Shirin Mohebbi

Evolutionary Computing

Assignment #3

problem: System reliability optimization

parameter: number of redundant components(ni), component reliability(ri)

fitness function: fitness function calculate based on two thing, 1.constraints 2.objective, if any of the constraints voilated fitness will be -10.

```
def fitness(self, chrom):
    c1 = self.constraint1(chrom[2])
    c2 = self.constraint2(chrom[0], chrom[2])
    c3 = self.constraint3(chrom[2])
    if (c1 and c2 and c3):
        if (self.obj == 1):
            return self.obj1(chrom[0], chrom[2])  #complex system
        else:
            return self.obj2(chrom[0], chrom[2])  #series system
    else:  #if any constraint voilated, with would be negative number
        return -10
```

objective function system1:

$$Max\ f(r.\ nc) = R_1R_2 + R_3R_4 + R_1R_4R_5 + R_2R_3R_5 - R_1R_2R_3R_4 - R_1R_2R_3R_5 - (2)\ R_1R_2R_4R_5 - R_1R_3R_4R_5 - R_2R_3R_4R_5 + 2R_1R_2R_3R_4R_5$$

```
def obj1(self, r, n):
    R = []
    for i in range(len(r)):
        R.append(1 - (1 - r[i])**n[i])
    res = (R[0] * R[1]) + (R[2] * R[3]) + (R[0] * R[3] * R[4]) + (R[1] * R[2] * R[4])
    - (R[0] * R[1] * R[2] * R[3]) - (R[0] * R[1] * R[2] * R[4]) - (R[0] * R[1] * R[3] * R[4])
    - (R[0] * R[2] * R[3] * R[4]) - (R[1] * R[2] * R[3] * R[4]) + (2 * R[0] * R[1] * R[2] * R[3] * R[4])
    return res
```

objective function system2:

$$Max\, f(r.n)= \prod R_i = \prod(1-(1-r_i)^{n_i})$$

```
def obj2(self, r, n):
    s = 1
    for i in range(self.m):
        a = (1 - (1 - r[i])**n[i])
        s *= a
    return s
```

and the constrains is as below

```
def constraint1(self, n):
    s = 0
    for i in range(self.m):
        s += (self.wv2[i] * n[i]**2)
    if s - self.vAll <= 0:
        return True
    return False
```

```
def constraint2(self, r, n):
    s = 0
    for i in range(self.m):
        a = self.alpha[i] * (float(-1000)/np.log(r[i]))**self.b[i] * (n[i] + np.exp(0.25 * n[i]))
        s += a
    if s - self.cAll <= 0:
        return True
    return False
```

```
def constraint3(self, n):
    s = 0
    for i in range(self.m):
        s += (self.w[i] * n[i] * np.exp(0.25 * n[i]))
    if s - self.wAll <= 0:
        return True
    return False
```

chromosome: each chromosome contain of 4 parts:

1.ri 2.sigma ri 3.ni 4.sigma ni

each part is length of 5(m)

initial ri are randomly between 0 and 1

initial ni are randomly between 1 and m(5)

initial sigma ri is randomly between -0.01 and 0.01

initial sigma ri is randomly between -1 and 1

```
def initialization(self, m):
    #each chrom is like this: [[r1,r2,r3,r3,r5],
    pop = []
    for i in range(self.numPop):
        r = []
        sr = []
        n = []
        sn = []
        for j in range(5):
            r.append(random.uniform(0, 1))
            n.append(random.randint(1, m))
            sr.append(random.uniform(-0.01, 0.01))
            sn.append(random.uniform(-1,1))
        pop.append([r, sr, n, sn])
    self.population = pop
```

population:100

parent selection: random uniform

no. of offsprings: 700 (7 * population)

cross over: local intermediary

```
def crossOver(self, parents):   #local intermediary
    #[[r1,r2,r3,r4,r5],[sr1,sr2,sr3,sr4,sr5],[n1,n2,n3,n4,n5],[sn1,sn2,sn3,sn4,sn5]]
    r = []
    sr = []
    n = []
    sn = []
    for i in range(5):
        newR = ( parents[0][0][i] + parents[1][0][i] )/ 2
        if newR > 1:  #r should be between 0 and 1
            newR = 1
        elif newR < 0:
            newR = 0
        r.append(newR)
        newN = round( ( parents[0][2][i] + parents[1][2][i] ) / 2 )  #n should be integer number
        # if newN > self.m:
        #   newN = self.m
        if newN < 1:
            newN = 1
        n.append(newN)
        sr.append(( parents[0][1][i] + parents[1][1][i] )/2)
        sn.append(( parents[0][3][i] + parents[1][3][i] )/2)
    child = [r, sr, n, sn]
    return child
```

mutation: Uncorrelated mutation with n sigma's

for mutation we first mutate sigmas, and then with new sigmas we mutate ri and ni

$$1)\ \sigma'_i = \sigma_i \cdot \exp(\tau' \cdot N(0,1) + \tau \cdot N_i(0,1))$$

$$2)\ x'_i = x_i + \sigma'_i \cdot N_i(0,1)$$

$$t = 1/(2n)^{1/2} \text{ and } t'=1/(2n^{1/2})^{1/2}$$

and if sigma' < ep : sigma' = ep to avoid very small sigmas

```
def mutation(self, child):
    #mutate sigmas:
    overAllLearning = 0.47 * np.random.normal(0.0, 1.0)
    for i in range(5):
        coordinateLearning = 0.22 * np.random.normal(0.0, 1.0)
        child[1][i] = child[1][i] * np.exp(overAllLearning + coordinateLearning)
        if (child[1][i] < self.epsilon):
            child[1][i] = self.epsilon
        coordinateLearning = 0.22 * np.random.normal(0.0, 1.0)
        child[3][i] = child[3][i] * np.exp(overAllLearning + coordinateLearning)
        if (child[3][i] < self.epsilon):
            child[3][i] = self.epsilon
    #mutate x:
    for i in range(5):
        child[0][i] = child[0][i] + (child[1][i] * np.random.normal(0.0, 1.0))
        if child[0][i] < 0:
            child[0][i] = 0
        elif child[0][i] > 1:
            child[0][i] = 1
        child[2][i] = round(child[2][i] + (child[3][i] * np.random.normal(0.0, 1.0)))
        if child[2][i] < 1:
            child[2][i] = 1
        # elif child[2][i] > self.m:
        #   child[2][i] = self.m
```

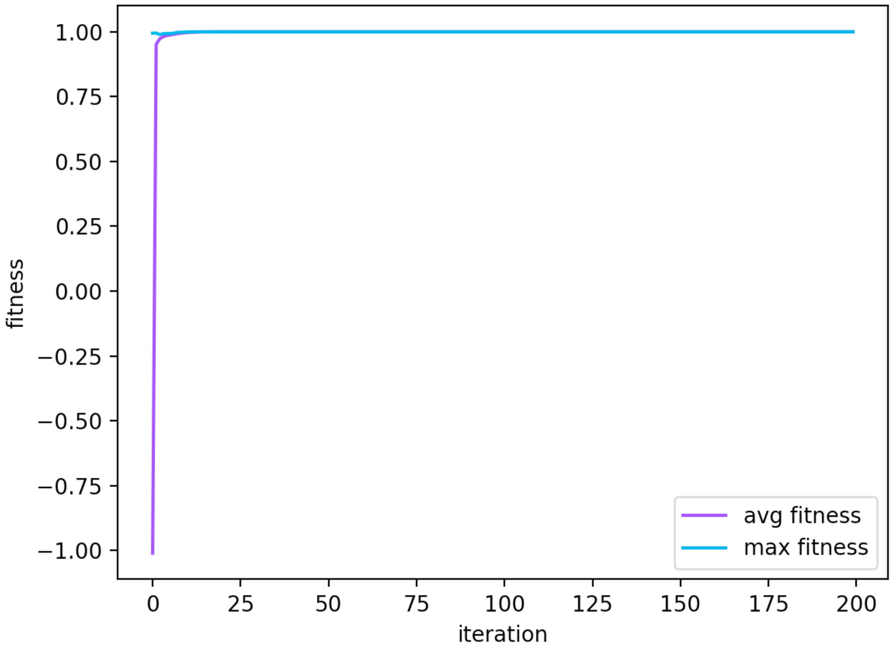
survival selection: set of children only: (μ, λ) selection

```
def survivalSelection(self):
    childWithFitness = []
    for child in self.offSprings:
        f = self.fitness(child)
        # print ("f", f)
        childWithFitness.append((child, f))
    childWithFitness.sort(key=lambda tup: tup[1], reverse=True)
    newpop = []
    avgFitness = 0
    for i in range(numPop):
        newpop.append(childWithFitness[i][0])
        avgFitness += childWithFitness[i][1]
    self.avgFitnesses.append(avgFitness/self.numPop)
    self.maxFitness.append(childWithFitness[0][1])
    self.population = newpop
```

results

complex system average and max fitness within 200 iterations:

max fitness: 0.9997351104120658



series system average and max fitness within 200 iterations:

max fitness: 0.9271813382353742