<u>DELHI TECHNOLOGICAL UNIVERSITY</u>

(Formerly Delhi College of Engineering)
Shahbad Daulatpur, Bawana Road, Delhi 110042

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CSE5216: Information and Network Security Lab File

<u>Submitted To:</u>
Mr. Sanjay Patidar
Associate Professor
Science
Department of Computer
Science and Engineering

<u>Submitted By:</u>
Kunal Sinha
B.Tech Computer
7th Semester
2K17/CO/164

INDEX

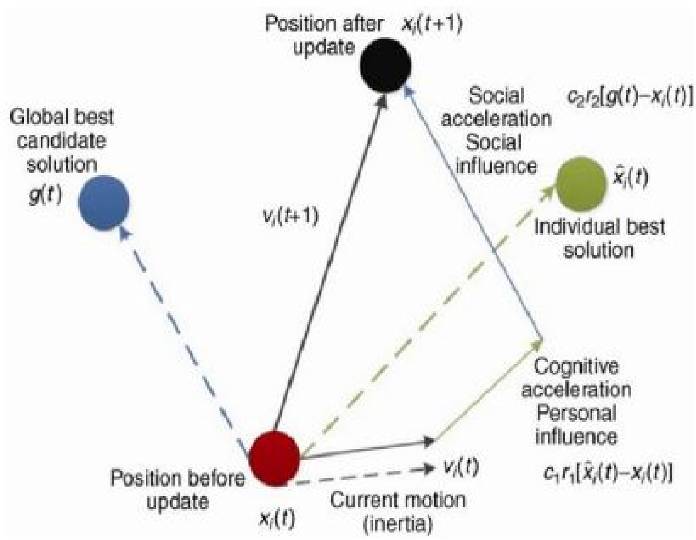
S.No.	Topic	Date	Signature

Experiment 1

Aim: Write a program to implement Particle swarm optimization algorithm. .

Theory:

PSO shares many similarities with evolutionary computation techniques such as Genetic Algorithms (GA). The system is initialized with a population of random solutions and searches for optima by updating generations. However, unlike GA, PSO has no evolution operators such as crossover and mutation. In PSO, the potential solutions, called particles, fly through the problem space by following the current optimum particles PSO is initialized with a group of random particles (solutions) and then searches for optima by updating generations. In every iteration, each particle is updated by following two "best" values. The first one is the best solution (fitness) it has achieved so far. (The fitness value is also stored.) This value is called pbest. Another "best" value that is tracked by the particle swarm optimizer is the best value, obtained so far by any particle in the population. This best value is a global best and called gbest.



Let's take a closer look to the equation that defines the velocity of the next iteration of a particle dimension:

- $V_i(k+1)$ is the next iteration velocity
- W is an inertial parameter. This parameter affects the movement propagation given by the last velocity value.
- C_1 and C_2 are acceleration coefficients. C_1 value gives the importance of personal best value and C_2 is the importance of social best value.
- P_i is the best individual position and Pg is the best position of all particles. In the equation, the distance of each of these parameters to the particle's actual position.

- $rand_1$ and $rand_2$ are random numbers where $0 \le rand \le 1$ and they control the influence of each value: Social and individual as shown below.

After that is calculated the new particle's position until the number of iterations specified or an error criteria be reached

Algorithm:

Input: Data of 101 cities with distance between them

```
FOR each particle i
    FOR each dimension d
      Initialize position x_id randomly within permissible range
      Initialize velocity v_id randomly within permissible range
    End FOR
END FOR
Iteration k=1
DO
   FOR each particle i
       Calculate fitness value
       IF the fitness value is better than p_best_id in history
         Set current fitness value as the p_best_id
       END IF
   END FOR
   Choose the particle having the best fitness value as the g_best_d
   FOR each particle i
       FOR each dimension d
              Calculate velocity according to the equation
              v_id(k+1)= w v_id(k)+c_1 rand_1(p_id-x_id)+c_2 rand_2 (p_gd-x_id)
              Update particle position according to the equation
            x_id(k+1)= x_id(k)+v_id(k+1)
       END FOR
   END FOR
k=k+1
WHILE maximum iterations or minimum error criteria are not attained
```

Source Code:

util.py

```
import math
import random
import matplotlib.pyplot as plt

class City:
   def __init__(self, x, y):
      self.x = x
      self.y = y
```

```
    def distance(self, city):
        return math.hypot(self.x - city.x, self.y - city.y)

    def __repr__(self):
        return f"({self.x}, {self.y})"

def read_cities(size):
    cities = []
    with open(f'test_data/cities_{size}.data', 'r') as handle:
        lines = handle.readlines()
        for line in lines:
            z, x, y = map(float, line.split())
            cities.append(City(x, y))
    return cities

def write_cities_and_return_them(size):
    cities = generate_cities(size)
    with open(f'test_data/cities_{size}.data', 'w+') as handle:
        for city in cities:
            handle.write(f'{city.x} {city.y}\n')
    return cities

def generate_cities(size):
    return [City(x=int(random.random() * 1000), y=int(random.random() * 1000)) for _ in
range(size)]

def path_cost(route):
    return sum([city.distance(route[index - 1]) for index, city in enumerate(route)])

def visualize_tsp(title, cities):
    fig = plt.figure()
    fig.suptitle(title)
    x_list, y_list = [], []
    for city in cities:
        x_list.append(city.x)
        y_list.append(city.y)
    x_list.append(cities[0].x)
    y_list.append(cities[0].y)
    plt.plot(x_list, y_list, 'ro')
```

```
    plt.plot(x_list, y_list, 'g')
    plt.show(block=True)
```

pso.py

```
import random
import math
import matplotlib.pyplot as plt
from util import City, read_cities, write_cities_and_return_them, generate_cities, path_cost

class Particle:
    def __init__(self, route, cost=None):
        self.route = route
        self.pbest = route
        self.current_cost = cost if cost else self.path_cost()
        self.pbest_cost = cost if cost else self.path_cost()
        self.velocity = []

    def clear_velocity(self):
        self.velocity.clear()

    def update_costs_and_pbest(self):
        self.current_cost = self.path_cost()
        if self.current_cost < self.pbest_cost:
            self.pbest = self.route
            self.pbest_cost = self.current_cost

    def path_cost(self):
        return path_cost(self.route)

class PSO:
    def __init__(self, iterations, population_size, gbest_probability=1.0, pbest_probability=1.0,
cities=None):
        self.cities = cities
        self.gbest = None
        self.gcost_iter = []
        self.iterations = iterations
        self.population_size = population_size
        self.particles = []
        self.gbest_probability = gbest_probability
```

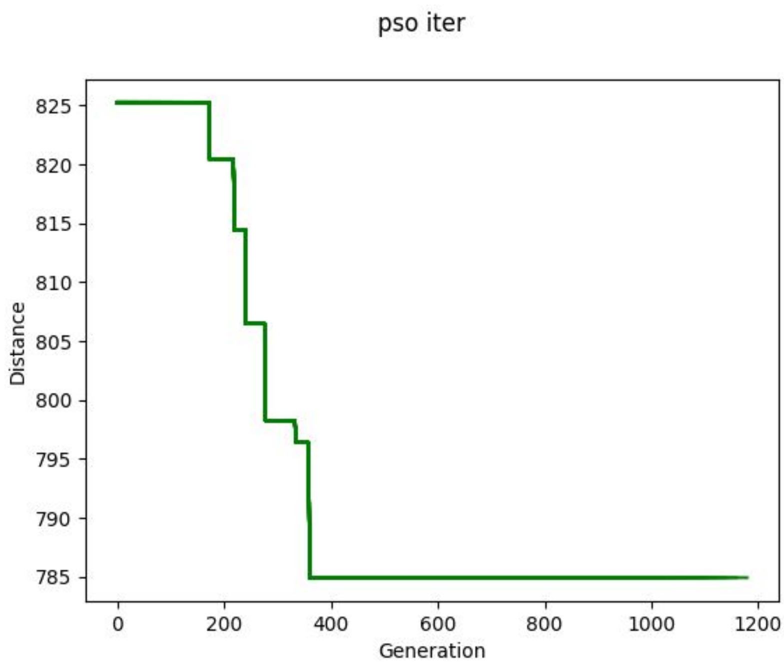
```
        self.pbest_probability = pbest_probability

        solutions = self.initial_population()
        self.particles = [Particle(route=solution) for solution in solutions]

    def random_route(self):
        return random.sample(self.cities, len(self.cities))

    def initial_population(self):
        random_population = [self.random_route() for _ in range(self.population_size - 1)]
        greedy_population = [self.greedy_route(0)]
        return [*random_population, *greedy_population]
        # return [*random_population]

    def greedy_route(self, start_index):
        unvisited = self.cities[:]
        del unvisited[start_index]
        route = [self.cities[start_index]]
        while len(unvisited):
            index, nearest_city = min(enumerate(unvisited), key=lambda item:
item[1].distance(route[-1]))
            route.append(nearest_city)
            del unvisited[index]
        return route

    def run(self):
        self.gbest = min(self.particles, key=lambda p: p.pbest_cost)
        print(f"initial cost is {self.gbest.pbest_cost}")
        plt.ion()
        plt.draw()
        for t in range(self.iterations):
            self.gbest = min(self.particles, key=lambda p: p.pbest_cost)
            if t % 20 == 0:
                plt.figure(0)
                plt.plot(pso.gcost_iter, 'g')
                plt.ylabel('Distance')
                plt.xlabel('Generation')
                fig = plt.figure(0)
                fig.suptitle('pso iter')
```

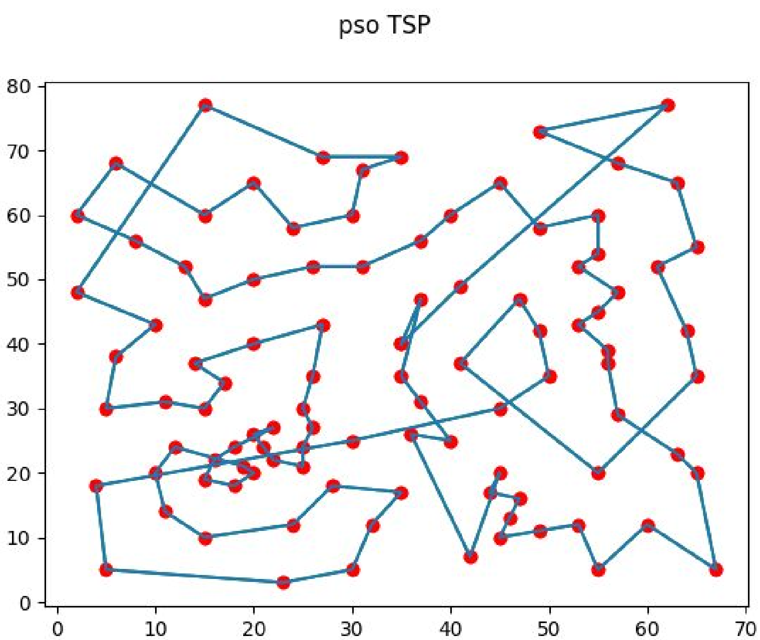
```
        x_list, y_list = [], []
        for city in self.gbest.pbest:
            x_list.append(city.x)
            y_list.append(city.y)
        x_list.append(pso.gbest.pbest[0].x)
        y_list.append(pso.gbest.pbest[0].y)
        fig = plt.figure(1)
        fig.clear()
        fig.suptitle(f'pso TSP iter {t}')
        plt.plot(x_list, y_list, 'ro')
        plt.plot(x_list, y_list, 'g')
        plt.draw()
        plt.pause(.001)
    self.gcost_iter.append(self.gbest.pbest_cost)

    for particle in self.particles:
        particle.clear_velocity()
        temp_velocity = []
        gbest = self.gbest.pbest[:]
        new_route = particle.route[:]
        for i in range(len(self.cities)):
            if new_route[i] != particle.pbest[i]:
                swap = (i, particle.pbest.index(new_route[i]), self.pbest_probability)
                temp_velocity.append(swap)
                new_route[swap[0]], new_route[swap[1]] = \
                    new_route[swap[1]], new_route[swap[0]]
        for i in range(len(self.cities)):
            if new_route[i] != gbest[i]:
                swap = (i, gbest.index(new_route[i]), self.gbest_probability)
                temp_velocity.append(swap)
                gbest[swap[0]], gbest[swap[1]] = gbest[swap[1]], gbest[swap[0]]
        particle.velocity = temp_velocity

        for swap in temp_velocity:
            if random.random() <= swap[2]:
                new_route[swap[0]], new_route[swap[1]] = \
                    new_route[swap[1]], new_route[swap[0]]

        particle.route = new_route
```

```
            particle.update_costs_and_pbest()

if __name__ == "__main__":
    cities = read_cities(101)
    pso = PSO(iterations=1200, population_size=300, pbest_probability=0.9,
gbest_probability=0.02, cities=cities)
    pso.run()
    print(f'cost: {pso.gbest.pbest_cost}\t| gbest: {pso.gbest.pbest}')

    x_list, y_list = [], []
    for city in pso.gbest.pbest:
        x_list.append(city.x)
        y_list.append(city.y)
    x_list.append(pso.gbest.pbest[0].x)
    y_list.append(pso.gbest.pbest[0].y)
    fig = plt.figure(1)
    fig.suptitle('pso TSP')

    plt.plot(x_list, y_list, 'ro')
    plt.plot(x_list, y_list)
    plt.show(block=True)
```

Output:

```
C:\Users\Admin\Desktop\webd\projects\Lab Programs\Swarm and Evolutionary Computing>python pso.py
initial cost is 825.2423227277445

cost: 784.923608033959

gbest:
 [(35.0, 40.0), (37.0, 47.0), (35.0, 35.0), (37.0, 31.0), (40.0, 25.0), (36.0, 26.0), (42.0, 7.0), (45.0, 2
0.0), (44.0, 17.0), (47.0, 16.0), (46.0, 13.0), (45.0, 10.0), (49.0, 11.0), (53.0, 12.0), (55.0, 5.0), (60.
0, 12.0), (67.0, 5.0), (65.0, 20.0), (63.0, 23.0), (57.0, 29.0), (56.0, 37.0), (56.0, 39.0), (53.0, 43.0),
(55.0, 45.0), (57.0, 48.0), (53.0, 52.0), (55.0, 54.0), (55.0, 60.0), (49.0, 58.0), (45.0, 65.0), (40.0, 60
.0), (37.0, 56.0), (31.0, 52.0), (26.0, 52.0), (20.0, 50.0), (15.0, 47.0), (13.0, 52.0), (8.0, 56.0), (2.0,
 60.0), (6.0, 68.0), (15.0, 60.0), (20.0, 65.0), (24.0, 58.0), (30.0, 60.0), (31.0, 67.0), (35.0, 69.0), (2
7.0, 69.0), (15.0, 77.0), (2.0, 48.0), (10.0, 43.0), (6.0, 38.0), (5.0, 30.0), (11.0, 31.0), (15.0, 30.0),
(17.0, 34.0), (14.0, 37.0), (20.0, 40.0), (27.0, 43.0), (26.0, 35.0), (25.0, 30.0), (26.0, 27.0), (25.0, 24
.0), (25.0, 21.0), (22.0, 22.0), (21.0, 24.0), (20.0, 26.0), (22.0, 27.0), (18.0, 24.0), (16.0, 22.0), (15.
0, 19.0), (18.0, 18.0), (20.0, 20.0), (19.0, 21.0), (12.0, 24.0), (10.0, 20.0), (11.0, 14.0), (15.0, 10.0),
 (24.0, 12.0), (28.0, 18.0), (35.0, 17.0), (32.0, 12.0), (30.0, 5.0), (23.0, 3.0), (5.0, 5.0), (4.0, 18.0),
 (30.0, 25.0), (45.0, 30.0), (50.0, 35.0), (49.0, 42.0), (47.0, 47.0), (41.0, 37.0), (55.0, 20.0), (65.0, 3
5.0), (64.0, 42.0), (61.0, 52.0), (65.0, 55.0), (63.0, 65.0), (57.0, 68.0), (49.0, 73.0), (62.0, 77.0), (41
.0, 49.0)]
```





Finding and Learnings:

We have successfully implemented the Particle Search Optimization Algorithm on Travelling salesman problem in python .PSO does not use the gradient of the problem being optimized, which means PSO does not require that the optimization problem be differentiable as is required by classic optimization methods.

Experiment 2

Aim: Write a program to implement Cuckoo Search Optimization algorithm. .

Theory:

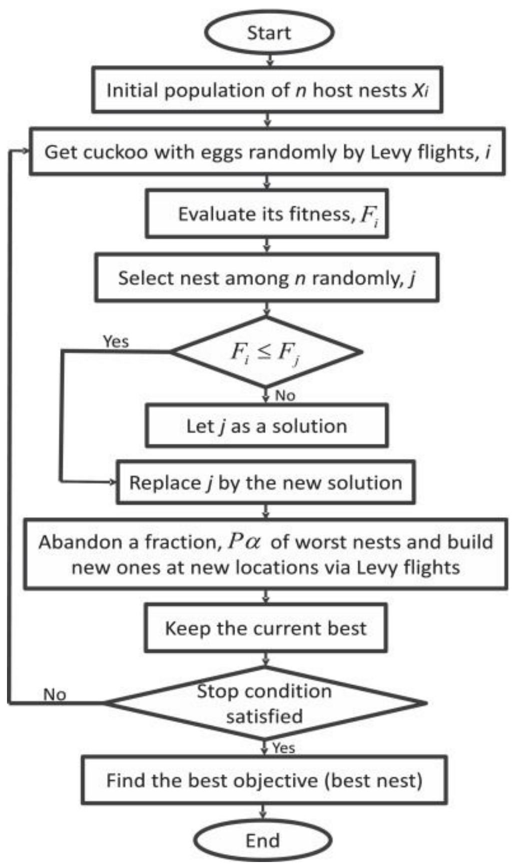
Cuckoo Search (CS) is a meta-heuristic algorithm based on the breeding pattern of certain species of cuckoo birds. In our research, we have implemented CS for the NP-hard optimization problem, the Traveling Salesman Problem (TSP). CS is based on three idealized rules:

1. Each cuckoo lays one egg at a time, and dumps its egg in a randomly chosen nest;
2. The best nests with high quality of eggs will carry over to the next generation;
3. The number of available hosts nests is fixed, and the egg laid by a cuckoo is discovered by the host bird with a probability pa $\in$ (0,1).

Discovering operate on some set of worst nests, and discovered solutions dumped from farther calculations.The algorithm can be extended to more complicated cases in which each nest has multiple eggs representing a set of solutions.

Algorithm:

Input: Input is the distance between cities given in form of a matrix. (distanceMatrix in code).



Source Code:

cuckoo.py

```
from random import uniform
from random import randint
import math
distanceMatrix = [
                [0, 29, 20, 21, 16, 31, 100, 12, 4, 31, 18],
                [29, 0, 15, 29, 28, 40, 72, 21, 29, 41, 12],
                [20, 15, 0, 15, 14, 25, 81, 9, 23, 27, 13],
                [21, 29, 15, 0, 4, 12, 92, 12, 25, 13, 25],
                [16, 28, 14, 4, 0, 16, 94, 9, 20, 16, 22],
                [31, 40, 25, 12, 16, 0, 95, 24, 36, 3, 37],
                [100, 72, 81, 92, 94, 95, 0, 90, 101, 99, 84],
                [12, 21, 9, 12, 9, 24, 90, 0, 15, 25, 13],
                [4, 29, 23, 25, 20, 36, 101, 15, 0, 35, 18],
                [31, 41, 27, 13, 16, 3, 99, 25, 35, 0, 38],
                [18, 12, 13, 25, 22, 37, 84, 13, 18, 38, 0]             ]

def levyFlight(u):
    return math.pow(u, -1.0/3.0)

def randF():
    return uniform(0.0001, 0.9999)

def calculateDistance(path):
    index = path[0]
    distance = 0
    for nextIndex in path[1:]:
        distance += distanceMatrix[index][nextIndex]
        index = nextIndex
        return distance+distanceMatrix[path[-1]][path[0]]

def swap(sequence, i, j):
    temp = sequence[i]
    sequence[i] = sequence[j]
    sequence[j] = temp
```

```
def twoOptMove(nest, a, c):
   nest = nest[0][:]
   swap(nest, a, c)
   return (nest, calculateDistance(nest))

def doubleBridgeMove(nest, a, b, c, d):
   nest = nest[0][:]
   swap(nest, a, b)
   swap(nest, b, d)
   return (nest, calculateDistance(nest))

numNests = 10
pa = int(0.2*numNests)
pc = int(0.6*numNests)
maxGen = 50
n = len(distanceMatrix)
nests = []
initPath = list(range(0, n))
index = 0
for i in range(numNests):
   if index == n-1:
      index = 0
   swap(initPath, index, index+1)
   index += 1
   nests.append((initPath[:], calculateDistance(initPath)))
nests.sort(key=lambda tup: tup[1])
for t in range(maxGen):
   cuckooNest = nests[randint(0, pc)]
   if(levyFlight(randF()) > 2):
      cuckooNest = doubleBridgeMove(cuckooNest, randint(0, n-1), randint(0, n-1), randint(0,
                        n-1), randint(0, n-1))
   else:
      cuckooNest = twoOptMove(cuckooNest, randint(0, n-1), randint(0, n-1))
   randomNestIndex = randint(0, numNests-1)
   if(nests[randomNestIndex][1] > cuckooNest[1]):
     nests[randomNestIndex] = cuckooNest
   for i in range(numNests-pa, numNests):
      nests[i] = twoOptMove(nests[i], randint(0, n-1), randint(0, n-1))
   nests.sort(key=lambda tup: tup[1])
```

```
    if (t+1) % 5 == 0:
        print("\nGEN#", t+1, ": ", nests[0])
print("\nCUCKOO's SOLUTION", end=': ')
print(nests[0])
```

Output:

```
kunal@DESKTOP-AITAEP7:/mnt/c/Users/Admin/Desktop/webd/projects/Lab Programs/Swarm and Evolutionary Computing$ python3 cuckoo.py

GEN# 5 :  ([1, 2, 0, 3, 4, 5, 6, 7, 8, 9, 10], 27)

GEN# 10 :  ([1, 2, 0, 3, 4, 5, 6, 7, 8, 9, 10], 27)

GEN# 15 :  ([4, 3, 8, 1, 9, 6, 5, 0, 7, 10, 2], 18)

GEN# 20 :  ([4, 3, 8, 1, 9, 6, 5, 0, 7, 10, 2], 18)

GEN# 25 :  ([4, 3, 8, 1, 9, 6, 5, 0, 7, 10, 2], 18)

GEN# 30 :  ([4, 3, 8, 1, 9, 6, 5, 0, 7, 10, 2], 18)

GEN# 35 :  ([4, 3, 8, 1, 9, 6, 5, 0, 7, 10, 2], 18)

GEN# 40 :  ([4, 3, 8, 1, 9, 6, 5, 0, 7, 10, 2], 18)

GEN# 45 :  ([4, 3, 8, 1, 9, 6, 5, 0, 7, 10, 2], 18)

GEN# 50 :  ([4, 7, 9, 2, 0, 6, 1, 8, 5, 10, 3], 13)

CUCKOO's SOLUTION: ([4, 7, 9, 2, 0, 6, 1, 8, 5, 10, 3], 13)
```

Finding and Learnings:

We have successfully implemented cuckoo search algorithm technique in python. The optimal solution was calculated using a Naïve brute force approach which has a complexity of (n!).An important advantage of this algorithm is its simplicity. In fact, compared with other population- or agent-based metaheuristic algorithms such as particle swarm optimization and harmony search, there is essentially only a single parameter pa in Cuckoo Search (apart from the population size n). Therefore, it is very easy to implement

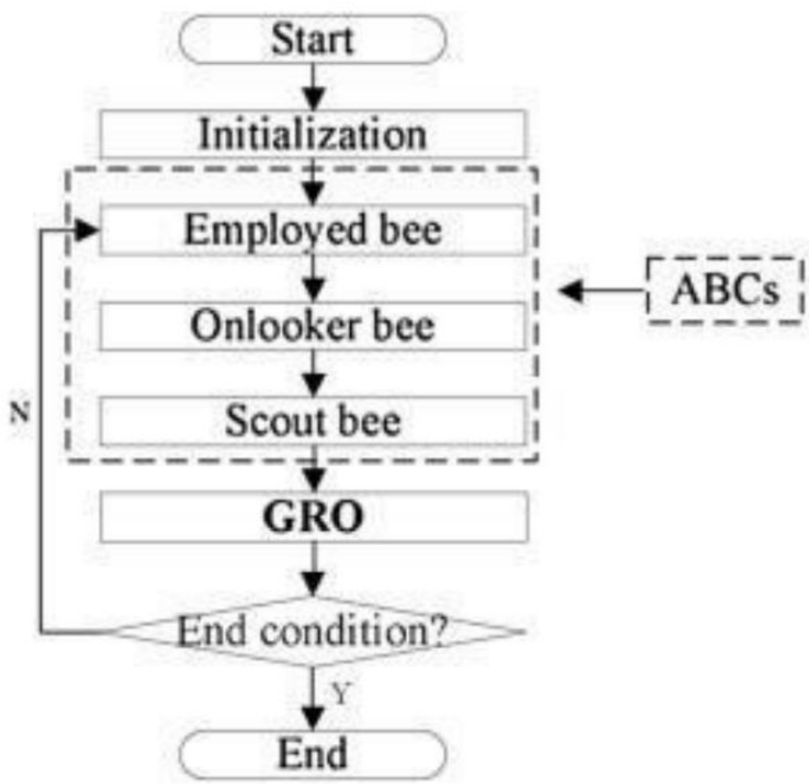
Experiment 3

Aim: Write a program to implement Artificial Bee Colony (ABC) optimization algorithm.

Theory:

In the Artificial Bee Algorithm model, the colony consists of three groups of bees: employed bees, onlookers and scouts. Scouts perform random searches, employed bees collect previously found food and onlookers watch the dances of employed bees and choose food sources depending on dances. Onlookers and scouts are called non-working bees. Communication between bees is based on dances. Before a bee starts to collect food it watches dances of other bees. A dance is the way bees describe where food is.

Working and non-working bees search for rich food sources near their hive. A working bee keeps the information about a food source and shares it with onlookers. Working bees whose solutions can't be improved after a definite number of attempts become scouts and their solutions are not used after that. The number of food sources represents the number of solutions in the population. The position of a food source represents a possible solution to the optimization problem and the nectar amount of a food source corresponds to the quality (fitness) of the associated solution.



Algorithm:

1. BEGIN
2. Initialize the population
3. Find current best agent for the initial iteration
4. Calculate the number of scouts, onlookers and employed bees
5. SET global best to current best
6. FOR iterator = 0 : iteration
    a. evaluate fitness for each agent
    b. sort fitness in ascending order and get best agents
    c. from best agents list select agents from a to c
    d. Create new bees which will fly to the best solution
    e. Evaluate current best agent
    f. IF function(current best) < function (global best)
        i. global best = current best
    g. END IF
7. END FOR
8. Save global best

Source Code:

Artificialbeecolony.py

```
import random
from collections import Iterable
class ABC:
    def __init__(self, objective_function, sn, bound, trial_limit, maximum_cycle_number):
        self.objective_function = objective_function
        self.bound = bound
        self.maximum_cycle_number = maximum_cycle_number
        self.trial_limit = trial_limit
        self.trial = [0] * sn

        self.solutions = \
            [
                [random.uniform(-bound, bound) for arg in
range(self.objective_function.__code__.co_argcount)]
                for f in range(sn)
            ]
```

```
        self._eval_solutions()
        for c in range(self.maximum_cycle_number):
            self._employed_phase()
            self._eval_prob()
            self._onlookers_phase()

    @staticmethod
    def _fitness_function(function_f):
        if function_f >= 0:
            return 1 / (1 + function_f)
        else:
            return 1 + function_f

    def _eval_prob(self):
        sum_fit = sum(self.fit)
        self.prob = [self.fit[i] / sum_fit for i in range(len(self.solutions))]

    def eval_solution(self, solution):
        """Calculates objective_function and fitness_function values"""
        if isinstance(solution, int):
            obj_val = self.objective_function(self.solutions[solution])
        elif isinstance(solution, Iterable):
            obj_val = self.objective_function(*solution)
        else:
            raise Exception("Expected solution to be int or Iterable, instead found ", type(solution))
        fit_val = ABC._fitness_function(obj_val)
        return obj_val, fit_val

    def _eval_solutions(self):
        self.function = list(map(lambda args: self.objective_function(*args), self.solutions))
        self.fit = list(map(ABC._fitness_function, self.function))

    def best_solution(self):
        i = self.fit.index(max(self.fit))
        return self.solution_detail(i)

    def worst_solution(self):
        i = self.fit.index(min(self.fit))
        return self.solution_detail(i)
```

```
def solution_detail(self, i):
    return {"solution": self.solutions[i], "function": self.function[i], "fitness": self.fit[i],
            "trial": self.trial[i]}

def _new_v_solution(self, i):
    k = random.choice([k for k in range(len(self.solutions)) if k != i])
    j = random.randrange(self.objective_function.__code__.co_argcount)
    xkj = self.solutions[k][j]
    xij = self.solutions[i][j]
    phi = random.uniform(-1, 1)
    new_xj = xij + phi * (xij - xkj)
    new_xj = self._bound(new_xj)
    new_solution = self.solutions[i][:]
    new_solution[j] = new_xj
    return new_solution

def _new_x_solution(self, i):
    # Randomly select a variable j
    j = random.randrange(self.objective_function.__code__.co_argcount)
    # Generate new solution new_x and bound it
    xij = self.solutions[i][j]
    r = random.uniform(0, 1)
    new_xj = -self.bound + r * (self.bound - (-self.bound))
    new_xj = self._bound(new_xj)
    new_solution = self.solutions[i][:]
    new_solution[j] = new_xj
    return new_solution

def _bound(self, value):
    if value >= self.bound:
        return self.bound
    elif value <= -self.bound:
        return -self.bound
    return value

def _accept_solution(self, i, new_solution, new_obj_val=None, new_fit_val=None):
    if not new_obj_val:
        new_fit_val = ABC._fitness_function(new_obj_val)
```

```
            if not new_fit_val:
                new_obj_val, new_fit_val = self.eval_solution(new_solution)
        self.solutions[i] = new_solution
        self.fit[i] = new_fit_val
        self.function[i] = new_obj_val
        self.trial[i] = 0

    def _employed_phase(self):
        for i in range(len(self.solutions)):
            new_solution = self._new_v_solution(i)
            self._general_phase(new_solution, i)

    def _onlookers_phase(self):
        for n in range(len(self.solutions)):
            i = random.choices(range(len(self.solutions)), weights=self.prob)[0]
            new_solution = self._new_v_solution(i)
            self._general_phase(new_solution, i)

    def _scout_phase(self, i):
        new_solution = self._new_x_solution(i)
        self._general_phase(new_solution, i)

    def _general_phase(self, new_solution, i=None):
        new_obj_val, new_fit_val = self.eval_solution(new_solution)

        if new_fit_val > self.fit[i]:
            self._accept_solution(i, new_solution, new_obj_val, new_fit_val)
        else:
            self.trial[i] += 1
            if self.trial[i] >= self.trial_limit:
                self.trial[i] = 0
                self._scout_phase(i)
```

main.py

```
from Artificialbeecolony import ABC
import math
Bukin_function_N_6 = lambda x, y: 100 * (math.sqrt(abs(y - 0.01 * x ** 2)) + 0.01 * abs(x +
10))
```

Ackley_function = lambda x, y: -20 * math.exp(-.02 * math.sqrt(0.5 * (x ** 2 + y ** 2))) - math.exp(0.5 * (math.cos(2 * math.pi * x) + math.cos(2 * math.pi * y))) + math.e + 20

sphere_function = lambda x1, x2, x3, x4, x5, x6: x1 ** 2 + x2 ** 2 + x3 ** 2 + x4 ** 2 + x5 ** 2 + x6 ** 2

SN = 10
limit = 50
MCN = 1000
bound = 40
result = ABC(Ackley_function, SN, bound, limit, MCN)
print(result.best_solution())

Output:

Ackley_function

```
kunal@DESKTOP-AITAEP7:/mnt/c/Users/Admin/Desktop/Artificial_Bee_Colony_Algo
rithm$ python3 main.py
{'solution': [3.150455108665474e-16, 3.821856448668951e-15], 'function': 0.0
, 'fitness': 1.0, 'trial': 21}
```

Bukin_function_N_6

```
kunal@DESKTOP-AITAEP7:/mnt/c/Users/Admin/Desktop/Artificial_Bee_Colony_Algo
rithm$ python3 main.py
{'solution': [-9.395428521459555, 0.8825940531728784], 'function': 1.815842
9001402223, 'fitness': 0.35513344865588997, 'trial': 25}
```

sphere_function

```
kunal@DESKTOP-AITAEP7:/mnt/c/Users/Admin/Desktop/Artificial_Bee_Colony_Algo
rithm$ python3 main.py
{'solution': [5.293217762757041e-09, -3.7870524915479095e-09, 4.50093245353
21514e-10, 4.761543663105327e-09, 4.397164296132926e-09, 2.8092130569673537
e-09], 'function': 9.24615346894999e-17, 'fitness': 1.0, 'trial': 8}
```

Finding and Learnings:

We have successfully implemented the Artificial Bee colony Algorithm in python. The ABC(Artificial Bee Colony) model consists of four phases that are accomplished sequentially, Initialization Phase, Exploitation Phase, Refinement Phase and Exploration Phase where scout bees are sent out to unexplored regions of the search domain.

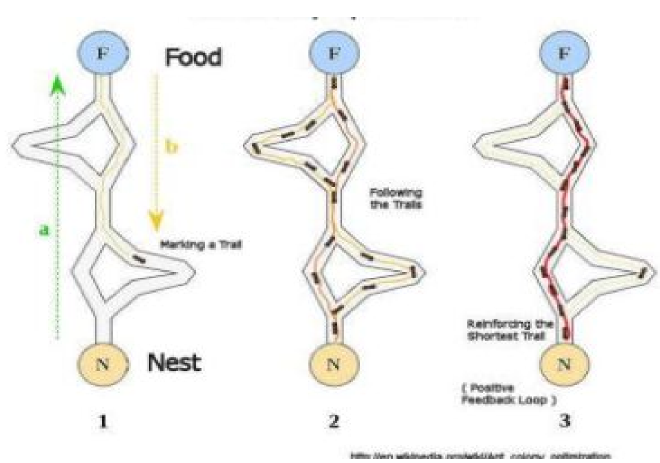
Experiment 4

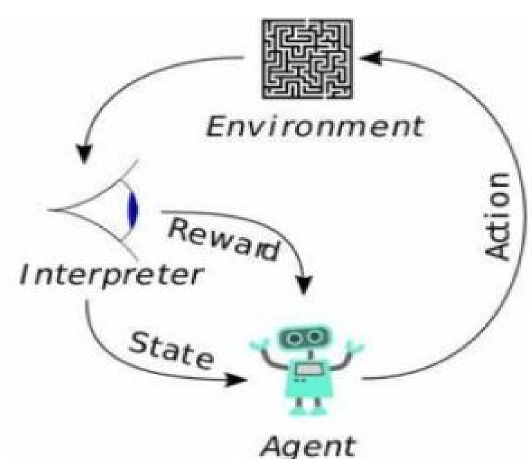
Aim: Write a program to implement Ant Colony optimization (ACO) algorithm.

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:

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)]  # probabilities for moving to a node in the
next step
        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
```

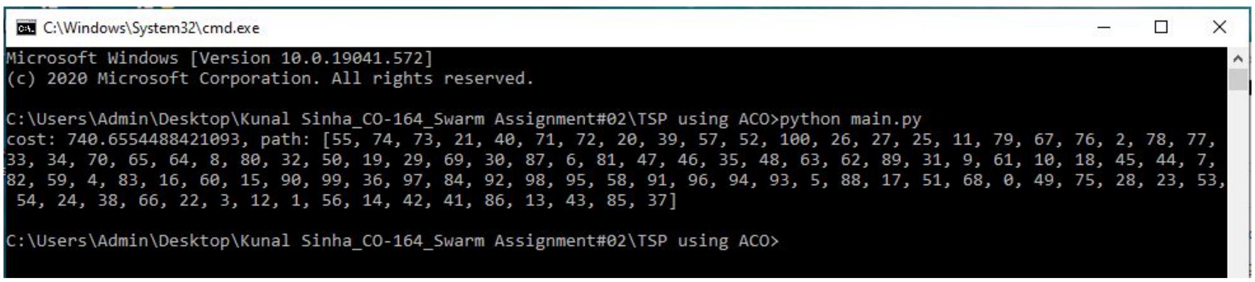
```
def plot(points, path: list):
    x = []
    y = []
    for point in points:
        x.append(point[0])
        y.append(point[1])
    y = list(map(operator.sub, [max(y) for i in range(len(points))], y))
    plt.plot(x, y, 'co')

    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()
```

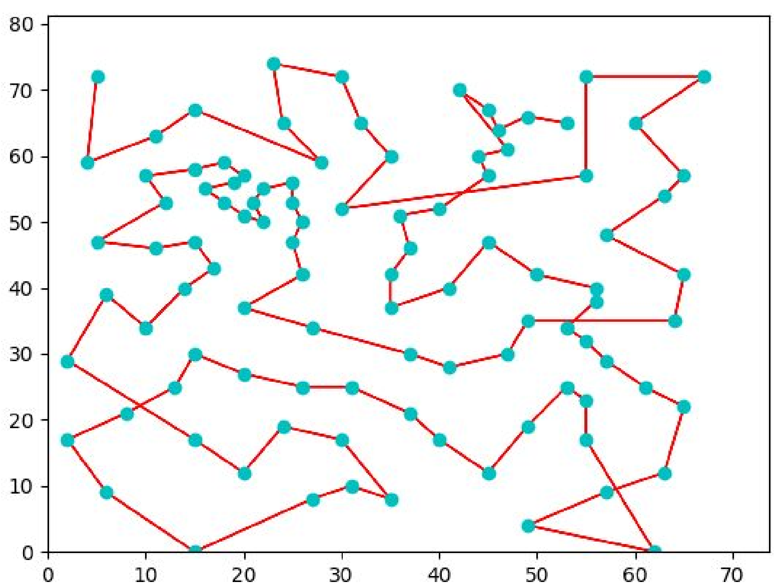
main.py

```
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):
```

```python
            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:





Finding and Learnings:

We have successfully implemented the ant colony optimization technique 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.

Experiment 5

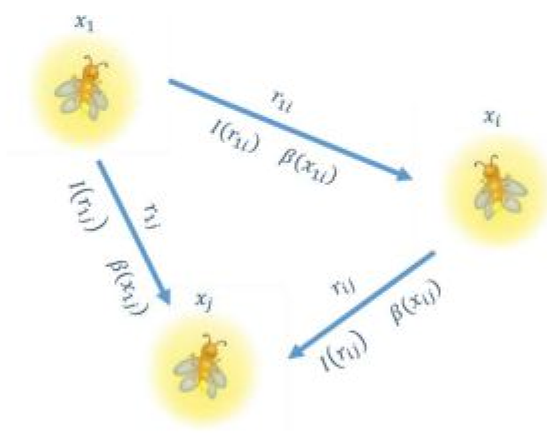
Aim: Write a program to implement Firefly algorithm (FA).

Theory:

Most species of fireflies are able to glow producing short flashes. It is considered that the main function of flashes is to attract fireflies of the opposite gender and potential prey. Besides, a signal flash can communicate to a predator that a firefly has a bitter taste.

The Firefly Algorithm is based on two important things: the change in light intensity and attractiveness. For simplicity, it is assumed that the attractiveness of a firefly is defined by its brightness which is connected with the objective function.
The algorithm utilizes the following firefly behaviour model:

- All fireflies are able to attract each other independently of their gender;
- A firefly attractiveness for other individuals is proportional to its brightness.
- Less attractive fireflies move in the direction of the most attractive one.
- As the distance between two fireflies increases, the visible brightness of the given firefly for the other decreases.
- If a firefly sees no firefly that is brighter than itself, it moves randomly.



Working of firefly algorithm meta-heuristic function

Algorithm:

1. Objective function f(x), x=(x1, x2, ... , xd)T
2. Initialize a population of fireflies xi(i = 1, 2, ... , n)
3. Define light absorption coefficient gamma
4. WHILE count < Maximum Generations
    a. FOR i = 1 : n (all n fireflies)
        i. FOR j = 1 : i
        ii. Light intensity Ii at xi is determined by f(xi)
        iii. IF Ii > Ij
            1. Move firefly i towards j in all d dimensions
            2. ELSE
            3. Move firefly i randomly
            4. END IF
            5. Attractiveness changes with distance r via exp[-γ r2]
        iv. Determine new solutions and revise light intensity
        v. END FOR j
    b. END FOR i
    c. Rank the fireflies according to light intensity and find the current best
5. END WHILE

Source Code:

firefly.py

```
from math import exp
import numpy as np
from . import intelligence

class fa(intelligence.sw):
    """  Firefly Algorithm """

    def __init__(self, n, function, lb, ub, dimension, iteration, csi=1, psi=1,
                 alpha0=1, alpha1=0.1, norm0=0, norm1=0.1):
        """
        :param n: number of agents
        :param function: test function
```

```
:param lb: lower limits for plot axes
:param ub: upper limits for plot axes
:param dimension: space dimension
:param iteration: number of iterations
:param csi: mutual attraction
:param psi: light absorption coefficient of the medium
:param alpha0: initial value of the free randomization parameter alpha
:param alpha1: final value of the free randomization parameter alpha
:param norm0: first parameter for a normal (Gaussian) distribution
:param norm1: second parameter for a normal (Gaussian) distribution
"""
super(fa, self).__init__()
self.__agents = np.random.uniform(lb, ub, (n, dimension))
self._points(self.__agents)

Pbest = self.__agents[np.array([function(x)
                        for x in self.__agents]).argmin()]
Gbest = Pbest

for t in range(iteration):
    alpha = alpha1 + (alpha0 - alpha1) * exp(-t)
    for i in range(n):
        fitness = [function(x) for x in self.__agents]
        for j in range(n):
            if fitness[i] > fitness[j]:
                self.__move(i, j, t, csi, psi, alpha, dimension,
                            norm0, norm1)
            else:
                self.__agents[i] += np.random.normal(norm0, norm1,
                                                     dimension)

    self.__agents = np.clip(self.__agents, lb, ub)
    self._points(self.__agents)
    Pbest = self.__agents[
        np.array([function(x) for x in self.__agents]).argmin()]
    if function(Pbest) < function(Gbest):
        Gbest = Pbest
self._set_Gbest(Gbest)
```

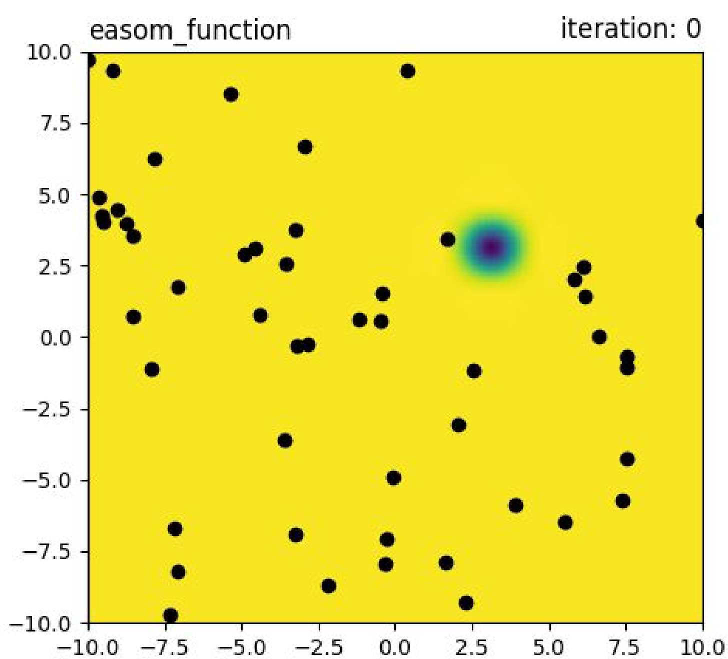
```
    def __move(self, i, j, t, csi, psi, alpha, dimension, norm0, norm1):
        r = np.linalg.norm(self.__agents[i] - self.__agents[j])
        beta = csi / (1 + psi * r ** 2)

        self.__agents[i] = self.__agents[j] + beta * ( self.__agents[i] - self.__agents[j]) + alpha *
                               exp(-t) * \ np.random.normal(norm0, norm1, dimension)
```

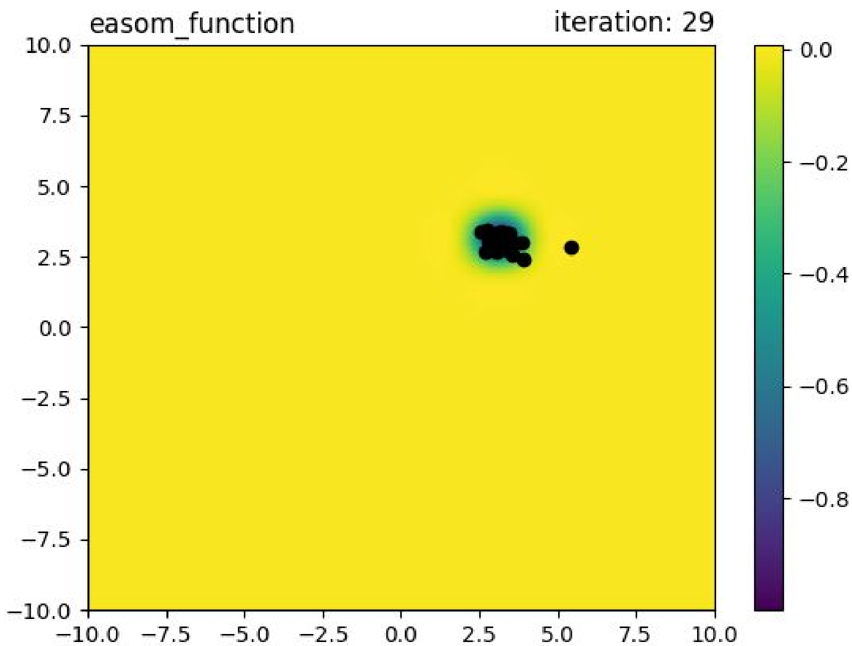
main.py

```
import firerfly.py as fa
import matplotlib.pyplot as plt

def easom_function(x):
    return -cos(x[0])*cos(x[1])*exp(-(x[0] - pi)**2 - (x[1] - pi)**2)

alh = fa(50, easom_function, -10, 10, 2, 30,1,1,1,0.1,0,0.1)
plt(alh.get_agents(),easom_function, -10, 10)
```

Output:

For 30 iterations





Finding and Learnings:

We have successfully implemented the Firefly Algorithm in python. The "firefly algorithm" (FFA) is a modern metaheuristic algorithm, inspired by the behavior of fireflies. This algorithm and its variants have been successfully applied to many continuous optimization problems.

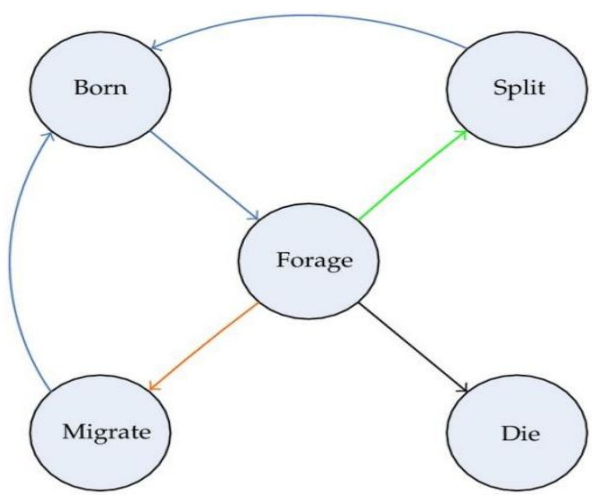
Experiment 6

Aim: Write a program to implement the Bacterial Foraging algorithm.

Theory:

The Bacterial Foraging Optimization, is inspired by the social foraging behavior of E.coli. During foraging of the real bacteria, locomotion is achieved by a set of tensile flagella. Tumble or swim, are two basic operations performed by a bacterium at the time of foraging. When they rotate the flagella in the clockwise direction, each flagellum pulls on the cell.In the above-mentioned algorithm the bacteria undergoes chemotaxis, where they like to move towards a nutrient gradient and avoid a noxious environment. Generally the bacteria move for a longer distance in a friendly environment.

When they get food in sufficient quantities, they are increased in length and in presence of suitable temperature they break in the middle to from an exact replica of itself. Due to the occurrence of sudden environmental changes or attack, the chemotactic progress may be destroyed and a group of bacteria may move to some other places or some other may be introduced in the swarm of concern. This constitutes the event of elimination-dispersal in the real bacterial population, where all the bacteria in a region are killed or a group is dispersed into a new part of the environment.



Working of Bacterial Foraging algorithm

Bacterial Foraging Optimization has four main steps:

- Chemotaxis
- Reproduction
- Elimination
- Dispersal

Algorithm:

1. Initialize randomly the bacteria foraging optimization population
2. Calculate the fitness of each agent
3. Set global best agent to best agent
4. FOR number of iterations
    a. FOR number of chemotactic steps
        i. FOR each search agent
            1. Move agent to the random direction
            2. Calculate the fitness of the moved agent
            3. FOR swimming length
                a. IF current fitness is better than previous
                    i. Move agent to the same direction
                b. ELSE
                    i. Move agent to the random direction
        ii. Calculate the fitness of each agent
    b. END FOR
    c. Compute and sort sum of fitness function of all chemotactic loops (health of agent)
    d. Let live and split only half of the population according to their health
    e. IF not the last iteration
        i. FOR each search agent
            1. With some probability replace agent with new random generated
    f. END IF
    g. Update the best search agent
5. Calculate the fitness of each agent

Source Code:

bacteria.py

```
import numpy as np
from random import random
from . import intelligence

class bfo(intelligence.sw):
    def __init__(self, n, function, lb, ub, dimension, iteration, Nc=2, Ns=12, C=0.2, Ped=1.15):
        """
n: number of agents, function: test function , lb&ub: lower and  upper limits for plot axes
dimension: space dimension , iteration: the number of iterations
Nc: number of chemotactic steps , Ns: swimming length
```

```
C: the size of step taken in the random direction specified by the tumble
Ped: elimination-dispersal probability """

    super(bfo, self).__init__()
    self.__agents = np.random.uniform(lb, ub, (n, dimension))
    self._points(self.__agents)

    n_is_even = True
    if n & 1:
        n_is_even = False

    J = np.array([function(x) for x in self.__agents])
    Pbest = self.__agents[J.argmin()]
    Gbest = Pbest

    C_list = [C - C * 0.9 * i / iteration for i in range(iteration)]
    Ped_list = [Ped - Ped * 0.5 * i / iteration for i in range(iteration)]
    J_last = J[::1]

    for t in range(iteration):
        J_chem = [J[::1]]
        for j in range(Nc):
            for i in range(n):
                dell = np.random.uniform(-1, 1, dimension)
                self.__agents[i] += C_list[t] * np.linalg.norm(dell) * dell

                for m in range(Ns):
                    if function(self.__agents[i]) < J_last[i]:
                        J_last[i] = J[i]
                        self.__agents[i] += C_list[t] * np.linalg.norm(dell) \ * dell
                    else:
                        dell = np.random.uniform(-1, 1, dimension)
                        self.__agents[i] += C_list[t] * np.linalg.norm(dell) \* dell

            J = np.array([function(x) for x in self.__agents])
            J_chem += [J]

        J_chem = np.array(J_chem)
        J_health = [(sum(J_chem[:, i]), i) for i in range(n)]
```

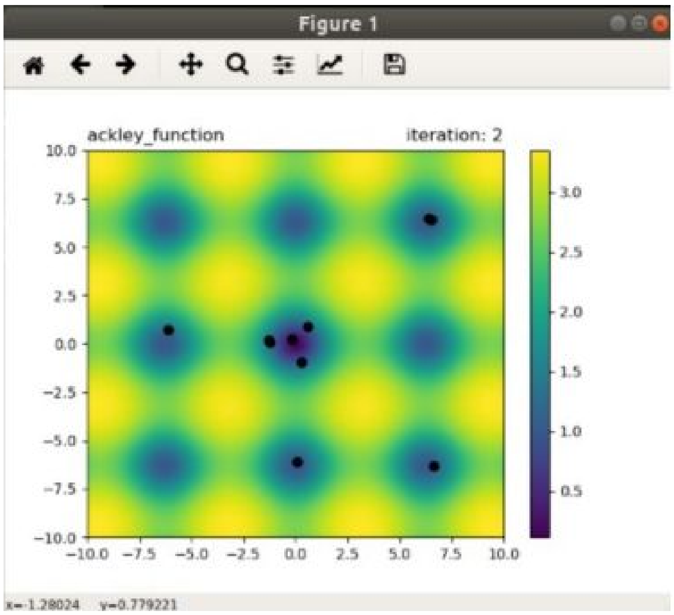
```
        J_health.sort()
        alived_agents = []
        for i in J_health:
            alived_agents += [list(self.__agents[i[1]])]

        if n_is_even:
            alived_agents = 2*alived_agents[:n//2]
            self.__agents = np.array(alived_agents)
        else:
            alived_agents = 2*alived_agents[:n//2] +\
                            [alived_agents[n//2]]
            self.__agents = np.array(alived_agents)

        if t < iteration - 2:
            for i in range(n):
                r = random()
                if r >= Ped_list[t]:
                    self.__agents[i] = np.random.uniform(lb, ub, dimension)

        J = np.array([function(x) for x in self.__agents])
        self._points(self.__agents)

        Pbest = self.__agents[J.argmin()]
        if function(Pbest) < function(Gbest):
            Gbest = Pbest
    self._set_Gbest(Gbest)
```

main.py

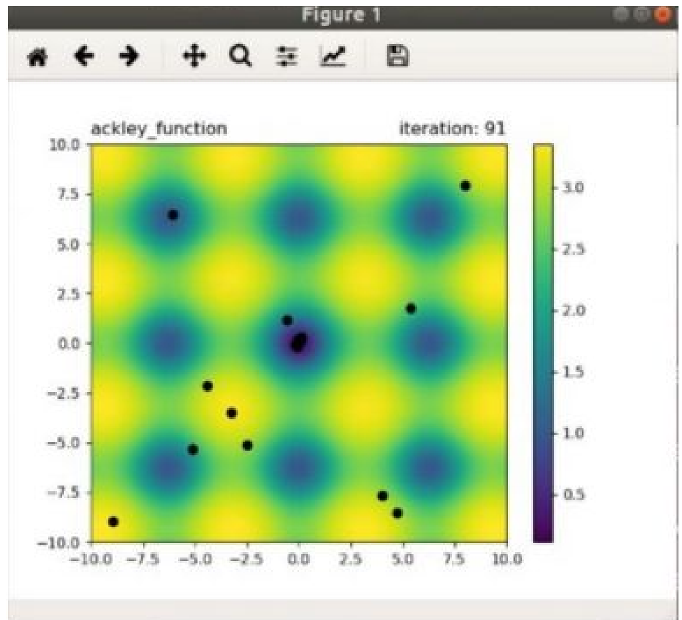
```
from math import *
import bacteria.py as bfo
import matplotlib.pyplot as plt

def ackley_function(x):
    return -exp(-sqrt(0.5*sum([i**2 for i in x]))) - \exp(0.5*sum([cos(i) for i in x])) + 1 + exp(1)

alh = bfo(50, ackely_function, -10,10, 2, 90,2, 12,0.2, 1.15)
plt(alh.get_agents(),easom_function, -10, 10)
```

Output:

For 90 iterations



Initial epoch



Final epoch

Finding and Learnings:

We have successfully implemented the Bacterial Foraging Algorithm (BFOA) in python. BFOA has been widely accepted as a global optimization algorithm of current interest for optimization and control. BFOA has already drawn the attention of researchers because of its efficiency in solving real-world optimization problems arising in several application domains.

Experiment 7

Aim: Write a program to implement the Genetic algorithm.

Theory:

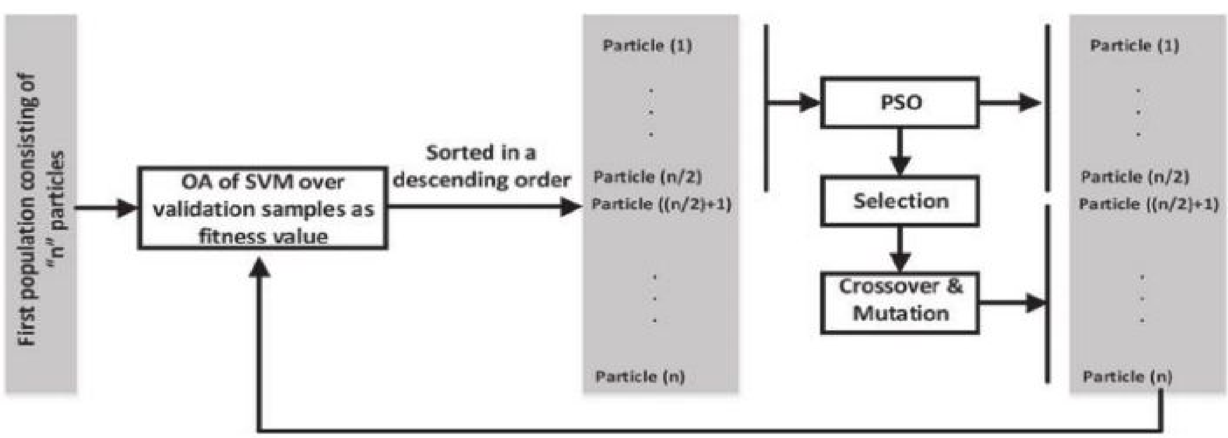
A genetic algorithm is a search heuristic that is inspired by Charles Darwin's theory of natural evolution. This algorithm reflects the process of natural selection where the fittest individuals are selected for reproduction in order to produce offspring of the next generation.

The process of natural selection starts with the selection of fittest individuals from a population. They produce offspring which inherit the characteristics of the parents and will be added to the next generation. If parents have better fitness, their offspring will be better than parents and have a better chance at surviving. This process keeps on iterating and at the end, a generation with the fittest individuals will be found. This notion can be applied for a search problem. We consider a set of solutions for a problem and select the set of best ones out of them. Five phases are considered in a genetic algorithm.

1. Initial population 2. Fitness function 3. Selection 4.Crossover 5.Mutation

Algorithm:

1. Generate the initial population
2. Compute fitness
3. REPEAT
4. Selection
5. Crossover
6. Mutation
7. Compute fitness
8. UNTIL population has converged



Source Code:

genetic.py

```
import numpy

def cal_pop_fitness(equation_inputs, pop):
    fitness = numpy.sum(pop*equation_inputs, axis=1)
    return fitness

def select_mating_pool(pop, fitness, num_parents):
    parents = numpy.empty((num_parents, pop.shape[1]))
    for parent_num in range(num_parents):
        max_fitness_idx = numpy.where(fitness == numpy.max(fitness))
        max_fitness_idx = max_fitness_idx[0][0]
        parents[parent_num, :] = pop[max_fitness_idx, :]
        fitness[max_fitness_idx] = -99999999999
    return parents

def crossover(parents, offspring_size):
    offspring = numpy.empty(offspring_size)
    crossover_point = numpy.uint8(offspring_size[1]/2)

    for k in range(offspring_size[0]):
        parent1_idx = k%parents.shape[0]
        parent2_idx = (k+1)%parents.shape[0]
        offspring[k, 0:crossover_point] = parents[parent1_idx, 0:crossover_point]
        offspring[k, crossover_point:] = parents[parent2_idx, crossover_point:]
    return offspring

def mutation(offspring_crossover, num_mutations=1):
    mutations_counter = numpy.uint8(offspring_crossover.shape[1] / num_mutations)
    for idx in range(offspring_crossover.shape[0]):
        gene_idx = mutations_counter - 1
        for mutation_num in range(num_mutations):
            random_value = numpy.random.uniform(-1.0, 1.0, 1)
            offspring_crossover[idx, gene_idx] = offspring_crossover[idx, gene_idx] +
random_value
            gene_idx = gene_idx + mutations_counter
    return offspring_crossover
```

Main.py

```
import numpy
import genetic
equation_inputs = [4,-2,3.5,5,-11,-4.7]
num_weights = len(equation_inputs)
sol_per_pop = 8
num_parents_mating = 4
pop_size = (sol_per_pop,num_weights)
new_population = numpy.random.uniform(low=-4.0, high=4.0, size=pop_size)
print(new_population)
best_outputs = []
num_generations = 10
for generation in range(num_generations):
    print("Generation : ", generation)
    fitness = genetic.cal_pop_fitness(equation_inputs, new_population)
    print("Fitness")
    print(fitness)
    best_outputs.append(numpy.max(numpy.sum(new_population*equation_inputs, axis=1)))
    print("Best result : ", numpy.max(numpy.sum(new_population*equation_inputs, axis=1)))
    parents = genetic.select_mating_pool(new_population, fitness,num_parents_mating)
    print("Parents")
    print(parents)
    offspring_crossover = genetic.crossover(parents,
                                    offspring_size=(pop_size[0]-parents.shape[0], num_weights))
    print("Crossover")
    print(offspring_crossover)
    offspring_mutation =genetic.mutation(offspring_crossover, num_mutations=2)
    print("Mutation")
    print(offspring_mutation)
    new_population[0:parents.shape[0], :] = parents
    new_population[parents.shape[0]:, :] = offspring_mutation

fitness = genetic.cal_pop_fitness(equation_inputs, new_population)
best_match_idx = numpy.where(fitness == numpy.max(fitness))
print("Best solution : ", new_population[best_match_idx, :])
print("Best solution fitness : ", fitness[best_match_idx])

import matplotlib.pyplot
matplotlib.pyplot.plot(best_outputs)
```

matplotlib.pyplot.xlabel("Iteration")
matplotlib.pyplot.ylabel("Fitness")
matplotlib.pyplot.show()

Output:

For 10 generations

Initial population

```
C:\Users\Admin\Desktop>python geneticmain.py
[[ 0.10437277 -3.06269698  2.61504437 -1.56204928 -0.77732296 -2.80023349]
 [ 1.90703394 -0.18198742  3.98895785  2.86320899  0.88686627 -2.59161741]
 [ 3.31783698  1.40325753 -3.00092359 -1.24947539  0.88036932 -3.74177034]
 [-3.57553258  3.75161393 -1.43787474  3.53667965  1.54109297  3.79998988]
 [-2.96670332  1.31053164 -3.63234149  0.40062619 -2.71984275 -2.56084743]
 [ 0.06446086 -0.73115168 -1.15465806 -1.60236106  1.27753059 -1.32643179]
 [ 1.07586711  3.28407854 -0.76774762 -0.96239531  0.71842609 -1.6781868 ]
 [ 3.02939884 -3.10276914 -1.3491814  -3.36410929  3.50079599  1.62793956]]
```

1st Generation

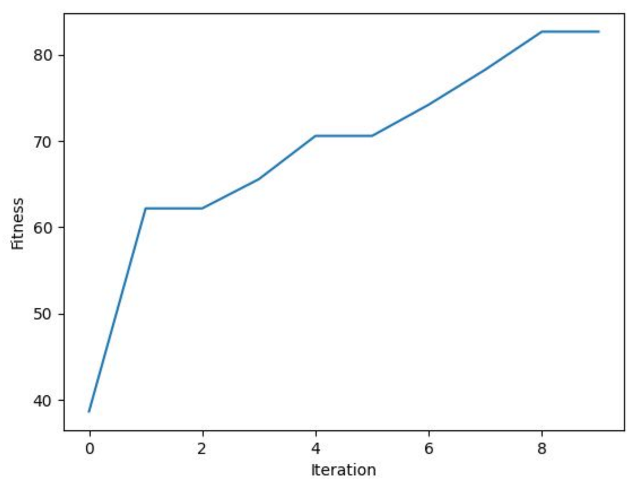
```
Generation :  0
Fitness
[ 29.59694383  38.69458088   1.61648135 -43.96649653  16.75631228
 -18.15156873  -9.77899084 -49.37961943]
Best result :  38.6945808806662
Parents
[[ 1.90703394 -0.18198742  3.98895785  2.86320899  0.88686627 -2.59161741]
 [ 0.10437277 -3.06269698  2.61504437 -1.56204928 -0.77732296 -2.80023349]
 [-2.96670332  1.31053164 -3.63234149  0.40062619 -2.71984275 -2.56084743]
 [ 3.31783698  1.40325753 -3.00092359 -1.24947539  0.88036932 -3.74177034]]
Crossover
[[ 1.90703394 -0.18198742  3.98895785 -1.56204928 -0.77732296 -2.80023349]
 [ 0.10437277 -3.06269698  2.61504437  0.40062619 -2.71984275 -2.56084743]
 [-2.96670332  1.31053164 -3.63234149 -1.24947539  0.88036932 -3.74177034]
 [ 3.31783698  1.40325753 -3.00092359  2.86320899  0.88686627 -2.59161741]]
Mutation
[[ 1.90703394 -0.18198742  3.33085439 -1.56204928 -0.77732296 -3.4889268 ]
 [ 0.10437277 -3.06269698  3.04059286  0.40062619 -2.71984275 -2.7838638 ]
 [-2.96670332  1.31053164 -4.50093676 -1.24947539  0.88036932 -3.99398814]
 [ 3.31783698  1.40325753 -2.22932092  2.86320899  0.88686627 -1.92043003]]
```

10th Generation

```
Generation :  9
Fitness
[82.6340379  81.27575319 78.24350761 75.28014413 77.55408117 76.10816489
 69.72476571 80.80394507]
Best result :  82.6340378950662
Parents
[[ 1.90703394 -0.18198742  4.01069215  0.40062619 -2.71984275 -6.102788  ]
 [ 1.90703394 -0.18198742  4.06366855  0.40062619 -2.71984275 -5.77434074]
 [ 1.90703394 -0.18198742  3.96449148  0.40062619 -2.71984275 -5.74781131]
 [ 1.90703394 -0.18198742  3.67155091  0.40062619 -2.71984275 -5.42118461]]
Crossover
[[ 1.90703394 -0.18198742  4.01069215  0.40062619 -2.71984275 -5.77434074]
 [ 1.90703394 -0.18198742  4.06366855  0.40062619 -2.71984275 -5.74781131]
 [ 1.90703394 -0.18198742  3.96449148  0.40062619 -2.71984275 -5.42118461]
 [ 1.90703394 -0.18198742  3.67155091  0.40062619 -2.71984275 -6.102788  ]]
Mutation
[[ 1.90703394 -0.18198742  4.27942301  0.40062619 -2.71984275 -5.870571  ]
 [ 1.90703394 -0.18198742  3.16050915  0.40062619 -2.71984275 -5.07097764]
 [ 1.90703394 -0.18198742  4.13891506  0.40062619 -2.71984275 -5.39482972]
 [ 1.90703394 -0.18198742  3.64479644  0.40062619 -2.71984275 -6.58908534]]
```

Best solution

```
Best solution :  [[[ 1.90703394 -0.18198742  3.64479644  0.40062619 -2.71984275
   -6.58908534]]]
Best solution fitness :  [83.63900039]
```



Finding and Learnings:

We have successfully implemented the Genetic Algorithm (GA) in python. The fitness function determines how fit an individual is (the ability of an individual to compete with other individuals). It gives a fitness score to each individual. The probability that an individual will be selected for reproduction is based on its fitness score.

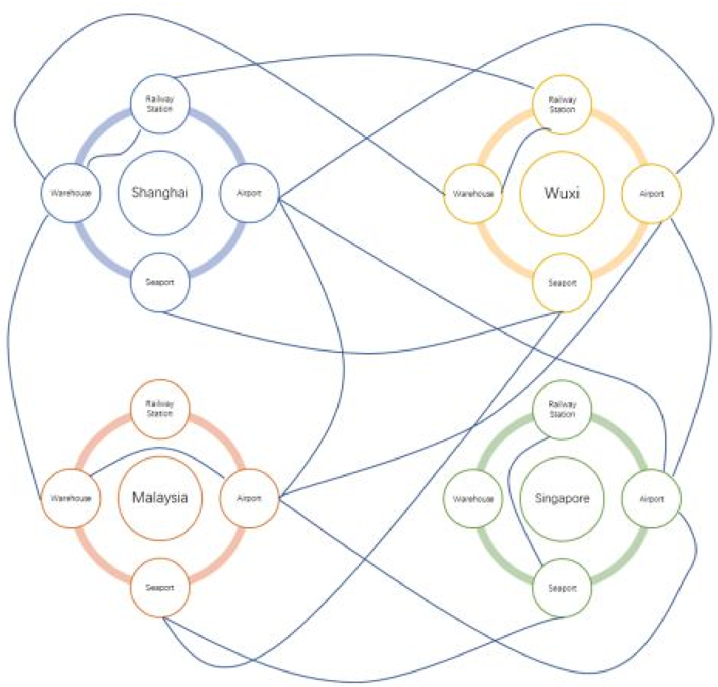
Experiment 8

Aim: Write a program to implement any kind of optimization on a multimodal dataset

Theory:

In this experiment, we would be using swarm algorithms to solve multi-modal transportation cost minimization in goods delivery and do the required optimization for better results.

In our simulated case, there are 8 goods, 4 cities/countries (Shanghai, Wuxi, Singapore, Malaysia), 16 ports and 4 transportation tools. The 8 goods originate from different cities and have different destinations. Each city/country has 4 ports, the airport, railway station, seaport and warehouse. There are in total 50 direct routes connecting different ports. Each route has a specific transportation tool, transportation cost, transit time and weekly schedule. Warehouses in each city allow goods to be deposited for a period of time so as to fit certain transportation schedules or wait for other goods to be transported together. All goods might have different order dates and different delivery deadlines. With all these criteria, how can we find out solution routes for all goods that minimize the overall cost?



Optimizing of routes using various swarm algorithms

Algorithm:

In order to make the criteria logic clearer and the calculation more efficient, we use the concept of the matrix to build the necessary components in the model. In our case, there are totally 4 dimensions:

1. Start Port: i

Indicating the start port of a direct transport route. The dimension length equals the total number of ports in the data.

2. End Port: j

Indicating the end port of a direct transport route. The dimension length equals the total number of ports in the data.

3. Time: t

Indicating the departure time of direct transport. The dimension length equals the total number of days between the earliest order date and the latest delivery deadline date of all goods in the data.

4. Goods: k

Indicating the goods to be transported. The dimension length equals the total number of goods in the data. All the variable or parameter matrices to be introduced in the later parts will have one or more of these 4 dimensions.

The objective of the model is to minimize the overall cost, which includes 3 parts, transportation cost, warehouse cost and tax cost. Firstly, the transportation cost includes container cost and route fixed cost. Container cost equals the number of containers used in each route times per container cost while route fixed cost equals the sum of the fixed cost of all routes.

Source Code:

multi.py

```
from
docplex.mp.model
import Model
from itertools import product
import numpy as np
import cvxpy as cp
import pandas as pd
import json
class MMT:
 '''a Model class that solves the multi-model transportation optimization problem.'''
 def __init__(self, framework='DOCPLEX'):
        self.portSpace = None
```

```
self.dateSpace = None
self.goods = None
self.indexPort = None
self.portIndex = None
self.maxDate = None
self.minDate = None
self.tranCost = None
self.tranFixedCost = None
self.tranTime = None
self.ctnVol = None
self.whCost = None
self.kVol = None
self.kValue = None
self.kDDL = None
self.kStartPort = None
self.kEndPort = None
self.kStartTime = None
self.taxPct = None
self.transitDuty = None
self.route_num = None
self.available_routes = None
# decision variables
self.var = None
self.x = None
self.var_2 = None
self.y = None
self.var_3 = None
self.z = None
# result & solution
self.xs = None
self.ys = None
self.zs = None
self.whCostFinal = None
self.transportCost = None
self.taxCost = None
self.solution_ = None
self.arrTime_ = None
self.objective_value = None
# helping variables
```

```
        self.var_location = None
        self.var_2_location = None
        self.var_3_location = None

        if framework not in ['CVXPY', 'DOCPLEX']:
            raise ValueError('Framework not supported, the model only supports CVXPY and
DOCPLEX')
        else:
            self.framework = framework

    def set_param(self, route, order):
    '''set model parameters based on the read-in route and order information.'''
        bigM = 100000
        route = route[route['Feasibility'] == 1]
        route['Warehouse Cost'][route['Warehouse Cost'].isnull()] = bigM
        route = route.reset_index()
        portSet = set(route['Source']) | set(route['Destination'])
        self.portSpace = len(portSet)
        self.portIndex = dict(zip(range(len(portSet)), portSet))
        self.indexPort = dict(zip(self.portIndex.values(), self.portIndex.keys()))
        self.maxDate = np.max(order['Required Delivery Date'])
        self.minDate = np.min(order['Order Date'])
        self.dateSpace = (self.maxDate - self.minDate).days
        startWeekday = self.minDate.weekday() + 1
        weekday = np.mod((np.arange(self.dateSpace) + startWeekday), 7)
        weekday[weekday == 0] = 7
        weekdayDateList = {i: [] for i in range(1, 8)}
        for i in range(len(weekday)):
            weekdayDateList[weekday[i]].append(i)
        for i in weekdayDateList:
            weekdayDateList[i] = json.dumps(weekdayDateList[i])

        source = list(route['Source'].replace(self.indexPort))
        destination = list(route['Destination'].replace(self.indexPort))
        DateList =list(route['Weekday'].replace(weekdayDateList).apply(json.loads))
        self.goods = order.shape[0]
        self.tranCost = np.ones([self.portSpace, self.portSpace, self.dateSpace])* bigM
        self.tranFixedCost = np.ones([self.portSpace, self.portSpace, self.dateSpace]) * bigM
        self.tranTime = np.ones([self.portSpace, self.portSpace, self.dateSpace])* bigM
```

```
        for i in range(route.shape[0]):
            self.tranCost[source[i], destination[i], DateList[i]] = route['Cost'][i]
            self.tranFixedCost[source[i], destination[i], DateList[i]] =route['Fixed Freight 
Cost'][i]
            self.tranTime[source[i], destination[i], DateList[i]] = route['Time'][i]
        self.transitDuty = np.ones([self.portSpace, self.portSpace]) * bigM
        self.transitDuty[source, destination] = route['Transit Duty']

        # make the container size of infeasible routes to be small enough, similar to bigM
        self.ctnVol = np.ones([self.portSpace, self.portSpace]) * 0.1
        self.ctnVol[source, destination] = route['Container Size']
        self.ctnVol = self.ctnVol.reshape(self.portSpace, self.portSpace, 1)
        self.whCost = route[['Source', 'Warehouse Cost']].drop_duplicates()
        self.whCost['index'] = self.whCost['Source'].replace(self.indexPort)
        self.whCost = np.array(self.whCost.sort_values(by='index')['WarehouseCost'])
        self.kVol = np.array(order['Volume'])
        self.kValue = np.array(order['Order Value'])
        self.kDDL = np.array((order['Required Delivery Date'] - self.minDate).dt.days)
        self.kStartPort = np.array(order['Ship From'].replace(self.indexPort))
        self.kEndPort = np.array(order['Ship To'].replace(self.indexPort))
        self.kStartTime = np.array((order['Order Date'] - self.minDate).dt.days)
        self.taxPct = np.array(order['Tax Percentage'])

        # add available route indexes
        self.route_num = route[['Source','Destination']].drop_duplicates().shape[0]
        routes = route[['Source','Destination']].drop_duplicates().replace(self.indexPort)
        self.available_routes = list(zip(routes['Source'], routes['Destination']))

        # localization variables of decision variables in the matrix
        var_location = product(self.available_routes, range(self.dateSpace),range(self.goods))
        var_location = [(i[0][0], i[0][1], i[1], i[2]) for i in var_location]
        self.var_location = tuple(zip(*var_location))
        var_2_location = product(self.available_routes, range(self.dateSpace))
        var_2_location = [(i[0][0], i[0][1], i[1]) for i in var_2_location]
        self.var_2_location = tuple(zip(*var_2_location))
        self.var_3_location = self.var_2_location

    def build_model(self):
    '''overall function to build up model objective and constraints'''
```

```
        if self.framework == 'CVXPY':
            self.cvxpy_build_model()
        elif self.framework == 'DOCPLEX':
            self.cplex_build_model()

    def cvxpy_build_model(self):
    '''build up the mathematical programming model's objective and constraints using CVXPY
framework.'''
    # 4 dimensional binary decision variable matrix
        self.var = cp.Variable(self.route_num * self.dateSpace * self.goods, boolean=True,
name='x')
        self.x = np.zeros((self.portSpace, self.portSpace, self.dateSpace,
self.goods)).astype('object')
        self.x[self.var_location] = list(self.var)

        # 3 dimensional container number matrix
        self.var_2 = cp.Variable(self.route_num * self.dateSpace, integer=True, name='y')
        self.y = np.zeros((self.portSpace, self.portSpace, self.dateSpace)).astype('object')
        self.y[self.var_2_location] = list(self.var_2)

        self.var_3 = cp.Variable(self.route_num * self.dateSpace, boolean=True, name='z')
        self.z = np.zeros((self.portSpace, self.portSpace, self.dateSpace)).astype('object')
        self.z[self.var_3_location] = list(self.var_3)
        # warehouse related cost
        warehouseCost, arrTime, stayTime = self.warehouse_fee(self.x)
        transportCost = np.sum(self.y * self.tranCost) + np.sum(self.z *self.tranFixedCost)
        transitDutyCost = np.sum(np.sum(np.dot(self.x, self.kValue), axis=2) * self.transitDuty)
        taxCost = np.sum(self.taxPct * self.kValue) + transitDutyCost
        objective = cp.Minimize(transportCost + warehouseCost + taxCost)

        constraints = []
        constraints += [np.sum(self.x[self.kStartPort[k], :, :, k]) == 1 for k in range(self.goods)]
        constraints += [np.sum(self.x[:, self.kEndPort[k], :, k]) == 1 for k in range(self.goods)]

        constraints += [np.sum(self.x[:, self.kStartPort[k], :, k]) == 0 for k in range(self.goods)]
        constraints += [np.sum(self.x[self.kEndPort[k], :, :, k]) == 0 for k in range(self.goods)]

        for k in range(self.goods):
            for j in range(self.portSpace):
```

```
                    if (j != self.kStartPort[k]) & (j != self.kEndPort[k]):
                        constraints.append(np.sum(self.x[:, j, :, k]) == np.sum(self.x[j, :, :,
k]))

    constraints += [np.sum(self.x[i, :, :, k]) <= 1 for k in range(self.goods)
    for i in range(self.portSpace)]
        constraints += [np.sum(self.x[:, j, :, k]) <= 1 for k in range(self.goods)
    for j in range(self.portSpace)]
        constraints += [stayTime[j, k] >= 0 for j in range(self.portSpace) for k in
range(self.goods)]

    numCtn = np.dot(self.x, self.kVol) / self.ctnVol
    constraints += [self.y[i, j, t] - numCtn[i, j, t] >= 0 \
                            for i in range(self.portSpace) for j in
    range(self.portSpace) for t in
                            range(self.dateSpace) if not isinstance(self.y[i, j, t] -
    numCtn[i, j, t] >= 0, bool)]

    constraints += [self.z[i, j, t] >= (np.sum(self.x[i, j, t, :]) * 10e-5) \
     for i in range(self.portSpace)
        for j in range(self.portSpace)
            for t in range(self.dateSpace)
                if not isinstance(self.z[i, j, t] >= (np.sum(self.x[i, j, t, :]) * 10e-5),
bool)]
                            constraints += [np.sum(arrTime[:, self.kEndPort[k], :, k])
<= self.kDDL[k]
    for k in range(self.goods)
     if not isinstance(np.sum(arrTime[:, self.kEndPort[k], :, k]) <= self.kDDL[k], bool)]
        model = cp.Problem(objective, constraints)
     self.objective = objective
     self.constraints = constraints
     self.model = model

 def cplex_build_model(self):
 '''build up the mathematical programming model's objective and constraints using DOCPLEX
framework.'''
    model = Model()

     self.var = model.binary_var_list(self.route_num * self.dateSpace *self.goods, name='x')
```

```
        self.x = np.zeros((self.portSpace, self.portSpace,
self.dateSpace,self.goods)).astype('object')
        self.x[self.var_location] = self.var

        # 3 dimensional container number matrix
        self.var_2 = model.integer_var_list(self.route_num * self.dateSpace,name='y')
        self.y = np.zeros((self.portSpace, self.portSpace,self.dateSpace)).astype('object')
        self.y[self.var_2_location] = self.var_2

        self.var_3 = model.binary_var_list(self.route_num * self.dateSpace,name='z')
        self.z = np.zeros((self.portSpace, self.portSpace,self.dateSpace)).astype('object')
        self.z[self.var_3_location] = self.var_3
        warehouseCost, arrTime, stayTime = self.warehouse_fee(self.x)
        transportCost = np.sum(self.y * self.tranCost) + np.sum(self.z *self.tranFixedCost)
        transitDutyCost = np.sum(np.sum(np.dot(self.x, self.kValue), axis=2) *self.transitDuty)
        taxCost = np.sum(self.taxPct * self.kValue) + transitDutyCost
        model.minimize(transportCost + warehouseCost + taxCost)

        model.add_constraints(np.sum(self.x[self.kStartPort[k], :, :, k]) == 1 for k in
range(self.goods))
            model.add_constraints(np.sum(self.x[:, self.kEndPort[k], :, k]) == 1 for k in
range(self.goods))
            model.add_constraints(np.sum(self.x[:, self.kStartPort[k], :, k]) == 0 for k in
range(self.goods))
            model.add_constraints(np.sum(self.x[self.kEndPort[k], :, :, k]) == 0 for k in
range(self.goods))

        for k in range(self.goods):
        for j in range(self.portSpace):
            if (j != self.kStartPort[k]) & (j != self.kEndPort[k]):
                model.add_constraint(np.sum(self.x[:, j, :, k]) ==np.sum(self.x[j, :, :, k]))

        model.add_constraints(np.sum(self.x[i, :, :, k]) <= 1 for k in range(self.goods) for i in
range(self.portSpace))
        model.add_constraints(np.sum(self.x[:, j, :, k]) <= 1 for k in  range(self.goods) for j in
range(self.portSpace))
        # 5.transition-out should be after transition-in
        model.add_constraints(stayTime[j, k] >= 0 for j in range(self.portSpace)
        for k in range(self.goods))
```

```
            # 6.constraint for number of containers used
            numCtn = np.dot(self.x, self.kVol) / self.ctnVol
            model.add_constraints(self.y[i, j, t] - numCtn[i, j, t] >= 0 \
            for i in range(self.portSpace) for j in
           range(self.portSpace) for t in
            range(self.dateSpace) if not isinstance(self.y[i, j,
           t] - numCtn[i, j, t] >= 0, bool))
            # 7. constraint to check whether a route is used
            model.add_constraints(self.z[i, j, t] >= (np.sum(self.x[i, j, t, :]) *
           10e-5) \
            for i in range(self.portSpace) for j in range(self.portSpace) for t in range(self.dateSpace)
            if not isinstance(self.z[i, j, t] >= (np.sum(self.x[i, j, t, :]) * 10e-5), bool))
                    model.add_constraints(np.sum(arrTime[:, self.kEndPort[k], :, k])
<=self.kDDL[k] for k in range(self.goods)
            if not isinstance(np.sum(arrTime[:,self.kEndPort[k], :, k]) <= self.kDDL[k], bool))
            self.objective = model.objective_expr
            self.constraints = list(model.iter_constraints())
            self.model = model

 def solve_model(self, solver=cp.CBC):
            try:
            if self.framework == 'CVXPY':
            self.objective_value = self.model.solve(solver)
            self.xs = np.zeros((self.portSpace, self.portSpace,
           self.dateSpace, self.goods))
            self.xs[self.var_location] = self.var.value
            self.ys = np.zeros((self.portSpace, self.portSpace,
           self.dateSpace))
            self.ys[self.var_2_location] = self.var_2.value
            self.zs = np.zeros((self.portSpace, self.portSpace,
           self.dateSpace))
            self.zs[self.var_3_location] = self.var_3.value
            elif self.framework == 'DOCPLEX':
            ms = self.model.solve()
            self.objective_value = self.model.objective_value
            self.xs = np.zeros((self.portSpace, self.portSpace,
           self.dateSpace, self.goods))
            self.xs[self.var_location] = ms.get_values(self.var)
            self.ys = np.zeros((self.portSpace, self.portSpace,
```

```
            self.dateSpace))
             self.ys[self.var_2_location] = ms.get_values(self.var_2)
             self.zs = np.zeros((self.portSpace, self.portSpace,
            self.dateSpace))
             self.zs[self.var_3_location] = ms.get_values(self.var_3)
             except:
             raise Exception('Model is not solvable, no solution will be provided')
             nonzeroX = list(zip(*np.nonzero(self.xs)))
             nonzeroX = sorted(nonzeroX, key=lambda x: x[2])
             nonzeroX = sorted(nonzeroX, key=lambda x: x[3])
             nonzeroX = list(map(lambda x: (self.portIndex[x[0]], self.portIndex[x[1]],
            \
             (self.minDate + pd.to_timedelta(x[2],
            unit='days')).date().isoformat(),
             x[3]), nonzeroX))
             self.whCostFinal, arrTime, _ = self.warehouse_fee(self.xs)
             self.transportCost = np.sum(self.ys * self.tranCost) + np.sum(self.zs *
            self.tranFixedCost)
             self.taxCost = np.sum(self.taxPct * self.kValue) + \
             np.sum(np.sum(np.dot(self.xs, self.kValue), axis=2) *
            self.transitDuty)
             self.solution_ = {}
             self.arrTime_ = {}
             for i in range(self.goods):
             self.solution_['goods-' + str(i + 1)] = list(filter(lambda x: x[3] ==
            i, nonzeroX))
             self.arrTime_['goods-' + str(i + 1)] = (self.minDate + pd.to_timedelta
            \
             (np.sum(arrTime[:, self.kEndPort[i], :, i]),
            unit='days')).date().isoformat()

 def get_output_(self):
            return self.objective_value, self.solution_, self.arrTime_

 def warehouse_fee(self, x):
             startTime = np.arange(self.dateSpace).reshape(1, 1, self.dateSpace, 1) * x
             arrTimeMtrx = startTime + self.tranTime.reshape(self.portSpace, \ self.portSpace,
self.dateSpace, 1) * x
             arrTime = arrTimeMtrx.copy()
```

```
        arrTimeMtrx[:, self.kEndPort.tolist(), :, range(self.goods)] = 0
        stayTime = np.sum(startTime, axis=(1, 2)) - np.sum(arrTimeMtrx, axis=(0, 2))
        stayTime[self.kStartPort.tolist(), range(self.goods)] -= self.kStartTime
        warehouseCost = np.sum(np.sum(stayTime * self.kVol, axis=1) * self.whCost)
        return warehouseCost, arrTime, stayTime

    def txt_solution(self, route, order):
        '''transform the cached results to text.'''
        travelMode = dict(zip(zip(route['Source'], route['Destination']), route['Travel Mode']))
        txt = "Solution"
        txt += "\nNumber of goods: " + str(order['Order Number'].count())
        txt += "\nTotal cost: " + str(self.transportCost + self.whCostFinal + self.taxCost)
        txt += "\nTransportation cost: " + str(self.transportCost)
        txt += "\nWarehouse cost: " + str(self.whCostFinal)
        txt += "\nTax cost: " + str(self.taxCost)
        for i in range(order.shape[0]):
        txt += "\n-----------------------------------"
        txt += "\nGoods-" + str(i + 1) + " Category: " + order['Commodity'][i]
        txt += "\nStart date: " + pd.to_datetime(order['Order Date']) \ .iloc[i].date().isoformat()
        txt += "\nArrival date: " + str(self.arrTime_['goods-' + str(i + 1)])
        txt += "\nRoute:"
        solution = self.solution_['goods-' + str(i + 1)]
        route_txt = ''
        a = 1
        for j in solution:
            route_txt += "\n(" + str(a) + ")Date: " + j[2]
            route_txt += " From: " + j[0]
            route_txt += " To: " + j[1]
            route_txt += " By: " + travelMode[(j[0], j[1])]
            a += 1
            txt += route_txt
        return txt

def transform(filePath):
        order = pd.read_excel(filePath, sheet_name='Order Information')
        route = pd.read_excel(filePath, sheet_name='Route Information')
        order['Tax Percentage'][order['Journey Type'] == 'Domestic'] = 0
        route['Cost'] = route[route.columns[7:12]].sum(axis=1)
        route['Time'] = np.ceil(route[route.columns[14:18]].sum(axis=1) / 24)
```

```
route = route[list(route.columns[0:4]) +
                  ['Fixed Freight Cost', 'Time', \ 'Cost', 'Warehouse Cost', 'Travel Mode',
'Transit Duty'] + list(
route.columns[-9:-2])]
route = pd.melt(route, id_vars=route.columns[0:10], value_vars=route.columns[-7:] \ ,
var_name='Weekday', value_name='Feasibility')
route['Weekday'] = route['Weekday'].replace({'Monday': 1, 'Tuesday': 2,'Wednesday': 3,
\ 'Thursday': 4, 'Friday': 5,'Saturday': 6, 'Sunday': 7})
return order, route

if __name__ == '__main__':
order, route = transform("model data.xlsx")
m = MMT()
m.set_param(route, order)
m.build_model()
m.solve_model()
txt = m.txt_solution(route, order)
with open("Solution.txt", "w") as text_file:
text_file.write(txt)
```

Output:

```
Solution
          Number of goods: 8
          Total cost: 196959.0
          Transportation cost: 6645.0
          Warehouse cost: 1410.0
          Tax cost: 188904.0
          -----------------------------------
          Goods-1  Category: Honey
          Start date: 2018-02-01
          Arrival date: 2018-02-12
          Route:
          (1)Date: 2018-02-01  From: Singapore Warehouse  To: Malaysia Warehouse  By: Truck
          (2)Date: 2018-02-02  From: Malaysia Warehouse  To: Malaysia Port  By: Truck
          (3)Date: 2018-02-03  From: Malaysia Port  To: Shanghai Port  By: Sea
          (4)Date: 2018-02-10  From: Shanghai Port  To: Shanghai Warehouse  By: Truck
          (5)Date: 2018-02-11  From: Shanghai Warehouse  To: Wuxi Warehouse  By: Truck
          -----------------------------------
          Goods-2  Category: Furniture
          Start date: 2018-02-02
          Arrival date: 2018-02-11
          Route:
          (1)Date: 2018-02-02  From: Malaysia Warehouse  To: Malaysia Port  By: Truck
          (2)Date: 2018-02-03  From: Malaysia Port  To: Shanghai Port  By: Sea
          (3)Date: 2018-02-10  From: Shanghai Port  To: Shanghai Warehouse  By: Truck

          -----------------------------------
          Goods-3  Category: Paper plates
          Start date: 2018-02-03
          Arrival date: 2018-02-15
          Route:
          (1)Date: 2018-02-03  From: Singapore Warehouse  To: Malaysia Warehouse  By: Truck
          (2)Date: 2018-02-06  From: Malaysia Warehouse  To: Malaysia Port  By: Truck
          (3)Date: 2018-02-07  From: Malaysia Port  To: Shanghai Port  By: Sea
          (4)Date: 2018-02-14  From: Shanghai Port  To: Shanghai Warehouse  By: Truck
          -----------------------------------
          Goods-4  Category: Pharmaceutical drugs
          Start date: 2018-02-04
          Arrival date: 2018-02-15
```

```
Route:
(1)Date: 2018-02-04  From: Singapore Warehouse  To: Malaysia Warehouse  By: Truck
(2)Date: 2018-02-06  From: Malaysia Warehouse  To: Malaysia Port  By: Truck
(3)Date: 2018-02-07  From: Malaysia Port  To: Shanghai Port  By: Sea
(4)Date: 2018-02-14  From: Shanghai Port  To: Shanghai Warehouse  By: Truck
----------------------------------
Goods-5  Category: Cigarette
Start date: 2018-02-05
Arrival date: 2018-02-15
Route:
(1)Date: 2018-02-05  From: Wuxi Warehouse  To: Shanghai Warehouse  By: Truck
(2)Date: 2018-02-06  From: Shanghai Warehouse  To: Shanghai Port  By: Truck
(3)Date: 2018-02-07  From: Shanghai Port  To: Malaysia Port  By: Sea
(4)Date: 2018-02-14  From: Malaysia Port  To: Malaysia Warehouse  By: Truck
----------------------------------
Goods-6  Category: Apple
Start date: 2018-02-06
Arrival date: 2018-02-16
Route:
(1)Date: 2018-02-06  From: Shanghai Warehouse  To: Shanghai Port  By: Truck
(2)Date: 2018-02-07  From: Shanghai Port  To: Malaysia Port  By: Sea
(3)Date: 2018-02-14  From: Malaysia Port  To: Malaysia Warehouse  By: Truck
(4)Date: 2018-02-15  From: Malaysia Warehouse  To: Singapore Warehouse  By: Truck
----------------------------------

Goods-7  Category: Durian
Start date: 2018-02-07
Arrival date: 2018-02-08
Route:
(1)Date: 2018-02-07  From: Malaysia Warehouse  To: Singapore Warehouse  By: Truck
----------------------------------
Goods-8  Category: Furniture
Start date: 2018-02-08
Arrival date: 2018-02-09
Route:
(1)Date: 2018-02-08  From: Wuxi Warehouse  To: Shanghai Warehouse  By: Truck
```

Finding and Learnings:

We have successfully implemented the optimization Algorithm on a multimodal dataset in python.