ANNAMALAI UNIVERSITY

FACULTY OF ENGINEERING AND TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

B.E. COMPUTER SCIENCE & ENGINEERING

(Artificial Intelligence and Machine Learning)

SEMESTER – VII

AICP706 – OPTIMIZATION TECHNIQUES LAB

LABORATORY RECORD

(JULY 2022 – DECEMBER 2022)

Name: ……………………………………………………

Reg. No : ………………………………………………….

ANNAMALAI UNIVERSITY

FACULTY OF ENGINEERING AND TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

B.E. COMPUTER SCIENCE AND ENGINEERING
(ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING)

VII SEMESTER

AICP706 – OPTIMIZATION TECHNIQUES LAB

Bonafide Certificate

Certified that this is the Bonafide Record of work done by Mr./Ms. ____ Reg. No. __________________ of VII semester B.E. Computer Science and Engineering (Artificial Intelligence and Machine Learning) in the **AICP706 – Optimization Techniques Lab** During the odd semester (July 2022 – December 2022).

Internal Examiner Staff In-Charge

Place: Annamalai Nagar External Examiner

Date:

Vision and Mission of the Department

VISION

To provide a congenial ambience for individuals to develop and blossom as academically superior, socially conscious and nationally responsible citizens.

MISION

M1: Impart high quality computer knowledge to the students through a dynamic scholastic environment wherein they learn to develop technical, communication and leadership skills to bloom as a versatile professional.

M2: Develop life-long learning ability that allows them to be adaptive and responsive to the changes in career, society, technology, and environment

M3: Build student community with high ethical standards to undertake innovative research and development in thrust areas of national and international needs

M4: Expose the students to the emerging technological advancements for meeting the demands of the industry.

Program Educational Objectives (PEOs)

PEOs	PEO Statements
PEO1	To prepare graduates with potential to get employed in the right role and/or become entrepreneurs to contribute to the society.
PEO2	To provide the graduates with the requisite knowledge to pursuehigher education and carry out research in the field of Computer Science.
PEO3	To equip the graduates with the skills required to stay motivated andadapt to the dynamically changing world so as to remain successful in their career.
PEO4	To train the graduates with effectively, work collaboratively and exhibit high levels of professionalism and ethical responsibility.

Program Outcomes (PO):

Sl. No.	Program Outcomes
PO1	**Engineering Knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
PO2	**Problem Analysis:** Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences and engineering sciences.
PO3	**Design/Development of Solutions:** Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
PO4	**Conduct Investigations of Complex Problems:** Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
PO5	**Modern Tool Usage:** Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
PO6	**The Engineer and Society:** Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
PO7	**Environment and Sustainability:** Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
PO8	**Ethics:** Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
PO9	**Individual and Team Work:** Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
PO10	**Communication:** Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
PO11	**Project Management and Finance:** Demonstrate knowledge and understanding of the engineering and management principles and apply these to one’s own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
PO12	**Life-long Learning:** Recognize the need for, and have the preparation and ability to engage in independent and lifelong learning in the broadest context of technological change.

COURSE OUTCOMES:

At the end of this course, the students will be able to

1. Understand and implement constrained and unconstrained optimization problems.
2. Implement biogeography based optimization techniques.
3. Appreciate the principles of multi objective optimization techniques.

Mapping of Course Outcomes with Program Outcomes												
	PO1	**PO2**	**PO3**	**PO4**	**PO5**	**PO6**	**PO7**	**PO8**	**PO9**	**PO10**	**PO11**	**PO12**
CO1	-	-	2	-	2	-	-	-	-	-	-	-
CO2	-	3	3	1	3	1	-	-	-	-	-	2
CO3	2	2	-	-	-	-	-	-	-	2	-	2

Rubric for CO3

Rubric for CO3 in Laboratory Courses				
Rubric	**Distribution of 10 Marks for CIE/SEE Evaluation Out of 40/60 Marks**			
	Up To 2.5 Marks	**Up To 5 Marks**	**Up To 7.5 Marks**	**Up To 10 marks**
Demonstrate an ability to listen and answer the viva questions related to programming skills needed for solving real-world problems in Computer Science and Engineering.	Poor listening and communication skills. Failed to relate the programming skills needed for solving the problem.	Showed better communication skill by relating the problem with the programming skills acquired but the description showed serious errors.	Demonstrated good communication skills by relating the problem with the programming skills acquired with few errors.	Demonstrated excellent communication skills by relating the problem with the programming skills acquired and have been successful in tailoring the description.

INDEX

1. GENETIC ALGORITHM FOR CONTINUOUS OPTIMIZATION

Aim:

To write a python code to implement genetic algorithm for continuous function optimization.

Algorithm:

1. Initialize the population **P** in the given range:

   The process begins with a set of individuals which is called a **Population**. Each individual is a solution to the problem you want to solve.

2. Decode each individual of the population to convert them from bits to continuous value.
3. Compute the fitness of the population F.
4. Repeat the following steps for number of iterations:
   i. Select the parents with good fitness score.
   ii. Select a random crossover point.
   iii. Crossover the parents and generate new population:

      Child1 = Parent1[:crossover_point] + Parent2[crossover_point:]

      Child2 = Parent2[:crossover_point] + Parent1[crossover_point:]

   iv. **Mutation:** Flip the bits of each in each of at random positions.

      Bitstring[i] = 1 – bitstring[i]

   v. Compute the fitness of the new population.
5. Return the best solution and its score.

Source code:

```
from numpy.random import randint

from numpy.random import rand
```

```
# objective function
def objective(x):
    return x[0]**2.0 + x[1]**2.0


# decode bitstring to numbers
def decode(bounds, n_bits, bitstring):
    decoded = list()
    largest = 2**n_bits
    for i in range(len(bounds)):
        # extract the substring
        start, end = i * n_bits, (i * n_bits)+n_bits
        substring = bitstring[start:end]
        # convert bitstring to a string of chars
        chars = ''.join([str(s) for s in substring])
        # convert string to integer
        integer = int(chars, 2)
        # scale integer to desired range
        value = bounds[i][0] + (integer/largest) * (bounds[i][1] - bounds[i][0])
         # store
        decoded.append(value)
    return decoded


# tournament selection
def selection(pop, scores, k=3):
    # first random selection
    selection_ix = randint(len(pop))
    for ix in randint(0, len(pop), k-1):
        # check if better (e.g. perform a tournament)
        if scores[ix] < scores[selection_ix]:
            selection_ix = ix
    return pop[selection_ix]
```

```
# crossover two parents to create two children
def crossover(p1, p2, r_cross):
    # children are copies of parents by default
    c1, c2 = p1.copy(), p2.copy()
    # check for recombination
    if rand() < r_cross:
        # select crossover point that is not on the end of the string
        pt = randint(1, len(p1)-2)
        # perform crossover
        c1 = p1[:pt] + p2[pt:]
        c2 = p2[:pt] + p1[pt:]
    return [c1, c2]

# mutation operator
def mutation(bitstring, r_mut):
    for i in range(len(bitstring)):
        # check for a mutation
        if rand() < r_mut:
            # flip the bit
            bitstring[i] = 1 - bitstring[i]

# genetic algorithm
def genetic_algorithm(objective, bounds, n_bits, n_iter, n_pop, r_cross, r_mut):
    # initial population of random bitstring
    pop = [randint(0, 2, n_bits*len(bounds)).tolist() for _ in range(n_pop)]
    # keep track of best solution
    best, best_eval = 0, objective(decode(bounds, n_bits, pop[0]))
    # enumerate generations
    for gen in range(n_iter):
        # decode population
        decoded = [decode(bounds, n_bits, p) for p in pop]
        # evaluate all candidates in the population
        scores = [objective(d) for d in decoded]
        # check for new best solution
        for i in range(n_pop):
            if scores[i] < best_eval:
                best, best_eval = pop[i], scores[i]
                    print(">iteration %d, new best f(%s) = %f" % (gen,  decoded[i],
scores[i]))
        # select parents
        selected = [selection(pop, scores) for _ in range(n_pop)]
        # create the next generation
        children = list()
        for i in range(0, n_pop, 2):
            # get selected parents in pairs
            p1, p2 = selected[i], selected[i+1]
            # crossover and mutation
            for c in crossover(p1, p2, r_cross):
                # mutation
                mutation(c, r_mut)
                # store for next generation
                children.append(c)
        # replace population
        pop = children
    return [best, best_eval]
```

```
# define range for input
bounds = [[-5.0, 5.0], [-5.0, 5.0]]
# define the total iterations
n_iter = 100
# bits per variable
n_bits = 16
# define the population size
n_pop = 100
# crossover rate
r_cross = 0.9
# mutation rate
r_mut = 1.0 / (float(n_bits) * len(bounds))
# perform the genetic algorithm search
print(f'Starting genetic algorithm\n')
best, score = genetic_algorithm(objective, bounds, n_bits, n_iter, n_pop, r_cross,
r_mut)
decoded = decode(bounds, n_bits, best)
print(f'\nGenetic algorithm completed\n')
print(f'Best solution: {decoded}')
print(f'Fitness score of the best solution: {score:.5f}')
```

Output:

```
Starting genetic algorithm
>iteration 0, new best f([-2.047271728515625, -1.97540283203125]) = 8.093538
>iteration 0, new best f([0.15594482421875, -1.57745361328125]) = 2.512679
>iteration 0, new best f([-0.55755615234375, -1.076812744140625]) = 1.470395
>iteratio0, new best f([-0.089263916015625, 0.421295166015625]) = 0.185458
>iteration 1, new best f([-0.089263916015625, 0.108795166015625]) = 0.019804
>iteration 4, new best f([-0.0140380859375, 0.016632080078125]) = 0.000474
>iteration 7, new best f([-0.0189208984375, 0.00213623046875]) = 0.000363
```

```
>iteration 8, new best f([-0.0128173828125, 0.000457763671875]) = 0.000164
>iteration 9, new best f([0.009307861328125, 0.006103515625]) = 0.000124
>iteration 11, new best f([-0.00396728515625, 0.0018310546875]) = 0.000019
>iteration 12, new best f([-0.00396728515625, 0.000457763671875]) = 0.000016
>iteration 15, new best f([-0.0030517578125, 0.001983642578125]) = 0.000013
>iteration 16, new best f([-0.00030517578125, 0.001373291015625]) = 0.000002
>iteration 17, new best f([-0.000152587890625, 0.001373291015625]) = 0.000002
>iteration 19, new best f([-0.001068115234375, 0.000457763671875]) = 0.000001
>iteration 19, new best f([-0.00030517578125, 0.00030517578125]) = 0.000000
>iteration 24, new best f([-0.000152587890625, 0.00030517578125]) = 0.000000
>iteration 30, new best f([-0.000152587890625, 0.000152587890625]) = 0.000000
>iteration 39, new best f([-0.000152587890625, 0.0]) = 0.000000

Genetic algorithm completed

Best solution: [-0.000152587890625, 0.0]
Fitness score of the best solution: 0.00000
```

Result:

The python code to implement genetic algorithm for continuous function optimization has been successfully implemented and verified successfully.

2. GENETIC ALGORITHM FOR BINARY OPTIMIZATION

Aim:

To write a python code to implement genetic algorithm for binary function optimization.

Algorithm:

1. Initialize the population **P** in the given range:

   The process begins with a set of individuals which is called a **Population**. Each individual is a solution to the problem you want to solve

2. Compute the fitness of the population F.
3. Repeat the following steps for number of iterations:
   i. Select the parents with good fitness score.
   ii. Select a random crossover point.
   iii. Crossover the parents and generate new population:

      Child1 = Parent1[:crossover_point] + Parent2[crossover_point:]

      Child2 = Parent2[:crossover_point] + Parent1[crossover_point:]

   iv. **Mutation:** Flip the bits of each in each of at random positions.

      Bitstring[i] = 1 – bitstring[i]

   v. Compute the fitness of the new population.
4. Return the best solution and its score.

Source code:

```
from numpy.random import randint

from numpy.random import rand


# objective function

def onemax(x):

    return -sum(x)
```

```
# tournament selection
def selection(pop, scores, k=3):
    # first random selection
    selection_ix = randint(len(pop))
    for ix in randint(0, len(pop), k-1):
        # check if better (e.g. perform a tournament)
        if scores[ix] < scores[selection_ix]:
            selection_ix = ix
    return pop[selection_ix]

# crossover two parents to create two children
def crossover(p1, p2, r_cross):
    # children are copies of parents by default
    c1, c2 = p1.copy(), p2.copy()
    # check for recombination
    if rand() < r_cross:
        # select crossover point that is not on the end of the string
        pt = randint(1, len(p1)-2)
        # perform crossover
        c1 = p1[:pt] + p2[pt:]
        c2 = p2[:pt] + p1[pt:]
    return [c1, c2]

# mutation operator
def mutation(bitstring, r_mut):
    for i in range(len(bitstring)):
        # check for a mutation
        if rand() < r_mut:
            # flip the bit
            bitstring[i] = 1 - bitstring[i]

# genetic algorithm
def genetic_algorithm(objective, n_bits, n_iter, n_pop, r_cross, r_mut):
    # initial population of random bitstring
    pop = [randint(0, 2, n_bits).tolist() for _ in range(n_pop)]
    # keep track of best solution
    best, best_eval = 0, objective(pop[0])
    # enumerate generations
    for gen in range(n_iter):
        # evaluate all candidates in the population
        scores = [objective(c) for c in pop]
        # check for new best solution
        for i in range(n_pop):
            if scores[i] < best_eval:
                best, best_eval = pop[i], scores[i]
                print(">iteration %d, new best f(%s) = %.3f" % (gen,  pop[i], scores[i]))
        # select parents
        selected = [selection(pop, scores) for _ in range(n_pop)]
        # create the next generation
        children = list()
        for i in range(0, n_pop, 2):
            # get selected parents in pairs
            p1, p2 = selected[i], selected[i+1]
            # crossover and mutation
            for c in crossover(p1, p2, r_cross):
                # mutation
                mutation(c, r_mut)
                # store for next generation
                children.append(c)
        # replace population
        pop = children
    return [best, best_eval]

# define the total iterations
n_iter = 100
# bits
n_bits = 20
# define the population size
n_pop = 100
# crossover rate
r_cross = 0.9
# mutation rate
r_mut = 1.0 / float(n_bits)
# perform the genetic algorithm search
print(f'Starting genetic algorithm\n')
best, score = genetic_algorithm(onemax, n_bits, n_iter, n_pop, r_cross, r_mut)
print(f'\nGenetic algorithm completed\n')
print(f'Best solution: {best}')
print(f'Fitness score of the best solution: {score:.5f}')
```

Output:

```
Starting genetic algorithm
>iteration 0, new best f([1, 1, 0, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 0, 1]) = -14.000
>iteration 0, new best f([1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0]) = -16.000
>iteration 1, new best f([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 0]) = -17.000
>iteration 1, new best f([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1]) = -18.000
>iteration 2, new best f([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1]) = -19.000
 >iteration 4, new best f([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]) = -20.000
Genetic algorithm completed

Best solution: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
Fitness score of the best solution: -20.00000
```

Result:

The python code to implement genetic algorithm for continuous function optimization has been successfully implemented and verified successfully.

3. Simulated Annealing

Aim:

To write a python code to implement simulated annealing algorithm for function optimization.

Algorithm:

1. Pick a random value for the variable within the given bounds as best value and current value.
2. Compute the fitness of the variable as best evaluation and current evaluation.
3. Repeat the following steps for number of iterations:
   i. Initialize a candidate value using the current value.
   ii. Compute the fitness value of the candidate value as candidate evaluation.
   iii. If the candidate evaluation is better than best evaluation, Replace best value with candidate value.
   iv. Reduce the temperature.
   v. Calculate the metropolis value using candidate value, current value and temperature.

   $$\text{Metropolis value} = e^{-(\text{candidate evaluation} - \text{current evaluation})/\text{temperature}}$$

   vi. If difference between current value and candidate value is less than 0 (or) metropolis value is greater than some random value, replace current value with candidate value.
4. Return the best value and best evaluation.

Source code:

```
from numpy import asarray

from numpy import exp

from numpy.random import randn

from numpy.random import rand
from numpy.random import seed

# objective function
def objective(x):
    return x[0]**2.0

# simulated annealing algorithm
def simulated_annealing(objective, bounds, n_iterations, step_size, temp):
    # generate an initial point
    best = bounds[:, 0] + rand(len(bounds)) * (bounds[:, 1] - bounds[:, 0])
    # evaluate the initial point
    best_eval = objective(best)
    # current working solution
    curr, curr_eval = best, best_eval
    # run the algorithm
    for i in range(n_iterations):
        # take a step
        candidate = curr + randn(len(bounds)) * step_size
        # evaluate candidate point
        candidate_eval = objective(candidate)
        # check for new best solution
        if candidate_eval < best_eval:
            # store new best point
            best, best_eval = candidate, candidate_eval
            # report progress
            print('>iteration %d: f(%s) = %.5f' % (i, best, best_eval))
        # difference between candidate and current point evaluation
        diff = candidate_eval - curr_eval
        # calculate temperature for current epoch
        t = temp / float(i + 1)
        # calculate metropolis acceptance criterion
        metropolis = exp(-diff / t)
        # check if we should keep the new point
        if diff < 0 or rand() < metropolis:
            # store the new current point
            curr, curr_eval = candidate, candidate_eval
    return [best, best_eval]

# seed the pseudorandom number generator
seed(1)
# define range for input
bounds = asarray([[-5.0, 5.0]])
# define the total iterations
n_iterations = 1000
# define the maximum step size
step_size = 0.1
# initial temperature
temp = 10
# perform the simulated annealing search
print('Starting simulated annealing algorithm\n')
best, score = simulated_annealing(objective, bounds, n_iterations, step_size,
temp)
print('\nSimulated annealing completed\n')
print(f'Best solution: {best}')
print(f'Fitness score of the best solution: {score:.5f}')
```

Output:

```
Starting simulated annealing algorithm

>34 f([-0.78753544]) = 0.62021
>35 f([-0.76914239]) = 0.59158
>37 f([-0.68574854]) = 0.47025
>39 f([-0.64797564]) = 0.41987
>40 f([-0.58914623]) = 0.34709
>41 f([-0.55446029]) = 0.30743
>42 f([-0.41775702]) = 0.17452
>43 f([-0.35038542]) = 0.12277
>50 f([-0.15799045]) = 0.02496

Simulated annealing completed

Best solution: [0.00013605]
Fitness score of the best solution: 0.00000
```

Result:

The python code to implement simulated annealing algorithm for function optimization has been successfully implemented and verified successfully.

4. ANT COLONY OPTIMIZATION

Aim:

To write a python code to implement ant colony optimization algorithm to optimize the travelling salesman problem.

Algorithm:

1. Initialize
2. For t = 1 to iteration_number do
   I. For k = 1 to l do
      i. Repeat until ant k has completed a tour Select the city j to be visited next with probability p_{ij} given by

$$p_{ij}^k = \begin{cases} \dfrac{[\tau_{ij}]^\alpha \cdot [\nu_{ij}]^\beta}{\sum_{s \in \text{allowed}_k} [\tau_{is}]^\alpha \cdot [\nu_{is}]^\beta} & j \in \text{allowed}_k \\ 0 & \text{otherwise} \end{cases}$$

      ii. Calulate L_k
   II. Update the trail levels according to the below equation

$$\tau_{ij}(t+1) = \rho \cdot \tau_{ij}(t) + \Delta\tau_{ij}$$

$$\Delta\tau_{ij} = \sum_{k=1}^{l} \Delta\tau_{ij}^k$$

$$\Delta\tau_{ij}^k = \begin{cases} Q/L_k & \text{if an ant travels on the edge}(i,j) \\ 0 & \text{otherwise} \end{cases}$$

3. End

Source code:

```
import numpy as np

import networkx as nx

import matplotlib.pyplot as plt

from itertools import combinations


def plot_graph(g,title="",highlight_edges=[]):
    pos = nx.get_node_attributes(g,"pos")
    plt.figure(figsize=(17,17))
    plt.title(title)
    nx.draw(g,pos=pos,labels = {x:x for x in g.nodes},width=2)
    weights = nx.get_edge_attributes(g,"weight")
    # draw labels for edges
    nx.draw_networkx_edge_labels(g,pos,edge_labels=weights,label_pos=.4);

    # highlight highlighted_edges
    nx.draw_networkx_edges(g,pos,edgelist =
highlight_edges,edge_color="r",width=3)
    # highlight labels of highlighted edges
    nx.draw_networkx_edge_labels(
        g,pos,
        edge_labels={
            e:w
            for e,w in weights.items()
            if e in map(lambda x:tuple(sorted(x)),highlight_edges)
        },
        font_color="r",
        label_pos=.4
    )
    plt.show()


def zero_divide(a,b):

    '''Utility function to remove divide by zero error'''

    return np.divide(a,b, out = np.zeros_like(a),where=b!=0)
```

```
class ACOTSP:
    def __init__(self,g,n_ants = 100, alpha=1,beta=5,Q=100,rho = .6) -> None:
        self.g = g # networkx graph
        self.n_nodes = len(g.nodes)
        distances = nx.to_numpy_array(g)
        self.visibility = zero_divide(np.ones_like(distances),distances) # visibility
\nu_ij= 1/d_ij without self
        self.n_ants = n_ants
        self.alpha = alpha
        self.beta = beta
        self.Q = Q
        self.rho= rho
        self.phe_trail = np.ones((self.n_nodes,self.n_nodes))

    def compute_prob(self,visited):
        self.prob = self.phe_trail**self.alpha*self.visibility**self.beta
        self.prob[:,np.array(list(visited))] = 0 # zeroing out visited nodes/columns
        prob_sum = self.prob.sum(-1,keepdims=True)
        self.prob = zero_divide(self.prob,prob_sum) # normalization - divide by
row sum
        return self.prob

    def initialize(self):
        nodes = list(self.g.nodes)
        self.ant_pos = np.random.choice(nodes,self.n_ants)

    def path_length(self,path):
        edge_weights = nx.get_edge_attributes(self.g,"weight")
        return sum((edge_weights[tuple(sorted(edge))] for edge in path))

    def ant_tour(self,k):
        current = self.ant_pos[k]
        visited = {current}
        path = []
        self.compute_prob(visited)
        while True:
            prev = current
            current = np.random.choice(self.n_nodes, p = self.prob[current])
            visited.add(current)
            path.append((prev,current))
            if np.all(self.prob[current]==0):
                break
            self.compute_prob(visited)
        self.paths[k] = path

    def update_pheromone_trails(self):
        d_phe_trail = np.zeros((self.n_nodes,self.n_nodes,self.n_ants))
        for k in range(self.n_ants):
            if len(self.paths[k])==self.n_nodes-1:
                for i,j in self.paths[k]:
                    d_phe_trail[i,j,k]= d_phe_trail[j,i,k] = self.Q/self.path_lengths[k]
        d_phe_trail = d_phe_trail.sum(-1)
        self.phe_trail = self.rho*self.phe_trail + d_phe_trail
    def run(self,n_iter = 1):
        self.initialize()
```

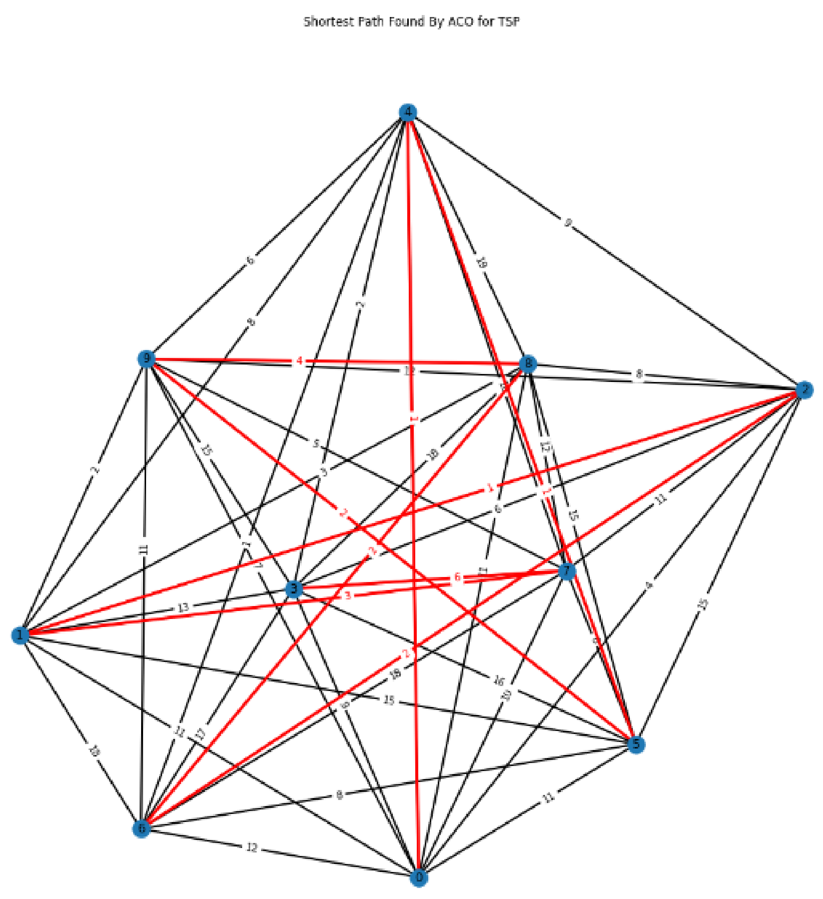
```
        for t in range(n_iter):
            self.paths= [0 for _ in range(self.n_ants)]
            for k in range(self.n_ants):
                self.ant_tour(k)
            self.path_lengths = list(map(self.path_length,self.paths))
            self.update_pheromone_trails()
            self.hamiltonian_paths = [path for path in self.paths if len(path) == 
self.n_nodes -1]
            self.hamiltonian_path_lengths = 
list(map(self.path_length,self.hamiltonian_paths))
            print("Shortest hamiltonian Path length:",self.min_path_length)


    @property
    def min_path_length(self):
        try:
            return min(self.hamiltonian_path_lengths)
        except:
            return None
    @property
    def min_path(self):
        try:
            return self.hamiltonian_paths[np.argmin(self.hamiltonian_path_lengths)]
        except:
            return None


np.random.seed(3)
g = generate_random_weighted_graph(10,50,1,20,seed=10)
plot_graph(g,"Graph for TSP")
```

```
acotsp = ACOTSP(g,n_ants=10,alpha=3,beta=5,Q=10,rho=.1)
acotsp.run(n_iter = 50)
plot_graph(g,"Shortest Path Found By ACO for TSP",acotsp.min_path)
```

Output:



Result:

The python code to implement ant colony optimization algorithm to optimize the travelling salesman problem has been implemented and verified successfully.

5. PARTICLE SWARM OPTIMIZATION

Aim:

To write a python code to implement particle swarm optimization algorithm for function optimization.

Algorithm:

1. Randomly initialize swarm population of N particles.
2. Select hyperparameter values dim(no. of dimensions), minx(lower bound), maxx(upper bound).
3. Repeat the following steps for number of iterations:
4. Repeat the below steps for each individual of the population:
   i. Compute new velocity of the particle.
      self.velocity[i] = ((maxx - minx) *self.rnd.random() + minx)
   ii. If velocity is not in range [minx, max] then clip it.
   iii. Compute new position of ith particle using its new velocity.
      self.position[i] = ((maxx - minx) *self.rnd.random() + minx)
   iv. Update new best of this particle and new best of Swarm.
5. Return the best particle of the swarm.

Source code:

```
import random
import copy
import sys

#sphere function
def fitness_sphere(position):
    fitnessVal = 0.0
    for i in range(len(position)):
        xi = position[i]
        fitnessVal += (xi*xi)
    return fitnessVal

#particle class
class Particle:
    def __init__(self, fitness, dim, minx, maxx, seed):
        self.rnd = random.Random(seed)

        # initialize position of the particle with 0.0 value
        self.position = [0.0 for i in range(dim)]

        # initialize velocity of the particle with 0.0 value
        self.velocity = [0.0 for i in range(dim)]

        # initialize best particle position of the particle with 0.0 value
        self.best_part_pos = [0.0 for i in range(dim)]

        # loop dim times to calculate random position and velocity
        # range of position and velocity is [minx, max]
        for i in range(dim):
            self.position[i] = ((maxx - minx) *self.rnd.random() + minx)
            self.velocity[i] = ((maxx - minx) *self.rnd.random() + minx)

        # compute fitness of particle
        self.fitness = fitness(self.position) # curr fitness

        # initialize best position and fitness of this particle
        self.best_part_pos = copy.copy(self.position)
        self.best_part_fitnessVal = self.fitness # best fitness

# particle swarm optimization function
def pso(fitness, max_iter, n, dim, minx, maxx):

    # hyper parameters
    w = 0.729 # inertia
    c1 = 1.49445 # cognitive (particle)
    c2 = 1.49445 # social (swarm)

    rnd = random.Random(0)

    # create n random particles
    swarm = [Particle(fitness, dim, minx, maxx, i) for i in range(n)]

    # compute the value of best_position and best_fitness in swarm
    best_swarm_pos = [0.0 for i in range(dim)]
    best_swarm_fitnessVal = sys.float_info.max # swarm best
    # computer best particle of swarm and it's fitness
    for i in range(n): # check each particle
        if swarm[i].fitness < best_swarm_fitnessVal:
            best_swarm_fitnessVal = swarm[i].fitness
            best_swarm_pos = copy.copy(swarm[i].position)

    # main loop of pso
    Iter = 0
    while Iter < max_iter:

        # after every 10 iterations
        # print iteration number and best fitness value so far
        if Iter % 10 == 0 and Iter > 1:
            print("Iter = " + str(Iter) + " best fitness = %.3f" %
best_swarm_fitnessVal + " Best position: " + str(["%.6f"%best_swarm_pos[k]
for k in range(dim)]))

        for i in range(n): # process each particle

            # compute new velocity of curr particle
            for k in range(dim):
                r1 = rnd.random() # randomizations
                r2 = rnd.random()

                swarm[i].velocity[k] = (
                                        (w * swarm[i].velocity[k]) +
                                        (c1 * r1 * (swarm[i].best_part_pos[k] -
swarm[i].position[k])) +
                                        (c2 * r2 * (best_swarm_pos[k] -swarm[i].position[k]))
                                    )

            # if velocity[k] is not in [minx, max] then clip it
            if swarm[i].velocity[k] < minx:
                swarm[i].velocity[k] = minx
            elif swarm[i].velocity[k] > maxx:
                swarm[i].velocity[k] = maxx

        # compute new position using new velocity
        for k in range(dim):
            swarm[i].position[k] += swarm[i].velocity[k]

        # compute fitness of new position
        swarm[i].fitness = fitness(swarm[i].position)

        # is new position a new best for the particle?
        if swarm[i].fitness < swarm[i].best_part_fitnessVal:
            swarm[i].best_part_fitnessVal = swarm[i].fitness
            swarm[i].best_part_pos = copy.copy(swarm[i].position)

        # is new position a new best overall?
        if swarm[i].fitness < best_swarm_fitnessVal:
            best_swarm_fitnessVal = swarm[i].fitness
            best_swarm_pos = copy.copy(swarm[i].position)

        # for-each particle
        Iter += 1
    #end_while

    return best_swarm_pos
    # end pso

# Driver code for rastrigin function
dim = 3
fitness = fitness_sphere
num_particles = 50
max_iter = 100

print("\nStarting PSO algorithm\n")

best_position = pso(fitness, max_iter, num_particles, dim, -10.0, 10.0)

print("\nPSO completed\n")
print("\nBest solution found:")
print(["%.6f"%best_position[k] for k in range(dim)])
fitnessVal = fitness(best_position)
print("fitness of best solution = %.6f" % fitnessVal)
```

Output:

```
Starting PSO algorithm

Iter = 10 best fitness= 9.700 Best position: ['-2.963075', '-0.926583', '0.248887']
Iter = 20 best fitness= 9.700 Best position: ['-2.963075', '-0.926583', '0.248887']
Iter = 30 best fitness= 1.100 Best position: ['-0.411780', '-0.854872', '0.446743']
Iter = 40 best fitness= 0.873 Best position: ['-0.417564', '-0.829184', '0.105651']
Iter = 50 best fitness= 0.855 Best position: ['-0.405815', '-0.829811', '0.036998']
Iter = 60 best fitness = 0.851 Best position: ['-0.403051', '-0.829691', '0.013340']
Iter = 70 best fitness = 0.850 Best position: ['-0.402990', '-0.829325', '0.015951']
Iter = 80 best fitness = 0.850 Best position: ['-0.402987', '-0.829222', '0.020165']
Iter = 90 best fitness = 0.850 Best position: ['-0.402987', '-0.829218', '0.020344']

PSO completed

Best solution found: ['-0.402987', '-0.829218', '0.020351']
fitness of best solution = 0.850415
```

Result:

The python code to implement particle swarm optimization algorithm for function optimization has implemented and verified successfully.

6. GRAYWOLF OPTIMIZATION

Aim:

To write a python code to implement graywolf optimization algorithm for function optimization.

Algorithm:

1. Randomly initialize the population of grey wolves X_i (i = 1,2,…,n)
2. Initialize the value of a=2, A and C where A and C are the coefficient vectors
3. Calculate the fitness of each member of the population
    i. Xα=member with the best fitness value
    ii. Xβ=second best member (in terms of fitness value)
    iii. Xδ=third best member (in terms of fitness value)
4. Repeat the below steps for the number of iterations:
    i. Update the position of all the omega wolves by eq. 4, 5 and 6
    ii. Update a, A, C (using eq. 3)
    iii. a = 2(1-t/T)
    iv. Calculate Fitness of all search agents
    v. Update Xα, Xβ, Xδ.
5. return X_α

Source code:

```
import random
import copy

#Sphere function
def fitness_sphere(position):
    fitness_value = 0.0
    for i in range(len(position)):
        xi = position[i]
        fitness_value += (xi*xi)
    return fitness_value

# wolf class
class wolf:
    def __init__(self, fitness, dim, minx, maxx, seed):
        self.rnd = random.Random(seed)
        self.position = [0.0 for i in range(dim)]

        for i in range(dim):
            self.position[i] = ((maxx - minx) * self.rnd.random() + minx)
            self.fitness = fitness(self.position) # curr fitness

# grey wolf optimization (GWO)
def gwo(fitness, max_iter, n, dim, minx, maxx):
    rnd = random.Random(0)

    # create n random wolves
    population = [ wolf(fitness, dim, minx, maxx, i) for i in range(n)]

    # On the basis of fitness values of wolves, sort the population in asc order
    population = sorted(population, key = lambda temp: temp.fitness)

    # best 3 solutions will be called as alpha, beta and gaama
    alpha_wolf, beta_wolf, gamma_wolf = copy.copy(population[: 3])

    # main loop of gwo
    Iter = 0
    while Iter < max_iter:

        # after every 10 iterations print iteration number and best fitness value so far
        if Iter % 10 == 0 and Iter > 1:
            print("Iter = " + str(Iter) + " best fitness = %.3f" % alpha_wolf.fitness + "
Best position = " + str(["%.6f"%alpha_wolf.position[k] for k in range(dim)]))

        # linearly decreased from 2 to 0
        a = 2*(1 - Iter/max_iter)

        # updating each population member with the help of best three members
        for i in range(n):
            A1, A2, A3 = a * (2 * rnd.random() - 1), a * (2 * rnd.random() - 1), a * (2
* rnd.random() - 1)
            C1, C2, C3 = 2 * rnd.random(), 2*rnd.random(), 2*rnd.random()

            X1 = [0.0 for i in range(dim)]
            X2 = [0.0 for i in range(dim)]
            X3 = [0.0 for i in range(dim)]
            Xnew = [0.0 for i in range(dim)]
            for j in range(dim):
                X1[j] = alpha_wolf.position[j] - A1 * abs(C1 * alpha_wolf.position[j]
- population[i].position[j])
                 X2[j] = beta_wolf.position[j] - A2 * abs(C2 * beta_wolf.position[j] -
population[i].position[j])
                                X3[j] = gamma_wolf.position[j] - A3 * abs(C3 *
gamma_wolf.position[j] - population[i].position[j])
                Xnew[j]+= X1[j] + X2[j] + X3[j]

            for j in range(dim):
                Xnew[j]/=3.0

            # fitness calculation of new solution
            fnew = fitness(Xnew)

            # greedy selection
            if fnew < population[i].fitness:
                population[i].position = Xnew
                population[i].fitness = fnew

        population = sorted(population, key = lambda temp: temp.fitness)

        alpha_wolf, beta_wolf, gamma_wolf = copy.copy(population[: 3])

        Iter+= 1

    # returning the best solution
    return alpha_wolf.position

dim = 3
fitness = fitness_sphere
num_particles = 10
max_iter = 50

print(f'Starting graywolf algorithm\n')
best_position = gwo(fitness, max_iter, num_particles, dim, -10.0, 10.0)
print(f'\nGraywolf algorithm completed\n')
print("\nBest solution found:")
print(["%.6f"%best_position[k] for k in range(dim)])
err = fitness(best_position)
print("fitness of best solution = %.6f" % err)
```

Output:

```
Starting graywolf algorithm

Iter = 10 best fitness = 0.012 Best position = ['-0.044360', '0.084065', '-0.050042']
Iter = 20 best fitness = 0.000 Best position = ['-0.004129', '0.005473', '0.000494']
Iter = 30 best fitness = 0.000 Best position = ['-0.001647', '0.001247', '-0.000390']
Iter = 40 best fitness = 0.000 Best position = ['-0.000896', '0.001025', '-0.000212']

Graywolf algorithm completed

Best solution found: ['-0.000871', '0.000903', '-0.000206']
fitness of best solution = 0.000002
```

Result:

The python code to implement graywolf optimization algorithm for function optimization has been implemented and verified successfully.

7. TABU SEARCH

Aim:

To write a python code to implement tabu search for job scheduling.

Algorithm:

1. Start with an initial solution ***s* = *S*$_0$**.
2. Generate a set of neighbouring solutions to the current solution ***s*** labeled ***N(s)***. From this set of solutions, the solutions that are in the Tabu List are removed with the exception of the solutions that fit the Aspiration Criteria. This new set of results is the new ***N(s)***.
3. Choose the best solution out of ***N(s)*** and label this new solution ***s'***. If the solution ***s'*** is better than the current best solution, update the current best solution.
4. Update the Tabu List ***T(s)*** by removing all moves that are expired past the Tabu Tenure and add the new move ***s'*** to the Tabu List. Additionally, update the set of solutions that fit the Aspiration Criteria ***A(s)***.
5. If the Termination Criteria are met, then the search stops or else it will move onto the next iteration.

Source code:

```
import pandas as pd
import random as rd
from itertools import combinations
import math

class TS():
    def __init__(self, Path, seed, tabu_tenure):
        self.Path = Path
        self.seed = seed
        self.tabu_tenure = tabu_tenure
        self.instance_dict = self.input_data()
        self.Initial_solution = self.get_InitialSolution()
        self.tabu_str, self.Best_solution, self.Best_objvalue = self.TSearch()

    def input_data(self):
        return pd.read_excel(self.Path, names=['Job', 'weight', "processing_time",
"due_date"],
                             index_col=0).to_dict('index')

    def get_tabuestructure(self):
        dict = {}
        for swap in combinations(self.instance_dict.keys(), 2):
            dict[swap] = {'tabu_time': 0, 'MoveValue': 0}
        return dict

    def get_InitialSolution(self):
        n_jobs = len(self.instance_dict) # Number of jobs
        # Producing a random schedule of jobs
        initial_solution = list(range(1, n_jobs+1))
        rd.seed(self.seed)
        rd.shuffle(initial_solution)
        return initial_solution

    def Objfun(self, solution):
        dict = self.instance_dict
        t = 0   #starting time
        objfun_value = 0
        for job in solution:
            C_i = t + dict[job]["processing_time"]  # Completion time
            d_i = dict[job]["due_date"]   # due date of the job
            T_i = max(0, C_i - d_i)    #tardiness for the job
            W_i = dict[job]["weight"]  # job's weight

            objfun_value +=  W_i * T_i
            t = C_i
        return objfun_value

    def SwapMove(self, solution, i ,j):
        solution = solution.copy()
        # job index in the solution:
        i_index = solution.index(i)
        j_index = solution.index(j)
        #Swap
        solution[i_index], solution[j_index] = solution[j_index], solution[i_index]
        return solution
    def TSearch(self):
        # Parameters:
        tenure =self.tabu_tenure
        tabu_structure = self.get_tabuestructure()  # Initialize the data structures
        best_solution = self.Initial_solution
        best_objvalue = self.Objfun(best_solution)
        current_solution = self.Initial_solution
        current_objvalue = self.Objfun(current_solution)

        iter = 1
        Terminate = 0
        while Terminate < 100:
            if iter<=10:
                print(f'Iteration {iter}: Best_objvalue: {best_objvalue}')

            # Searching the whole neighborhood of the current solution:
            for move in tabu_structure:
                candidate_solution = self.SwapMove(current_solution, move[0],
move[1])
                candidate_objvalue = self.Objfun(candidate_solution)
                tabu_structure[move]['MoveValue'] = candidate_objvalue

            # Admissible move
            while True:
                # select the move with the lowest ObjValue in the neighborhood
(minimization)
                best_move = min(tabu_structure, key =lambda x:
tabu_structure[x]['MoveValue'])
                MoveValue = tabu_structure[best_move]["MoveValue"]
                tabu_time = tabu_structure[best_move]["tabu_time"]
                # Not Tabu
                if tabu_time < iter:
                    # make the move
                    current_solution = self.SwapMove(current_solution, best_move[0],
best_move[1])
                    current_objvalue = self.Objfun(current_solution)
                    # Best Improving move
                    if MoveValue < best_objvalue:
                        best_solution = current_solution
                        best_objvalue = current_objvalue
                        Terminate = 0
                    else:
                        Terminate += 1
                    # update tabu_time for the move
                    tabu_structure[best_move]['tabu_time'] = iter + tenure
                    iter += 1
                    break
                # If tabu
                else:
                    # Aspiration
                    if MoveValue < best_objvalue:
                        # make the move
                      current_solution = self.SwapMove(current_solution, best_move[0],
best_move[1])
                        current_objvalue = self.Objfun(current_solution)
                        best_solution = current_solution
                        best_objvalue = current_objvalue
                        Terminate = 0
                        iter += 1
                        break
                    else:
                        tabu_structure[best_move]["MoveValue"] = float('inf')
                        continue

        print("\nTabu search completed")
         print("\nPerformed iterations: {}".format(iter), "Best found Solution: {} ,
Objvalue: {}".format(best_solution,best_objvalue), sep="\n")
        return tabu_structure, best_solution, best_objvalue

print("Starting Tabu search\n")
test = TS(Path="Instance_10.xlsx", seed = 2012, tabu_tenure=3)
```

Output:

```
Starting Tabu search

Iteration 1: Best_objvalue: 29.220000000000002
Iteration 2: Best_objvalue: 21.62
Iteration 3: Best_objvalue: 15.650000000000004
Iteration 4: Best_objvalue: 14.850000000000003
Iteration 5: Best_objvalue: 14.140000000000002
Iteration 6: Best_objvalue: 13.680000000000003
Iteration 7: Best_objvalue: 13.370000000000001
Iteration 8: Best_objvalue: 13.260000000000002
Iteration 9: Best_objvalue: 13.240000000000002
Iteration 10: Best_objvalue: 13.240000000000002

Tabu search completed

Performed iterations: 110
Best found Solution: [3, 2, 1, 4, 8, 10, 5, 9, 7, 6] , Objvalue: 13.240000000000002
```

Result:

The python code to implement tabu search for job scheduling has been implemented and verified successfully.

8. SHUFFLED FROG LEAPING ALGORITHM

Aim:

To write a python code to implement shuffled frog leaping algorithm for function optimization.

Algorithm:

1. Initialization of Population F = M*N where M is the number is the memeplexes and N is the number of frogs in each memeplex.
2. From the available space, sample F virtual frogs U(1), U(2),…, U(F). Calculate the competency value of *ƒ*(i) for each U(i).
3. arrange the F frogs in descending order based on their fitness function and place them in a list X
4. Divide frogs into memeplexes
5. Evolution of memetics in each memeplex using local search algorithm.
6. Mix up memeplexes
7. Stop if the convergence conditions are met. Otherwise, go back to step 3.

Source code:

```
import numpy as np

def opt_func(value):
    return np.sqrt((value ** 2).sum())

def gen_frogs(frogs, dimension, sigma, mu):
    return sigma * (np.random.randn(frogs, dimension)) + mu

def sort_frogs(frogs, mplx_no, opt_func):
    # Find fitness of each frog
    fitness = np.array(list(map(opt_func, frogs)))
    # Sort the indices in decending order by fitness
    sorted_fitness = np.argsort(fitness)
    # Empty holder for memeplexes
    memeplexes = np.zeros((mplx_no, int(frogs.shape[0]/mplx_no)))
    # Sort into memeplexes
    for j in range(memeplexes.shape[1]):
        for i in range(mplx_no):
            memeplexes[i, j] = sorted_fitness[i+(mplx_no*j)]
    return memeplexes

def local_search(frogs, memeplex, opt_func, sigma, mu):
    # Select worst, best, greatest frogs
    frog_w = frogs[int(memeplex[-1])]
    frog_b = frogs[int(memeplex[0])]
    frog_g = frogs[0]
    # Move worst wrt best frog
    frog_w_new = frog_w + (np.random.rand() * (frog_b - frog_w))
    # If change not better, move worst wrt greatest frog
    if opt_func(frog_w_new) > opt_func(frog_w):
        frog_w_new = frog_w + (np.random.rand() * (frog_g - frog_w))
    # If change not better, random new worst frog
    if opt_func(frog_w_new) > opt_func(frog_w):
        frog_w_new = gen_frogs(1, frogs.shape[1], sigma, mu)[0]
    # Replace worst frog
    frogs[int(memeplex[-1])] = frog_w_new
    return frogs

def shuffle_memeplexes(frogs, memeplexes):
    # Flatten the array
    temp = memeplexes.flatten()
    #Shuffle the array
    np.random.shuffle(temp)
    # Reshape
    temp = temp.reshape((memeplexes.shape[0], memeplexes.shape[1]))
    return temp

def sfla(opt_func, frogs=30, dimension=2, sigma=1, mu=0, mplx_no=5,
mplx_iters=10, solun_iters=50):
    # Generate frogs around the solution
    frogs = gen_frogs(frogs, dimension, sigma, mu)
    # Arrange frogs and sort into memeplexes
    memeplexes = sort_frogs(frogs, mplx_no, opt_func)
    # Best solution as greatest frog
    best_solun = frogs[int(memeplexes[0, 0])]
    # For the number of iterations
    for i in range(solun_iters):
        if i%10==0 and i>1:
            print(f'iteration {i}: best solution: {best_solun} score:
{opt_func(best_solun)}')
        # Shuffle memeplexes
        memeplexes = shuffle_memeplexes(frogs, memeplexes)
        # For each memeplex
        for mplx_idx, memeplex in enumerate(memeplexes):
            # For number of memeplex iterations
            for j in range(mplx_iters):
                # Perform local search
                frogs = local_search(frogs, memeplex, opt_func, sigma, mu)
            # Rearrange memeplexes
            memeplexes = sort_frogs(frogs, mplx_no, opt_func)
            # Check and select new best frog as the greatest frog
            new_best_solun = frogs[int(memeplexes[0, 0])]
            if opt_func(new_best_solun) < opt_func(best_solun):
                best_solun = new_best_solun

    return best_solun, frogs, memeplexes.astype(int)

print("Starting shuffled frog leaping algorithm \n")
solun, frogs, memeplexes = sfla(opt_func, 100, 2, 1, 0, 5, 25, 50)
print("\Shuffled frog leaping algorithm completed")
print(f'\nBest solution: {solun} Score: {opt_func(solun)}')
```

Output:

```
Starting shuffled frog leaping algorithm

iteration 10: best solution: [ 0.14935511 -0.00351332] score: 0.1493964236590642
iteration 20: best solution: [ 0.14935511 -0.00351332] score: 0.1493964236590642
iteration 30: best solution: [-0.10851772 0.03890909] score: 0.11528231934701044
iteration 40: best solution: [-0.10851772 0.03890909] score: 0.11528231934701044

Shuffled frog leaping algorithm completed
Best solution: [-0.09648238 -0.00341754] Score: 0.09654288417646839
```

Result:

The python code to implement shuffled frog leaping algorithm for function optimization has been implemented and verified successfully.

TSP USING GENETIC ALGORITHM

Aim:

To write a python code to implement different initialization/ crossover/ mutation operations of genetic algorithm on travelling salesman problem.

Algorithm:

Nearest neighbor initialization:

1. Initialize i = 1.
2. Randomly select a city as the starting city.
3. Find the city that is closest to s(i) that has not yet been assigned to an element of s, and assign it to s(i + 1).
4. Increment i by one.
5. If i = n, terminate; otherwise, go to step3.

Shortest edge initialization:

1. Define T as the set of edges in the tour. Initialize T to the empty set.
2. Find the shortest edge in {L_k} that satisfies the following constraints:
   a. It is not in T.
   b. If added to T, it will not result in a closed tour with less than n edges.
   c. If it joins cities i and j and it is added to T, then T will not have more than two edges associated with city i or city j.
3. If T has n edges, then we are done; otherwise, go to step (2).

Partially matched crossover:

1. Initialize a random crossover point.
2. Combine the elements of from one parent 1 upto the crossover and the rest of the element from the parent 2 to create a child and vice versa to create another child.

   Child1 = Parent1[:crossover_point] + Parent2[crossover_point:]
   Child2 = Parent2[:crossover_point] + Parent1[crossover_point:]

Order crossover:

1. Randomly select and copy a section of a tour from one parent to the child.
2. Copy the remaining required cities from the second parent to the child, while maintaining the relative order of those cities from Parent 2.

Mutation by inversion:

1. Randomly select two indices.
2. Reverse the order of the tour between two randomly-selected indices.

Mutation by insertion:

1. Randomly select two indices A and B.
2. Move the city in position A to position B.

Source code:

Graph generation and display:

```
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt
from itertools import combinations

def generate_random_weighted_graph(n,low,high):
    g = nx.generators.complete_graph(n)
    # add random weights to the graph
    g.add_weighted_edges_from([(a,b,np.random.randint(low,high)) for a,b in
g.edges()])
    # creare layout fro plotting and set pos as attribute
    nx.set_node_attributes(g,nx.spring_layout(g),"pos")
    return g

def plot_graph(g,title="",highlight_edges=[]):
    pos = nx.get_node_attributes(g,"pos")
    plt.figure(figsize=(17,17))
    plt.title(title)
    nx.draw(g,pos=pos,labels = {x:x for x in g.nodes},width=2)
    weights = nx.get_edge_attributes(g,"weight")
    # draw labels for edges
    nx.draw_networkx_edge_labels(g,pos,edge_labels=weights,label_pos=.4);

    # highlight highlighted_edges
    nx.draw_networkx_edges(g,pos,edgelist =
highlight_edges,edge_color="r",width=3)
    # highlight labels of highlighted edges
    nx.draw_networkx_edge_labels(
        g,pos,
        edge_labels={
            e:w
            for e,w in weights.items()
            if e in map(lambda x:tuple(sorted(x)),highlight_edges)
        },
        font_color="r",
        label_pos=.4
    )
    plt.show()

np.random.seed(3)
g = generate_random_weighted_graph(7,1,20)
plot_graph(g,"Graph for TSP")
```

Nearest neighbor initialization:

```
def nearest_neighbour_initialization(g, closed_tour=False):
    curr_node = np.random.choice(g.nodes)
    path = [curr_node]
    not_visited = set(g.nodes)-{curr_node}
    while not_visited:
        not_visited_neighbours = not_visited&set(g.neighbors(curr_node))
        key =lambda x: g[curr_node][x]["weight"]
        curr_node = min(not_visited_neighbours,key = key)
        path.append(curr_node)
        not_visited.remove(curr_node)

    # closing the loop if necessary
    if closed_tour:
        path.append(path[0])
    return path

np.random.seed(1)
print(nearest_neighbour_initialization(g))
print(nearest_neighbour_initialization(g, closed_tour=True))
```

Shortest edge initialization:

```
from collections import defaultdict

def has_cycle(g):
    try:
        nx.find_cycle(g)
    except nx.NetworkXNoCycle:
        return False
    return True

def get_path_from_edges(edges,closed_tour=False):
    path_graph = nx.Graph(edges)
    # if it is an open tour start from a node with a single degree
    curr = min(path_graph.nodes,key=path_graph.degree)
    path,visited = [curr],{curr}
    while len(path)<len(path_graph):
        curr = (set(path_graph.neighbors(curr))-visited).pop()
        visited.add(curr)
        path.append(curr)
    if closed_tour:
        path.append(path[0])
    return path

def shortest_edge_initialization(g, closed_tour = False):
    edge_list = set(g.edges)
    times_visited  = defaultdict(int)
    tour = set()
    max_tour_len = len(g) if closed_tour else len(g)-1
    key = nx.get_edge_attributes(g,"weight").get
    while len(tour)<max_tour_len:
        u,v = min(edge_list, key=key)
        times_visited[u]+=1
        times_visited[v]+=1
```

```
            tour.add((u,v))

            # removing edges that not satisfying the conditions
            edge_list.remove((u,v))

            for u,v in set(edge_list):
                if (
                    # closed loop condition
                    (has_cycle(nx.Graph(tour|{(u,v)})) and len(tour) != len(g)-1)
                    # not more than two edges condition
                    or times_visited[u] ==2 or times_visited[v] ==2

                ):
                    edge_list.remove((u,v))

    return get_path_from_edges(tour,closed_tour=closed_tour)

np.random.seed(1)
print(shortest_edge_initialization(g))
print(shortest_edge_initialization(g, closed_tour=True))
```

Partially matched crossover:

```
# Note: during cross over use open tour's path
def make_valid_tour(p,nodes):
    unvisited = set(nodes)-set(p)
    indices = defaultdict(list)
    for i in range(len(p)):
        indices[p[i]].append(i)
    visited_twice = {node for node in indices if len(indices[node])==2}
    for node in visited_twice:
        change_index = np.random.choice(indices[node])
        p[change_index] = unvisited.pop()
    return p

def partially_matched_crossover(p1,p2):
    pt = np.random.randint(1,len(p1)-1) # crossover point
    c1 = p1[:pt] + p2[pt:]
    c2 = p2[:pt] + p1[pt:]
    nodes = set(p1)
    return make_valid_tour(c1,nodes),make_valid_tour(c2,nodes)

np.random.seed(2)
n_population = 8
population = [shortest_edge_initialization(g, closed_tour=False) for _ in
range(n_population)]
selected_population =  roulette_wheel_selection(path_length,population)
parents = selected_population[:2]
print(parents)
print(partially_matched_crossover(*parents))
```

Order crossover:

```
def order_crossover(p1,p2):
    start = np.random.randint(0,len(p1)-1)
    end = np.random.randint(start+1,len(p1) if start !=0 else len(p1)-1)
    def fill_blanks(p1,p2,s,e):

        unvisited_nodes = p2.copy()
        for node in p1[s:e]:
            unvisited_nodes.remove(node)

        c = p1.copy()
        for i in range(len(p1)):
            if i<s or i>=e:
                c[i] = unvisited_nodes.pop(0)
        return c

    c1 = fill_blanks(p1,p2,start,end)
    c2 = fill_blanks(p2,p1,start,end)
    return c1,c2

np.random.seed(2)
n_population = 8
population = [shortest_edge_initialization(g, closed_tour=False) for _ in
range(n_population)]
selected_population =  roulette_wheel_selection(inv_path_length,population)
parents = selected_population[:2]
print(parents)
print(order_crossover(*parents))
```

Mutation by inversion:

```
def inversion_mutation(p):
    start = np.random.randint(0,len(p)-1)
    end = np.random.randint(start+1,len(p)+1)
    subtour = p[start:end]
    c = p.copy()
    for i in range(start,end):
        c[i] = subtour.pop()
    return c

np.random.seed(3)
n_population = 8
population = [shortest_edge_initialization(g, closed_tour=False) for _ in
range(n_population)]
subject = population[0]
print(subject,inversion_mutation(subject))
```

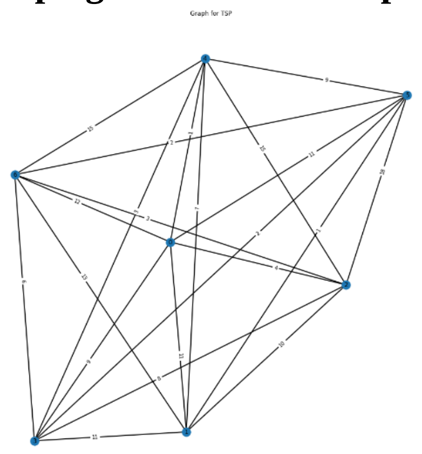
Mutation by insertion:

```
def insertion_mutation(p):
    i = np.random.randint(1,len(p))
    k = np.random.randint(0,len(p)-1)
    c = p.copy()
    c.insert(k,c.pop(i))
    return c

# np.random.seed(2)
n_population = 8
population = [shortest_edge_initialization(g, closed_tour=False) for _ in
range(n_population)]
subject = population[0]
print(subject,insertion_mutation(subject))
```

Output:

Graph generation and display:



Nearest neighbor initialization:

```
[5, 1, 4, 0, 2, 6, 3]
[3, 5, 1, 4, 0, 2, 6, 3]
```

Shortest edge initialization:

```
[3, 4, 0, 2, 6, 5, 1]
[0, 2, 6, 5, 1, 3, 4, 0]
```

Partially matched crossover:

```
[[1, 2, 0, 4, 6, 5, 3], [0, 1, 4, 2, 6, 5, 3]]
([1, 0, 4, 2, 6, 5, 3], [2, 1, 0, 4, 6, 5, 3])
```

Order crossover:

```
[[1, 2, 0, 4, 6, 5, 3], [0, 1, 4, 2, 6, 5, 3]]
([0, 1, 4, 2, 6, 5, 3], [1, 2, 0, 4, 6, 5, 3])
```

Mutation by inversion:

```
2 7
[2, 0, 4, 1, 3, 6, 5] [2, 0, 5, 6, 3, 1, 4]
```

Mutation by insertion:

```
1 5
[4, 1, 5, 3, 2, 0, 6] [4, 5, 3, 2, 0, 1, 6]
```

Result:

The python code to implement different initialization/ crossover/ mutation operations of genetic algorithm on travelling salesman problem has been implemented and verified successfully.