ISE 5113 Advanced Analytics and Metaheuristics

Homework # 5

Saied Hosseinpoor
Gowtham Talluru
Sai Srikanth Kola

May 2017

1 Genetic Algorithm Implementation

1.1 Finalize the code

i)

```
def createChromosome(d, lBnd, uBnd):  # d dimensions lBnd, uBnd lower n Upper bound
    x = []
    for i in range(d):
        x.append(myPRNG.uniform(lBnd, uBnd))    #creating a randomly located solution
    return x

def initializePopulation(): #n is size of population; d is dimensions of chromosome
    population = []          #Population
    populationFitness = []   #Population fitness

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

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

ii) Mutate function will select a random element in the chromosome and change it to a random number between upperBound and lowerBound

```
def mutate(x):

    xx=x[:]      #Create different object
    altIndex= myPRNG.randint(0, dimensions-1) # Random Index
    a = myPRNG.randrange(lowerBound, upperBound)  # Random number
    xx[altIndex] = a

    return xx
```

iii) The breeding process will select crossover with a probability of "*crossOverRate"* given in the input parameters. It will select mutation with a probability of "*MutateRate*" also given in the input parameters.

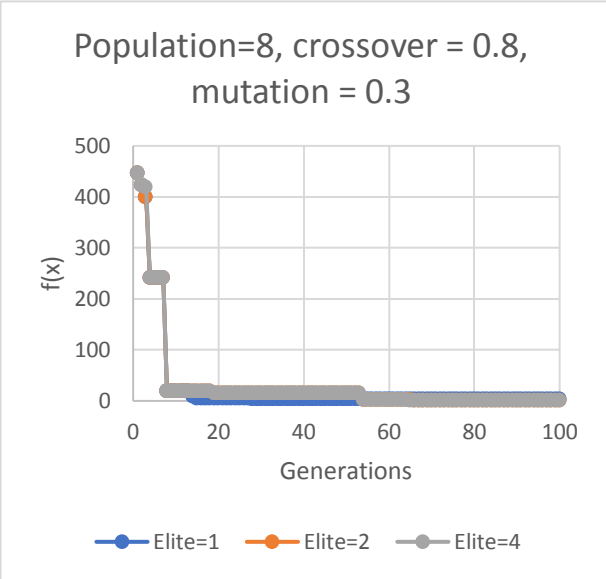
```
def breeding(matingPool):
    children = []
    childrenFitness = []
    for i in range(0,populationSize-1,2):

        if (myPRNG.random() < crossOverRate):     #Cross over
            child1,child2=crossover(matingPool[i],matingPool[i+1])
        else:
            child1=matingPool[i]  #New offspring same as parents
            child2=matingPool[i+1]

        if (myPRNG.random() < mutateRate):  #Mutation rate
            child1= mutate(child1)
            child2= mutate(child2)
```

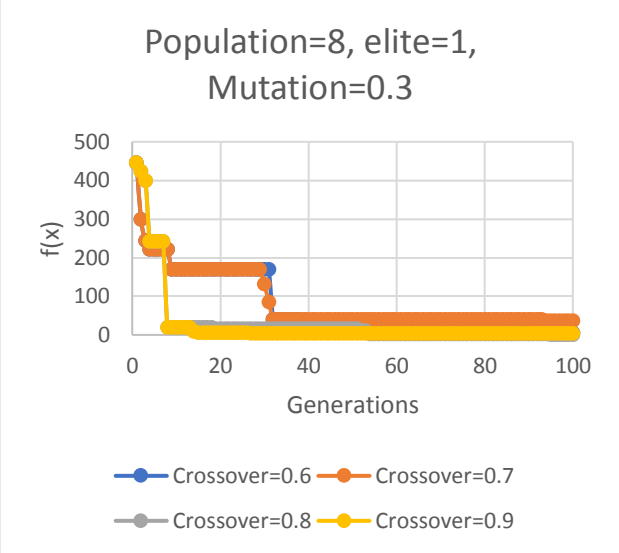
iv) Elitism: The top "*elite_group"* number of parent chromosomes which have best fitness value will survive for next generation. The bottom *elite_group* number of children chromosomes will die immediately and will not enter the population to compensate for the elite parent chromosomes and to maintain constant population number.

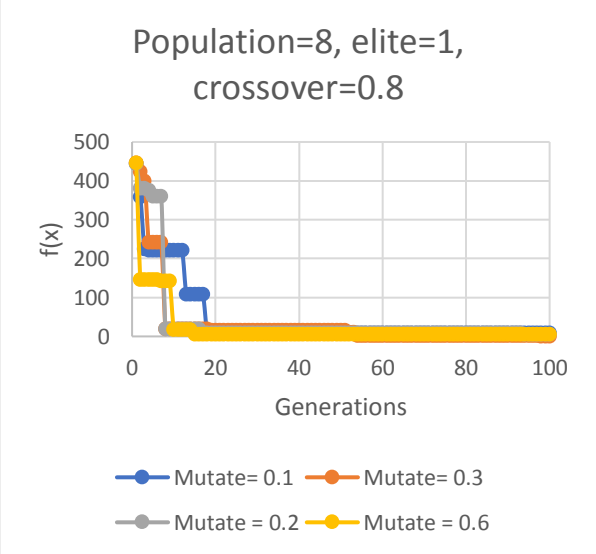
v) Appropriate changes for stopping condition were made to the code

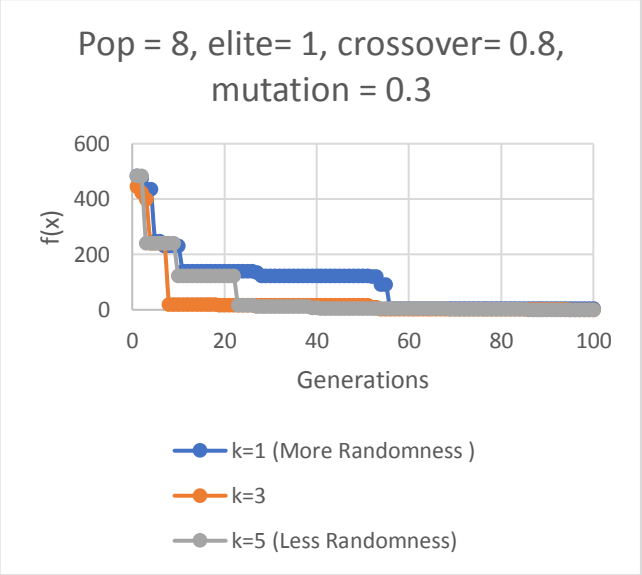
1.2 Empirically Decide on parameters

The trend of solution for 100 generations for various crossover rate, mutation rate, elitism, selection k are shown in figures below









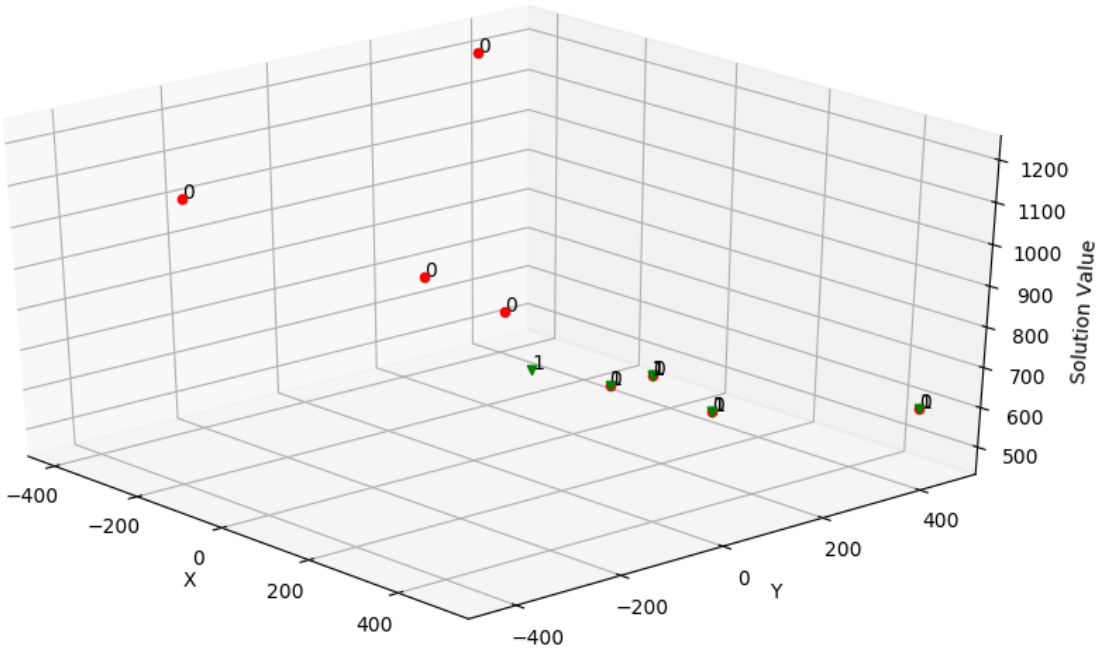
The solution value for various combinations of crossover rate, mutation rate, population size, elitism, selection k are shown in table below

Population	Crossover Rate	Mutation Rate	No of elite pop survived to next Gen	Selection k (Randomness)	Solution Value
8	0.8	0.	1	3	7.936
8	**0.8**	**0.3**	**1**	**3**	**0.162**
8	0.8	0.3	1	3	4.711
8	0.8	0.3	1	3	3.750
8	0.6	0.3	1	3	4.945
8	0.7	0.3	1	3	35.687
8	0.9	0.3	1	3	3.381
8	0.8	0.3	1	3	3.381
8	0.8	0.3	2	3	1.013
8	0.8	0.3	4	3	0.445
8	0.8	0.3	1	3	7.936
15	0.6	0.3	1	3	0.025
30	**0.7**	**0.3**	**1**	**3**	**0.000**
8	0.8	0.3	1	1	3.344
8	0.7	0.3	1	5	0.234

- **Stopping criterion:** If the best chromosome does not change for *"maxNonImproveGen"* then the algorithm will stop. The algorithm will also stop if the specified number of generations are reached.
- The best solution within 100 generations was found for combination of Population of 8, cross over rate of 0.8, mutation rate of 0.3, elite number of 1, and selection k of 3.
- Better solutions can be found with increases the size of the population but at a different investment of computational power.

1.3 *Solve the 2D Schwefel*

i)



0s- initial random set

1s- 1st generation

ii) Genetic algorithm is run with the parameters found in part-b

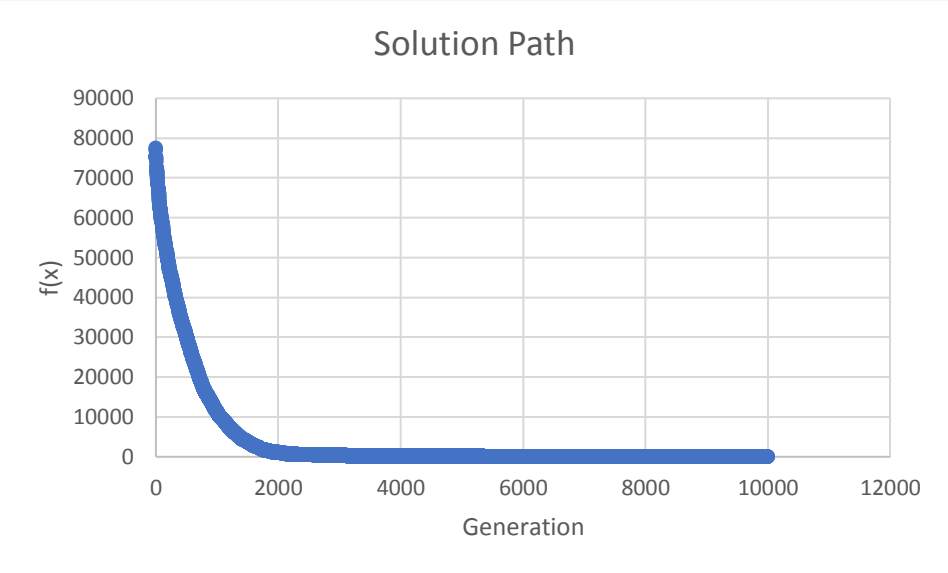
Parameter	Value
Population	30
Cross over Rate	0.8
Mutation Rate	0.3
Tournament k	3
Elite group	5
Max non Improving Gen	100
Solution	[420.97, 421]
Solution Value	0.000149
Generations taken	243

- **Quality and performance:** The algorithm reached near to global minimum effectively. But it did not reach or took many generations to reach global minimum even after reaching close to it.

1.4 *200D Schwefel*

One of the best solutions we obtained to 200D Schwefel problem is as follows

Parameter	Value
Populatin size	100
Cross over Rate	0.9
Mutation Rate	0.4
Tournament k	10
Elite group	10
Max non-Improving Gen	1000
Solution Value	**1.0071**
Generations taken	10000



Performance and Quality

- As discussed previously Genetic algorithm reached near to global minimum effectively.
- It can also be observed from the graph above that the algorithm reached near to global minimum at 2000th generation. Then even after 8000 generations it did not reach the global minimum.
- Combining genetic algorithm with hill climbing techniques may help GA to reach global optimum in more efficient way.

2 Particle Swarm Optimization Implementation

2.1 *Part a – Code Completion*

In this section, as the first step, the sample Python code which was provided be Dr. Nicholson has been completed based on the Particle Swarm Optimization (PSO) algorithm described in course. The given model is:

$$V_{i+1}^{t} = x\left(\omega V_i^t + \varphi_1 r_1 (P_i - X_i^t) + \varphi_2 r_2 \left(P_g - X_i\right)\right)$$

where x is the constriction factor, ω is the inertia weight, φ_1 and φ_2 are acceleration constants, r_1 and r_2 are uniform random number between 0 and 1, X_i^t is the current position of particle i, V_i^t is the current velocity of particle i, P_i is the local best position of particle i, P_g is the global best position and V_{i+1}^t is the next step velocity of particle i.

If the velocity goes beyond the maximum value it would be set to maximum value. The same procedure was implemented for particle positions in minimum and maximum values. Two criteria have been considered to stop the

1) Maximum number of iteration,
2) Stagnation

Following table summarizes the parameters and results:

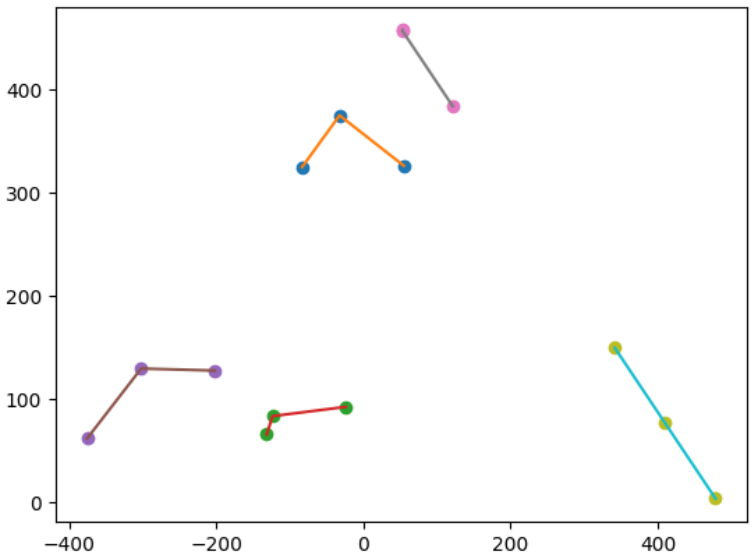
Parameter	Value
Dimension	2
Particles in Swarm	200
Cognitive Acceleration	2
Social Acceleration	2
Maximum Velocity	100
Maximum Iteration	500
Stagnation	100
Inertia Weight	1
Best Value	6.40e-5
Best Position	(420.98, 420.95)

2.2 *Part b - 2D Schwefel*

The same code as used in part a was implemented for this part with different parameters which are available in following table:

Parameter	Value
Dimension	2
Particles in Swarm	5
Cognitive Acceleration	2
Social Acceleration	2
Maximum Velocity	100
Maximum Iteration	6
Stagnation	100
Inertia Weight	1
Best Value	386.44
Best Position	(410.24, 76.88)

The first 3 positions and velocities have been recorded and plotted in python:



2.3 *Part c – PSO Local Best*

Two methods have been applied here: Ring and Von Neumann.

2.3.1 Ring

The ring topology was used for the PSO implementation every particle k has two neighbors, particles k − 1 and k + 1. Following table shows the results and parameters:

Parameter	Value
Dimension	2
Particles in Swarm	200
Cognitive Acceleration	2
Social Acceleration	3
Maximum Velocity	1
Maximum Iteration	1000
Stagnation	100
Inertia Weight	1
Best Value	2.67e-5
Best Position	(420.97, 420.97)

2.3.2 Von Neumann

The Von Neumann neighborhood topology was implemented where each particle k has four neighbors:

1) k – 1 to the left

2) k + 1 to the right

3) k + Δ down

4) k - Δ top

Where Δ is the distance determined by the swarm size which in this case was 200 with 20 rows and 10 columns. Following table shows the results and parameters:

Parameter	Value
Dimension	2
Particles in Swarm	200
Cognitive Acceleration	2
Social Acceleration	3
Maximum Velocity	1
Maximum Iteration	1000
Stagnation	100
Inertia Weight	1
Von Neumann Matrix Columns	10
Best Value	2.67e-5
Best Position	(420.97, 420.97)

2.4 *Part d - 2D and 200D Schwefel problem*

The problem has been solved for 2 and 200 dimensions with same parameters. The following table shows the comparative illustration of parameters and results:

Parameter	Value (2D)	Value (200D)
Dimension	2	2
Particles in Swarm	200	500
Cognitive Acceleration	2	2
Social Acceleration	3	1
Maximum Velocity	1	2000
Maximum Iteration	1000	20000
Stagnation	1000	100
Inertia Weight	1	1
Best Value	2.55e-5	28003.71
Best Position	(420.97, 420.97)	200D vector

Tuning the parameter is a very difficult job; we change the parameter with some different point respect to a base point. All the parameters were held unchanged but one and some values were selected for each parameter. Then model was ran with these parameters. Results shows that it is very time consuming process to get good results from this method for 200 dimensional problem.

3 Extra-Credit Option

Guided Local Search

Ans:

The strategy for the Guided Local Search algorithm is to use penalties to encourage a Local Search technique to escape local optima and discover the global optima. A Local Search algorithm is run until it gets stuck in some local optima. The features from the local optima are evaluated and penalized, the results of which are used in an augmented cost function employed by the Local Search procedure. The Local Search is repeated several times using the last local optima discovered and the augmented cost function that guides exploration away from solutions with features present in discovered local optima.

Input: $Iter_{max}, \lambda$
Output: S_{best}
$f_{penalties} \leftarrow \emptyset$
$S_{best} \leftarrow$ `RandomSolution()`
For ($Iter_i \in Iter_{max}$)
  $S_{curr} \leftarrow$ `LocalSearch`($S_{best}, \lambda, f_{penalties}$)
  $f_{utilities} \leftarrow$ `CalculateFeatureUtilities`($S_{curr}, f_{penalties}$)
  $f_{penalties} \leftarrow$ `UpdateFeaturePenalties`($S_{curr}, f_{penalties}, f_{utilities}$)
  If (`Cost`(S_{curr}) $\leq$ `Cost`(S_{best}))
    $S_{best} \leftarrow S_{curr}$
  End
End
Return (S_{best})

Pseudocode for Guided Local Search.

The Local Search algorithm used by the Guided Local Search algorithm uses an augmented cost function in the form

$$\mathbf{h(s) = g(s) + \lambda * \Sigma^{m}_{i=1}\ fi}$$

where **h(s)** is the augmented cost function,

g(s) is the problem cost function,

λ is the 'regularization parameter' (a coefficient for scaling the penalties),

s is a locally optimal solution of **m** features, and ***fi*** is the ***i*** 'th feature in locally optimal solution.

The augmented cost function is only used by the local search procedure, the Guided Local Search algorithm uses the problem specific cost function without augmentation.

In guided local search after for calculating local Optimum and if the points are selected the distance between two points in the feasible space is calculated using Euclidean Distance method which is:

Python code:

```
# Function which calculates the euclidean distance between two points
def euclideanDistance(v1, v2):
    sum = 0.0
    i = 0
    count = 0
    for coord1, coord2 in (zip(v1, v2)):
        while count < len(coord1):
            sum += ((coord1[i] - coord2[i])**2)
            i = i + 1
            count += 1
        break
    x = math.sqrt(sum)
    return x
```

Where in the above code v1 and v2 are two vectors

After calculating the distance as to delete that link and to make a new link we used function

```
# Function that deletes two edges and reverses the sequence in between the deleted edges
def stochasticTwoOpt(perm):
    result = perm[:]  # make a copy
    # select indices of two random points in the tour
    p1, p2 = (random.randrange(0, len(result)), random.randrange(0, len(result)))
    # do this so as not to overshoot tour boundaries
    exclude = set([p1])
    if p1 == 0:
        exclude.add(len(result) - 1)
    else:
        exclude.add(p1 - 1)
    if p1 == len(result) - 1:
        exclude.add(0)
    else:
        exclude.add(p1 + 1)
    while p2 in exclude:
        p2 = random.randint(0, len(result))
    # to ensure we always have p1<p2
    if p2 < p1:
        p1, p2 = p2, p1
    # now reverse the tour segment between p1 and p2
    result[p1:p2] = reversed(result[p1:p2])

    return result
```

This code after getting a local optimum it selects to points and deletes that edge and reverse the sequence in between the deleted edges.

Now as mentioned in the above function h(s) and g(s) are calculated using the functions:

```
def augmentedCost(perm, penalties, scalingFactor):
    distance, augmented = 0, 0
    size = len(perm)
    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 = perm[index1]
        v2 = perm[index2]
        d = euclideanDistance(v1, v2)
        distance += d
        augmented += d + (scalingFactor * penalties[index1][index2])
    return distance, augmented


def cost(candidate, penalties, scalingFactor):
    cost, augCost = augmentedCost(candidate , penalties, scalingFactor)
    return cost, augCost
```

Where in the above function scaling factor is nothing but **λ** parameter and penalties is also given as a parameter with initial values as zero's.

Local search code:

```
def localSearch(current, scalingFactor, penalties, maxNoImprove):
    count = 0
    while count < maxNoImprove:
        current_cost, current_augcost = cost(current, penalties, scalingFactor)
        candidate = []
        candidate = stochasticTwoOpt(current)
        candidate_cost,candidate_augcost = cost(candidate, penalties, scalingFactor)
        # Now we encourage diversification by aiming for a 'larger' augmented cost to escape local minima
        if candidate_augcost < current_augcost:
            # reset the counter to restart the search
            current = candidate[:]
            count = 0
        else:
            count += 1
    return current
```

Penalties are only updated for those features in a locally optimal solution that maximize utility, updated by adding 1 to the penalty for the future (a counter)

The utility for a feature is calculated as

$$U_{feature} = C_{feature}/1+P_{feature},$$

Where,

$U_{feature}$ is the utility for penalizing a feature (maximizing),

C feature is the cost of the feature, and

P feature is the current penalty for the feature

Python code:

```
def calculateFeatureUtilities(perm, penalties):
    size = len(perm)
    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 = perm[index1]
        v2 = perm[index2]
        utilities[index] = euclideanDistance(v1, v2) / (1 + penalties[index1][index2])

    return utilities
```

This code is to calculate the Utilities for a feature and if the feature is considered then we have to update the feature utility which is done by

```
def updateFeaturePenalties(perm, penalties, utilities):
    size = len(perm)
    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
```

Final search function to execute the Guided local search is :

```
def search(points, maxIterations, maxNoImprove, scalingFactor):
    # Create a random solution
    current = points[:]
    best = None
    # Initialize penalties. We create a list of lists. Each element of penalties is a list of penalties for each point
    penalties = [[0] * len(points)] * len(points)
    while maxIterations > 0:
        # Execute the local search taking into account lambda and penalties
        current = localSearch(current, scalingFactor, penalties, maxNoImprove)
        print(current)
        # Calculate feature utilities
        utilities = calculateFeatureUtilities(current , penalties)
        print(utilities)
        # Update feature penalties
        penalties = updateFeaturePenalties(current , penalties, utilities)
        print(penalties)
        # Compare current candidate cost with best and update if less
        current_cost, current_augcost = cost(current, penalties, scalingFactor)
        if best == None :
            current_best, current_best = cost(current, penalties, scalingFactor)
            if current_cost < current_best:
                best = current
        maxIterations -= 1
    return best
```

Results:

Parameter	Value (2D)	Value (200D)
Dimension	2	2
Particles in Swarm	200	200
Maximum Iteration	10000	10000
Best Value	1028.3	83903.3
Best Position	(483.29,332.2)	200D vector

Overall results:

Parameters	Genetic Algorithm		Particle swarm Algorithm		Guided Local search	
Dimension	2D	200D	2D	200D	2D	200D
Particles in swarm	200	200	200	500	200	200
Maximum Iterations	10000	10000	1000	20000	10000	10000
Best Value	0.000149	1.0071	2.55e-5	28003.71	1028.3	83903.3
Best Position	(420.97,421)	200D vector	(420.97,420.97)	200D vector	(483.29,332.2)	200D vector