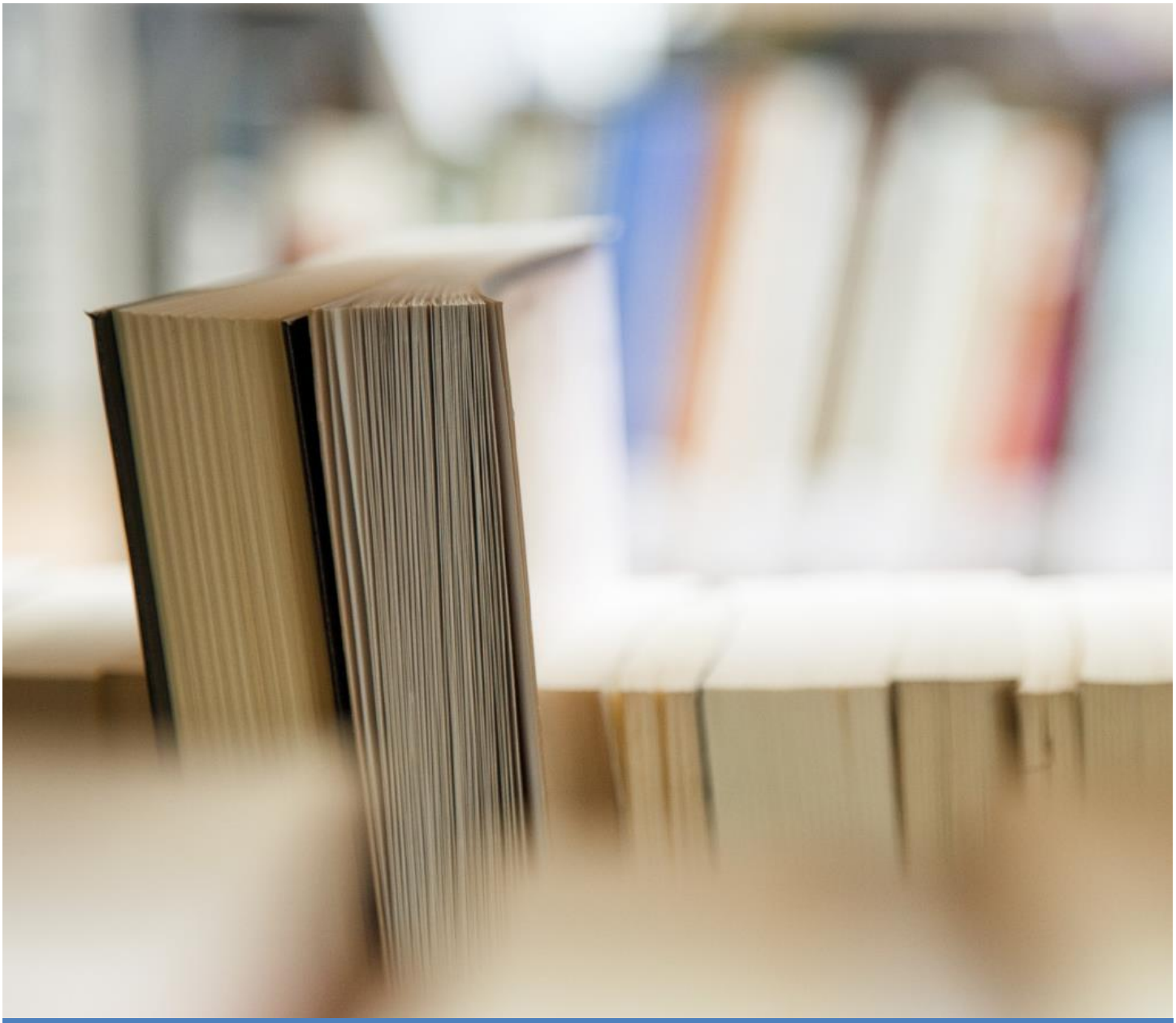


Homework - 5

Sai Saketh Boyanapalli, Sarvesh Prattipati

Advance Analytics and Metaheuristics

5/7/17

ISE 5113

Question 1: Genetic Algorithm Implementation (40 points)

(a) Finalize the code.

i. Create code to generate chromosomes in the initial population.

Initial population is generated randomly using the predefined seed, So the random numbers are generated depending on the size of dimensions of the problem and are then appended to get chromosome. Which is then evaluated to get fitness score, zipped with chromosome and then appended to population list. Size of the list depends on size of the population. And finally, the list is sorted by keeping the chromosome with highest fitness value at top.

Code:

```
# create a continuous valued chromosome
def createChromosome(d, lBnd, uBnd):
    x = []
    for i in range(d):
        x.append(myPRNG.uniform(lBnd, uBnd))  # creating a randomly located solution
    return x
# create initial population
def initializePopulation():  # n is size of population; d is dimensions of chromosome
    population = []
    populationFitness = []

    for i in range(populationSize):
        population.append(createChromosome(dimensions, lowerBound, upperBound))
        populationFitness.append(evaluate(population[i]))

    tempZip = zip(population, populationFitness)
    popVals = sorted(tempZip, key=lambda tempZip: tempZip[1])
    return popVals
```

ii. Create code to mutate chromosomes.

A function is created to mutate solutions i.e., mutate chromosomes.

```
# function to mutate solutions

def mutate(x):

  a = myPRNG.random()  # random number between 0 and 1


  if a < mutationRate:

    index = myPRNG.randint(0, dimensions - 1)

    #Inserts random number at position index

    x[index] = myPRNG.uniform(-500, 500)


  return x
```

iii. Implement logic for crossover rate and mutation rate.

The crossover rate is used to decide whether the two chromosomes should undergo crossover or not. So, if the crossover rate is low than a certain probability then the crossover of parents is performed.

Code:

```
def crossover(x1, x2):
crossover_rand = myPRNG.random() # random number between 0 and 1
if crossover_rand < crossOverRate:

    crossover_rand = myPRNG.random()  # random number between 0 and 1
    if crossover_rand < crossOverRate:
        d = len(x1)  # dimensions of solution

        # choose crossover point

        # we will choose the smaller of the two [0:crossOverPt] and
[crossOverPt:d] to be unchanged
        # the other portion be linear combo of the parents

        crossOverPt = myPRNG.randint(1,
```

```
                                        d - 1)  # notice I choose the crossover
point so that at least 1 element of parent is copied

        beta = myPRNG.random()  # random number between 0 and 1

        # note: using numpy allows us to treat the lists as vectors
        # here we create the linear combination of the soltuions
        new1 = list(np.array(x1) - beta * (np.array(x1) - np.array(x2)))
        new2 = list(np.array(x2) + beta * (np.array(x1) - np.array(x2)))

        # the crossover is then performed between the original solutions "x1"
and "x2" and the "new1" and "new2" solutions
        if crossOverPt < d / 2:
            offspring1 = x1[0:crossOverPt] + new1[crossOverPt:d]  # note the
"+" operator concatenates lists
            offspring2 = x2[0:crossOverPt] + new2[crossOverPt:d]
        else:
            offspring1 = new1[0:crossOverPt] + x1[crossOverPt:d]
            offspring2 = new2[0:crossOverPt] + x2[crossOverPt:d]

        return offspring1, offspring2  # two offspring are returned
    return x1, x2
```

Mutation Rate:

The mutation rate is to decide whether the children should undergo mutation or not. If the random probability is less than the mutation rate then only the mutation of child is performed, or else mutation is not performed. Below is the code snippet for crossover rate and mutation rate implementation.

Code:

```
# function to mutate solutions

def mutate(x):

  a = myPRNG.random()  # random number between 0 and 1

  # if a is less than mutation rate than it will mutate or else it will not

  #if a < mutationRate:

  x2 = x[:]

  # this will generate random integer depending on the dimension

  index = myPRNG.randint(0, dimensions - 1)
```

```
    # this will change the value at the index from the list
    x2[index] = myPRNG.uniform(-500, 500)
    return x2
```

iv. Implement some type of elitism in the insertion step.

Depending on the population size we are taking about 70% of the population from the new generation(kids) and 30% from the previous generation. Combining them to from new population.

Code:

```
# insertion step
def insert(pop, kids):
    # size of population from previous generation
    pop_count = int(populationSize*0.3)
    # size of population from new generation
    kids_count = int(populationSize * 0.7)
    # creating new population
    new_population = pop[:pop_count] + kids[:kids_count]

    return new_population
```

v. Complete/modify any other logic as you see fit.

Initial insertion was set to returning the whole new generation but we have changed it to perform elitism. To keep best 30% from the population and taking 70% from the new generation.

Introduced crossover rate for parent chromosomes and mutation rate for child chromosomes.

(b) Empirically decide on parameters for population size, stopping criterion, cross over rate, mutation rate, selection, and elitism.

The parameters need to change as we change dimensions of the problem to quickly arrive at optimal solution.

Population size:

Population size depends on the dimensions of the problem and no of generations. If we increase population size for 2D problem then it may yield bad solution. So, if must be set to a value where it improves over solution for the 2D problem we found that it was yielding better result when we set it to 40. And for 200D when we set to 100.

No of generations:

This also depends on the size of the problem for a 2D problem 500 generations is enough to get a solution very close to that of optimal but for a 200D problem it greatly increases. And it also depends on the size of population and other parameters.

Crossover Rate:

The crossover rate was set to 0.5 so the child is having equal features from both parents. When we tried to increase, or decrease this value it was affecting optimal solution badly. But when we tried to set it to 1 or 0 than the objective value was not improving by much.

Mutation Rate:

Too much of this will change the child completely so, we tried to reduce the value of mutation and calculate the optimal solution we got closer to optimal when the mutation rate was set to 1- 0.1. we got very close to optimal value when it was set to 0.1.

200-dimensional problem

This is entirely different case here by increasing dimensions leads to more computational time, increasing generations can lead to better output but eventually triggers stopping criteria when set to a very high value.

But the crossover and mutation rate were showing the same characteristics so, they were set to same values.

(c) Solve the 2D Schwefel.

i. Create a small population of 8 chromosomes and depict their locations on a graph for the initial random set and the first generation.

Parameters – population size = 8, generations = 1, mutation rate = 0.1, crossover rate = 0.5

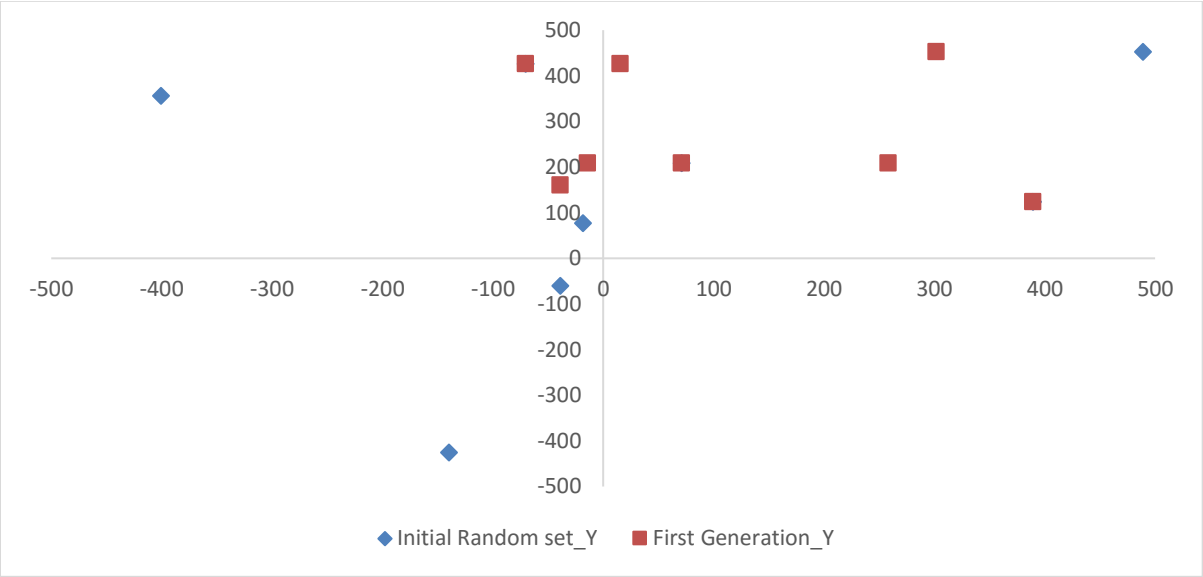
Initial Random set:

Initial random set	
x	**y**
-70.02697476	**426.1143718**
-18.26574476	**76.95165755**
-139.4184373	**-425.5633733**
71.36415689	**208.8684584**
-39.07001993	**-60.28815582**
488.8932897	**452.2448034**
389.1849048	**123.706994**
-400.5701021	**355.5580644**

First Generation:

First generation set	
15.52444463	**426.1143718**
-70.02697476	**426.1143718**
71.36415689	**208.8684584**
-14.18726251	**208.8684584**
389.1849048	**123.706994**
258.4615395	**208.8684584**
-39.07001993	**160.6133068**
301.7959072	**452.2448034**

Graph:



ii. Solve the problem as best as possible and provide information on the quality of the best solution and the performance of the approach overall.

Summary of the best solution

(min Objective Value, Mean of the population, Variance of the population)

(0.00010382266600572621, 42.390245113930291, 16186.536598289942)

Best solution found.

([x – position, y – position], objective value for that point)

([420.94480886635, 430.4482486857305], 0.00010382266600572621)

Best solution was found for following parameters

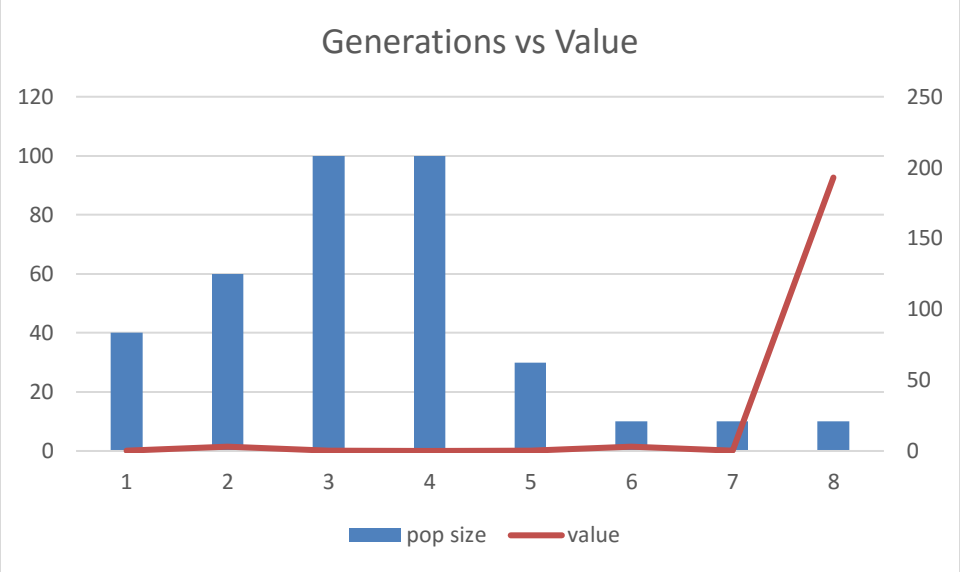
Population size = 40, generations = 500, crossover rate = 0.5, mutation rate = 0.1.

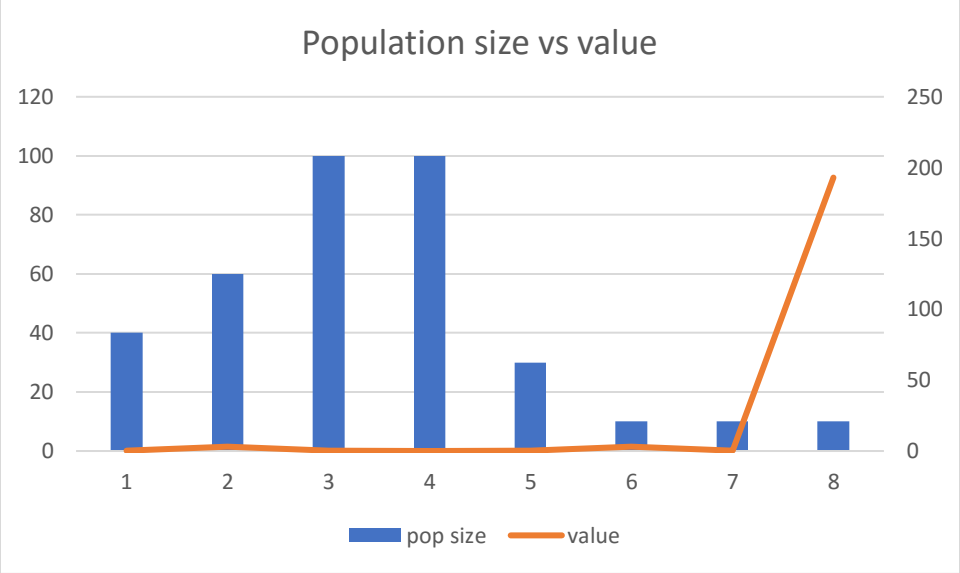
The solution that we got here is very close to the schwefel optimal solution.

Performance analysis

pop size	generations	crossover rate	mutation rate	insert rate	value
40	500	0.5	0.1	0.7	0.0001
60	500	0.5	0.1	0.7	3.15409

100	500	0.5	0.1	0.7	0.201
100	1000	0.5	0.1	0.7	0.0088
30	1000	0.9	0.1	0.7	0.0621
10	1000	0.9	0.1	0.7	3.18
10	10000	0.9	0.1	0.7	0.036
10	10000	0.5	0.2	0.7	192.97





Crossover rate, mutation rate are only varied slightly and elitism is kept constant.

The performance of the genetic algorithm mainly varies with that of parameters and how we model the problem. The time taken for the model to execute is 1 second. The 2D Schwefel function is modeled properly so that it can yield results very close to the optimal solutions in fraction of a time.

(d) Solve the 200D Schwefel problem as best as possible. Provide information on the quality of the solution and the performance of the approach overall.

Summary of the best solution

(min Objective Value, Mean of the population, Variance of the population)

(6.8699176343507133, 58.566947340996265, 28579.306875358008)

Best solution was found for following parameters

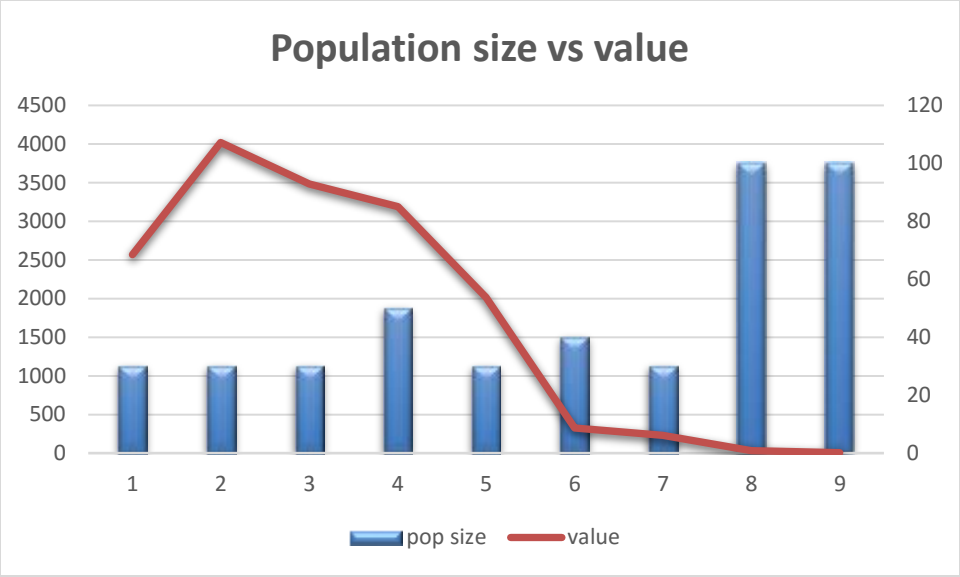
Population size = 100, generations = 10000, crossover rate = 0.5, mutation rate = 0.1.

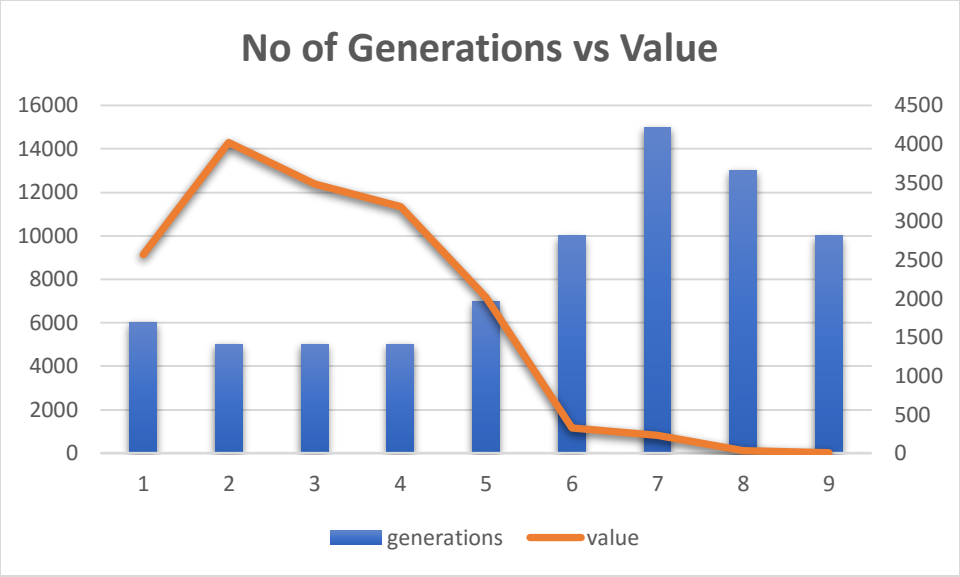
With elitism keeping 30% from the population and 70% from the children for next iteration.

The solution that we got here is very close to the schwefel optimal solution.

Performance of GA for various parameters

pop size	generations	crossover rate	mutation rate	insert rate	value
30	6000	0.8	0.15	0.3,0.7	2571
30	5000	0.8	0.15	0.3,0.7	4020
30	5000	0.9	0.15	0.3,0.7	3482
50	5000	0.9	0.15	0.3,0.7	3188
30	7000	0.7	0.1	0.3,0.7	2020
40	10000	0.9	0.1	0.3,0.7	329
30	15000	0.7	0.1	0.5,0.5	228
100	13000	0.5	0.1	0.3,0.7	29
100	10000	0.5	0.1	0.3,0.7	6





Crossover rate, mutation rate are only varied slightly and elitism is kept constant.

Compared to the 2D Schwefel the algorithm takes lot more time to run in case of 200D schwefel this is because of the increase in size of the solution space by 200-fold and the problem gets way more complicated at 200 dimensions. The time taken to execute is 176 seconds compared to 1 seconds for 2D. The quality of the solution also decreases because it requires many more iterations to arrive at the optimal value.

The performance can be improved more by increasing dimensions and increasing iterations but will take lot of time to compute.

Question 2: Particle Swarm Optimization Implementation (60 points)

(a) Complete the original PSO implementation based on "particle best" and "global best"; no inertia or constriction is necessary. Make sure to include the following elements:

i. limits on particle position (e.g., particles should primarily remain within the feasible region)

The limits to particle position is set to 500 and -500. All the initial particle positons are generated randomly from the range -500 to 500

```
def uP(pos, vel):
    for i in range(n):
        pos[i] = pos[i] + vel[i]
        if pos[i] > uBnd:
            pos[i] = uBnd - 5 * myPRNG.random()
        elif pos[i] < lBnd:
            pos[i] = lBnd + 5 * myPRNG.random()
    return pos
```

ii. limits on particle velocity

The min velocity is set to -1 and the maximum particle velocity is set to 1. The range of -1 to 1 is same for velocity and position of the particles.

```
def UpdatedvelocityVector(pos, pBest, gBest, vprev):
    UpdatedVelocity = []
    for i in range(0, len(pos)):
        upVel = w * vprev[i] + c1 * r1 * (pBest[i] - pos[i]) + r2 * c2 * (gBest[i] - pos[i])
        if upVel > vMax:
            upVel = vMax
        if upVel < vmin:
            upVel = vmin
        UpdatedVelocity.append(upVel)
    return UpdatedVelocity
```

iii. one or more stopping criteria

The stopping criteria is linked with no of iterations i.e., no of times particles undergo a change in position. So, when the set iterations are reached the program is terminated and the best objective value is printed from the list. In this case iterations are used as a stopping criteria.

```
    # stopping criteria
    if iter_counter == stopping:
            done = 1
```

In the other case the time is set as stopping criteria and it is set to 10 seconds so when the code does not execute over ten seconds the code is terminated. This can be used when calculating the performance. So, when the swarm size, iterations are set high this will let us terminate the code under a set of times.

```
# Stopping criteria
start = time.time()
end = time.time()
if end-start > 10:
    done = 1
```

(b) Using your code from Part (a), create a swarm of size 5 and solve the 2D Schwefel problem.

i. Record and list in a table the first 3 positions and velocities of each particle.

Positions	pos1_X	pos1_Y	pos2_X	pos2_Y	pos3_X	pos3_Y
1	327.5059056	-119.7899577	332.4440279	-95.82410839	335.7891784	-80.15106099
2	73.30532471	-358.4907133	88.41122421	-324.2260375	111.0020135	-273.6395496
3	69.48087871	61.58320231	84.76979573	78.42398678	94.87832985	89.36493973
4	457.2473733	478.4642344	456.4299615	478.2884731	455.6125497	478.1127119
5	12.46789844	223.6283976	30.04650447	233.4725925	56.29704176	248.2959358

Velocity	vel1_X	vel1_Y	vel2_X	vel2_Y	vel3_X	vel3_Y
1	2.299396352	11.7983528	4.938122293	23.96584932	3.34515047	15.6730474
2	7.297155985	17.24240243	15.10589949	34.26467581	22.5907893	50.58648791
3	7.402390636	8.362116745	15.28891702	16.84078446	10.10853411	10.94095295
4	-0.817411802	-0.175761214	-0.817411802	-0.175761214	-0.817411802	-0.175761214
5	8.532530331	4.661257295	17.57860603	9.84419491	26.25053729	14.82334326

Pos1_X => Says Position of first iteration on X axis

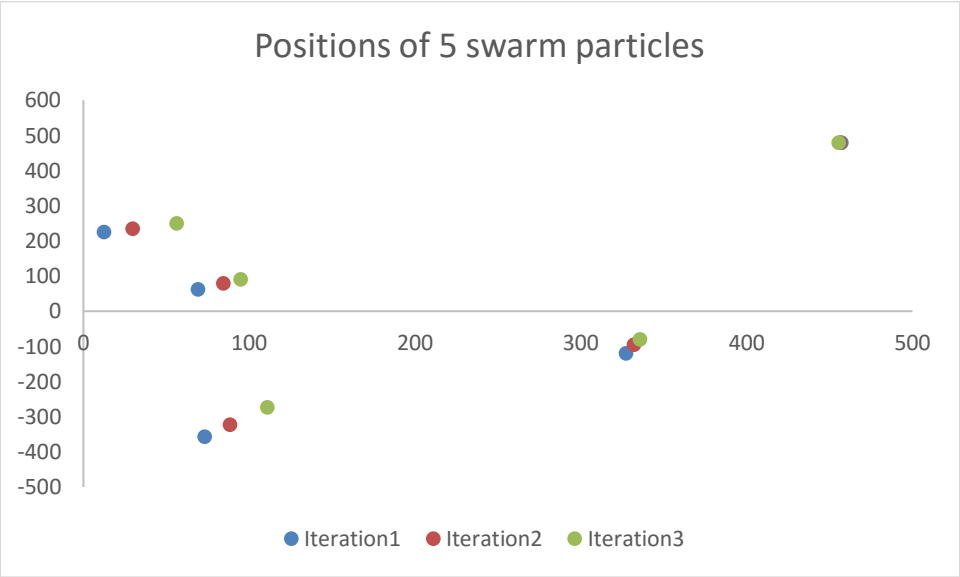
vel1_Y => Says velocity of first iteration on Y axis

The highlighted shows the global best of all the five particles for all 3 iterations

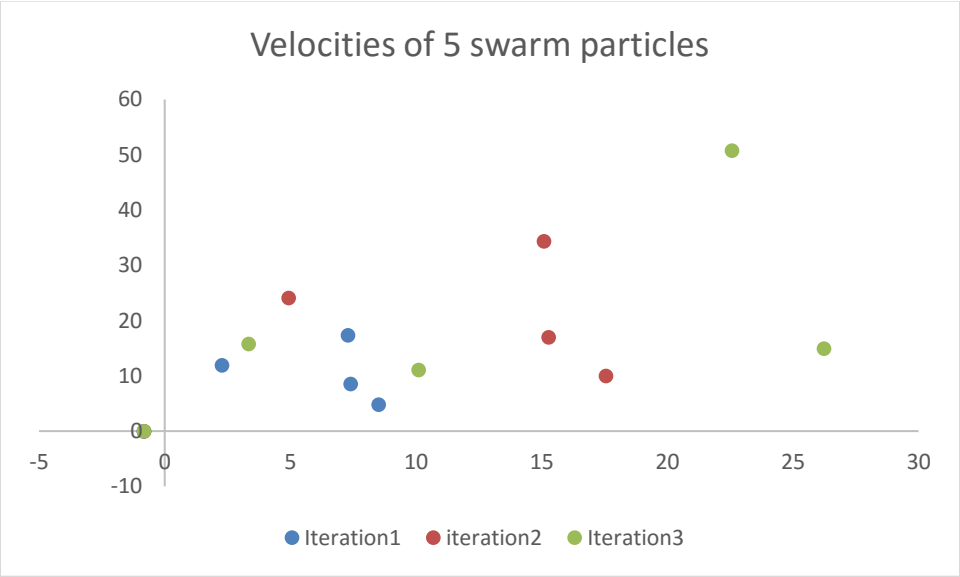
ii. Determine and highlight the particle that represents the \global best" particle position in each of these iterations.

Particle	iteration	Objective Value	Position
4	1	529.4944403	457.247373297302, 478.4642344
4	2	520.840568	456.429961495055, 478.2884731
4	3	512.3032775	455.612549692809,478.1127119

iii. Plot the first 3 positions of each of the 5 particles (you can use Python, R, Excel, or even draw it by hand)



The above plot shows positions of 5 swarm particles for first three iterations.



The above plot shows velocities of 5 swarm particles for first three iterations.

(c) Implement a PSO algorithm that uses the \local best" in place of the global best. You may also add inertia weighting or constriction if you like. Implement one of the following neighborhood topologies, ring, star, von Neumann:

We have implemented ring topology neighborhood for our PSO local best.

Neighborhood topology Ring:

Code:

```
# local best ring
def localbestring(pBest, i):
    # for the first element we are defining before element as the last element
    if i==0:
        prev = swarmSize -1
    # for the other elements before is set to previous element
    else:
        prev = i -1
    # setting next element for the last element
    if i == swarmSize - 1:
        next = 0
```

```
    # setting next element for other elements
    else:
        next = i + 1
    # taking local best for the current neighborhood
    if pBest[i:n] == max(pBest[prev:n], pBest[i:n], pBest[next:n]):
        pBestl = i
    if pBest[prev:n] == max(pBest[prev:n], pBest[i:n], pBest[next:n]):
        pBestl = prev
    if pBest[next:n] == max(pBest[prev:n], pBest[i:n], pBest[next:n]):
        pBestl = next

    return pBestl
```

We have added linear inertia weight to it.

Curiter keeps track of current iteration and noIter is the total no of iterations

```
w = wmax*(wmax -    wmin)*(curiter/noIter)
```

When we have solved it for 2D schwefel problem the optimal objective value is 0.000030

For the following parameters n = 2, swarmSize = 650, no of iterations = 100, inertia weight is set to 0.9, accelerations constants are c1 = 0.5 and c2 = 0.5.

For Current iteration of 100 Objective Value: 3.0120152018753288e-05 at the position: [420.9657496915311, 420.963455787274]

(d) Solve the 2D and 200D Schwefel problem as best as possible. Provide information on the quality of the solution and the performance of the approach overall. Compare the results with the Genetic Algorithm results.

2D schwefel

Best output:

For Current iteration of 150 Objective Value: 0.0000252248510009522 at the position: [421.0388798680618, 420.95780961210625]

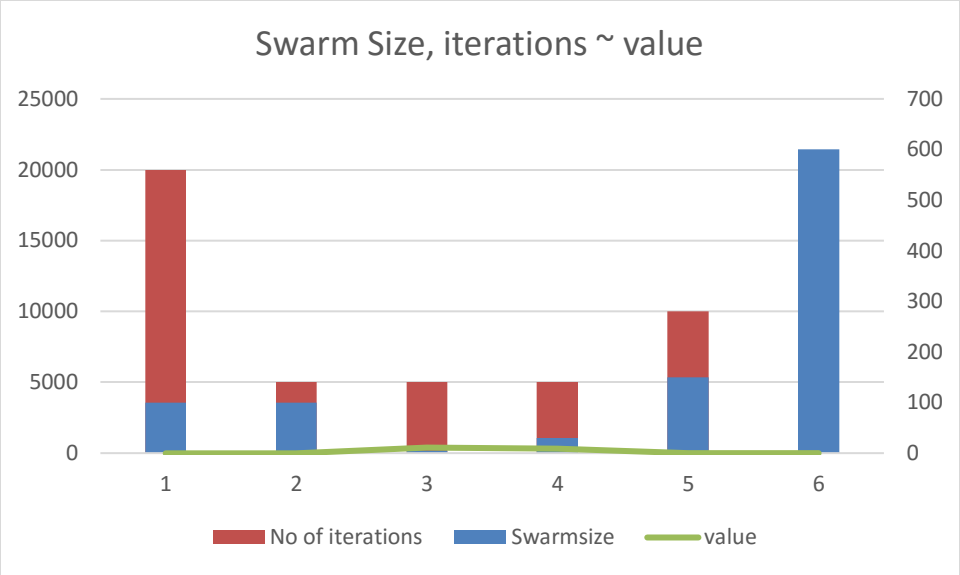
We got the objective value very close to that of optimal value of schwefel problem.

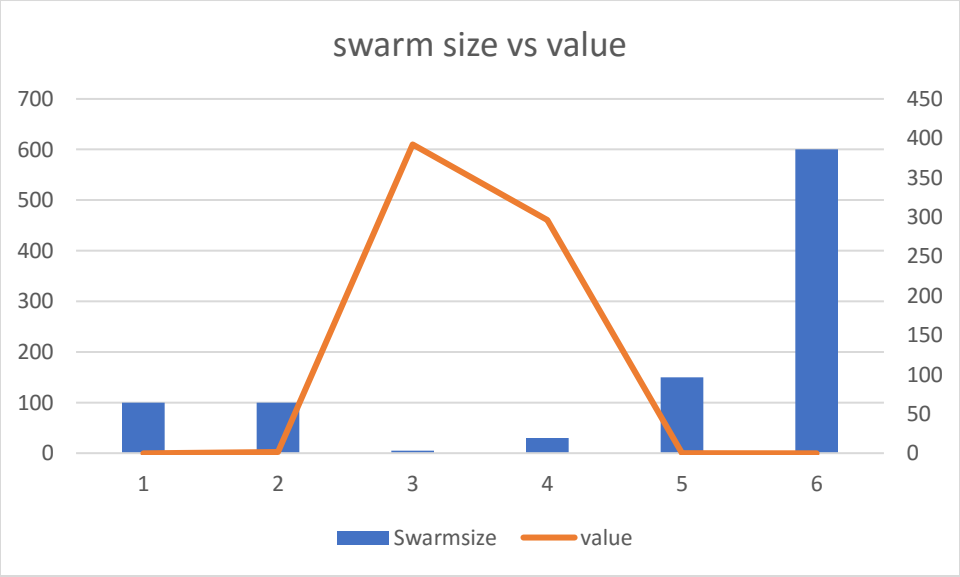
The parameters are set to
Iterations = 10000s
Swarm size = 150
Acceleration constants c1 = 0.3, c2 = 0.9
Constant inertia weight of 0.9

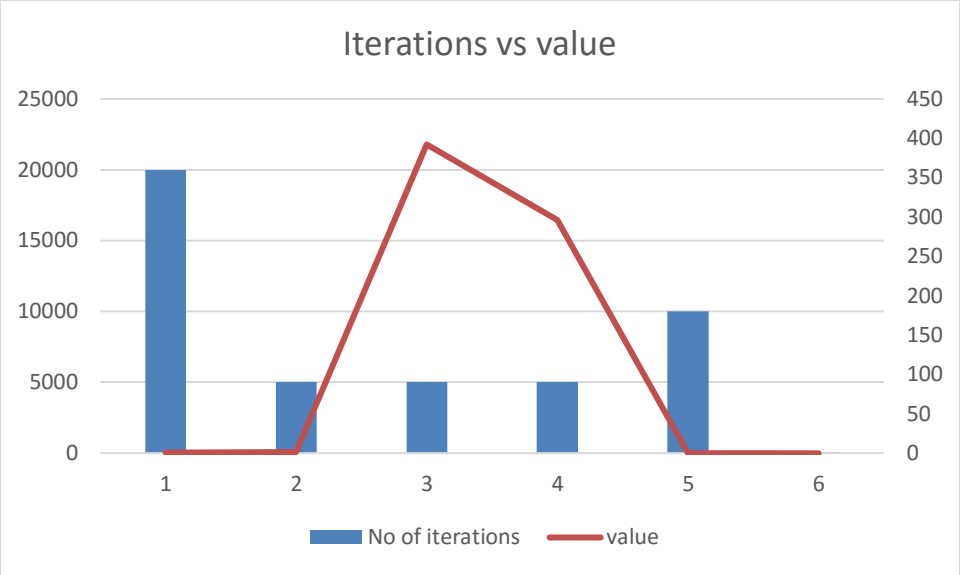
For a 2D problem the search space is small compared to that of 200D so by increasing the swarm size and decreasing the no of iterations we could arrive at an objective value very close to that of optimal value. And the performance of Particle swarm optimization is better than that of genetic algorithm and was able to arrive at better solution in lesser time than that of GA.

Performance analysis:

Swarmsize	No of iterations	c1(cognitive factor)	c2 (social Factor)	Inertia Weight	value
100	20000	0.8	0.55	1.5	0.2
100	5000	0.8	0.55	1.5	1.531
5	5000	0.8	0.55	0.9	392.19
30	5000	0.5	0.3	0.9	296
150	10000	0.3	0.1	0.9	0.000025
600	100	0.3	0.1	0.9	0.0021


Swarm Size, iterations ~ value
25000
20000
15000
10000
5000
0
700
600
500
400
300
200
100
0
1
2
3
4
5
6
No of iterations
Swarmsize
value


swarm size vs value
700
600
500
400
300
200
100
0
450
400
350
300
250
200
150
100
50
0
1
2
3
4
5
6
Swarmsize
value



200D schwefel

For the 200D problem the size of the problem increases by 200 fold so by increasing swarm size will be lot more computationally demanding than that of 2D problem so that's not an option and by increasing no of iterations it is getting stuck at a particular value and is triggering the stopping criteria so, only by changing the acceleration constants and inertia weights we can come to a better optimal value but still the optimal value is far from optimal in case of 200D.

Best solution found:

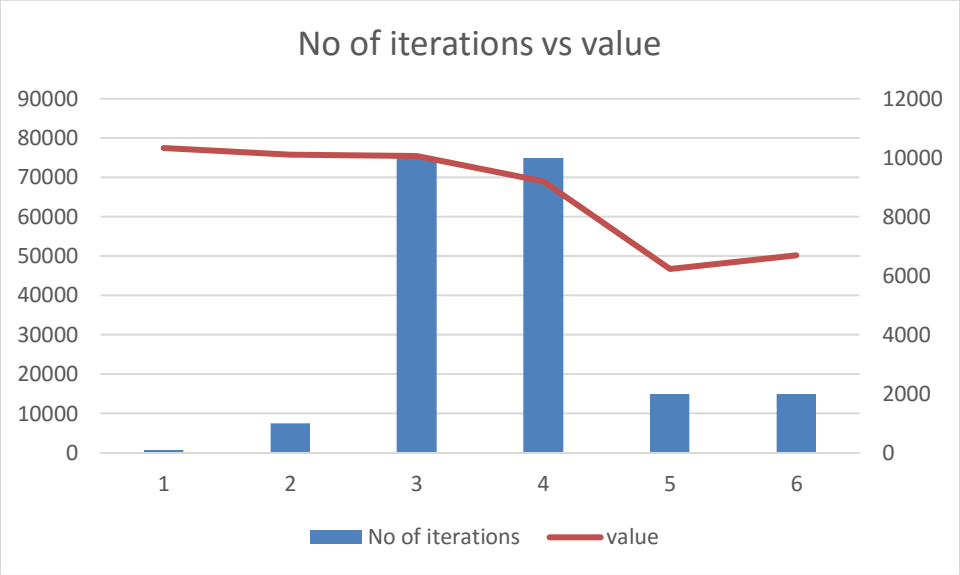
For Current iteration of 2000 Objective Value: 46750.9070630493

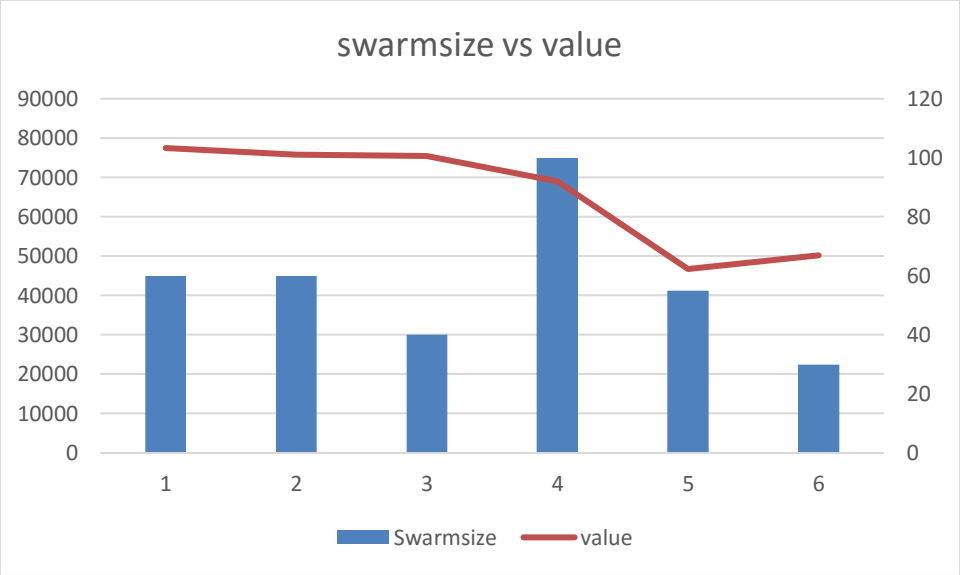
The parameters are set to
Iterations = 2000
Swarm size = 55
Acceleration constants c1 = 0.8, c2 = 0.7
Constant inertia weight of 1.5

Performance analysis:

Swarmsize	No of iterations	c1(cognitive factor)	c2 (social Factor)	Inertia Weight	value
60	100	0.3	0.1	0.9	77482
60	1000	0.3	0.1	0.9	75812

40	10000	0.2	0.1	0.9	75520
100	10000	0.5	0.7	0.9	68976
55	2000	0.8	0.7	1.5	46750
30	2000	0.8	0.7	1.5	50231





A better solution can be obtained for 200D by increasing the swarm size and number of iterations but the time it takes to compute is very high. This is because for 200D problem the solution space is 200 flod bigger than that's of 2D problem and it has now 200 position and velocity vectors so it is highly demanding for the system. The solution obtained is far from optimal in case of 200D problem and when comparing it to genetic algorithm. GA had a better optimal solution.

The result and the approach are very much comparable for that of genetic algorithm for 2D schwefel problem we are able to get a better optimal objective value in case of Particle swarm optimization. But for the 200D we were able to get a better optimal solution in case of genetic algorithm. We feel like it is because of the how particles behave in this case the particles are arriving at a pit and are getting stuck. And are triggering stopping criteria as there is no improvement in the optimal solution.

Question 3: Extra-Credit Option: Guided Local Search (25 points)

Solve the 2D and 200D Schwefel using Guided Local Search and compare the results to both the PSO and GA implementations. Make sure to explain your implementation well in your write-up. Since there is no Python code provided for this problem, you must comment well and you should include snippets of code in the write-up itself as you deem appropriate.

GLS escapes the local optimum using penalties and tries to explore global optimum. Here the augmented cost function guides the exploration away from the local optimum.

GLS tries to remove local optimum by removing them. Removing can be done through two ways:

1) Changes the topography of the search space
2) Uses an extended move evaluation function.

The extended move evaluation function is described as below:

$$f^*(s) = f(s) + \lambda \sum_{i=1}^{G} I_i(s) p_i$$

Here $I_i(s) = \begin{cases} 1, if\ solution\ s\ occured\ at\ i'th\ feature \\ 0, if\ solution\ s\ \ occured\ at\ i'th\ feature \end{cases}$

G = No of features

P_i = penalty for i'th feature

F(s) = problem cost function

F*(s) = augmented cost function

F(s) = evaluated cost function

Lambda = regularization parameter

So, to decide, which feature to penalize is given by the below function:

$$u_i(s) = I_i(s) \frac{c_i}{1 + p_i}$$

C_i = cost of the feature

P_i = current penalty value of the feature

Here $I_i(s) = \begin{cases} 1, if\ solution\ s\ occured\ at\ i'th\ feature \\ 0, if\ solution\ s\ \ occured\ at\ i'th\ feature \end{cases}$

The pseudo code is provided below:

Guided Local Search

```
1: input: starting solution, s_0
2: input: neighborhood operator, N
3: input: evaluation function, f
4: input: a set of features, F
5: input: a penalty factor, λ
6: current ⇐ s_0
7: best ⇐ s_0
8: p_i ⇐ 0 (for all i ∈ F)
9: while stopping criterion not met do
10:   Define f*(s) = f(s) + λ Σ_{i∈F} I_i(s) p_i
11:   s* ⇐ the best solution in N(current), according to f*
12:   if f*(s*) < f*(current) then
13:     current ⇐ s*
14:     if f(current) < f(best) then
15:       best ⇐ current
16:     end if
17:   else
18:     Define the utility, u_i(current) = I_i(current) c_i/(1+p_i), for all i ∈ F
19:     p_i ⇐ p_i + 1 for each feature i ∈ F having the maximum utility in
        solution current
20:   end if
21: end while
```

initialization

main loop

8

PYTHON CODE:

The implementation of the above Puesdo code step-wise is explained below.

Below is the snippet for calculating **lambda**:

```
lambdaa = alpha * (ls_Optima / float(len(eval_population)))
```

Below is the code snippet for the evaluation function:

```
def evaluate(x):
    val = 0
    d = len(x)
    for i in range(d):
        val = val + x[i] * sin(sqrt(abs(x[i])))
    val = 418.9829 * d - val
    return val
```

The **neighborhood operator** used is the stochastic two opt neighborhood structure. Below is the code for the same:

```
def stochasticTwoOpt(curr):
    current_result = curr[:]  # making a deep  and independent copy
    # selecting two random indices in the population
    index1, index2 = (random.randrange(0, len(current_result)), random.randrange(0, len(current_result)))
    to_delete = set([index1])
    if index1 == 0:
        to_delete.add(len(current_result) - 1)
    else:
        to_delete.add(index1 - 1)
    if index1 == len(current_result) - 1:
        to_delete.add(0)
    else:
        to_delete.add(index1 + 1)
    while index2 in to_delete:
        index2 = random.randint(0, len(current_result))
    # Ensuring always index1<index2
    if index2 < index1:
        index1, index2 = index2, index1
    # Reversing the line segment between index1 and index2
    current_result[index1:index2] = reversed(current_result[index1:index2])

    return current_result
```

The above function after getting the local optimum would delete the edges and structures new edge route.

The problem cost function and **augmented cost function (F*(s))** are mentioned below:

```
def augmentedCost(current_result, penalties, lambdaa):
    distance, augmented = 0, 0
    size = len(current_result)
    for index in range(0, size):
        index1 = index
        if index == size - 1:
            index2 = 0
        else:
            index2 = index + 1

        if index2 < index1:
            index1, index2 = index2, index1
        v1 = current_result[index1]
        v2 = current_result[index2]
        d = euclideanDistance(v1, v2)
        distance += d
        augmented += d + (lambdaa * penalties[index1][index2])
    return distance, augmented
```

The above is the augmented cost function, the modified cost function.

Below is the code snippet for the **problem cost function**(**F(s)**):

```
def cost(candidate                penalties                lambdaa):
```

```
    cost                augCost =                   penalties           lambdaa)
                        augmentedCost(candidate
    return cost         augCost
```

Going to the main code implementation, below is the code snippet of it:

```
def GLS(points, max_Iterations, max_itr_tostop, lambdaa):
    # Create a random solution
    current = points[:]
    best = None
    # Initializing the penalties with zeros
    penalties = [[0] * len(points)] * len(points)
    #Stopping criteria
    while max_Iterations > 0:

        # Executing the local search
        current = local_Search(current, lambdaa, penalties, max_itr_tostop)

        # Calculating the feature utilities
        utilities = calculate_Feature_Utilities(current , penalties)

        # Updating the feature penalties
        penalties = update_Feature_Penalties(current , penalties, utilities)

        # Comparing the current candidate cost with the best and update accordingly
        current_cost, current_augcost = cost(current, penalties, lambdaa)
        if best == None :
            current_best, current_best = cost(current, penalties, lambdaa)
            if current_cost < current_best:
                best = current
            max_Iterations = max_Iterations - 1

    return best
```

In the above code we see the **stopping criteria** is max_iterations.

The best value assignment is highlighted in the code

S* value is extracted from the local search function.

The code for local search is given below:

```
def local_Search(current, lambdaa, penalties, max_itr_tostop):
    count = 0
    while count < max_itr_tostop:
        current_cost, current_augcost = cost(current, penalties, lambdaa)
        candidate_Solution = []
        candidate_Solution = stochasticTwoOpt(current)
        candidate_cost,candidate_augcost = cost(candidate_Solution, penalties, lambdaa)
```

```
        #Checking for a larger augmented cost to escape the local minima
        if candidate_augcost < current_augcost:
            # resetting to restart the search
            current = candidate_Solution[:]
            count = 0
        else:
            count += 1
    return current
```

Coming to penalities, penalities are only done on the features of a local optimal solution that maximizes the utility, which is updated by adding 1 to the penalty of a feature.

Below Is the formulae for calculating the utility of a feature:

$$u_i(s) = I_i(s)\frac{c_i}{1+p_i}$$

C_i = cost of the feature i

P_i = penalty for the feature i

$U_i(s)$ = utility for penalizing the feature i

So once the utilities are calculated, i.e the outputs are binary. Then their penalties are added to the features depending upon the binary values from the utility function

Below is the code snippet implementing the above utility function:

```
def calculateFeatureUtilities(current_result, penalties):
    size = len(current_result)
    utilities = [0] * size
    for index in range(0, size):
        index1 = index
        if index == size - 1:
            index2 = 0
        else:
            index2 = index + 1
        if index2 < index1:
            index1, index2 = index2, index1
        v1 = current_result[index1]
        v2 = current_result[index2]
        utilities[index] = euclideanDistance(v1, v2) / (1 + penalties[index1][index2])

    return utilities
```

The below code snippet is to implement the updating of penalties for a feature.

```
def update_Feature_Penalties(current_result, penalties, utilities):
    size = len(current_result)
    maxUtil = max(utilities)
    for index in range(0, size):
        index1 = index
        if index == size - 1:
            index2 = 0
        else:
            index2 = index + 1
        if index2 < index1:
            index1, index2 = index2, index1
        # Update penalties
        if utilities[index] == maxUtil:
            penalties[index1][index2] += 1
    return penalties
```

Results:

Parameter	2D Value	200D Value
Number of elements	200	200
Max iterations	9000	900
Dimension	2	200
Best Value	1245.3	92334.5
Best position	[501.36,323.63]	**Cannot contain**

Final Result Table:

Parameters	Genetic Algorithm		Particle Swarm Algorithm		GLS	
Dimension	2D	200D	2D	200D	2D	200D
Number of Particles	40	100	650	55	200	200
Max Iterations	500	10000	100	2000	9000	900
Best Value	0.00010	6.869	0.00066	46750	1245.3	92334.5
Best Position	[420.94,430.44]	**Cannot contain**	[421.03, 420.957]	**Cannot contain**	[501.36,323.63]	**Cannot contain**