Name: **Kunal Sinha** Batch: Swarm **R1- G2** Roll no: **2K17/CO/164**

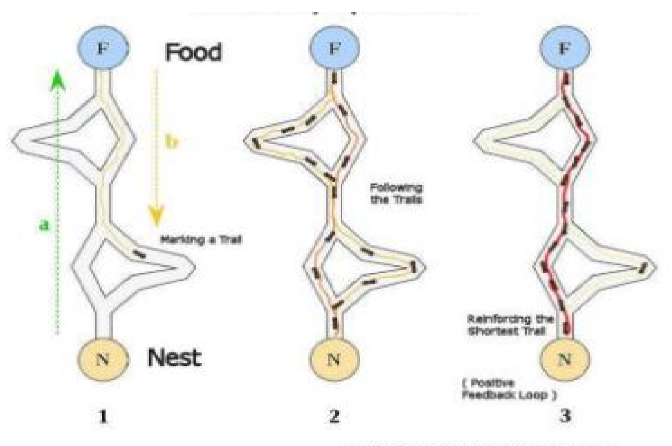
Experiment 4

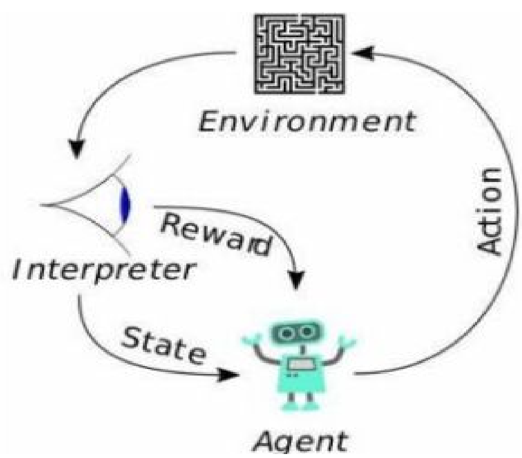
Aim: Write a program to implement Ant Colony optimization (ACO) algorithm.

Theory:

In the natural world, ants of some species (initially) wander randomly, and upon finding food return to their colony while laying down pheromone trails. If other ants find such a path, they are likely not to keep travelling at random, but instead to follow the trail, returning and reinforcing it if they eventually find food.

Over time, however, the pheromone trail starts to evaporate, thus reducing its attractive strength. The more time it takes for an ant to travel down the path and back again, the more time the pheromones have to evaporate. A short path, by comparison, gets marched over more frequently, and thus the pheromone density becomes higher on shorter paths than longer ones. Pheromone evaporation also has the advantage of avoiding the convergence to a locally optimal solution.
The overall result is that when one ant finds a good (i.e., short) path from the colony to a food source, other ants are more likely to follow that path, and positive feedback eventually leads to many ants following a single path.





Algorithm:

Input: Data of 101 cities with distance between them

1. BEGIN
2. Generate initial population of size nA(ants)
3. Initialize the pheromone trail and parameters
4. Evaluate initial population according to the fitness function
5. Find best solution of the population

6. While (current_iteration <= nI)
    a. Do Until each ant completely builds a solution
        i. Local trial update
    b. END Do
    c. Update pheromone
    d. Determine the global best ant
7. END While

Source Code:

aco.py

```
import random
class Graph(object):
    def __init__(self, cost_matrix: list, rank: int):
        self.matrix = cost_matrix
        self.rank = rank
        # noinspection PyUnusedLocal
        self.pheromone = [[1 / (rank * rank) for j in range(rank)] for i in range(rank)]

class ACO(object):
    def __init__(self, ant_count: int, generations: int, alpha: float, beta: float, rho: float, q: int,
strategy: int):
        self.Q = q
        self.rho = rho
        self.beta = beta
        self.alpha = alpha
        self.ant_count = ant_count
        self.generations = generations
        self.update_strategy = strategy

    def _update_pheromone(self, graph: Graph, ants: list):
        for i, row in enumerate(graph.pheromone):
            for j, col in enumerate(row):
                graph.pheromone[i][j] *= self.rho
                for ant in ants:
                    graph.pheromone[i][j] += ant.pheromone_delta[i][j]

    def solve(self, graph: Graph):
        best_cost = float('inf')
        best_solution = []
        for gen in range(self.generations):
            ants = [_Ant(self, graph) for i in range(self.ant_count)]
```

```
            for ant in ants:
                for i in range(graph.rank - 1):
                    ant._select_next()
                ant.total_cost += graph.matrix[ant.tabu[-1]][ant.tabu[0]]
                if ant.total_cost < best_cost:
                    best_cost = ant.total_cost
                    best_solution = [] + ant.tabu
                ant._update_pheromone_delta()
            self._update_pheromone(graph, ants)
        return best_solution, best_cost


class _Ant(object):
    def __init__(self, aco: ACO, graph: Graph):
        self.colony = aco
        self.graph = graph
        self.total_cost = 0.0
        self.tabu = []  # tabu list
        self.pheromone_delta = []  # the local increase of pheromone
        self.allowed = [i for i in range(graph.rank)]  # nodes which are allowed for the next
selection
        self.eta = [[0 if i == j else 1 / graph.matrix[i][j] for j in range(graph.rank)] for i in
                    range(graph.rank)]  # heuristic information
        start = random.randint(0, graph.rank - 1)  # start from any node
        self.tabu.append(start)
        self.current = start
        self.allowed.remove(start)

    def _select_next(self):
        denominator = 0
        for i in self.allowed:
            denominator += self.graph.pheromone[self.current][i] ** self.colony.alpha *
self.eta[self.current][ i] ** self.colony.beta
        probabilities = [0 for i in range(self.graph.rank)]
        for i in range(self.graph.rank):
            try:
                self.allowed.index(i)  # test if allowed list contains i
                probabilities[i] = self.graph.pheromone[self.current][i] ** self.colony.alpha * \
                    self.eta[self.current][i] ** self.colony.beta / denominator
            except ValueError:
                pass  # do nothing
        selected = 0
```

```
        rand = random.random()
        for i, probability in enumerate(probabilities):
            rand -= probability
            if rand <= 0:
                selected = i
                break
        self.allowed.remove(selected)
        self.tabu.append(selected)
        self.total_cost += self.graph.matrix[self.current][selected]
        self.current = selected

    def _update_pheromone_delta(self):
        self.pheromone_delta = [[0 for j in range(self.graph.rank)] for i in range(self.graph.rank)]
        for _ in range(1, len(self.tabu)):
            i = self.tabu[_ - 1]
            j = self.tabu[_]
            if self.colony.update_strategy == 1:  # ant-quality system
                self.pheromone_delta[i][j] = self.colony.Q
            elif self.colony.update_strategy == 2:  # ant-density system
                # noinspection PyTypeChecker
                self.pheromone_delta[i][j] = self.colony.Q / self.graph.matrix[i][j]
            else:  # ant-cycle system
                self.pheromone_delta[i][j] = self.colony.Q / self.total_cost
```

plot.py

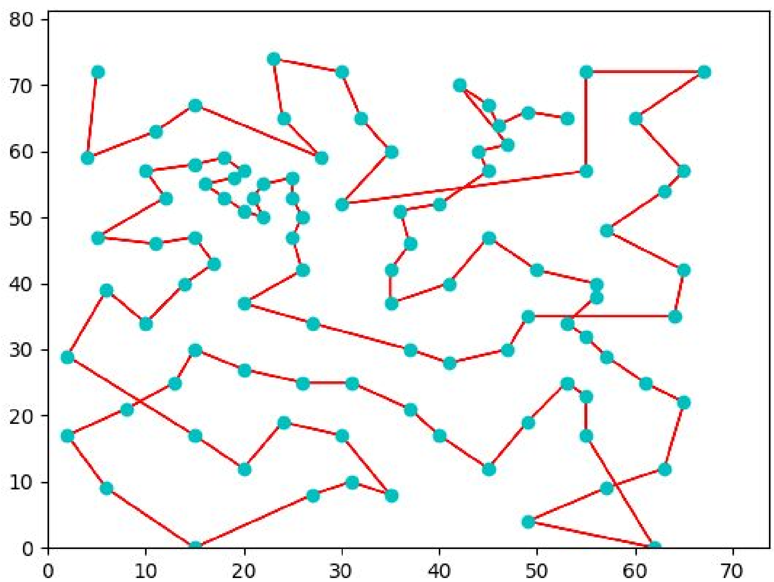
```
import operator
import matplotlib.pyplot as plt
def plot(points, path: list):
    x = []
    y = []
    for point in points:
        x.append(point[0])
        y.append(point[1])
    y = list(map(operator.sub, [max(y) for i in range(len(points))], y))
    plt.plot(x, y, 'co')
    for _ in range(1, len(path)):
        i = path[_ - 1]
        j = path[_]
        plt.arrow(x[i], y[i], x[j] - x[i], y[j] - y[i], color='r', length_includes_head=True)
    plt.xlim(0, max(x) * 1.1)
    plt.ylim(0, max(y) * 1.1)
    plt.show()
```

main.py

```python
import math
from aco import ACO, Graph
from plot import plot

def distance(city1: dict, city2: dict):
    return math.sqrt((city1['x'] - city2['x']) ** 2 + (city1['y'] - city2['y']) ** 2)

def main():
    cities = []
    points = []
    with open('./data/dataset.txt') as f:
        for line in f.readlines():
            city = line.split(' ')
            cities.append(dict(index=int(city[0]), x=int(city[1]), y=int(city[2])))
            points.append((int(city[1]), int(city[2])))
    cost_matrix = []
    rank = len(cities)
    for i in range(rank):
        row = []
        for j in range(rank):
            row.append(distance(cities[i], cities[j]))
        cost_matrix.append(row)
    aco = ACO(10, 100, 1.0, 10.0, 0.5, 10, 2)
    graph = Graph(cost_matrix, rank)
    path, cost = aco.solve(graph)
    print('cost: {}, path: {}'.format(cost, path))
    plot(points, path)

if __name__ == '__main__':
    main()
```

Output:

```
C:\Windows\System32\cmd.exe
Microsoft Windows [Version 10.0.19041.572]
(c) 2020 Microsoft Corporation. All rights reserved.

C:\Users\Admin\Desktop\Kunal Sinha_CO-164_Swarm Assignment#02\TSP using ACO>python main.py
cost: 740.6554488421093, path: [55, 74, 73, 21, 40, 71, 72, 20, 39, 57, 52, 100, 26, 27, 25, 11, 79, 67, 76, 2, 78, 77,
33, 34, 70, 65, 64, 8, 80, 32, 50, 19, 29, 69, 30, 87, 6, 81, 47, 46, 35, 48, 63, 62, 89, 31, 9, 61, 10, 18, 45, 44, 7,
82, 59, 4, 83, 16, 60, 15, 90, 99, 36, 97, 84, 92, 98, 95, 58, 91, 96, 94, 93, 5, 88, 17, 51, 68, 0, 49, 75, 28, 23, 53,
 54, 24, 38, 66, 22, 3, 12, 1, 56, 14, 42, 41, 86, 13, 43, 85, 37]

C:\Users\Admin\Desktop\Kunal Sinha_CO-164_Swarm Assignment#02\TSP using ACO>
```



Finding and Learnings:

We have successfully implemented the ant colony optimization technique on travelling salesman problem in python.The idea of the ant colony algorithm is to mimic this behavior with "simulated ants" walking around the graph representing the problem to solve.