**DELHI TECHNOLOGICAL UNIVERSITY**

(Formerly Delhi College of Engineering)

Shahbad Daulatpur, Bawana Road, Delhi 110042

**DEPARTMENT OF COMPUTER SCIENCE AND**

**ENGINEERING**



**CSE5216: Information and Network Security Lab File**

**Submitted To: Submitted By:**

**Mr. Sanjay Patidar Kunal Sinha**

**Associate Professor B.Tech Computer Science**

**Department of Computer 7th Semester**

**Science and Engineering 2K17/CO/164**

**INDEX**

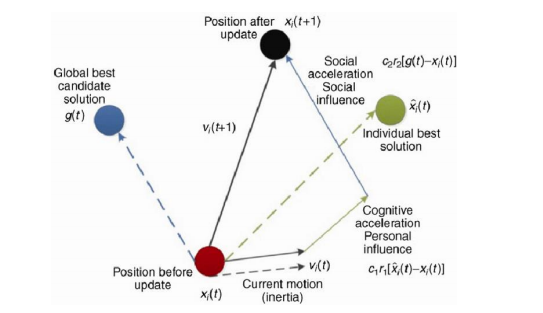
|  |  |  |  |
| --- | --- | --- | --- |
| **S.No.** | **Topic** | **Date** | **Signature** |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

**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:

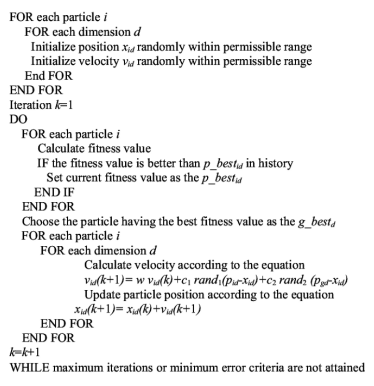
* Vᵢ(k+1) is the next iteration velocity
* W is an inertial parameter. This parameter affects the movement propagation given by the last velocity value.
* C₁ and C₂ are acceleration coefficients. C₁ value gives the importance of personal best value and C₂ is the importance of social best value.
* Pᵢ 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.
* rand₁ and rand₂ are random numbers where 0 ≤ rand ≤ 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**



**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)

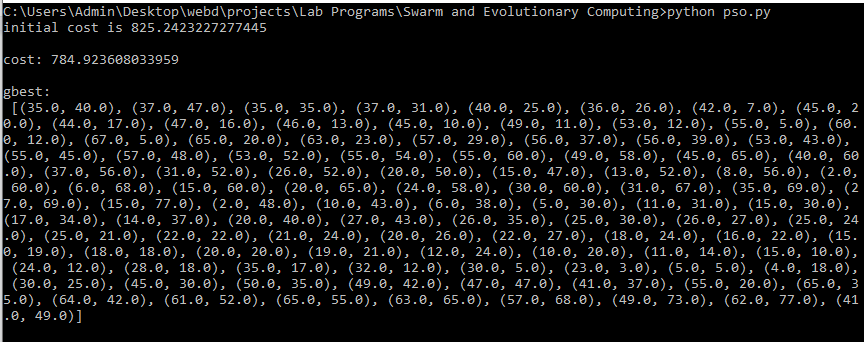
fig.suptitle('pso TSP')

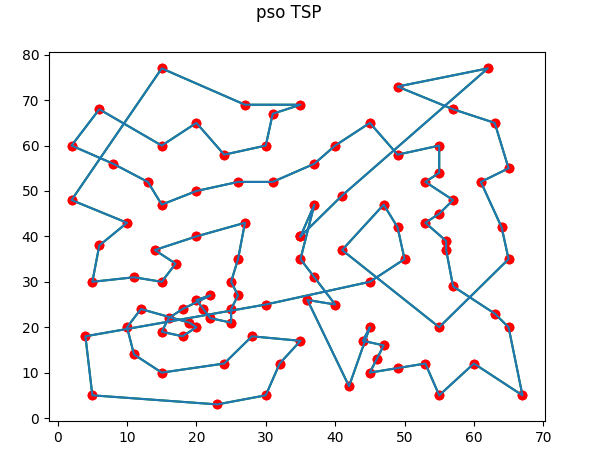
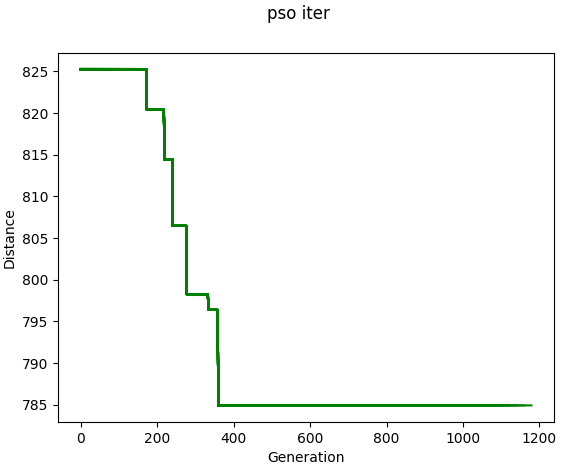
plt.plot(x\_list, y\_list, 'ro')

plt.plot(x\_list, y\_list)

plt.show(block=True)

**Output:**



****

**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:**

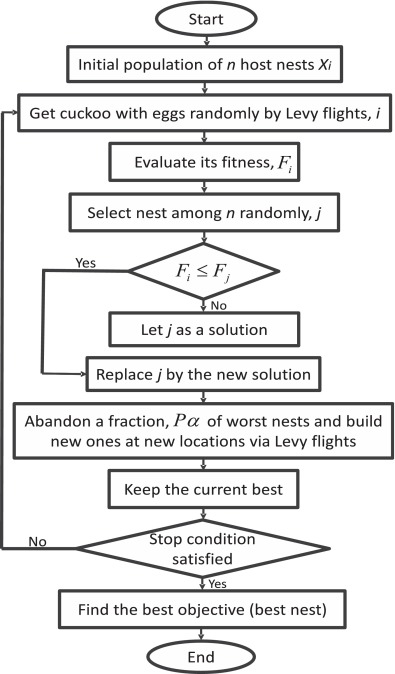
Cuckoo Search (CS) is a meta-heuristic algorithm based on the breeding pattern of certain species of cuckoo birds. In our research, we have implemented CS for the NP-hard optimization problem, the Traveling Salesman Problem (TSP). CS is based on three idealized rules:

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 ∈ (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])

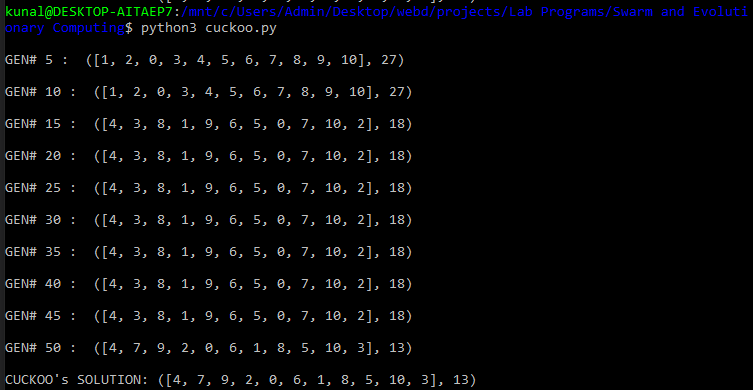
if (t+1) % 5 == 0:

print("\nGEN#", t+1, ": ", nests[0])

print("\nCUCKOO's SOLUTION", end=': ')

print(nests[0])

**Output:**



**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

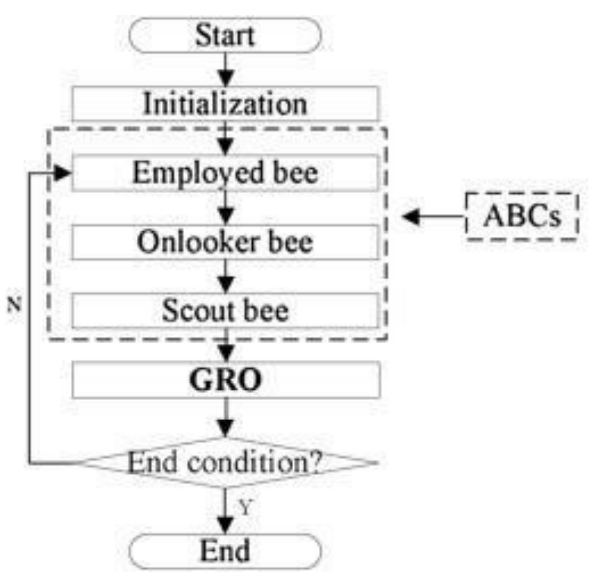
**Experiment 3**

**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
   1. evaluate fitness for each agent
   2. sort fitness in ascending order and get best agents
   3. from best agents list select agents from a to c
   4. Create new bees which will fly to the best solution
   5. Evaluate current best agent
   6. IF function(current best) < function (global best)
      1. global best = current best
   7. 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

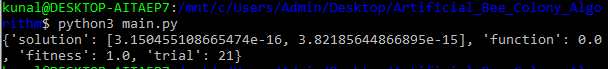
bound = 40

result = ABC(Ackley\_function, SN, bound, limit, MCN)

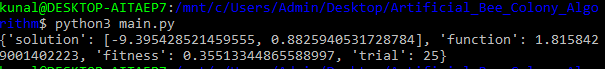
print(result.best\_solution())

**Output:**

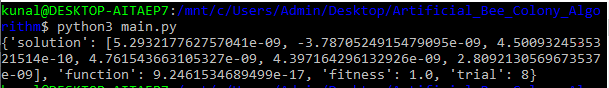
Ackley\_function

****

Bukin\_function\_N\_6

****

sphere\_function

****

**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**

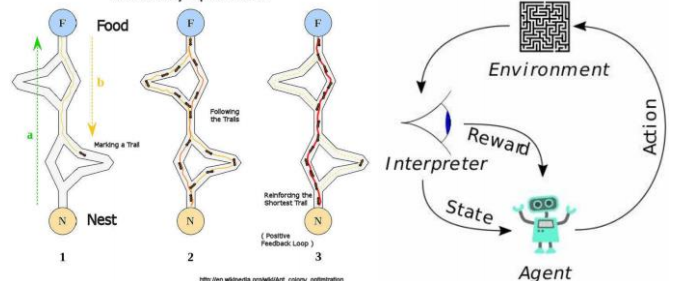
**Aim:** Write a program to implement Ant Colony optimization (ACO) algorithm.

**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 )
   1. Do Until each ant completely builds a solution
      1. Local trial update
   2. END Do
   3. Update pheromone
   4. 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):

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)

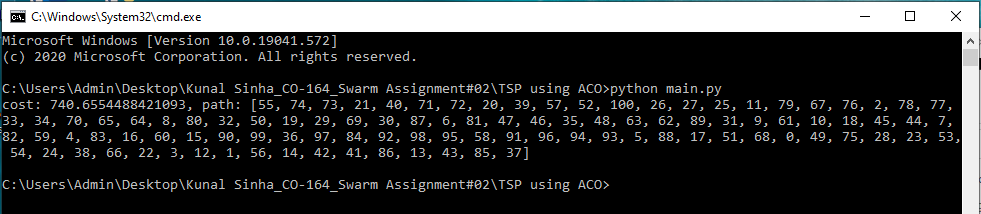
print('cost: {}, path: {}'.format(cost, path))

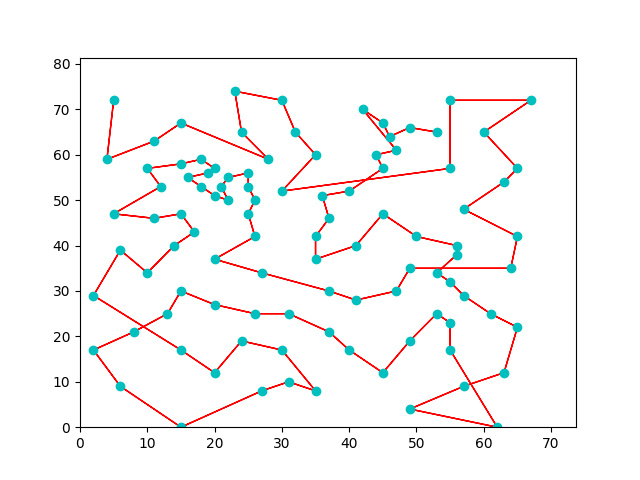
plot(points, path)

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).

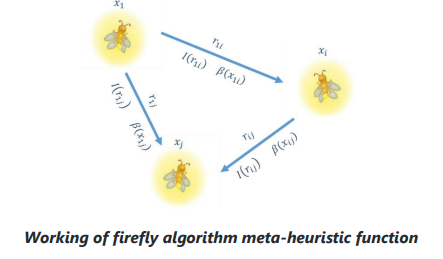
**Theory:**

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.



**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
   1. FOR i = 1 : n (all n fireflies)
      1. FOR j = 1 : i
      2. Light intensity Ii at xi is determined by f(xi)
      3. 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]
      4. Determine new solutions and revise light intensity
      5. END FOR j
   2. END FOR i
   3. 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):

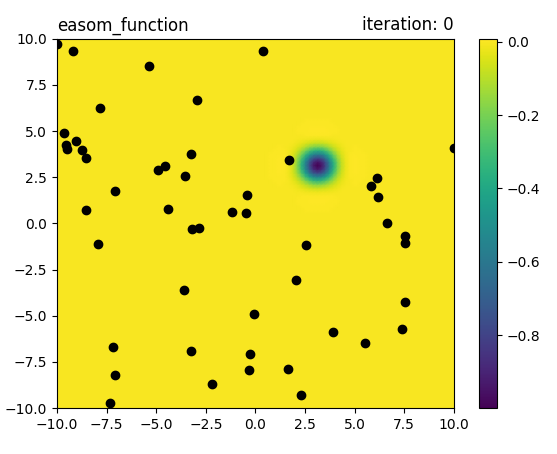
return -cos(x[0])\*cos(x[1])\*exp(-(x[0] - pi)\*\*2 - (x[1] - pi)\*\*2)

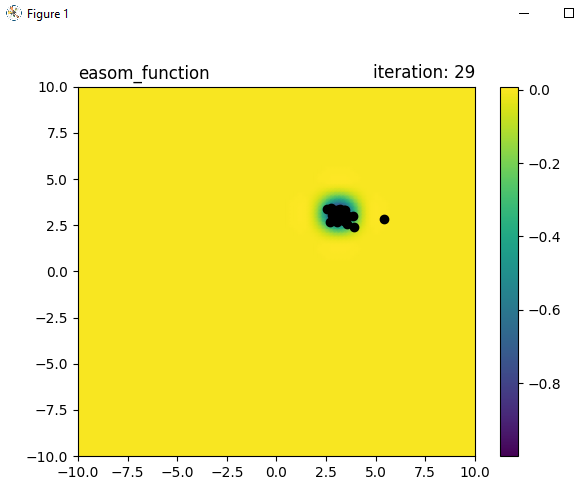
alh = fa(50, easom\_function, -10, 10, 2, 30,1,1,1,0.1,0,0.1)

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**

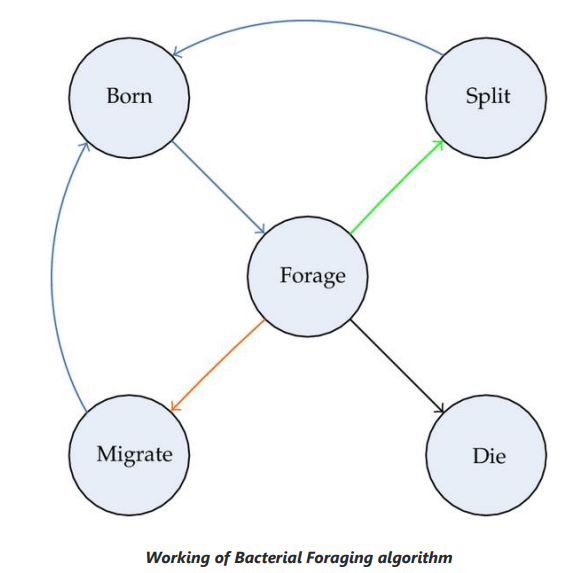
**Aim:** Write a program to implement the Bacterial Foraging algorithm.

**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.

****

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
   1. FOR number of chemotactic steps
      1. FOR each search agent
         1. Move agent to the random direction
         2. Calculate the fitness of the moved agent
         3. FOR swimming length
            1. IF current fitness is better than previous

Move agent to the same direction

* + - * 1. ELSE

Move agent to the random direction

* + 1. Calculate the fitness of each agent
  1. END FOR
  2. Compute and sort sum of fitness function of all chemotactic loops (health of agent)
  3. Let live and split only half of the population according to their health
  4. IF not the last iteration
     1. FOR each search agent
        1. With some probability replace agent with new random generated
  5. END IF
  6. Update the best search agent

1. 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):

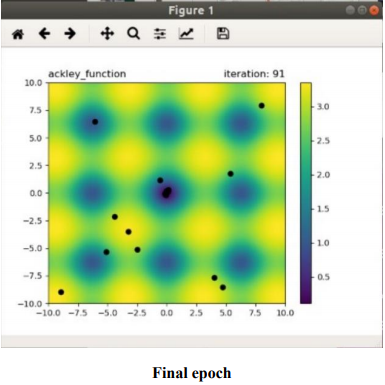
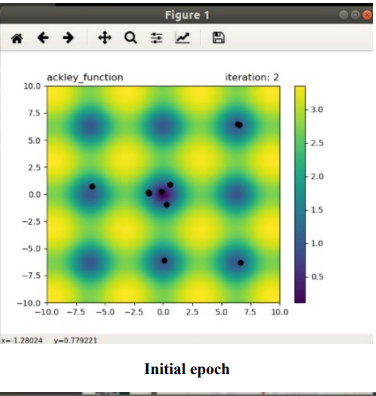
return -exp(-sqrt(0.5\*sum([i\*\*2 for i in x]))) - \exp(0.5\*sum([cos(i) for i in x])) + 1 + exp(1)

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**

****

**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:**

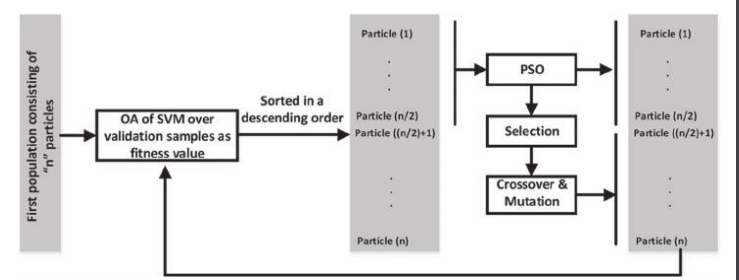
A genetic algorithm is a search heuristic that is inspired by Charles Darwin’s theory of natural evolution. This algorithm reflects the process of natural selection where the fittest individuals are selected for reproduction in order to produce offspring of the next generation.

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")

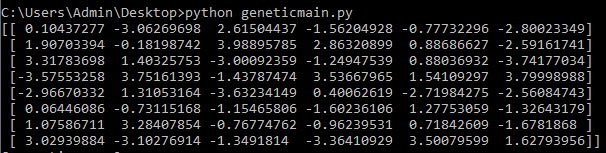
matplotlib.pyplot.ylabel("Fitness")

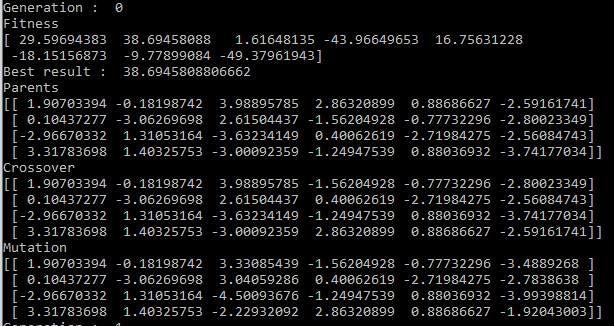
matplotlib.pyplot.show()

**Output:**

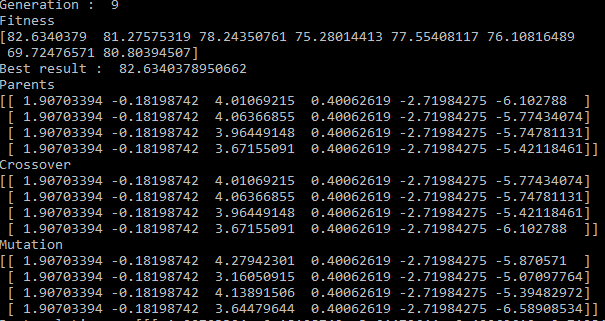
**For 10 generations**

**Initial population**

****

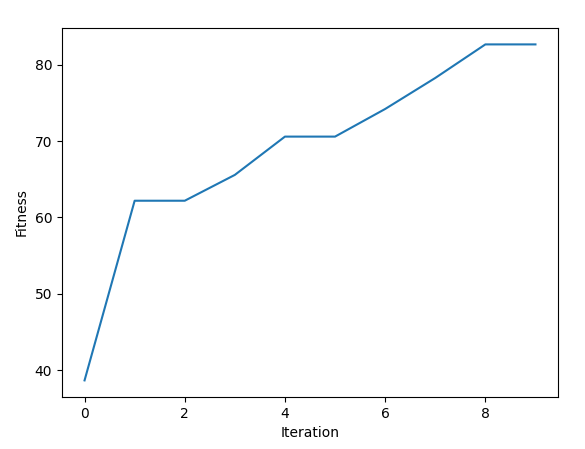
**1st Generation **

**10th Generation**

****

**Best solution**

****

****

**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.

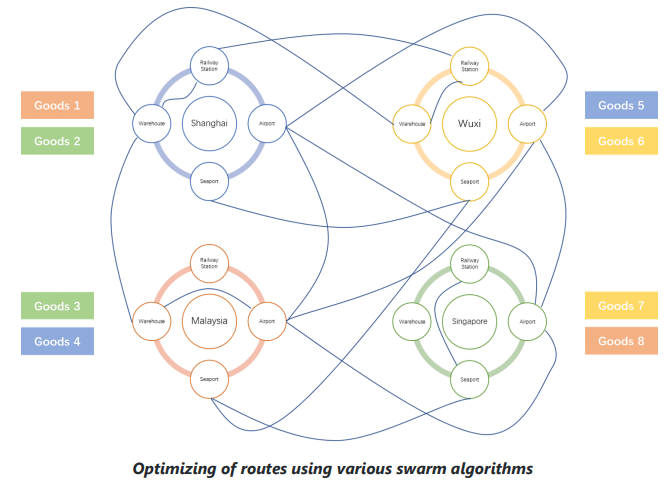
**Experiment 8**

**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?



**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()

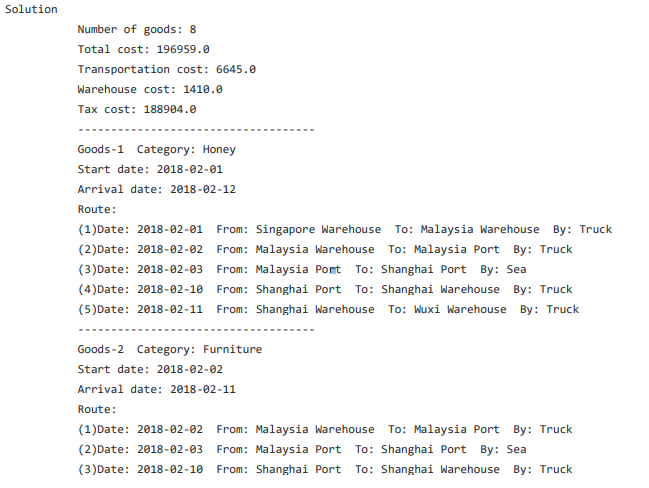
m.solve\_model()

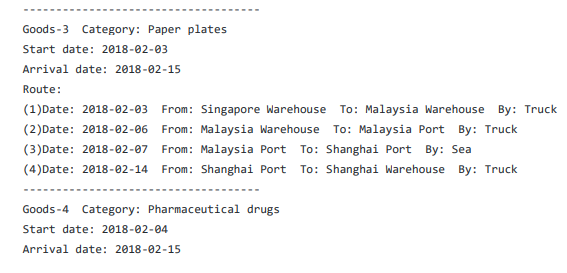
txt = m.txt\_solution(route, order)

with open("Solution.txt", "w") as text\_file:

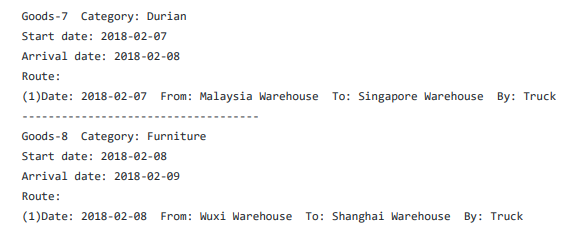
text\_file.write(txt)

**Output:**

****

****

****

****

**Finding and Learnings:**

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