Subject: Artificial Intelligence (DJ19DSC502)

AY: 2023-24

NAME:DIVYESH KHUNT SAPID:60009210116 BATCH:D12

Experiment 5

(Solution Space)

Aim: Implement Genetic Algorithm to solve Travelling Salesman Problem.

Theory:

Genetic algorithms are heuristic search algorithms inspired by the process that supports the evolution of life. The algorithm is designed to replicate the natural selection process to carry generation, i.e. survival of the fittest of beings. Standard genetic algorithms are divided into five phases which are:

1. Creating initial population.
2. Calculating fitness.
3. Selecting the best genes.
4. Crossing over.
5. Mutating to introduce variations.

These algorithms can be implemented to find a solution to the optimization problems of various types. One such problem is the Traveling Salesman Problem. The problem says that a salesman is given a set of cities, he has to find the shortest route to as to visit each city exactly once and return to the starting city.
Approach: In the following implementation, cities are taken as genes, string generated using these characters is called a chromosome, while a fitness score which is equal to the path length of all the cities mentioned, is used to target a population.
Fitness Score is defined as the length of the path described by the gene. Lesser the path length fitter is the gene. The fittest of all the genes in the gene pool survive the population test and move to the next iteration. The number of iterations depends upon the value of a cooling variable. The value of the cooling variable keeps on decreasing with each iteration and reaches a threshold after a certain number of

iterations.

Algorithm:

1. Initialize the population randomly.

2. Determine the fitness of the chromosome.

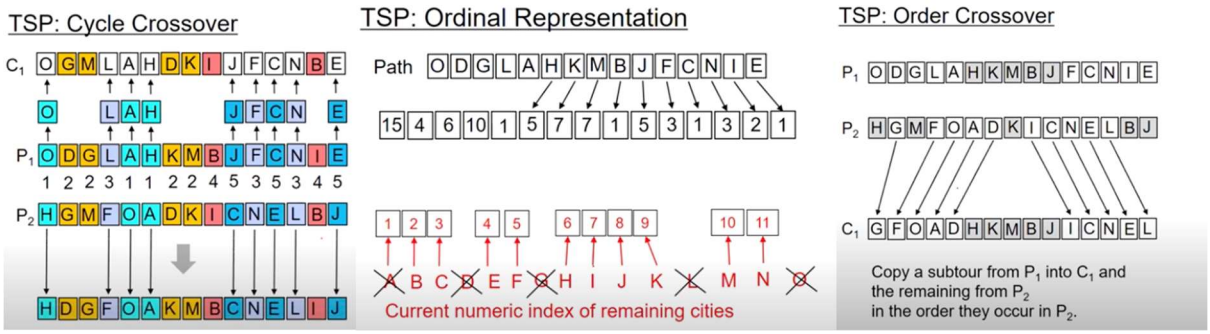
3. Until done repeat:

   1. Select parents.

   2. Perform crossover and mutation.

   3. Calculate the fitness of the new population.

   4. Append it to the gene pool.



Lab Assignment to do:

1. Implement Genetic algorithm for 20 cities using Cyclic crossover and find the number of iterations where the algorithm converges.
2. Compare the performance of Genetic algorithm using 5, 10, 20 and 40 cities.
3. Compare the time and space complexity of genetic algorithm using cyclic and ordinal crossovers for 10 cities.

```python
import numpy as np
n = int(input("Enter the number of cities: "))
distance_matrix = np.zeros((n, n))
print("Enter the distance matrix:")
for i in range(n):
  for j in range(n):
    if i != j:
      distance = float(input(f"Distance from city {i} to city {j}: "))
      distance_matrix[i][j] = distance

print("Distance Matrix:")
for i in range(n):
  for j in range(n):
    print(distance_matrix[i][j], end='\t')
  print()
```

```
Enter the number of cities: 5
Enter the distance matrix:
Distance from city 0 to city 1: 10
Distance from city 0 to city 2: 15
Distance from city 0 to city 3: 20
Distance from city 0 to city 4: 25
Distance from city 1 to city 0: 10
Distance from city 1 to city 2: 35
Distance from city 1 to city 3: 40
Distance from city 1 to city 4: 45
Distance from city 2 to city 0: 15
Distance from city 2 to city 1: 35
Distance from city 2 to city 3: 55
Distance from city 2 to city 4: 60
Distance from city 3 to city 0: 20
Distance from city 3 to city 1: 40
Distance from city 3 to city 2: 55
Distance from city 3 to city 4: 70
Distance from city 4 to city 0: 25
Distance from city 4 to city 1: 45
Distance from city 4 to city 2: 60
Distance from city 4 to city 3: 70
Distance Matrix:
0.0     10.0    15.0    20.0    25.0
10.0    0.0     35.0    40.0    45.0
15.0    35.0    0.0     55.0    60.0
20.0    40.0    55.0    0.0     70.0
25.0    45.0    60.0    70.0    0.0
```

```
[ ]  import itertools
     def fitness(tour, distance_matrix):
         total_distance = 0
         for i in range(len(tour) - 1):
             total_distance += distance_matrix[tour[i]][tour[i + 1]]
         total_distance += distance_matrix[tour[-1]][tour[0]]
         return total_distance
```

```
[ ]  def fitnessroulette_wheel(population, distances, num_parents):
         num_parents = 4
         fitness_values = np.array([1 / d for d in distances])
         total_fitness = np.sum(fitness_values)
         probabilities = fitness_values / total_fitness
         selected_parents_indices = np.random.choice(len(population), num_parents, p=probabilities, replace=False)
         selected_parents = [population[i] for i in selected_parents_indices]
         print("Selected Parents:")
         for parent in selected_parents:
             print(parent)
         return selected_parents
```

```
import random
def generatecombination(distance_matrix):
    n = len(distance_matrix)
    cities = list(range(n))
    all_permutations = list(itertools.permutations(cities))
    all_combinations = list(itertools.combinations(cities, n))

    print("All Permutations and Combinations of Cities:")
    for permutation in all_permutations:
        print("Permutation:", permutation)
        cost = fitness(permutation, distance_matrix)
        print("Total Distance:", cost)

    for combination in all_combinations:
        print("Combination:", combination)
        cost = fitness(combination, distance_matrix)
        print("Total Distance:", cost)

    selected_parents = random.sample(all_permutations, 4)
    selected_distances = [fitness(parent, distance_matrix) for parent in selected_parents]
    print("\n")
    print(f"Randomly selected {4} Parents:")
    for parent in selected_parents:
        print("Parent:", parent)
        cost = fitness(parent, distance_matrix)
        print("Total Distance:", cost)
    fitnessroulette_wheel(selected_parents, selected_distances, 4)
    return selected_parents, selected_distances
```

```
selected_parents_main, selected_distances_main = generate_permutations_combinations_and_costs(distance_matrix)
```

```
All Permutations and Combinations of Cities:
Permutation: (0, 1, 2, 3, 4)
Total Distance: 195.0
Permutation: (0, 1, 2, 4, 3)
Total Distance: 195.0
Permutation: (0, 1, 3, 2, 4)
Total Distance: 190.0
Permutation: (0, 1, 3, 4, 2)
Total Distance: 195.0
Permutation: (0, 1, 4, 2, 3)
Total Distance: 190.0
Permutation: (0, 1, 4, 3, 2)
Total Distance: 195.0
Permutation: (0, 2, 1, 3, 4)
Total Distance: 185.0
Permutation: (0, 2, 1, 4, 3)
Total Distance: 185.0
Permutation: (0, 2, 3, 1, 4)
Total Distance: 180.0
Permutation: (0, 2, 3, 4, 1)
Total Distance: 195.0
Permutation: (0, 2, 4, 1, 3)
Total Distance: 180.0
Permutation: (0, 2, 4, 3, 1)
Total Distance: 195.0
Permutation: (0, 3, 1, 2, 4)
Total Distance: 180.0
Permutation: (0, 3, 1, 4, 2)
Total Distance: 180.0
Permutation: (0, 3, 2, 1, 4)
Total Distance: 180.0
Permutation: (0, 3, 2, 4, 1)
Total Distance: 190.0
Permutation: (0, 3, 4, 1, 2)
Total Distance: 185.0
Permutation: (0, 3, 4, 2, 1)
Total Distance: 195.0
Permutation: (0, 4, 1, 2, 3)
Total Distance: 195.0
Permutation: (0, 4, 1, 2, 3)
Total Distance: 180.0
Permutation: (0, 4, 1, 3, 2)
Total Distance: 180.0
Permutation: (0, 4, 2, 1, 3)
Total Distance: 180.0
Permutation: (0, 4, 2, 3, 1)
Total Distance: 190.0
Permutation: (0, 4, 3, 1, 2)
Total Distance: 185.0
Permutation: (0, 4, 3, 2, 1)
Total Distance: 195.0
Permutation: (1, 0, 2, 3, 4)
Total Distance: 195.0
Permutation: (1, 0, 2, 4, 3)
Total Distance: 195.0
Permutation: (1, 0, 3, 2, 4)
Total Distance: 190.0
Permutation: (1, 0, 3, 4, 2)
Total Distance: 195.0
Permutation: (1, 0, 4, 2, 3)
Total Distance: 190.0
Permutation: (1, 0, 4, 3, 2)
Total Distance: 195.0
Permutation: (1, 2, 0, 3, 4)
Total Distance: 185.0
Permutation: (1, 2, 0, 4, 3)
Total Distance: 185.0
Permutation: (1, 2, 3, 0, 4)
Total Distance: 180.0
Permutation: (1, 2, 3, 4, 0)
Total Distance: 195.0
Permutation: (1, 2, 4, 0, 3)
Total Distance: 180.0
Permutation: (1, 2, 4, 3, 0)
Total Distance: 195.0
Permutation: (1, 3, 0, 2, 4)
Total Distance: 180.0
```

```
Randomly selected 4 Parents:
Parent: (2, 0, 4, 3, 1)
Total Distance: 185.0
Parent: (3, 0, 2, 1, 4)
Total Distance: 185.0
Parent: (3, 2, 0, 4, 1)
Total Distance: 180.0
Parent: (0, 1, 3, 2, 4)
Total Distance: 190.0
Selected Parents:
(3, 0, 2, 1, 4)
(3, 2, 0, 4, 1)
(2, 0, 4, 3, 1)
(0, 1, 3, 2, 4)
```

```python
def crossover(parent1, parent2):
    n = len(parent1)
    i, j = np.random.choice(n, 2, replace=False)
    if i > j:
        i, j = j, i

    child = [-1] * n

    child[i:j+1] = parent1[i:j+1]

    for k in range(n):
        if i <= k <= j:
            continue
        p2_city = parent2[k]
        while p2_city in child:
            idx = parent2.index(p2_city)
            p2_city = parent1[idx]
        child[k] = p2_city
```

```python
  print("Crossover Result:")
  print("Parent 1:", parent1)
  print("Parent 2:", parent2)
  print("Child:", child)

  return child
```

```
import random

def mutate(tour):
    idx1, idx2 = random.sample(range(len(tour)), 2)
    tour[idx1], tour[idx2] = tour[idx2], tour[idx1]
    return tour
```

```
costs = [0]
parent1_main = []
parent2_main = []
mutated_child_main = []
mutated_child_all = []
for i in range(len(selected_parents_main)):
  for j in range(len(selected_parents_main)):
    if i!=j:
      parent1 = selected_parents_main[i]
      parent2 = selected_parents_main[j]
      child_main = crossover(parent1, parent2)
      mutated_child = mutate(child_main)
      mutated_child_all.append(mutated_child)
      print("Mutated child: ", mutated_child)
      cost = calculate_total_distance(mutated_child, distance_matrix)
      costs.append(cost)
      print("Cost: ", cost)
```

```
      if cost>=max(costs):
        parent1_main.append(parent1)
        parent2_main.append(parent2)
        mutated_child_main.append(mutated_child)

print("Parent 1 main: ", parent1_main[-1])
print("Parent 2 main: ", parent2_main[-1])
print("Mutated child main: ", mutated_child_main[-1])
print("Total Cost: ", max(costs))
```

```
Crossover Result:
Parent 1: (3, 0, 2, 1, 4)
Parent 2: (2, 0, 4, 3, 1)
Child: [3, 0, 2, 1, 4]
Mutated child:  [3, 4, 2, 1, 0]
Cost:  195.0
Crossover Result:
Parent 1: (3, 0, 2, 1, 4)
Parent 2: (3, 2, 0, 4, 1)
Child: [3, 0, 2, 1, 4]
Mutated child:  [3, 1, 2, 0, 4]
Cost:  185.0
Crossover Result:
Parent 1: (3, 0, 2, 1, 4)
Parent 2: (0, 1, 3, 2, 4)
Child: [3, 0, 2, 1, 4]
Mutated child:  [3, 0, 1, 2, 4]
Cost:  195.0
Crossover Result:
Parent 1: (3, 2, 0, 4, 1)
Parent 2: (2, 0, 4, 3, 1)
Child: [3, 2, 0, 4, 1]
Mutated child:  [3, 1, 0, 4, 2]
Cost:  190.0
Crossover Result:
Parent 1: (3, 2, 0, 4, 1)
Parent 2: (3, 0, 2, 1, 4)
Child: [3, 2, 0, 4, 1]
Mutated child:  [3, 4, 0, 2, 1]
Cost:  185.0
Crossover Result:
Parent 1: (3, 2, 0, 4, 1)
Parent 2: (0, 1, 3, 2, 4)
Child: [3, 2, 0, 4, 1]
Mutated child:  [3, 2, 4, 0, 1]
Cost:  190.0
```

```
Crossover Result:
Parent 1: (0, 1, 3, 2, 4)
Parent 2: (2, 0, 4, 3, 1)
Child: [0, 1, 3, 2, 4]
Mutated child:  [0, 2, 3, 1, 4]
Cost:  180.0
Crossover Result:
Parent 1: (0, 1, 3, 2, 4)
Parent 2: (3, 0, 2, 1, 4)
Child: [0, 1, 3, 2, 4]
Mutated child:  [0, 1, 3, 4, 2]
Cost:  195.0
Crossover Result:
Parent 1: (0, 1, 3, 2, 4)
Parent 2: (3, 2, 0, 4, 1)
Child: [0, 1, 3, 2, 4]
Mutated child:  [0, 1, 4, 2, 3]
Cost:  190.0
Parent 1 main:  (0, 1, 3, 2, 4)
Parent 2 main:  (3, 0, 2, 1, 4)
Mutated child main:  [0, 1, 3, 4, 2]
Total Cost:  195.0
```