**Adaptive Multimodal continuous Ant colony Optimization**

**“ALGORITHM 1”**

**CODE:**

P = []

sizeP = 80

M = 16

C1 = []

C2 = []

C3 = []

C4 = []

C5 = []

for i in range(0, sizeP):

ele = random.randint(0 , 30)

P.append(ele)

print(P)

R = random.randint(1,30)

print("The random reference number : " , R)

for i in range(0, sizeP):

P[i] = abs(P[i] - R)

print(P)

temp = P[0]

for i in range(0,sizeP):

if temp > P[i]:

temp = P[i]

print ("The nearest P to R is : " ,temp)

temp1 = 0;

for i in range(0, sizeP):

for j in range(i+1, sizeP):

if(P[i] > P[j]):

temp1 = P[i]

P[i] = P[j]

P[j] = temp1

print (P)

for j in range(0 , M):

num = P[j]

C1.append(num)

for k in range(M , 2\*M):

num = P[k]

C2.append(num)

for l in range(2\*M , 3\*M):

num = P[l]

C3.append(num)

for m in range(3\*M , 4\*M):

num = P[m]

C4.append(num)

for n in range(4\*M , 5\*M):

num = P[n]

C5.append(num)

print("Cluster 1 : " , C1)

print("Cluster 2 : " , C2)

print("Cluster 3 : " , C3)

print("Cluster 4 : " , C4)

print("Cluster 5 : " , C5)

**Explanation:**

In this algorithm, which is used for the clustering of the crowding as we take population and the size of cluster which I set as 16 was given as input. Then we randomly generate values for the population in range of 0 to 30 and randomly generate a reference point and after that calculate their distance from the reference point and then sort the population set and the first value of the set is the nearest one after sorting. So the output is a set of 5 crowds of size 16.

**“ALGORITHM 2”**

**CODE:**

P = []

N = 80

M = 16

for i in range(0, N):

ele = random.randint(0 , 30)

P.append(ele)

print(P)

for i in range(0,N):

if 0 <= P[i] and P[i]<2.5:

P[i]=80\*(2.5-P[i])

elif 2.5 <= P[i] and P[i]<5:

P[i]=64\*(P[i]-2.5)

elif 5 <= P[i] and P[i]<7.5:

P[i]=64\*(7.5-P[i])

elif 7.5 <= P[i] and P[i]<12.5:

P[i]=28\*(P[i]-7.5)

elif 12.5 <= P[i] and P[i]<17.5:

P[i]=28\*(17.5-P[i])

elif 17.5 <= P[i] and P[i]<22.5:

P[i]=32\*(P[i]-17.5)

elif 22.5 <= P[i] and P[i]<27.5:

P[i]=32\*(27.5-P[i])

elif 27.5 <= P[i] and P[i]<=30:

P[i]=80\*(P[i]-27.5)

print (P)

temp1 = 0;

for i in range(0, N):

for j in range(i+1, N):

if(P[i] > P[j]):

temp1 = P[i]

P[i] = P[j]

P[j] = temp1

print (P)

E = []

F = []

for j in range(0,M):

num = P[j]

E.append(num)

print(E)

for k in range(M,M+M):

num = P[k]

F.append(num)

print(F)

temp = P[0]

for i in range(0,N):

if temp < P[i]:

temp = P[i]

print ("The best specie is : " ,temp)

C1 = []

C2 = []

C3 = []

C4 = []

C5 = []

for j in range(0 , M):

num = P[j]

C1.append(num)

for k in range(M , 2\*M):

num = P[k]

C2.append(num)

for l in range(2\*M , 3\*M):

num = P[l]

C3.append(num)

for m in range(3\*M , 4\*M):

num = P[m]

C4.append(num)

for n in range(4\*M , 5\*M):

num = P[n]

C5.append(num)

print("Cluster 1 : " , C1)

print("Cluster 2 : " , C2)

print("Cluster 3 : " , C3)

print("Cluster 4 : " , C4)

print("Cluster 5 : " , C5)

**Explanation:**

In this algorithm, we use it for clustering of speciation as of species. We take population which is 80 and cluster size of 16 as input. Then after initializing them random values from 0 to 30 then we calculate their fitness and then sort them accordingly. Then we select the best specie in P as the new seed and then we return the 5 clusters of size 16 as output as a set of species.

**“ALGORITHM 3”**

**Explanation:**

This algorithm is used if the termination criterion is not met so I have coded all its implementation in the 4th algorithm so it is just a check of whether to terminate or not.

**“ALGORITHM 4”**

**CODE:**

index = 5

FSimax = []

FSimin = []

#/////////////////// C1 //////////////////

zmax = C1[M-1]

FSimax.append(zmax)

zmin = C1[0]

FSimin.append(zmin)

#/////////////////// C2 //////////////////

zmax = C2[M-1]

FSimax.append(zmax)

zmin = C2[0]

FSimin.append(zmin)

#/////////////////// C3 //////////////////

zmax = C3[M-1]

FSimax.append(zmax)

zmin = C3[0]

FSimin.append(zmin)

#/////////////////// C4 /////////////////

zmax = C4[M-1]

FSimax.append(zmax)

zmin = C4[0]

FSimin.append(zmin)

#/////////////////// C5 //////////////////

zmax = C5[M-1]

FSimax.append(zmax)

zmin = C5[0]

FSimin.append(zmin)

print ("FSimax : " , FSimax)

print ("FSimin : " , FSimin)

#///////////////// oi ////////////////////

power = 1

e = 2.718

oi = []

n = 0.01

wj = []

mule1 = []

mule2 = []

mule3 = []

mule4 = []

mule5 = []

sigma = []

ee = random.uniform(0.1 , 1)

#Eq 5 (Adaptive adjusting strategy for o) :

for i in range(0 , index):

power = -(FSimax[i] - FSimin[i]) / (FSmax - FSmin + n)

fn = 0.1 + 0.3 \* e \*\* power

oi.append(fn)

print("The value of oi is : " , oi)

#Eq 2 (The weight of the jth solution) :

for i in range ( 0 , index):

wjnum = ( 1 / oi[i] \* NP \* ( (2 \* 3.1416)\*\* 0.5)) \* e \*\* ( ((FSimax[i] - 1) \*\* 2) / (2 \* (oi[i]\*\*2) \* (NP \*\* 2)))

wj.append(wjnum)

print("The value of wj is : " , wj)

for x in range ( 0 , M ):

xj = C1[x]

xseed = C1[0]

if(random.uniform(0.1, 1) <= 0.5 ):

mule1.append(xj)

else :

F = random.uniform(0.1 , 1)

cc= xj + F \* (xseed - xj)

mule1.append(cc)

print("The mule 1 is : " , mule1)

print()

for x in range ( 0 , M ):

xj = C2[x]

xseed = C2[0]

if(random.uniform(0.1, 1) <= 0.5 ):

mule2.append(xj)

else :

F = random.uniform(0.1 , 1)

cc= xj + F \* (xseed - xj)

mule2.append(cc)

print("The mule 2 is : " , mule2)

print()

for x in range ( 0 , M ):

xj = C3[x]

xseed = C3[0]

if(random.uniform(0.1, 1) <= 0.5 ):

mule3.append(xj)

else :

F = random.uniform(0.1 , 1)

cc= xj + F \* (xseed - xj)

mule3.append(cc)

print("The mule 3 is : " , mule3)

print()

for x in range ( 0 , M ):

xj = C4[x]

xseed = C4[0]

if(random.uniform(0.1, 1) <= 0.5 ):

mule4.append(xj)

else :

F = random.uniform(0.1 , 1)

cc= xj + F \* (xseed - xj)

mule4.append(cc)

print("The mule 4 is : " , mule4)

print()

for x in range ( 0 , M ):

xj = C5[x]

xseed = C5[0]

if(random.uniform(0.1, 1) <= 0.5 ):

mule5.append(xj)

else :

F = random.uniform(0.1 , 1)

cc= xj + F \* (xseed - xj)

mule5.append(cc)

print("The mule 5 is : " , mule5)

print()

total = 0

for j in range(0 , M):

total = total + (abs(mule1[0] - mule1[j]) / (NP - 1))

tt = ee \* total

sigma.append(tt)

total = 0

for k in range(0 , M):

total = total + (abs(mule2[0] - mule2[k]) / (NP - 1))

tt = ee \* total

sigma.append(tt)

total = 0

for l in range(0 , M):

total = total + (abs(mule3[0] - mule3[l]) / (NP - 1))

tt = ee \* total

sigma.append(tt)

total = 0

for m in range(0 , M):

total = total + (abs(mule4[0] - mule4[m]) / (NP - 1))

tt = ee \* total

sigma.append(tt)

total = 0

for n in range(0 , M):

total = total + (abs(mule5[0] - mule5[n]) / (NP - 1))

tt = ee \* total

sigma.append(tt)

print("Sigma values for each Clusters are : " , sigma)

**Explanation:**

In this algorithm, we use it for the solution construction for ants. So, we take niching size NS, maximum and minimum fitness of the data set as input. In this we calculate the maximum and the minimum fitness of each cluster set and then calculate the value of σi through the equation 5 and then calculate the probabilities through the equations 2 and 1. Then through a nested loop we calculate the values of mule through the given conditions and equation 6 and then compute sigma from equation 4. We have NP number of solutions and their fitness as the output.

**“ALGORITHM 6”**

**CODE:**

G = []

FSmax = 0

FSmin = 0

NS = 0

rnum = 0

NP = 80

for i in range(0 , NP):

element = random.randint(0,30)

G.append(element)

print("Set G : " , G)

rnum = random.randint(0,NP)

print(rnum)

NS = G[rnum]

print("The niching size is : " , NS)

for i in range(0 , NP):

if 0 <= G[i] and G[i]<2.5:

G[i]=80\*(2.5-G[i])

elif 2.5 <= G[i] and G[i]<5:

G[i]=64\*(G[i]-2.5)

elif 5 <= G[i] and G[i]<7.5:

G[i]=64\*(7.5-G[i])

elif 7.5 <= G[i] and G[i]<12.5:

G[i]=28\*(G[i]-7.5)

elif 12.5 <= G[i] and G[i]<17.5:

G[i]=28\*(17.5-G[i])

elif 17.5 <= G[i] and G[i]<22.5:

G[i]=32\*(G[i]-17.5)

elif 22.5 <= G[i] and G[i]<27.5:

G[i]=32\*(27.5-G[i])

elif 27.5 <= G[i] and G[i]<=30:

G[i]=80\*(G[i]-27.5)

print ("The calculated distances from the reference point are : " , G)

temp = 0;

for i in range(0, NP):

for j in range(i+1, NP):

if (G[i] > G[j]):

temp = G[i]

G[i] = G[j]

G[j] = temp

print ("The sorted set after calculating distances from reference point are : " , G)

FSmax = G[NP-1]

FSmin = G[0]

print ("FSmax : " , FSmax)

print ("FSmin : " , FSmin)

**Explanation:**

In this algorithm, we use it for local search-based AMC-ACO as we take ant colony, niching size set G and local std sigma as input. Now we initialize random numbers to the set G and then calculate their fitness and then sort them out. Then we randomly select a number as the niching size NS and then call the algorithm 1 for partition into crowds and then algorithm 4 to construct NP solutions. Then for each new solutions as we put a loop for it and then we nearest solution to it and then replace it with the best. After that we call the algorithm 5 for adaptive local search. And then check for the termination criterion and conditions if met or not or else go to step 2 which is finding the minimum and the maximum fitness of the dataset. The output is the whole archive.

**“ALGORITHM 7”**

**CODE:**

G = []

FSmax = 0

FSmin = 0

NS = 0

rnum = 0

NP = 80

for i in range(0 , NP):

element = random.randint(0,30)

G.append(element)

print("Set G : " , G)

rnum = random.randint(0,NP)

print(rnum)

NS = G[rnum]

print("The niching size is : " , NS)

for i in range(0 , NP):

if 0 <= G[i] and G[i]<2.5:

G[i]=80\*(2.5-G[i])

elif 2.5 <= G[i] and G[i]<5:

G[i]=64\*(G[i]-2.5)

elif 5 <= G[i] and G[i]<7.5:

G[i]=64\*(7.5-G[i])

elif 7.5 <= G[i] and G[i]<12.5:

G[i]=28\*(G[i]-7.5)

elif 12.5 <= G[i] and G[i]<17.5:

G[i]=28\*(17.5-G[i])

elif 17.5 <= G[i] and G[i]<22.5:

G[i]=32\*(G[i]-17.5)

elif 22.5 <= G[i] and G[i]<27.5:

G[i]=32\*(27.5-G[i])

elif 27.5 <= G[i] and G[i]<=30:

G[i]=80\*(G[i]-27.5)

print ("The calculated distances from the reference point are : " , G)

temp = 0;

for i in range(0, NP):

for j in range(i+1, NP):

if (G[i] > G[j]):

temp = G[i]

G[i] = G[j]

G[j] = temp

print ("The sorted set after calculating distances from reference point are : " , G)

FSmax = G[NP-1]

FSmin = G[0]

print ("FSmax : " , FSmax)

print ("FSmin : " , FSmin)

**Explanation:**

In this algorithm, we use it for local search-based AMC-ACO as we take ant colony, niching size set G and local std sigma as input. Now we initialize random numbers to the set G and then calculate their fitness and then sort them out. Then we randomly select a number as the niching size NS and then call the algorithm 2 for partition into species and then algorithm 4 to construct NP solutions. Then for each new solutions as we put a loop for it and then we nearest solution to it and then replace it with the best. After that we call the algorithm 5 for adaptive local search. And then check for the termination criterion and conditions if met or not or else go to step 2 which is finding the minimum and the maximum fitness of the dataset. The output is the whole archive. This algorithm is identical to the algorithm 6, the only difference is of the crowds and species which is of the algorithm 1 and 2.