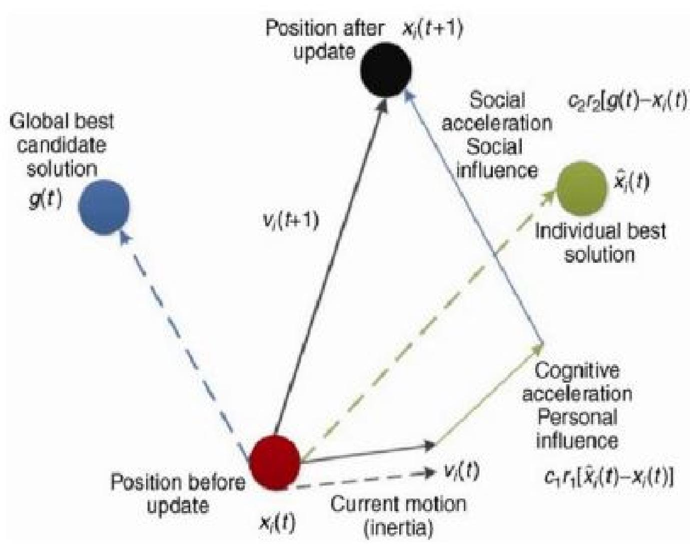
Name: **Kunal Sinha** Batch: Swarm **R1- G2** Roll no: **2K17/CO/164**

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 $0 \leq rand \leq 1$ 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):
```

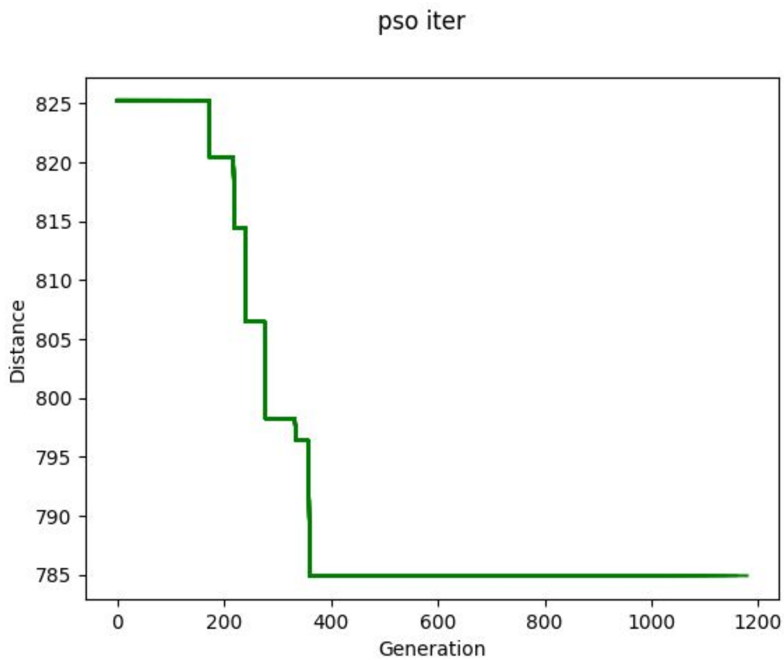
```
        return random.sample(self.cities, len(self.cities))

    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}')
```

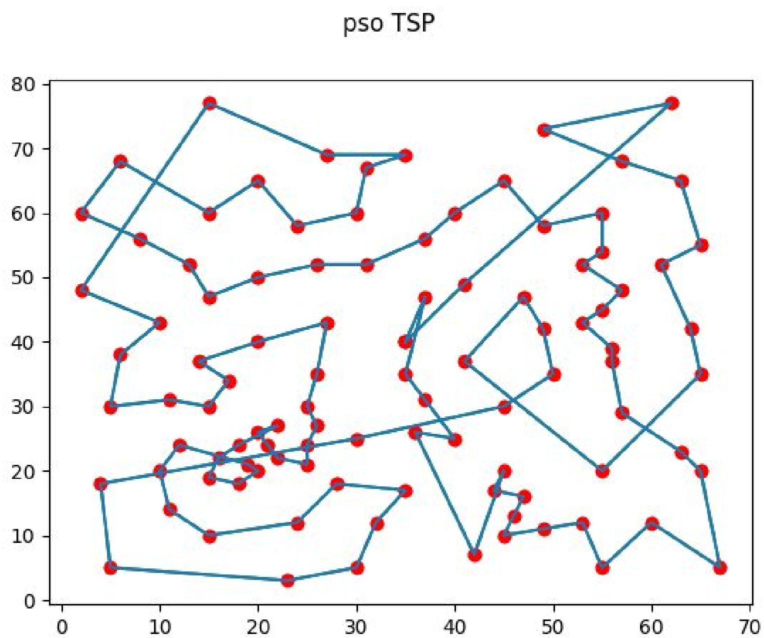
```
            plt.plot(x_list, y_list, 'ro')
            plt.plot(x_list, y_list, 'g')
            plt.draw()
            plt.pause(.001)
        self.gcost_iter.append(self.gbest.pbest_cost)

        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.