NUM_FEATURES)]
```

Step 4: Fitness Function

```
def fitness(individual):
    if sum(individual) == 0:
        return -1

    selected = [i for i in range(NUM_FEATURES) if individual[i] == 1]

    model = LinearRegression()
    model.fit(X_train[:, selected], y_train)

    preds = model.predict(X_test[:, selected])
    return r2_score(y_test, preds)
```

Step 5: GA Operators

```
def select(pop):
    return max(random.sample(pop, 3), key=fitness)

def crossover(p1, p2):
    point = random.randint(1, NUM_FEATURES-1)
    return p1[:point] + p2[point:]

def mutate(ind):
    if random.random() < 0.3:
        i = random.randint(0, NUM_FEATURES-1)
        ind[i] = 1 - ind[i]
    return ind
```

Step 6: Run GA

```
def run_ga():
    pop = [create_individual() for _ in range(20)]

    for gen in range(30):
```

```
        new_pop = []

        for _ in range(len(pop)):
            p1 = select(pop)
            p2 = select(pop)

            child = crossover(p1, p2)
            child = mutate(child)

            new_pop.append(child)

        pop = new_pop

        best = max(pop, key=fitness)
        print(f"Gen {gen}, R2: {fitness(best):.4f}")

    return best
```

Step 7: Run

```
best_features = run_ga()
print("Selected Features:", best_features)
```

Part 2: Classification (Zoo Dataset)

Step 1: Load Dataset

```
from sklearn.datasets import load_iris
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score

data = load_iris()
X = data.data
y = data.target

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)
NUM_FEATURES = X.shape[1]
```

Step 2: Fitness Function

```
def fitness(individual):
    if sum(individual) == 0:
        return 0

    selected = [i for i in range(NUM_FEATURES) if individual[i] == 1]

    model = RandomForestClassifier()
    model.fit(X_train[:, selected], y_train)
```

```
    preds = model.predict(X_test[:, selected])
    return accuracy_score(y_test, preds)
```

Step 3: Run GA (reuse same GA)

```
best_features = run_ga()
print("Best Feature Mask:", best_features)
```

Expected Results

- Reduced number of features
- Improved or similar performance
- Faster training

Lab implementation:

Imports

```
import numpy as np

import random
import matplotlib.pyplot as plt

from sklearn.datasets import make_friedman1, load_iris
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import r2_score, accuracy_score
```

Common GA Functions

```
NUM_FEATURES = None
X_train, X_test, y_train, y_test = None, None, None, None
```

Create Individual

```
def create_individual():
    return [random.randint(0, 1) for _ in range(NUM_FEATURES)]
```

Selection

```
def select(pop):

    return max(random.sample(pop, 3), key=fitness)
```

Crossover

```
def crossover(p1, p2):
    point = random.randint(1, NUM_FEATURES - 1)
```

```
    return p1[:point] + p2[point:]
```

Mutation

```
def mutate(ind):
    if random.random() < 0.3:
        i = random.randint(0, NUM_FEATURES - 1)
        ind[i] = 1 - ind[i]
    return ind
```

REGRESSION PART (Friedman Dataset)

```
X, y = make_friedman1(n_samples=200, n_features=10, noise=0.1)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)

NUM_FEATURES = X.shape[1]
```

Fitness Function (Regression)

```
def fitness(individual):
    if sum(individual) == 0:
        return -1

    selected = [i for i in range(NUM_FEATURES) if individual[i] == 1]

    model = LinearRegression()
    model.fit(X_train[:, selected], y_train)

    preds = model.predict(X_test[:, selected])
    return r2_score(y_test, preds)
```

GA Runner

```
def run_ga():
    pop = [create_individual() for _ in range(20)]

    for gen in range(30):
        new_pop = []

        for _ in range(len(pop)):
            p1 = select(pop)
            p2 = select(pop)

            child = crossover(p1, p2)
            child = mutate(child)

            new_pop.append(child)

        pop = new_pop

        best = max(pop, key=fitness)
```

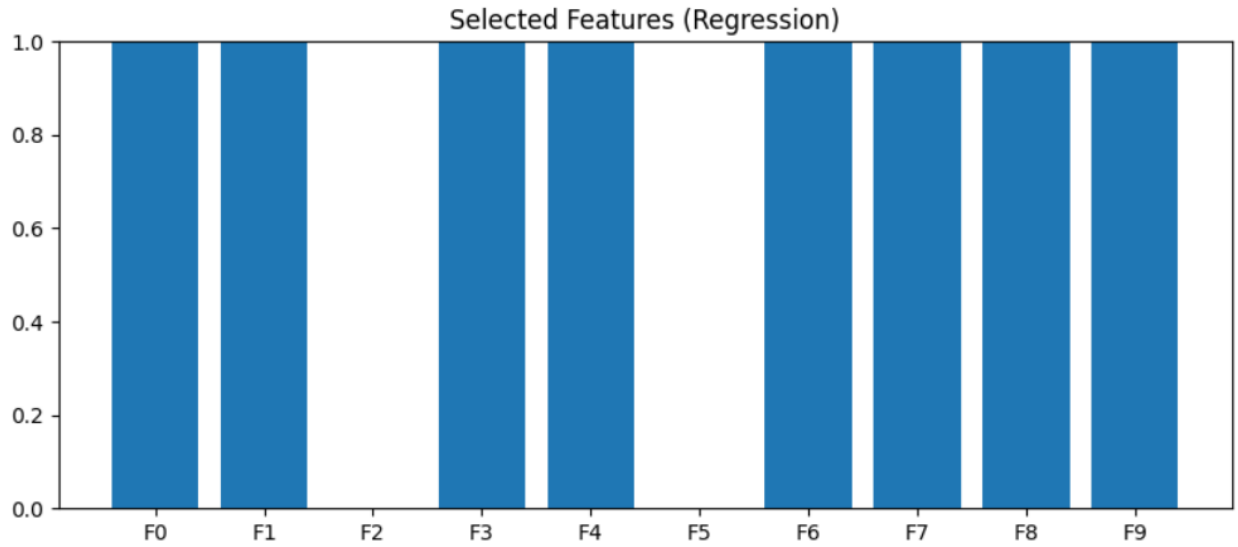
```
        print(f"Gen {gen}, Fitness: {fitness(best):.4f}")

    return best
```

Run GA (Regression)

```
best_features_reg = run_ga()
print("Best Features (Regression):", best_features_reg)
```

Feature Visualization

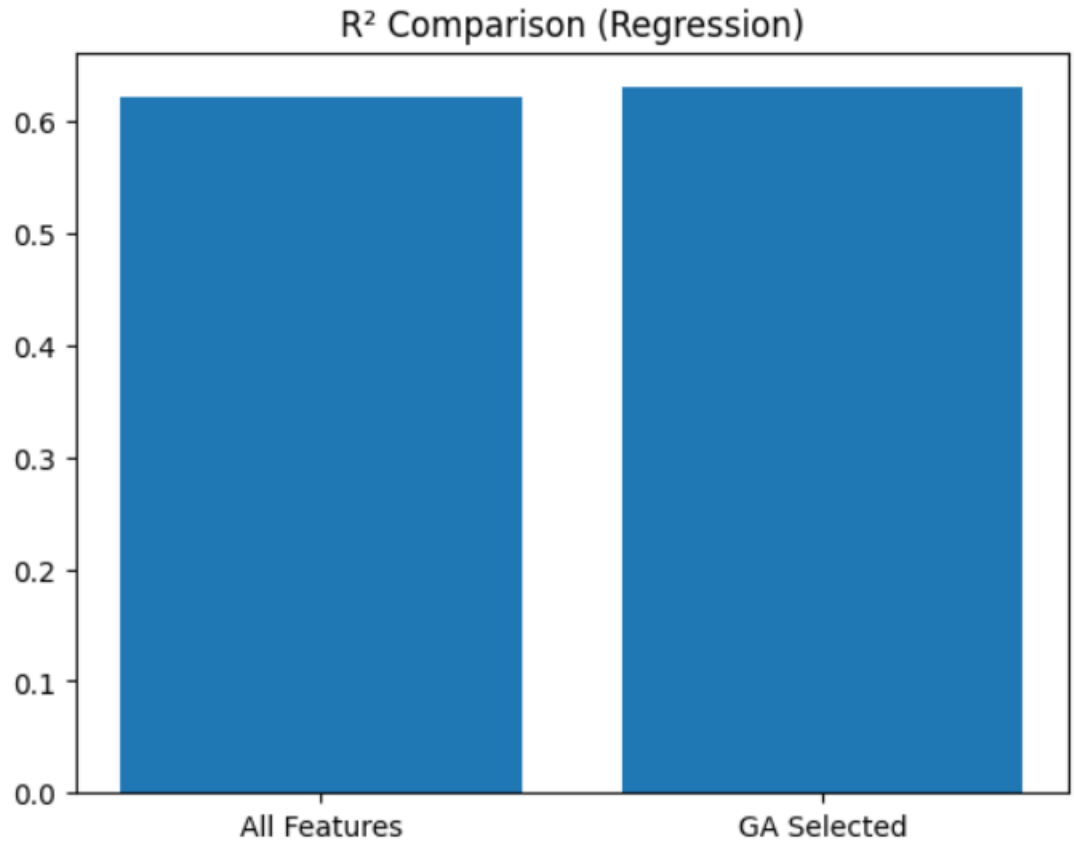
```
def plot_feature_selection(mask):
    plt.figure(figsize=(10,4))
    plt.bar([f"F{i}" for i in range(len(mask))], mask)
    plt.title("Selected Features (Regression)")
    plt.ylim(0, 1)
    plt.show()

plot_feature_selection(best_features_reg)
```



Before vs After (Regression)

```
# Full model
model_full = LinearRegression()
model_full.fit(X_train, y_train)
pred_full = model_full.predict(X_test)
r2_full = r2_score(y_test, pred_full)

# GA model
selected = [i for i in range(NUM_FEATURES) if best_features_reg[i] == 1]

model_ga = LinearRegression()
model_ga.fit(X_train[:, selected], y_train)
pred_ga = model_ga.predict(X_test[:, selected])
r2_ga = r2_score(y_test, pred_ga)
```

```
# Plot
plt.bar(["All Features", "GA Selected"], [r2_full, r2_ga])
plt.title("R² Comparison (Regression)")
plt.show()
```



Part 2 Classification

IRIS Dataset

```
data = load_iris()
X = data.data
y = data.target

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)

NUM_FEATURES = X.shape[1]
```

Fitness Function (Classification)

```
def fitness(individual):
    if sum(individual) == 0:
        return 0

    selected = [i for i in range(NUM_FEATURES) if individual[i] == 1]
```

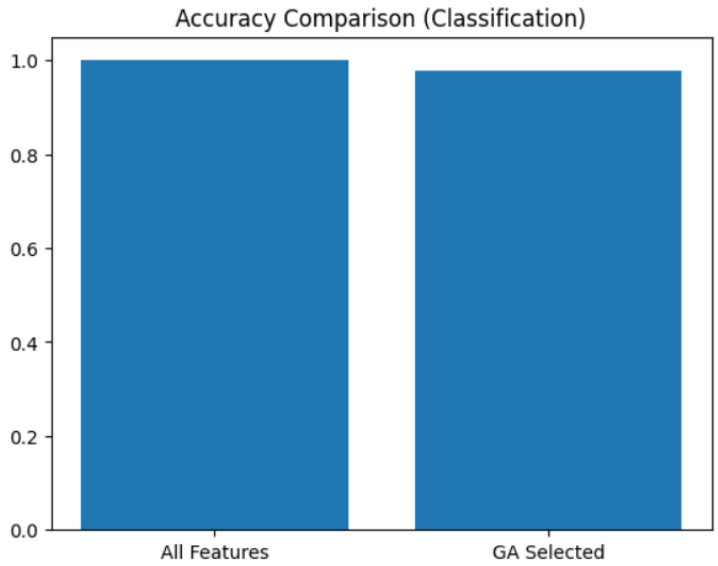
```
    model = RandomForestClassifier()
    model.fit(X_train[:, selected], y_train)

    preds = model.predict(X_test[:, selected])
    return accuracy_score(y_test, preds)
```

Run GA

```
best_features_cls = run_ga()
print("Best Features (Classification):", best_features_cls)
```

Before vs After (Classification)

```
# Full model
model_full = RandomForestClassifier()
model_full.fit(X_train, y_train)
pred_full = model_full.predict(X_test)
acc_full = accuracy_score(y_test, pred_full)

# GA model
selected = [i for i in range(NUM_FEATURES) if best_features_cls[i] == 1]

model_ga = RandomForestClassifier()
model_ga.fit(X_train[:, selected], y_train)
pred_ga = model_ga.predict(X_test[:, selected])
acc_ga = accuracy_score(y_test, pred_ga)

# Plot
plt.bar(["All Features", "GA Selected"], [acc_full, acc_ga])
plt.title("Accuracy Comparison (Classification)")
plt.show()
```





Beni-Suef University
College of Computers and AI
Department of Computer Science

Lab Manual

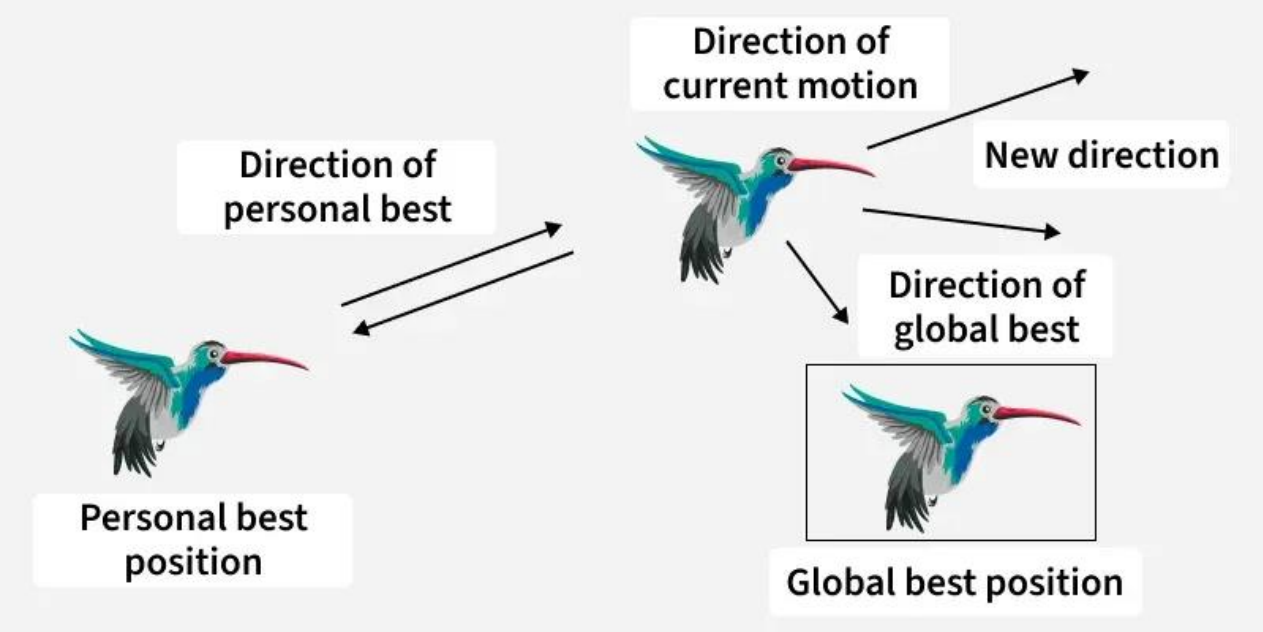
Particle Swarm Optimization (PSO)

Particle Swarm Optimization (PSO)

Lab - 10

Particle Swarm Optimization (PSO)

Optimisation aims to find the best solution from a set of feasible options under given constraints. In many machine learning and engineering problems, the search space is complex, non-linear and multimodal, where traditional gradient-based methods may be ineffective. This makes population-based optimization techniques more suitable.



Particle Swarm Optimization (PSO) is a stochastic population based optimization technique inspired by swarm intelligence in nature. It is designed to solve complex optimization problems where the search space is large, non-linear or unknown, where traditional deterministic methods are ineffective.

- Each particle represents a potential solution and moves through the search space
- Movement is guided by both individual experience and collective swarm knowledge
- The algorithm iteratively improves solutions using a fitness function
- Simple to implement with fast convergence and few control parameters

How Particle Swarm Optimization (PSO) Works

Particle Swarm Optimization (PSO) is an iterative, population based optimization algorithm. It works by moving a group of particles (candidate solutions) through the search space using simple mathematical rules based on personal and collective experience.

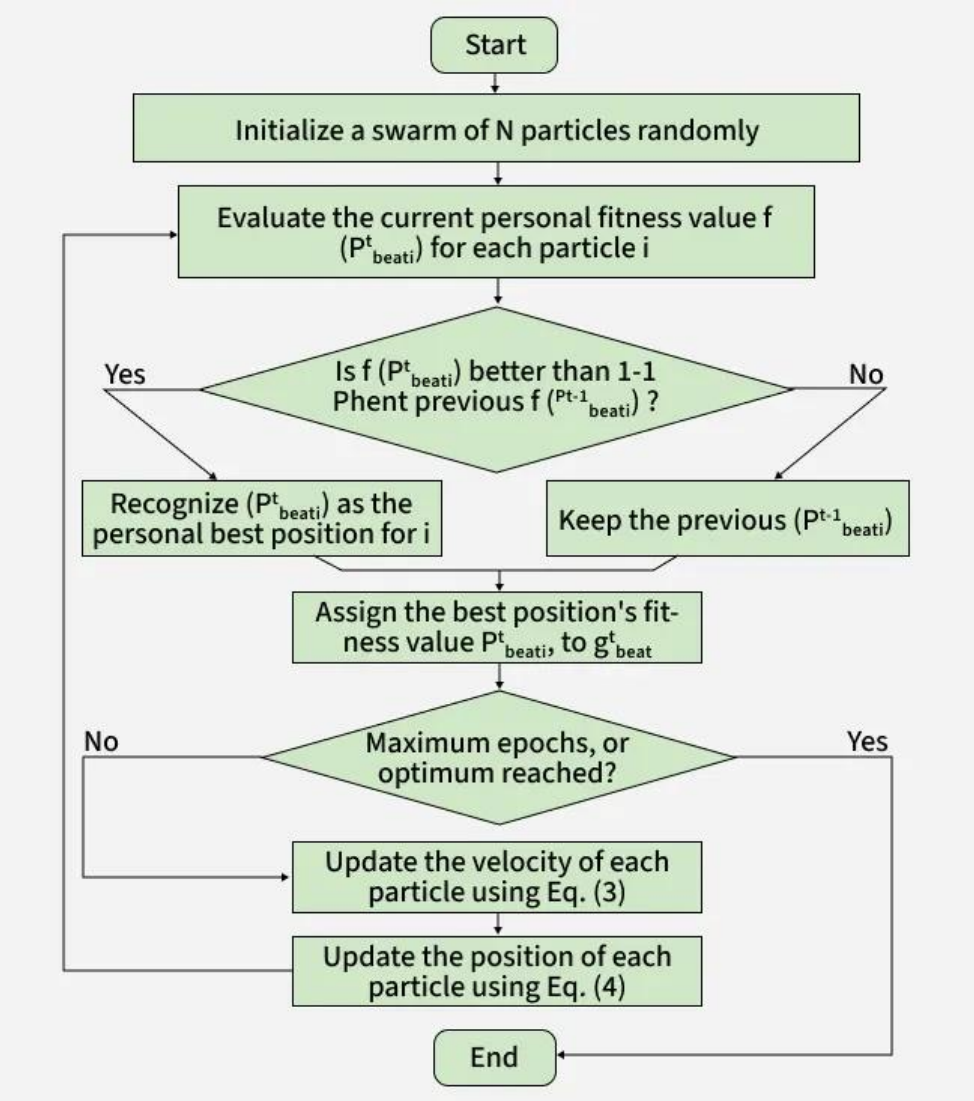
Each particle in PSO has

- **Position (x):** current solution
- **Velocity (v):** direction and speed of movement
- **Fitness value:** quality of the solution

Each particle stores:

- **pBest:** best position found by itself

- **gBest:** best position found by the entire swarm



Step 1: Initialization

- Randomly initialize N particles within the search space $[minx, maxx]$
- Assign a random velocity to each particle
- Evaluate the fitness value of each particle

pBest = current position
gBest = best pBest among all particles

Step 2: Velocity Update

At each iteration the velocity of a particle is updated using:

$$v_i^{t+1} = w \cdot v_i^t + c_1 \cdot r_1 \cdot (pBest_i - x_i^t) + c_2 \cdot r_2 \cdot (gBest - x_i^t)$$

where

- w: inertia weight (controls exploration)

- c_1: cognitive coefficient (self-learning)
- c_2: social coefficient (swarm learning)
- r_1, r_2: random values in [0,1]

Step 3: Position Update

After updating velocity the position is updated as:

$$x_i^{t+1} = x_i^t + v_i^{t+1}$$

If the new position goes outside [minx,maxx] clip it to the boundary.

Step 4: Update Best Positions

- If current fitness is better than pBest, update pBest
- If current fitness is better than gBest, update gBest

Step 5: Convergence

- Repeat Steps 2–4 for a fixed number of iterations or until convergence
- The swarm gradually moves toward the optimal solution

Step By Step Implementation

Here in this code we implements Particle Swarm Optimization (PSO) to find the global minimum of the Ackley function by iteratively updating a swarm of particles based on their personal best and the global best positions. It simulates collective behavior to efficiently search the solution space and converge to an optimal solution.

Step 1: Import Required Libraries

- Import numpy for numerical computations and vector operations
- Import math for mathematical constants and functions
- Import matplotlib.pyplot for possible visualization

```
import numpy as np
import math
import matplotlib.pyplot as plt
```

Step 2: Define the Cost (Objective) Function

- The Ackley function is a common benchmark function for optimization algorithms
- It is multimodal, non-convex and difficult for gradient-based methods
- PSO aims to minimize this function

```
def cost_function(x):
    a = 20
    b = 0.2
```

```
    c = 2 * math.pi
    d = len(x)

    term1 = -a * np.exp(-b * np.sqrt(np.sum(x**2) / d))
    term2 = -np.exp(np.sum(np.cos(c * x)) / d)

    return term1 + term2 + a + math.e
```

Step 3: Define PSO Hyperparameters

- **DIMENSIONS:** specifies the search space dimensionality
- **POPULATION:** represents the number of particles
- **MAX_ITER:** controls the number of optimization iterations
- **MIN_BOUND and MAX_BOUND:** define the search space limits
- **w, c1 and c2:** are inertia, cognitive and social coefficients

```
DIMENSIONS = 2
POPULATION = 30
MAX_ITER = 100

MIN_BOUND = -5
MAX_BOUND = 5

w = 0.7
c1 = 1.5
c2 = 1.5
V_MAX = 0.5
```

Step 4: Define the Particle Class

- Each particle represents a candidate solution
- Particles have position, velocity and personal best
- Fitness is calculated using the cost function
- Personal best is updated whenever a better solution is found

```
class Particle:
    def __init__(self):
        self.position = np.random.uniform(MIN_BOUND, MAX_BOUND, DIMENSIONS)
        self.velocity = np.random.uniform(-V_MAX, V_MAX, DIMENSIONS)
        self.best_position = self.position.copy()
        self.best_fitness = cost_function(self.position)
        self.fitness = self.best_fitness
```

Step 5: Initialize the Swarm and Global Best

- A swarm of particles is created

- The global best solution is selected from all personal bests
- This global best guides the swarm during optimization

```
def particle_swarm_optimization():
    swarm = [Particle() for _ in range(POPULATION)]

    gbest_position = swarm[0].best_position.copy()
    gbest_fitness = swarm[0].best_fitness

    for particle in swarm:
        if particle.best_fitness < gbest_fitness:
            gbest_fitness = particle.best_fitness
            gbest_position = particle.best_position.copy()
```

Step 6: Update Velocity and Position of Particles

- Velocity update includes inertia, cognitive and social components
- Random factors introduce stochastic behavior
- Velocity and position are clipped to avoid divergence
- This step enables exploration and exploitation

```
for iteration in range(MAX_ITER):
        for particle in swarm:
            r1 = np.random.rand(DIMENSIONS)
            r2 = np.random.rand(DIMENSIONS)

            particle.velocity = (
                w * particle.velocity
                + c1 * r1 * (particle.best_position - particle.position)
                + c2 * r2 * (gbest_position - particle.position)
            )

            particle.velocity = np.clip(particle.velocity, -V_MAX, V_MAX)
            particle.position = particle.position + particle.velocity
            particle.position = np.clip(particle.position, MIN_BOUND, MAX_BOUND)
```

Step 7: Update Personal Best and Global Best

- Fitness is recalculated after position update
- Personal best is updated if a better fitness is achieved
- Global best is updated if any particle outperforms the current best

```
particle.fitness = cost_function(particle.position)
```

```
                    if particle.fitness < particle.best_fitness:
                        particle.best_fitness = particle.fitness
                        particle.best_position = particle.position.copy()

                    if particle.fitness < gbest_fitness:
                        gbest_fitness = particle.fitness
                        gbest_position = particle.position.copy()
                    print(f"Iteration {iteration+1}/{MAX_ITER}, Best Fitness =
{gbest_fitness:.6f}")
                    return gbest_position, gbest_fitness
```

Step 8: Execute the PSO Algorithm

- The PSO function is executed from the main block
- Final optimal position and fitness value are printed
- The result approximates the global minimum of the Ackley function

```
if __name__ == "__main__":
    best_position, best_fitness = particle_swarm_optimization()

    print("\nOptimal Solution Found:")
    print("Best Position:", best_position)
    print("Best Fitness:", best_fitness)
```

```
Iteration 92/100, Best Fitness = 0.000002
Iteration 93/100, Best Fitness = 0.000002
Iteration 94/100, Best Fitness = 0.000002
Iteration 95/100, Best Fitness = 0.000002
Iteration 96/100, Best Fitness = 0.000001
Iteration 97/100, Best Fitness = 0.000001
Iteration 98/100, Best Fitness = 0.000001
Iteration 99/100, Best Fitness = 0.000001
Iteration 100/100, Best Fitness = 0.000001

Optimal Solution Found:
Best Position: [ 1.93715136e-07 -1.18610611e-07]
Best Fitness: 6.424593483878027e-07
```

Difference between Particle Swarm Optimization (PSO) and Genetic Algorithm (GA)

Here we compare Particle Swarm Optimization (PSO) and Genetic Algorithm(GA):

Parameter	PSO	Genetic Algorithm (GA)
Inspiration	Based on social behavior of birds or fish	Based on natural evolution and genetics

Parameter	PSO	Genetic Algorithm (GA)
Search Strategy	Particles move continuously through the search space	Uses randomized population evolution
Population Update	No creation or deletion particles only change position	New individuals are created using crossover and mutation
Operators Used	Velocity and position updates	Selection, crossover and mutation
Variable Handling	Best suited for continuous optimization	Handles both continuous and discrete variables
Complexity and Speed	Simple structure with faster convergence	More complex and generally slower

Application

1. Health Care Applications

- **Intelligent Diagnosis:** Optimizes models for faster and accurate disease detection.
- **Medical Robots:** Guides robots for precise surgical or therapeutic tasks.

2. Environmental Applications

- **Wild Vegetation Monitoring:** Tracks forest and vegetation growth efficiently.
- **Agriculture Monitoring:** Optimizes crop management and environmental monitoring.
- **Flood Control and Routing:** Predicts floods and finds optimal evacuation or routing paths.

3. Industrial Applications

- **WSN Deployment:** Optimizes sensor placement in Wireless Sensor Networks.
- **Product Defection Prediction:** Predicts defects to improve product quality.
- **Microgrid Design and Operation:** Optimizes design and operation for stability.

4 Smart City Applications

- **Smart City Planning:** Optimizes urban energy, traffic and resources.
- **Smart Home Automation:** Efficiently controls and schedules home appliances.
- **Appliance Scheduling:** Reduces energy consumption via optimal use.

Advantages

- Simple to implement with fewer control parameters.
- Fast convergence compared to many traditional algorithms.
- Does not require gradient information.

- Effective for non-linear and multimodal problems.
- Easily used with other optimization techniques.

Limitations

- May converge early to local optima.
- Weak local search capability in later iterations.
- Performance depends on proper parameter tuning.
- Computational cost increases with swarm size.
- Less effective for discrete optimization problems.