TSP的GA算法实现

给定TSP的一个实例$C=(c_{ij})_{n\times n}$,为n个城市之间的距离矩阵，对任一0-1解矩阵$S=(s_{ij})_{n\times n}$,其中s_{ij}取1若解包含城市i->j的路线，否则取0.那么路线距离长度为$d(S)=C*S$.

为实现遗传算法，做如下定义

染色体:$\sigma(S)=(s_1s_2\cdots s_n), s_{ij}=1 \iff s_is_j\in\sigma(S)$

适应度函数:定义为 $f(S)=n*C_m-C*S, C_m=\max\limits_{i,j} c_{ij}$,显然$f>0$,并且f的最大值恰好是TSP的最优解

遗传算子:

1.复制算子:按照适应度选择一个个体进行复制

2.选择算子:选择father和mother，son的前半部分和father一样，后半部分按mather的序对father的元排序

3.突变算子:随机选择$i\in\{1,2,\cdots n-1\}$,交换s_i,s_{i+1}

In [ ]:
```python
class SA_TSP():
    from random import sample, choice, choices, uniform
    def __init__(self, C):
        self.maxDepth = 5 # 最大迭代深度
        self.target = 1 # 目标优化值
        self.dim = len(C) # 维度
        self.Top = self.dim * max(max(row) for row in C)
        self.C = C # 距离矩阵
        self.Pm = 0.1
        self.Pc = 0.5
        self.Py = 1 - self.Pm - self.Pc
        # 每个sigma是一个list
        self.sigmas = []

    # 种群初始化
    def setX(self, target = 1 , X0 = [], depth = 1000,):
        if len(X0) == 0:
            self.sigmas.append(self.sample(range(self.dim),self.dim))
        else:
            self.sigmas = X0
        self.maxDepth = depth
        self.target = target

    # 设置种群行为概率
    def setProbability(self, Pc, Pm):
        self.Pm = Pm
        self.Pc = Pc
        self.Py = 1 - Pm - Pc
    # 染色体编码
    def code(self, S):
        # 设定起点都在城市0
        current_city = 0
        chromosome = [current_city]
        while len(chromosome) < self.dim:
            next_city = [col for col in range(self.dim) if S[current_city
            current_city = next_city
            chromosome.append(current_city)
        return chromosome

    # 舒适度
    def comfort(self, sigma):
        S = [[0 for _ in range(self.dim)] for _ in range(self.dim)]
        for i in range(len(sigma)-1):
            S[sigma[i]][sigma[i+1]] = 1
        S[sigma[-1]][sigma[0]] = 1
        s = self.Top - sum([self.C[i][j] for  i in range(self.dim) for j
        return s

    # 复制函数
    def Copy(self):
        newSigma = self.choices(self.sigmas, [self.comfort(sigma) for sig
        return newSigma

    # 选择函数
    def Select(self):
        father,mather = self.choices(self.sigmas, [self.comfort(sigma) fo
        son = [-1 for _ in range(self.dim)]
        mid = self.dim // 2
        son[:mid] = father[:mid]
        son[mid:] = sorted(father[mid:], key=lambda x: mather.index(x))
        return son

    # 突变函数
    def Mutation(self):
        newSigma = self.choice(self.sigmas)
        i,j = self.choices(range(self.dim),k=2)
        i,j = min(i,j),max(i,j)
        newSigma[i],newSigma[j] = newSigma[j],newSigma[i]
        return newSigma

    # 获取当前种群中的最优种群舒适度
    def best(self):
        a = max([self.comfort(sigma) for sigma in self.sigmas])
        return a

    def main(self):
        if len(self.sigmas) == 0:
            self.setX()
        # 当前迭代深度
        depth = 0
        bestRoute = self.sigmas[0]
        while depth < self.maxDepth and self.best() > self.target:
            newSigmas = []
            for _ in range(self.dim):
                rNum = self.uniform(0,1)
                if rNum < self.Py : # 复制
                    newSimga = self.Copy()
                elif rNum < self.Py + self.Pc:
                    newSimga = self.Select()
                else:
                    newSimga = self.Mutation()
                newSigmas.append(newSimga)
            depth += 1
            self.sigmas = newSigmas
            level = [self.comfort(item) for item in self.sigmas]
            bestRoute = max(self.sigmas,key=self.comfort) if max(level) >

        return self.Top - self.comfort(bestRoute)
```

In [ ]:
```
import math
city = [(12,12),(18,23),(24,21),(29,25),(31,52),(36,43),(37,14),(42,8),(5
    (62,53),(63,19),(69,39),(81,7),(82,18),(83,40),(88,30)]
# city = [(0,0),(1,1),(3,4)]

def d(i,j):
    city1,city2 = city[i],city[j]
    return math.sqrt( math.pow(city1[0]-city2[0], 2) + math.pow( city1[1]

M = [[0 for _ in range(len(city))] for _ in range(len(city))]

for i in range(len(city)):
    for j in range(len(city)):
        M[i][j] = d(i,j)


a = SA_TSP(M)
a.setX(target=1,X0=[],depth=1000)
result = a.main()
print("最优解：{},路线长度{}".format(str(max(a.sigmas,key=a.comfort))[1:-2].
```

```
最优解：13-> 14-> 9-> 2-> 3-> 0-> 1-> 5-> 15-> 7-> 11-> 6-> 10-> 4-> 8-> 1,
路线长度356.4133387461901
```

SA算法计算出的最优解为231左右，误差较大.原因可能在与GA的设计思路不合理，具体表现在染色体设计，也许可以取矩阵S为染色体

P5 求解矩阵积和式

In [ ]:
```
import numpy as np
from itertools import permutations
from random import choice, choices, sample
class perm():

    def __init__(self, n, m):
        self.Pm = 0.01
        self.Pc = 0.4
        self.T = 100
        self.target = 10000
        self.n = n
        self.m = m
        self.perms = []

    # 产生n:m的0-1矩阵
    def getMatrix(self):
        matrix = np.zeros((self.n, self.n))
        indices = np.random.choice(self.n*self.n, self.m, replace=False)
        matrix.flat[indices] = 1
        return matrix

    # 随机生成初始值
    def setInitial(self):
        for _ in range(12):
            self.perms.append(self.getMatrix())

    # 积和式值 适应度
    def getFitness(self, matrix):
        if any( [ sum([matrix[i][j] for j in range(self.n) ]) == 0 for i
            any( [ sum([matrix[i][j] for i in range(self.n)]) == 0 for j
            perm_sum = 0
        else:
            perm_sum = 0
            all_permutations = permutations(range(self.n))
            for permutation in all_permutations:
                product = 1
                for i in range(self.n):
                    product *= matrix[i, permutation[i]]
                perm_sum += product
        return perm_sum

    # 突变
    def mutation(self):
        newMatrix = choices(self.perms,[self.getFitness(perm) for perm in
        i, j = sample(range(self.n), 2)
        direction = choice(['N', 'W', 'S', 'E'])

        # 获取当前位置和相邻位置的值
        current = newMatrix[i][j]
        if direction == 'N':
            next_i = (i - 1) % self.n
            newMatrix[i][j], newMatrix[next_i][j] = newMatrix[next_i][j],
        elif direction == 'W':
            next_j = (j - 1) % self.n
            newMatrix[i][j], newMatrix[i][next_j] = newMatrix[i][next_j],
        elif direction == 'S':
            next_i = (i + 1) % self.n
            newMatrix[i][j], newMatrix[next_i][j] = newMatrix[next_i][j],
        else:  # direction == 'E'
            next_j = (j + 1) % self.n
            newMatrix[i][j], newMatrix[i][next_j] = newMatrix[i][next_j],

        return newMatrix
    # 交叉
    def crossover(self):
        A,B = choices(self.perms,[self.getFitness(perm) for perm in self.
        Alst = list(A.flat)
        Blst = list(B.flat)
        i,j = sample(range(self.n*self.n),2)
        Alst = Alst[i:] + Alst[:i]
        Blst = Blst[j:] + Blst[:j]
        for index in range(self.n * self.n):
            if Alst[index] != Blst[index]:
                Alst[index],Blst[index] = Blst[index],Alst[index]
        C = np.array(Alst).reshape(self.n,self.n)
        return C
    # 选择
    def copy(self):
        newMatrix = choices(self.perms,[self.getFitness(perm) for perm in
        return newMatrix

    # 获取当前种群中的最优种群舒适度
    def best(self):
        BestComfort = max([self.getFitness(perm) for perm in self.perms])
        return BestComfort

    # 主函数
    def main(self):
        self.setInitial()
        depth = 0
        while depth < self.T and self.best() < self.target:
            newPerms = []
            for _ in range(self.m):
                rNum = np.random.rand()
                if rNum < self.Pm:
                    newPerm = self.mutation()
                elif rNum < self.Pc + self.Pm:
                    newPerm = self.crossover()
                else:
                    newPerm = self.copy()
                newPerms.append(newPerm)
            depth += 1
            self.perms = newPerms
        return self.best()
```

In [ ]:
```
MA = perm(8,20)
result = MA.main()
print("the biggest M&A is :",result)
```

```
the biggest M&A is : 8.0
```

In [ ]:
```
MA = perm(8,20)
result = MA.main()
print("the biggest M&A is :",result)
```

多次运行，得到的最大结果为 16

P6

In [ ]:
```
from scipy.optimize import curve_fit
import numpy as np
# target_func
def f(t,A,r,K):
    return A * np.exp( r * t) / ( 1 + (A / K) * np.exp( r * t) )

populations = [3929214,5308483,7239881,9638453,12866020,17069453,23191876
    50155783,62947714,75994575,91972266,105710620,122775046,131669275,151
        203302031,226545805,248709873]

X = np.array(list(range(10,220,10)))
Y = np.array(populations)
popt, pcov = curve_fit(f,X,Y)
```

```python
A,r,K = popt[0],popt[1],popt[2]
print("A = {}, r = {}, K = {}".format(A, r, K))
```

```
A = 5716745.358744745, r = 0.022703488052232292, K = 387967738.12121123
```

```
/var/folders/2z/kvvrz9fx7575sz8vyx1php7c0000gn/T/ipykernel_56354/125126692
3.py:5: RuntimeWarning: overflow encountered in exp
  return A * np.exp( r * t) / ( 1 + (A / K) * np.exp( r * t) )
/var/folders/2z/kvvrz9fx7575sz8vyx1php7c0000gn/T/ipykernel_56354/125126692
3.py:5: RuntimeWarning: invalid value encountered in divide
  return A * np.exp( r * t) / ( 1 + (A / K) * np.exp( r * t) )
```

这里year是映射$year(x) = 1780 + x$