COMBINATORIAL OPTIMIZATION USING PARTICLE SWARM ALGORITHM (PSO)

Genetic Algorithms in engineering process modelling
Term Project: Spring 21''

RAGHUSRINIVASAN JV
19IM10023
DEVESH CHAUDHARY
19IM30027

INDEX

COMBINATORIAL OPTIMIZATION: INSIGHTS

Combinatorial Optimization is a subfield of Mathematical Optimization which is related to operations research, algorithm theory, and computational complexity theory. It has some important applications in various fields some of which are Artificial Intelligence, Machine Learning and other fields of computer science. It is concerned with the process of searching for optimal values of a objective function whose domain is a discrete but large configuration space (as opposed to N-dimensional continuous space. It operates on the domain of those optimization problems in which the set of feasible solutions is discrete or can be reduced to discrete , and in which the goal is to find the best solution.

PARTICLE SWARM OPTIMIZATION

HISTORY

PSO is a Swarm Intelligence method for global optimization, proposed by Kennedy and Eberhart. It is a population based self-adaptive, stochastic optimization technique. The PSO begins by creating the initial particles, and assigning them initial velocities. It evaluates the objective function at each particle location and determines the best function value and the best location. Every particle has a memory of its own in which it remembers its own best location of all it had till the present time and the global best of the whole population and at every iteration or epoch it updates its position and velocity based on its own best position, global best position and its own velocity. We set a upper bound to the velocity to keep the particles bounded inside the domain. We can have an error based stopping criteria or we can simply have fixed no. of iterations to achieve our objective. This algorithm is used to solve the continuous optimization problem where the search space is a continuous function.

PSO FOR COMBINATORIAL OPTIMIZATION

In this project we have taken up the Travelling Salesman (TSP) type combinatorial optimization problems to be solved using PSO. This is a permutation problem in which we have to cover all the cities once in a tour and the objective is to minimize the cost of traveling in the tour. There are a set of PSO based methods to solve this optimization problem like Swap Operator based PSO, Enhanced self-tentative PSO, Discrete PSO. Among the listed methods, Swap Operator based PSO is the pioneer and studied for other methods(give reference). This is the method we use in this project to solve TSP type problem.

SWAP OPERATOR BASED PSO

INSIGHTS

In this model the position of the i-th particle is represented as a D-dimensional tuple (or array) representing the D-dimensional search space. For the TSP type problems each particle represents a complete tour. At every step particle changes position based on its velocity which is also a D-dimensional tuple (or array). The velocity of the particle depends on its previous personal best position of the particle and the global best position of the swarm. However, the velocity updating scheme and the position updating scheme in this algorithm will be different than what we have in the classic PSO.

SWAP OPERATOR (SO)

The velocity for solving TSP in the Swap Operator based PSO is a Swap Sequence (SS). A Swap Sequence is a tuple (or array) of Swap Operators (SO), which indicates two positions in the tour that might be swap. All swap operators of a SS are applied maintaining order on a particle and gives a new tour i.e., a particle having new solution in PSO.

Position of the particle: $X_i = (x_{i1}, x_{i2}, x_{i3}, \ldots\ldots., x_{iD})$

Velocity of the particle: $V_i = (v_{i1}, v_{i2}, v_{i3}, \ldots\ldots., v_{iD})$

Personal best $(pBest_i) = (p_{i1}, p_{i2}, p_{i3}, \ldots\ldots.., p_{iD})$

Global best $(gBest) = (g_1, g_2, g_3, \ldots\ldots.., g_D)$

Swap Sequence $(SS) = (SO_1, SO_2, SO_3, \ldots\ldots., SO_D)$

SO_i are individual swap operators

SWAP OPERATOR: IN ACTION

Example of a swap operator:

We take a 5-city sequence and apply a swap operator SO(1,3)

$S = (1,3,2,5,4)$

$S' = S + SO(1,3) = (1,3,2,5,4) + SO(1,3) = (2,3,1,5,4)$

New sequence $S' = (2,3,1,5,4)$

SWAP SEQUENCE: IN ACTION

Example of Swap Sequence:

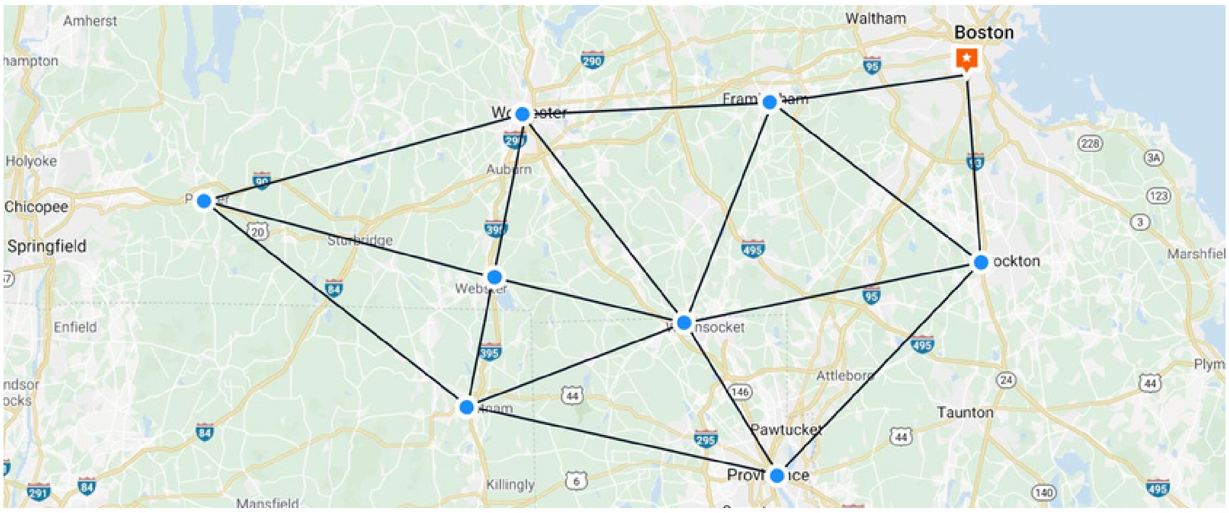
$SS = (SO_1, SO_2, SO_3, \ldots\ldots.., SO_D)$

$S_2 = S_1 + SS_{12} = S_1 + SS = (SO_1, SO_2, SO_3, \ldots\ldots.., SO_D)_{12}$

$S_2 - S_1 = SS = (SO_1, SO_2, SO_3, \ldots\ldots.., SO_D)_{12}$

TSP: AN INTRODUCTION

The Travelling Salesman Problem (TSP) is a combinatorial optimization problem in which we need to find the optimal sequence which represents a tour of a set of cities where the cost of travelling to each city once and returning to the city where the tour was started, is minimized. The problem statements goes as "Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?". It is a NP-hard problem in combinatorial optimization which has its importance in computer science and operations research. There is not computational algorithms which can run in polynomial time to solve this problem so the use of heuristic methods and metaheuristic methods hold great importace for solving this problem. It is important to note that the running time of TSP increases 'superpolynomially' (but less than exponentially) with the increase in the number of cities.



This is a small real life example TSP

FORMULATING THE PROBLEM

Given a finite set of N cities and a distance matrix the formulation of the TSP is given below

$$min \sum_{i \in \pi} (C_{i\pi(i)})$$

Where π runs over all cyclic permutations of N

Or can be formulated as -

$$\min \left(\sum_{i=1}^{n=i+1} C_{v(i)v(i+1)} + C_{v(n)v(1)} \right)$$

where v runs over all permutations of N; here v(k) is the k^th city

in a salesman's tour. If $C_{i,j} = C_{j,i}$ then the problem is symmetric,

otherwise, it is called asymmetric. If $C_{i,k} <= C_{i,j} + C_{j,k}$

DISTANCE MATRIX

CHARACTERISTICS OF DISTANCE MATRIX

The distance matrix of a general TSP is a square symmetric matrix with the dimension of N x N. This happens so because we assume the cost of traveling from city i to city j to be same as city j to city i. This is for a symmetric TSP problem if the cost of traveling up and traveling down are not same then the problem becomes an asymmetric TSP.
This algorithm performed really good to give consistent results with the Standard Travelling Salesman Problem as we increased the number of iterations in the code.

FIRST PROBLEM STATEMENT

At first we solve a symmetric TSP which has 8 cities and the cost matrix is given below -

$$\begin{vmatrix}
0 & 10 & 12 & 5 & 17 & 9 & 13 & 7 \\
10 & 0 & 9 & 20 & 8 & 11 & 3 & 5 \\
12 & 9 & 0 & 14 & 4 & 10 & 1 & 16 \\
5 & 20 & 14 & 0 & 20 & 5 & 8 & 10 \\
17 & 8 & 4 & 20 & 0 & 21 & 4 & 9 \\
9 & 11 & 10 & 5 & 21 & 0 & 2 & 3 \\
13 & 3 & 1 & 28 & 4 & 2 & 0 & 2 \\
7 & 5 & 16 & 10 & 9 & 3 & 2 & 0
\end{vmatrix}$$

SECOND PROBLEM STATEMENT

This is a 10 x 10 Job shop scheduling problem where we have 10 machine and 10 jobs, each machine has to be assigned 1 machine and 1 job can be done by 1 machine only (one on one assignment). Now we have to design the optimal schedule for the job assignment to machines. (The matrix here won't be symmetric as time taken by machine i to complete job j and time taken by machine j to complete job i are not equal).

0	0	1	2	3	4	5	6	7	8	9
0	9	58	44	5	97	58	89	84	77	96
1	77	87	81	85	15	39	21	31	57	73
2	40	49	48	70	10	71	22	80	34	82
3	55	91	17	7	11	75	72	62	47	35
4	64	90	94	75	12	50	71	67	15	20
5	52	57	29	68	58	93	93	70	77	7
6	95	63	82	76	26	6	87	56	48	36
7	36	8	36	76	30	15	84	76	41	78
8	75	78	81	13	54	88	82	13	29	40
9	52	26	6	6	54	82	64	32	54	88

APPLICATION: INTEGRATED CIRCUIT BOARDS

Circuit boards (such as those used in PC's) are drilled with holes for mounting different electronic components. The boards are fed one at a time under a moving drill. The matrix below provides the distances (in mm) between pairs of 6 holes of a specific circuit board.

$$\|d_{ij}\| = \begin{pmatrix} - & 1.3 & .5 & 2.6 & 4.1 & 3.2 \\ 1.3 & - & 3.5 & 4.7 & 3.0 & 5.3 \\ .5 & 3.5 & - & 3.5 & 4.6 & 6.2 \\ 2.6 & 4.7 & 3.5 & - & 3.8 & .9 \\ 4.1 & 3.0 & 4.6 & 3.8 & - & 1.9 \\ 3.2 & 5.3 & 6.2 & .9 & 1.9 & - \end{pmatrix}$$

This happens to be a TSP problem used to optimize the time taken for tthe hole drilling process in the Integrated Circuit board for the components.

OUR APPROACH

Like the general PSO we started by creating a template for particles of our swarm as our particle needs memory to store the data like its personal best, its velocity and its current position. We did that by creating a particle class where a particle stores its current position, personal best position and its current velocity along with the personal best cost and current cost. We defined some methods for the class where a particle can update its personal best position and personal best cost.

Next we defined a PSO class whose object takes in the parameters for the algorithm like number of iterations, size of the population, global best position (since global best is a property of the swarm so it is defined inside the PSO class) and a particle list (the swarm). We initiate the population inside the constructor of the PSO class.

There is a small difference in the classic Swap based PSO and our algorithm, we have one individual in the swarm which is generated by using a greedy brute force search (for having atleast one high fitness individual) in the swarm so that we can have better convergence, which is also reflected in our results.

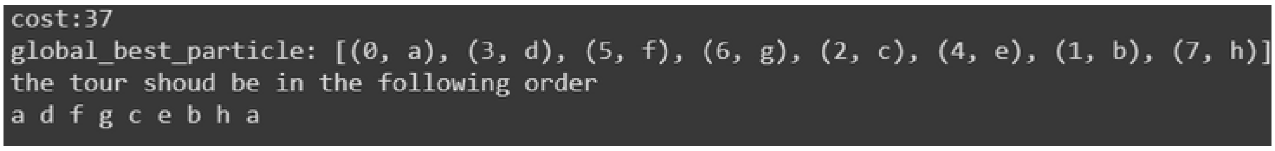
Inside the PSO class we have a run function in which we define all our velocity updating (Swap Sequence operation) and the position updating (Swap operation) for each partilce by iterating over the complete swarm.

We also have introduced a degree of randomness in the choice of swap sequence operation for updating the velocity. we have a 'p_best_prob' as an argument of the PSO class where we can tweak that value to have some velocities of the previous epoch in the next epoch too .

We have a main function where our PSO object is initiated with corresponding values.

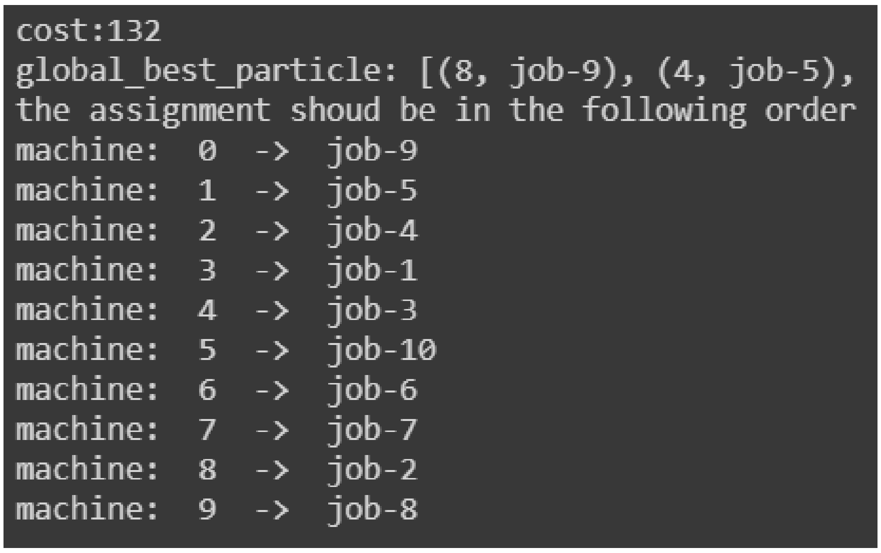
RESULTS

Problem statement 1 (TSP with 8 cities)
The cities have been named a, b, c, d, e, f, g, h in the order of their index in the distance matrix. City 'a' corresponds to index 0 of the matrix, City 'b' corresponds to index 1 and so on till City 'g' corresponding to index 7.



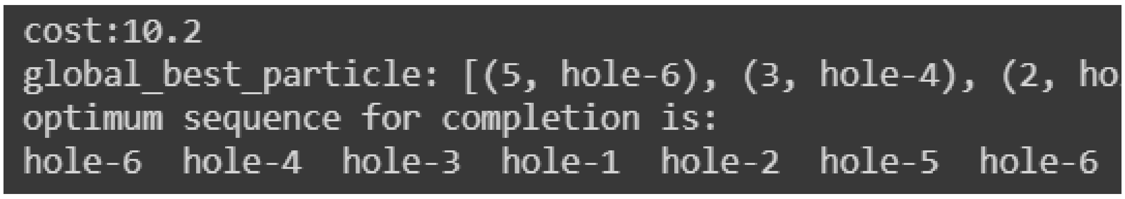
```
cost:37
global_best_particle: [(0, a), (3, d), (5, f), (6, g), (2, c), (4, e), (1, b), (7, h)]
the tour shoud be in the following order
a d f g c e b h a
```

Problem statement 2 (10 x 10 Job shop scheduling)
The job shop scheduling is done using our algorithm, the machines 1 - 10 correspond to the indexed matrix 0 - 9, and similarly for the jobs. The input is an asymmetric matrix having a dimension 10 x 10



```
cost:132
global_best_particle: [(8, job-9), (4, job-5),
the assignment shoud be in the following order
machine:  0  ->  job-9
machine:  1  ->  job-5
machine:  2  ->  job-4
machine:  3  ->  job-1
machine:  4  ->  job-3
machine:  5  ->  job-10
machine:  6  ->  job-6
machine:  7  ->  job-7
machine:  8  ->  job-2
machine:  9  ->  job-8
```

Problem statement 3 (Integrated Circuit board time optimization)
In this problem we optimize the time by choosing the optimum sequence of making holes in the Integrated Circuit, this is an industrial problem where we might have to do this for optimizing the time taken so as to minimize the wages and cost of operating the machine.


```
cost:10.2
global_best_particle: [(5, hole-6), (3, hole-4), (2, ho
optimum sequence for completion is:
hole-6  hole-4  hole-3  hole-1  hole-2  hole-5  hole-6
```

CONCLUSION

Our project implements a variant of Swap operator based Particle Swarm Optimization (PSO) method for solving TSP and TSP like problems. For TSP each particle represents a complete tour and velocity is measured by as a Swap Sequence (SS) consisting of several Swap Operators (SO). In this method, a new tour is calculated using after applying a complete SS with all its SO's of the velocity calculated. Since every swap operation gives new solution, we checked all the solutions that found for the velocity and the best one is considered for the update of the solution. Updating the velocity based on a personal update probability helps us to get some velocity of the previous iterations to the new or next iteration which helps us to maintain diversity and prevent premature convergence.

All the results which were given by the algorithm were optimal (or near optimal) and the results were consistent with what the literature claims. This is a very robust and relatively fast algorithm when it is compared to other evolutionary algorithms. The computation time taken is less as observed by us.
There have been many other methods for solving TSP or TSP-like problems in computer science where we have dynamic programming method, we can also formulate a TSP into a linear programming model and solve it using variety of Linear Programming Problem methods but this metaheuristic technique gives us a new horizon to solve these kind of problems in an effective manner and along with that using this algorithm we get a lot of other near optimal solutions which can be useful at times.

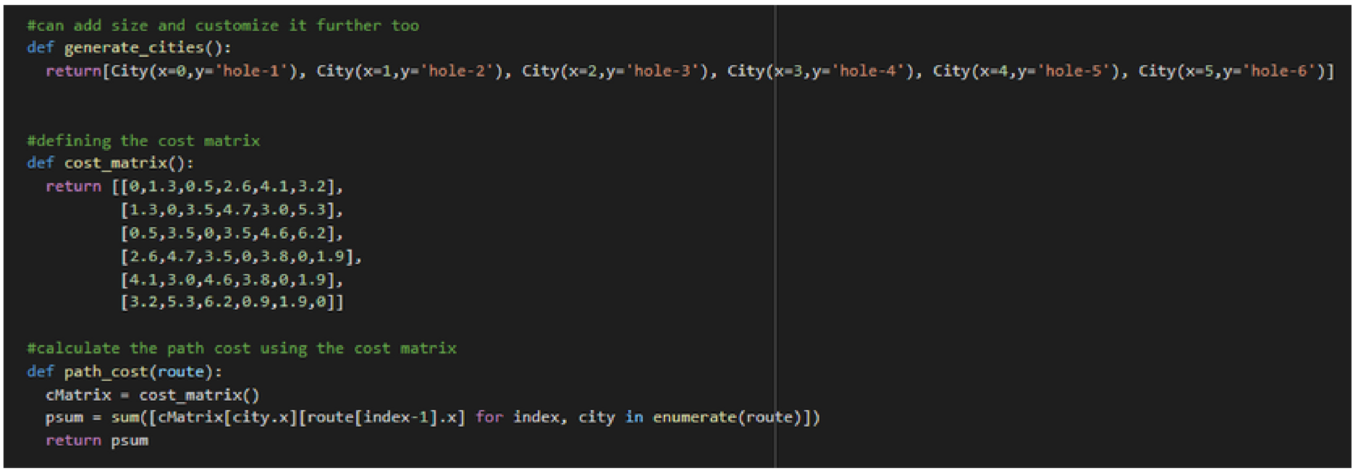
BIBLIOGRAPHY

- *J.K. LENSTRA and A.H.G. RINNOY KAN "Some simple Applications of Travelling Salesman Problem"*
- *N.K. JAIN, Uma NANGIA, Jyoti JAIN "A review of Particle Swarm Optimization"*
- *E.C. LASKARI, K.E. PARSOPOULOS and M.N. VRAHATIS "Particle Swarm Optimization for Integer Programming Problem"*
- *M.A.H. AKHAND, Shahina AKTER, S. Sazzadur RAHMAN, M.M. Hafizur RAHMAN "Particle Swarm Optimization with Partial Search to solve Travelling Salesman Problem"*
- *H. A. TAHA "Operations Research: An Introduction 10th ed."*

APPENDIX

CODE SNIPPETS

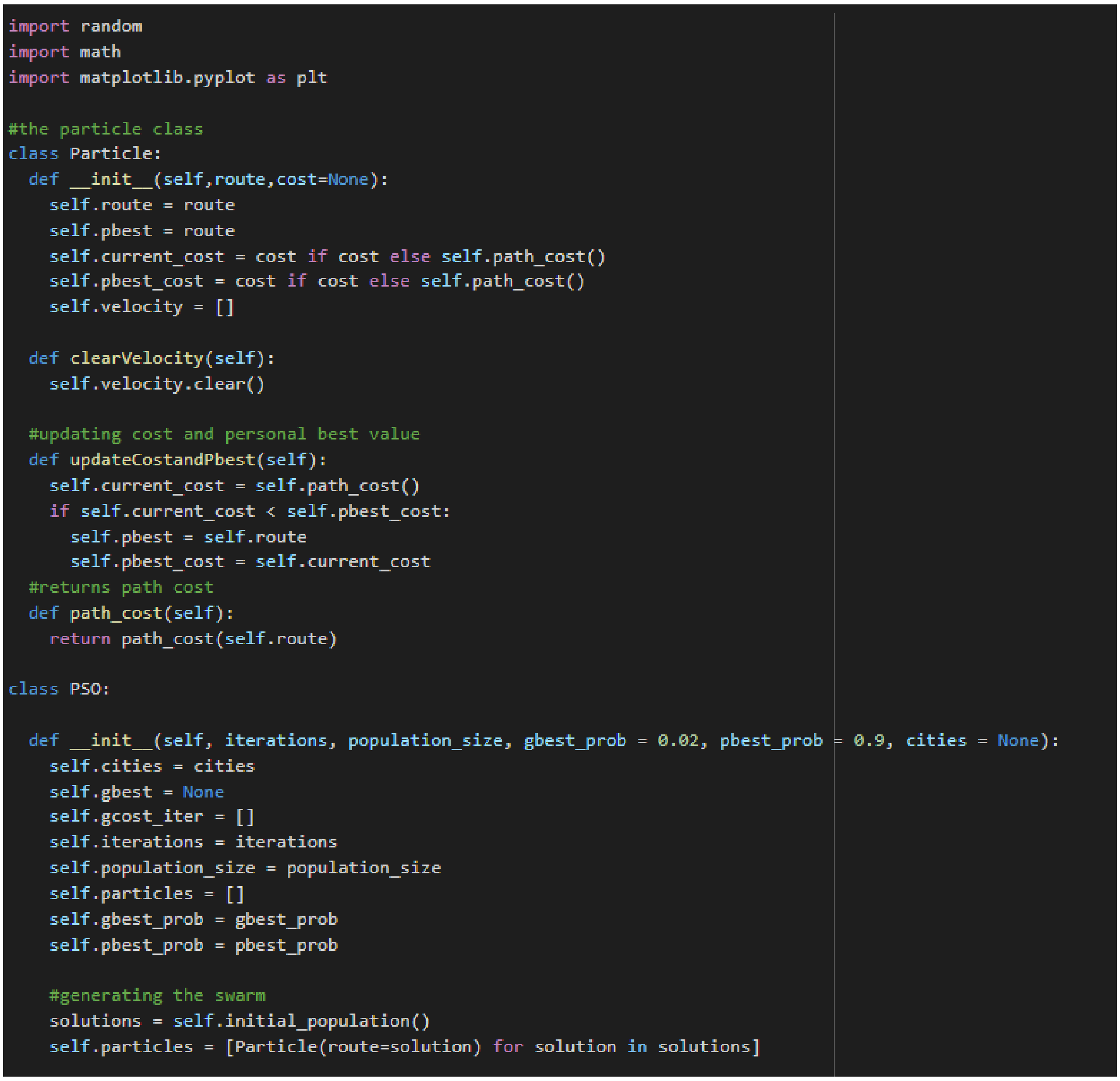
SOME UTILITIES DEFINED FOR THE ALGORITHM



```
#can add size and customize it further too
def generate_cities():
  return[City(x=0,y='hole-1'), City(x=1,y='hole-2'), City(x=2,y='hole-3'), City(x=3,y='hole-4'), City(x=4,y='hole-5'), City(x=5,y='hole-6')]


#defining the cost matrix
def cost_matrix():
  return [[0,1.3,0.5,2.6,4.1,3.2],
          [1.3,0,3.5,4.7,3.0,5.3],
          [0.5,3.5,0,3.5,4.6,6.2],
          [2.6,4.7,3.5,0,3.8,0,1.9],
          [4.1,3.0,4.6,3.8,0,1.9],
          [3.2,5.3,6.2,0.9,1.9,0]]

#calculate the path cost using the cost matrix
def path_cost(route):
  cMatrix = cost_matrix()
  psum = sum([cMatrix[city.x][route[index-1].x] for index, city in enumerate(route)])
  return psum
```

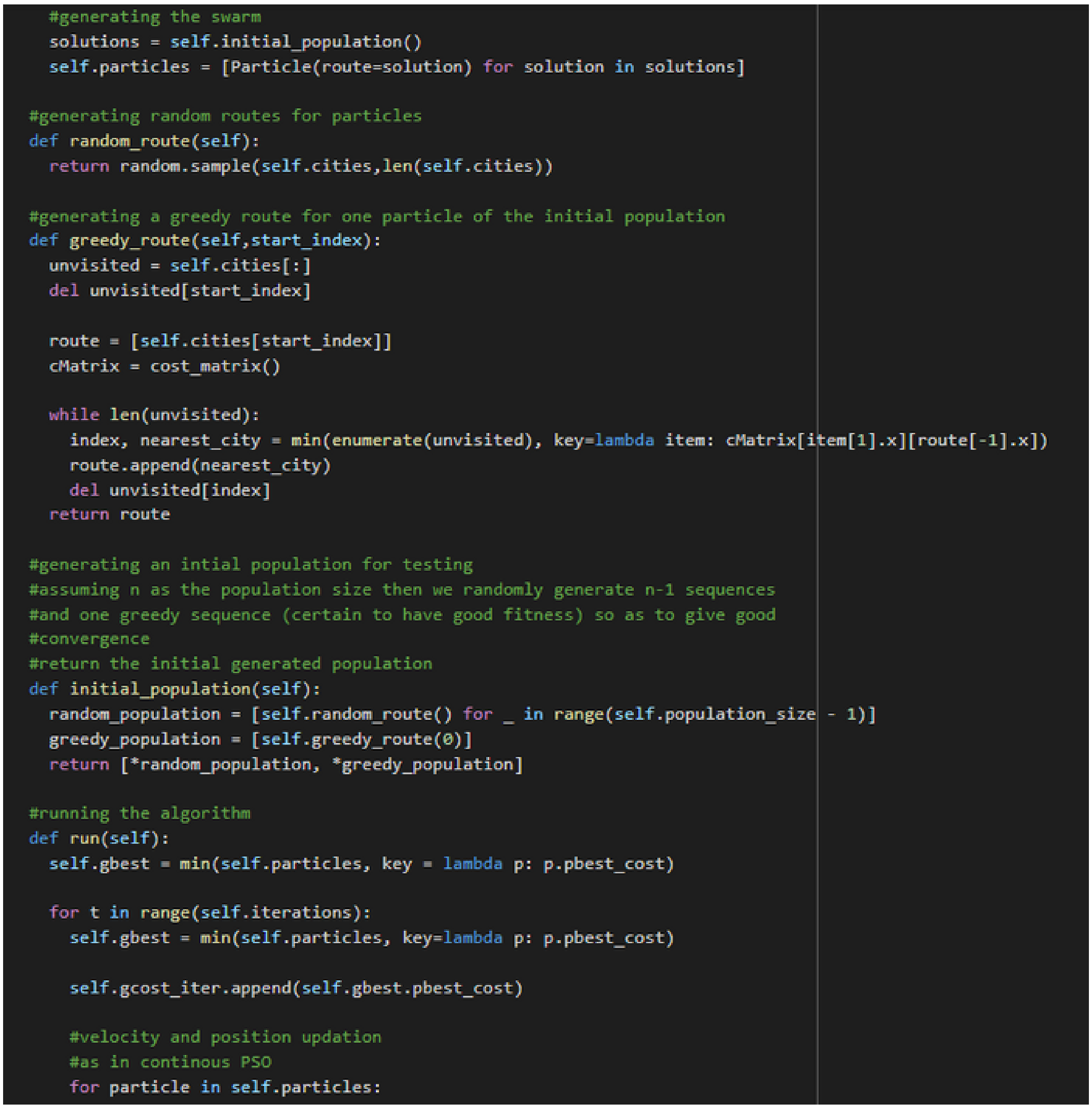
IMPLEMENTATION OF ALGORITHM: CODE

PART - 1



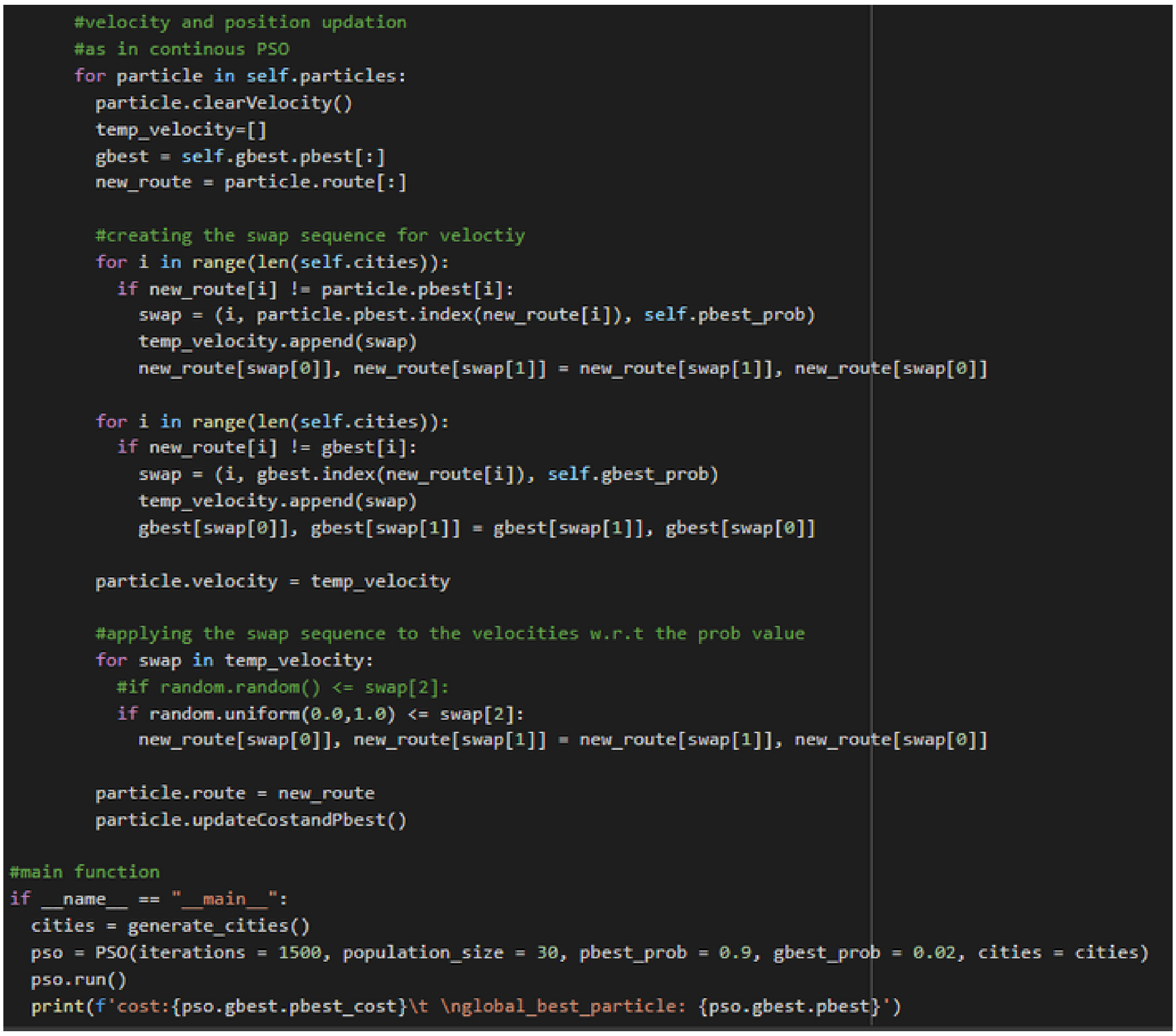
```python
import random
import math
import matplotlib.pyplot as plt

#the particle class
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 clearVelocity(self):
    self.velocity.clear()

  #updating cost and personal best value
  def updateCostandPbest(self):
    self.current_cost = self.path_cost()
    if self.current_cost < self.pbest_cost:
      self.pbest = self.route
      self.pbest_cost = self.current_cost
  #returns path cost
  def path_cost(self):
    return path_cost(self.route)

class PSO:

  def __init__(self, iterations, population_size, gbest_prob = 0.02, pbest_prob = 0.9, cities = None):
    self.cities = cities
    self.gbest = None
    self.gcost_iter = []
    self.iterations = iterations
    self.population_size = population_size
    self.particles = []
    self.gbest_prob = gbest_prob
    self.pbest_prob = pbest_prob

    #generating the swarm
    solutions = self.initial_population()
    self.particles = [Particle(route=solution) for solution in solutions]
```

PART - 2



```python
    #generating the swarm
    solutions = self.initial_population()
    self.particles = [Particle(route=solution) for solution in solutions]

  #generating random routes for particles
  def random_route(self):
    return random.sample(self.cities,len(self.cities))

  #generating a greedy route for one particle of the initial population
  def greedy_route(self,start_index):
    unvisited = self.cities[:]
    del unvisited[start_index]

    route = [self.cities[start_index]]
    cMatrix = cost_matrix()

    while len(unvisited):
      index, nearest_city = min(enumerate(unvisited), key=lambda item: cMatrix[item[1].x][route[-1].x])
      route.append(nearest_city)
      del unvisited[index]
    return route

  #generating an intial population for testing
  #assuming n as the population size then we randomly generate n-1 sequences
  #and one greedy sequence (certain to have good fitness) so as to give good
  #convergence
  #return the initial generated population
  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]

  #running the algorithm
  def run(self):
    self.gbest = min(self.particles, key = lambda p: p.pbest_cost)

    for t in range(self.iterations):
      self.gbest = min(self.particles, key=lambda p: p.pbest_cost)

      self.gcost_iter.append(self.gbest.pbest_cost)

      #velocity and position updation
      #as in continous PSO
      for particle in self.particles:
```

PART - 3



```
      #velocity and position updation
      #as in continous PSO
      for particle in self.particles:
        particle.clearVelocity()
        temp_velocity=[]
        gbest = self.gbest.pbest[:]
        new_route = particle.route[:]

        #creating the swap sequence for veloctiy
        for i in range(len(self.cities)):
          if new_route[i] != particle.pbest[i]:
            swap = (i, particle.pbest.index(new_route[i]), self.pbest_prob)
            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_prob)
            temp_velocity.append(swap)
            gbest[swap[0]], gbest[swap[1]] = gbest[swap[1]], gbest[swap[0]]

        particle.velocity = temp_velocity

        #applying the swap sequence to the velocities w.r.t the prob value
        for swap in temp_velocity:
          #if random.random() <= swap[2]:
          if random.uniform(0.0,1.0) <= swap[2]:
            new_route[swap[0]], new_route[swap[1]] = new_route[swap[1]], new_route[swap[0]]

        particle.route = new_route
        particle.updateCostandPbest()

#main function
if __name__ == "__main__":
  cities = generate_cities()
  pso = PSO(iterations = 1500, population_size = 30, pbest_prob = 0.9, gbest_prob = 0.02, cities = cities)
  pso.run()
  print(f'cost:{pso.gbest.pbest_cost}\t \nglobal_best_particle: {pso.gbest.pbest}')
```