Experiment:4

Aim: Write a Program to Implement Travelling Salesman Problem.

# Description of Travelling Salesman Problem:

The Traveling Salesman Problem (TSP) is a classic problem in the field of artificial intelligence and optimization. It is a combinatorial optimization problem that involves finding the shortest possible route that visits a set of cities exactly once and returns to the original city. The problem is formally defined as follows:

Given a set of *n* cities and the distances between each pair of cities, the objective is to find the shortest possible tour that visits each city exactly once and returns to the starting city.

Here are some key components and descriptions related to the Traveling Salesman Problem in AI:

1. Cities: The problem involves a finite set of cities, each representing a location that needs to be visited. The number of cities can vary depending on the problem instance.
2. Distances: The distance between each pair of cities is typically represented by a distance matrix. The distance could be measured in various units such as miles, kilometers, or any other appropriate metric.
3. Optimization Criterion: The objective of the TSP is to minimize the total distance traveled in the tour. The goal is to find the shortest route that visits all cities exactly once and returns to the starting city.
4. Combinatorial Nature: The TSP is a combinatorial optimization problem because it involves exploring many possible solutions. The number of possible tours grows factorially with the number of cities, making it computationally challenging to find the optimal solution for large instances.
5. NP-Hardness: The Traveling Salesman Problem is classified as NP-hard, which means that it is difficult to find an optimal solution in polynomial time. As the number of cities increases, the problem becomes increasingly difficult to solve using exact algorithms.
6. Applications: The TSP has numerous real-world applications beyond salesmen routing, including logistics, transportation planning, manufacturing, DNA sequencing, and network optimization.
7. Algorithms: Various algorithms are used to tackle the TSP, including exact algorithms (such as dynamic programming, branch and bound), approximation algorithms (such as nearest neighbor, minimum spanning tree), and heuristic algorithms (such as genetic algorithms, simulated annealing, ant colony optimization).
8. Heuristics and Metaheuristics: Due to the computational complexity of the problem, heuristic and metaheuristic approaches are often employed to find near-optimal solutions within reasonable time limits.

# Here are some common approaches to solving the Traveling Salesman Problem in AI:

1. Exact Algorithms:
   * Brute Force: Enumerate all possible permutations of cities and calculate the total distance for each permutation. While guaranteed to find the optimal solution, it is impractical for large problem instances due to its exponential time complexity.
   * Dynamic Programming: Utilizes memorization to avoid redundant calculations and solve smaller subproblems efficiently. The Held-Karp algorithm is a well-known dynamic programming approach for solving the TSP with a time complexity of *O*(*n2*2*n*).
2. Approximation Algorithms:
   * Nearest Neighbor Algorithm: Start from an arbitrary city and iteratively choose the nearest unvisited city until all cities are visited, then return to the starting city. While simple, it may not always produce optimal solutions.
   * Minimum Spanning Tree (MST) Heuristics: Construct a minimum spanning tree (MST) from the cities and then traverse the tree to form a tour. Algorithms like Christofides' algorithm extend the MST approach by adding shortcut edges and can achieve solutions within a factor of 3/2 of the optimal solution.
3. Heuristic Algorithms:
   * Genetic Algorithms (GA): Inspired by the process of natural selection, genetic algorithms evolve a population of candidate solutions through mutation, crossover, and selection operators to find a good solution to the TSP.
   * Ant Colony Optimization (ACO): Inspired by the foraging behavior of ants, ACO algorithms use pheromone trails and heuristic information to guide the search for good solutions. Ants construct solutions probabilistically and update pheromone trails based on the quality of solutions found.

**Ant Colony Optimization (ACO)** is a popular metaheuristic algorithm inspired by the foraging behaviour of ants. It has been widely applied to solve combinatorial optimization problems such as the Traveling Salesman Problem (TSP). The basic idea behind ACO is to mimic the pheromone communication among ants to guide the search for good solutions. Here's how ACO works for solving the TSP:

# Initialization:

* + Randomly place a number of ants at each city.
  + Initialize pheromone levels on the edges between cities.

# Ant Solution Construction:

* + Each ant constructs a solution (tour) by probabilistically selecting the next city to visit based on pheromone levels and heuristic information.
  + The probability of choosing a particular city depends on both the pheromone level on the edge and the distance to the city. Generally, ants tend to prefer shorter paths with higher pheromone levels.
  + Ants complete their tours by visiting all cities exactly once.

# Pheromone Update:

* + After all ants have completed their tours, pheromone levels on the edges are updated based on the quality of the solutions found.
  + Stronger pheromone trails are deposited on edges belonging to shorter tours.
  + Pheromone evaporation is applied to prevent stagnation and ensure exploration of the solution space.

# Termination Condition:

* + Repeat the construction and update process for a certain number of iterations or until a termination condition is met (e.g., a maximum number of iterations, convergence criteria).

# Solution Construction and Update:

* + Repeat the process until a termination condition is met.

# Solution Selection:

* + After multiple iterations, select the best solution found by the ants as the final solution to the TSP.

# Key Components of ACO for TSP:

* **Pheromone Trails**: Ants deposit pheromone on edges as they construct solutions. The amount of pheromone is updated based on the quality of the solution.
* **Heuristic Information**: Ants use heuristic information, such as the distance between cities, to guide their decision-making process.
* **Pheromone Update Rule**: The pheromone update rule balances exploration and exploitation. It usually involves depositing more pheromone on shorter tours and applying evaporation to prevent pheromone trails from becoming too strong.
* **Parameter Tuning**: ACO involves tuning parameters such as the evaporation rate, pheromone influence factor, heuristic information factor, and the number of ants to achieve good performance.

# Implementation of Travelling Salesman Problem using Ant Colony Optimization (ACO)

import numpy as np

class AntColony:

def init (self, distances, n\_ants, n\_best, n\_iterations, decay, alpha=1, beta=1): """

Args:

distances (2D numpy.array): Square matrix of distances. Diagonal is assumed to be np.inf. n\_ants (int): Number of ants running per iteration

n\_best (int): Number of best ants who deposit pheromone n\_iteration (int): Number of iterations

decay (float): Rate it which pheromone decays. The pheromone value is multiplied by decay, so 0.95 will lead to decay, 0.5 to much faster decay.

alpha (int or float): exponenet on pheromone, higher alpha gives pheromone more weight. Default=1 beta (int or float): exponent on distance, higher beta give distance more weight. Default=1

"""

self.distances = distances

self.pheromone = np.ones(self.distances.shape) / len(distances) self.all\_inds = range(len(distances))

self.n\_ants = n\_ants self.n\_best = n\_best self.n\_iterations = n\_iterations self.decay = decay

self.alpha = alpha self.beta = beta

def run(self): shortest\_path = None

shortest\_path\_length = np.inf for i in range(self.n\_iterations):

all\_paths = self.gen\_all\_paths()

self.spread\_pheronome(all\_paths, self.n\_best, shortest\_path, shortest\_path\_length) shortest\_path, shortest\_path\_length = self.choose\_best\_path(all\_paths) self.pheromone \* self.decay

return shortest\_path, shortest\_path\_length

def spread\_pheronome(self, all\_paths, n\_best, shortest\_path, shortest\_path\_length): sorted\_paths = sorted(all\_paths, key=lambda x: x[1])

for path, length in sorted\_paths[:n\_best]: for move in path:

self.pheromone[move] += 1.0 / self.distances[move]

def choose\_best\_path(self, all\_paths): # returns the best path and its length best\_path, best\_length = all\_paths[0] for path, length in all\_paths[1:]:

if length < best\_length: best\_length = length best\_path = path

return best\_path, best\_length

def gen\_path\_dist(self, path):

total\_dist = 0 for ele in path:

total\_dist += self.distances[ele] return total\_dist

def gen\_all\_paths(self): all\_paths = []

for i in range(self.n\_ants): path = self.gen\_path(0)

all\_paths.append((path, self.gen\_path\_dist(path))) return all\_paths

def gen\_path(self, start): path = []

visited = set() visited.add(start) prev = start

for i in range(len(self.distances) - 1):

move = self.pick\_move(self.pheromone[prev], self.distances[prev], visited) path.append((prev, move))

prev = move visited.add(move)

path.append((prev, start)) # going back to where we started return path

def pick\_move(self, pheromone, dist, visited): pheromone = np.copy(pheromone) pheromone[list(visited)] = 0

row = pheromone \*\* self.alpha \* (( 1.0 / dist) \*\* self.beta)

norm\_row = row / row.sum()

move = np\_choice(self.all\_inds, 1, p=norm\_row)[0] return move

# Example Usage:

if name == ' main ': # Example usage:

distances = np.array([[np.inf, 2, 2, 5, 7],

[2, np.inf, 4, 8, 2],

[2, 4, np.inf, 1, 3],

[5, 8, 1, np.inf, 2],

[7, 2, 3, 2, np.inf]])

n\_ants = 5

n\_best = 2

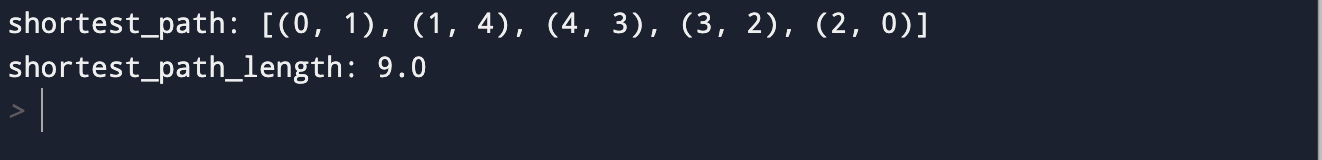
n\_iterations = 10

decay = 0.95

ant\_colony = AntColony(distances, n\_ants, n\_best, n\_iterations, decay) shortest\_path, length = ant\_colony.run()

print ("shorted\_path", shortest\_path) print ("shortest\_path\_length", length)

# Output:



**Description of the code:**

In order to solve the Traveling Salesman Problem (TSP), this Python code implements the Ant Colony Optimization (ACO) algorithm, which seeks to determine the shortest path that visits each city precisely once and then returns to the starting point. The program directs the ants to find better solutions over iterations by using pheromone levels and heuristic information (distance). The technique strives for an optimal or nearly optimal solution to the TSP by iteratively updating the pheromone levels on the antpaths.

# Date of experiment performed: Day of experiment performed: Date of experiment submission: Day of experiment Submission:

Faculty Co-ordinator Signature