免疫遗传算法的设计与实现

姓名：胡瑞

学号：2020300004044

1 概述

人工免疫算法是模拟生物免疫系统智能行为的算法；而遗传算法则是通过模拟生物进化中的自然选择和交配变异的算法。二者在都是确定性和随机性选择相结合并具有勘测与开采能力的启发式随机搜索算法。

虽然在生物学范畴上，免疫与遗传之间有着很大的差异，但从它们之中抽象出的人工免疫算法和遗传算法在思路与实现上高度相似。另一方面，两种算法又具有各自特点，二者结合能够做到优势互补。

本次实验将以经典的旅行社问题（traveling salesman problem, TSP）为优化目标，设计并实现基本的免疫遗传算法。同时，通过阅读近几年的文献，我们将对算法中的某些算子以及算法框架进行调整与优化，并展现改进后的效果。

2 生物遗传机制和免疫机制

2.1 生物遗传机制

达尔文的自然选择学说认为，在变化的生活条件下，生物几乎都表现出个体差异，并有过度繁殖的倾向。在生存斗争过程中，具有有利变异的个体能生存下来并繁殖后代，具有不利变异的个体则逐渐被淘汰。

其中，繁殖涉及到生物遗传过程。生物遗传是指亲代表达相应性状的基因通过无性繁殖或有性繁殖传给后代。从而使后代获得亲代遗传信息的现象。通过遗传，个体之间的差异可以积累并通过自然选择使物种进化。

在有性繁殖的减数分裂过程中，来自双亲的染色体之间可能会发生等位基因的交叉互换与基因突变。新的染色体将进入子代细胞，造成子代个体与双亲之间的差异。

2.2 生物免疫机制

生物学上，免疫指的是机体免疫系统识别自身与异己物质，并通过免疫应答排除抗原性异物，以维持 机体生理平衡的功能的过程。免疫细胞中包括了T细胞和B细胞。T细胞可以识别抗原，并刺激B细胞系，使之火花、增殖并产生特异性抗体来排除抗原。B细胞还会分化为记忆细胞。当同类抗原再次入侵时，记忆细胞能够被激活并产生大量抗体，缩短免疫反应的时间。

另一方面，大量产生的抗体会抑制抗原对免疫细胞的刺激，从而抑制抗体的产生；同时产生的抗体之间也有相互刺激和抑制的关系。抗原和抗体、抗体和抗体之间的制约关系使得免疫反应能维持到一定强度，保证机体的免疫平衡。

3 免疫遗传算法

3.1 基本原理

3.1.1 遗传算法

如前述，遗传算法是借鉴了生物进化中的某些现象而设计出的算法。这些现象包括了遗传、变异、杂交和自然选择等。

对于一个待优化的问题，将一个可能的解称为个体，可抽象为一条染色体。首先会随机生成一个拥有许多个体的种群。接着藉由适应度函数来评价某个个体的适应度，并保留适应度较大的个体。其次让被选出的个体进行杂交以及变异以产生子代。子代种群又会重复选择、杂交、变异的操作，直到结果收敛为止。

3.1.2 免疫算法

在免疫算法中，视待优化问题为抗原，一个可能的解为抗体。首先会随机产生初代的抗体。接着计算抗原与抗体之间的亲和力，并将与抗原有最大亲和力的抗体加入记忆库。如前述，高亲和力的抗体将受到促进而高浓度的抗体将受到抑制。最后根据亲和度和浓度来选择子代抗体。子代抗体又将重复该过程，直到结果收敛为止。

3.1.3 免疫遗传算法

遗传算法和免疫算法在思路上有诸多相似之处。二者都会随机初始化一些个体，并通过某种评价方式来选择个体。接着使用某些算子来更新种群。如此反复，直到结果收敛。

其中，遗传算法容易陷入局部收敛，这是因为种群中适应度大的越有可能存活，最终导致所有个体趋同。而免疫算法中的记忆机制和浓度控制机制则有助于保持种群的多样性。免疫遗传算法就是在保留原本的选择、交叉和变异三个算子的基础上加入了记忆和浓度机制。

3.2 实现步骤

3.2.1 染色体编码

染色体编码的方式要依据具体求解的问题而定。对于旅行商问题，可以采用实数编码。对于N个要遍历的城市，可以用一个长度为N的序列表示一个可能的解。例如$(1,2,3,\ldots,N)$，其含义就是从城市1出发，接着前往城市2，然后城市3.....最终抵达城市N。

3.2.2 适应度

适应度函数用于衡量某个个体的优劣程度。每个城市的位置可以用二维空间的坐标表示，于是可以计算城市i和城市j之间的欧氏距离：

$$distance(i,j)=\sqrt{(x_i-x_j)^2+(y_i-y_j)^2}$$

而某个个体的适应度则可以表示遍历所有城市距离的倒数，即：

$$fitness=\frac{1}{\sum distance(i,j)}$$

遗传算法仅依据适应度来选择个体。免疫遗传算法则在此基础上引入亲和度、浓度以及激励度。

亲和度就是上述适应度，表示某个抗体与抗原的亲和程度，即某个个体的好坏程度。

$$affinity = \frac{1}{\sum distance(i,j)}$$

某个个体的浓度则是与之相似个体占种群中全部个体的比重。在旅行商问题中，可以依据两个个体对应的总距离之差来判断相似度，差值越小则越相似。然后再设定一个阈值，如果二者距离之差小于该阈值则认为它们属于一类。

$$density(k) = \frac{N_k}{N}$$

其中N表示个体总数，N_k表示与第k个个体相似（包括该个体）的数目。

激励度则是最终用于选择个体的标准，受到亲和度和浓度的共同调节。亲和度大则激励度大；浓度高则激励度小。

$$simulation = alpha \times affinity - beta \times density$$

其中alpha与beta均为正实数，取值依实际需要而定。

在实际实现中并没有直接使用亲和度，而是亲和度概率，即某个个体亲和度占群体总亲和度的比重。另外，考虑到上述方法计算出的激励度可能出现负值，会对其做线性归一化，即：

$$simulation\_new = \frac{simulation\_old - simulation\_min}{simulation\_max - simulation\_min}$$

这样的到的激励度的值介于$[0,1]$，可以用作选择种群中个体的概率。

3.2.3 交叉

交叉即交换双亲染色体的等位基因片段，可以让子代继承双亲的基因片段。但在旅行商问题中，直接交叉会引起重复。针对该问题提出的交叉方式有很多，其中一种简单的方法如下。假设有双亲$Parent1$和$Parent2$，首先随机产生两个交叉点。然后将$P1$中位于交叉点之间的片段复制到子代相同的位置。接着从第二个交叉点开始循环遍历$P2$，并选出不再子代序列中的城市编号加入到子代中。

例如有双亲：

$$Parent1(1,2,3,4,5,6,7,8,9)$$
$$Parent2(7,4,1,9,2,5,3,6,8)$$

如果连交叉点分别为4和6，则依据该交叉方法得到的子代为：

$$Child(1,9,2,4,5,6,3,8,7)$$

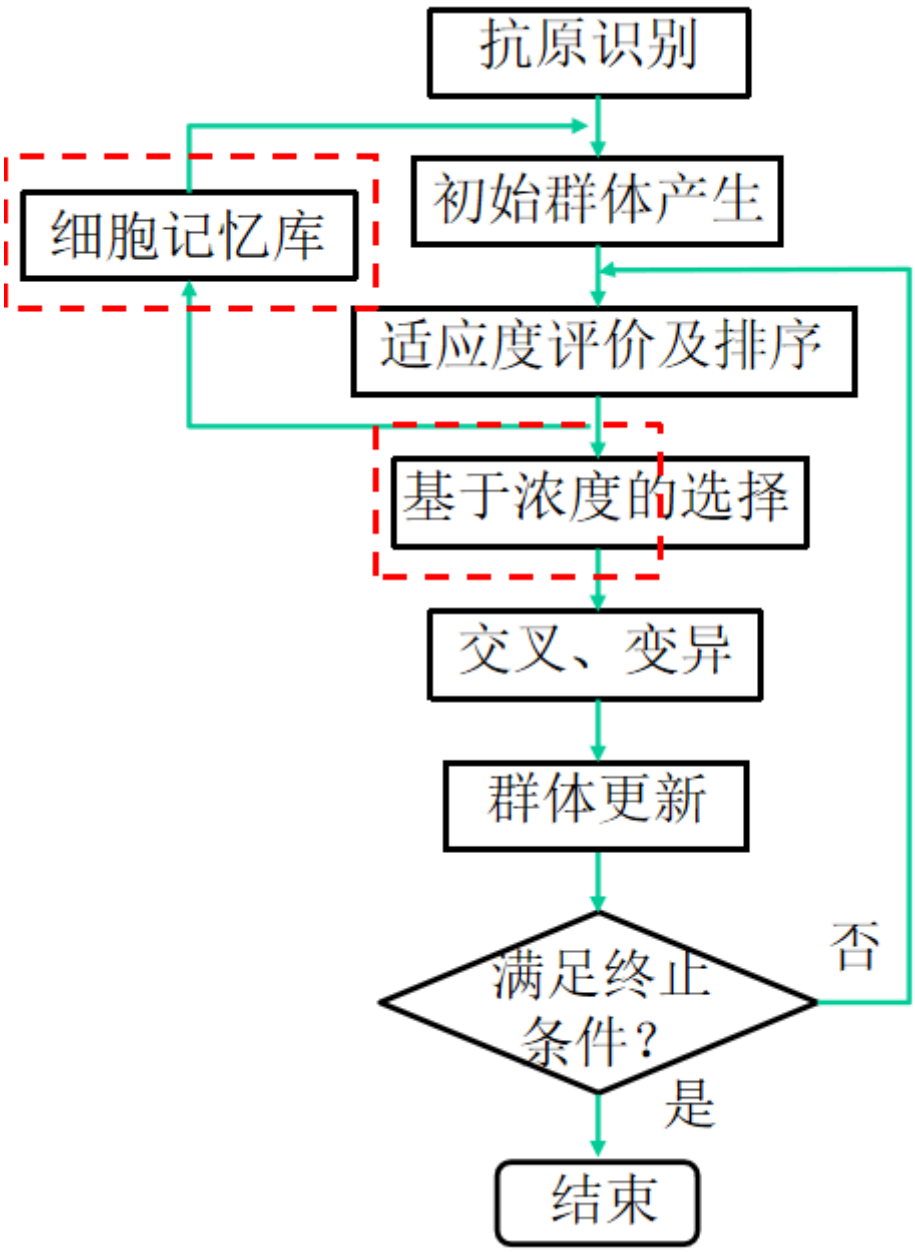
3.2.4 变异

变异中的基因突变指的是某个基因突变为其等位基因。在旅行社问题中，为了防止重复，突变的做法是随机产生两个突变位点，然后交换两个位点对应的城市编号。例如个体$(1,2,3,4,5,6,7,8,9)$在第1和第9处发生变异后得到的个体为$(9,2,3,4,5,6,7,8,1)$。变异有助于增加种群中个体的多样性。

3.2.5 记忆

记忆机制来自免疫算法。首先依据适应度，将旧种群的个体和记忆库中的个体排序，选出适应度最大的几个个体来更新记忆库。下次更新种群时，用记忆库中的个体替换掉现有种群中适应度较低的个体。

3.2.1 算法流程



免疫遗传算法的流程图如上所示。首先随机初始化种群。然后根据适应度函数来评价并排序每个个体，将优秀个体保存到记忆库中。接着计算每个个体的激励度以进行基于浓度的选择。选择出来的个体经过交叉变异后得到子代种群，并与记忆库中的个体共同组成下一次迭代的新种群。

4 仿真实验

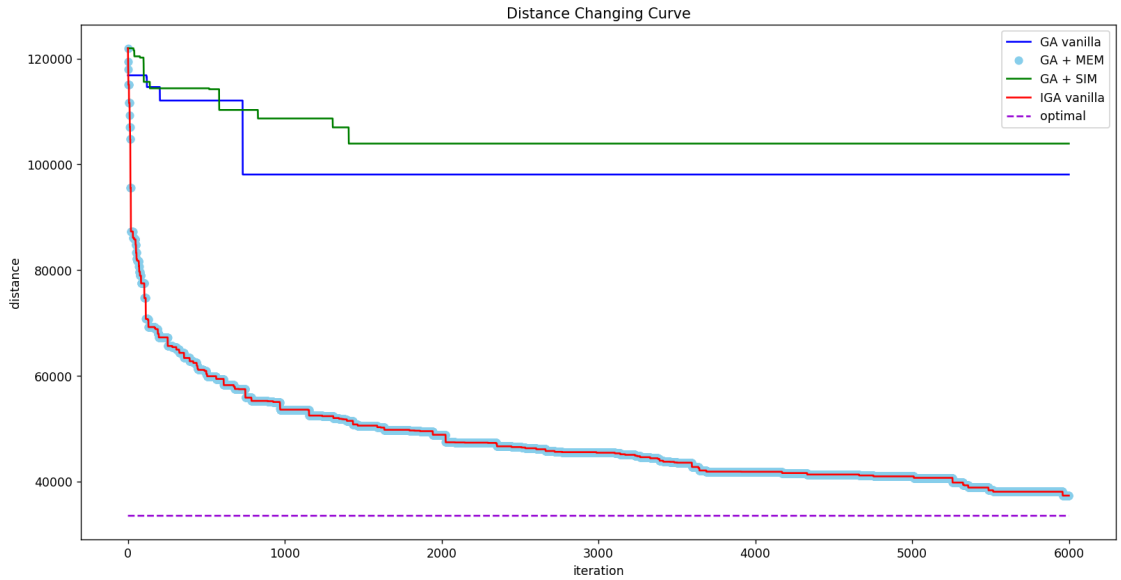
4.1 算法有效性

相较于基本的遗传算法，免疫遗传算法的两大改进之处在于加入了记忆库和基于浓度的选择机制。其中记忆库能够保留历史最优个体，并在每次种群更新时都加入到新种群中，从而有效防止种群退化并加速收敛。而基于浓度的选择机制则有助于维持种群中个体的多样性，防止个体趋同。

用于本次实验的旅行商问题中，共有48个城市，最优解（optimal）对应的总距离为33503。

首先进行的是遗传算法（GA vanilla）、带记忆库的遗传算法（GA + MEM）、基于浓度选择机制的遗传算法（GA + SIM）和免疫遗传算法（IGA vanilla）的实验结果之对比。其中免疫遗传算法也可以认为是基于浓度选择机制的，带记忆库的遗传算法（GA + MEM + SIM）。

实验结果如下图所示。



可以看到，在相同迭代次数下，基本的遗传算法出现了“早熟”现象，且与最优解之间有很大差距；一旦加入记忆库用于保存历史优秀个体后，算法收敛速度立刻有了提升，且有收敛到最优解的趋势；但基于浓度选择机制的算法表现反而差于基本的遗传算法，猜测可能的原因是该机制进一步抑制了种群找到优秀个体。有趣的是，免疫遗传算法和仅加入记忆库的遗传算法收敛情况完全一致，可见记忆库功能的强大。

实验中设定的超参数如下。

算法	种群大小	alpha	beta	delta	交叉概率	变异概率		迭代次数
GA vanilla	200	1.0	0.0	0.0	0.8	0.2		6000
GA + MEM	200	1.0	0.0	0.0	0.8	0.2		6000
GA + SIM	200	2.0	1.0	0.05	0.8	0.2		6000
IGA	200	2.0	1.0	0.05	0.8	0.2		6000

其中alpha和beta分别是基于浓度选择机制中亲和度和浓度的权重；delta是计算浓度时的阈值。

5 算法分析与改进

5.1 算法不足之处

算法的其中一处不足在于交叉算子设计不合理。如前述，该交叉算子只是随机地选取交叉点位并将双亲之一的基因片段复制，再用另一亲代的基因补全子代染色体。该做法的坏处在于不但没有考虑，反而打乱了亲代基因序列中优质的基因结构。

5.2 算法改进策略

通过阅读文献，我采用了一种称为MSCX（modified sequential constructive crossover）的交叉算子，它能有效找出双亲中的优秀基因结构并加入到子代中。

设长度为N的双亲染色体分别为$P1$和$P2$，其步骤如下：

1. 以$P1$的第一个基因为结点p，即$p = P1(1)$，并将其作为子代的第一个基因；
2. 按顺序在双亲中搜索位于p后一个的的合法基因（没有访问过的基因）。如果某个亲代中没有合法基因，**则继续在该亲代中向后搜索，直到找到一个合法基因。**
3. 经过步骤2，可以分别得到两个合法基因α和β；
4. 如果$Cost_{p\alpha} < Cost_{p\beta}$，即$p$距离$\alpha$更近，则将$\alpha$加入到子代中，反之则将$\beta$加入到子代中。同时将$p$设为选出的基因。如果子代已经是一条完整的基因序列，则停止，否则继续进行第2步。

例如，$P1$和$P2$的基因序列如下：

$$P1 \quad 1 \quad 5 \quad 7 \quad 3 \quad 6 \quad 4 \quad 2$$

$$P2 \quad 1 \quad 6 \quad 2 \quad 4 \quad 3 \quad 5 \quad 7$$

且cost matrix如下：

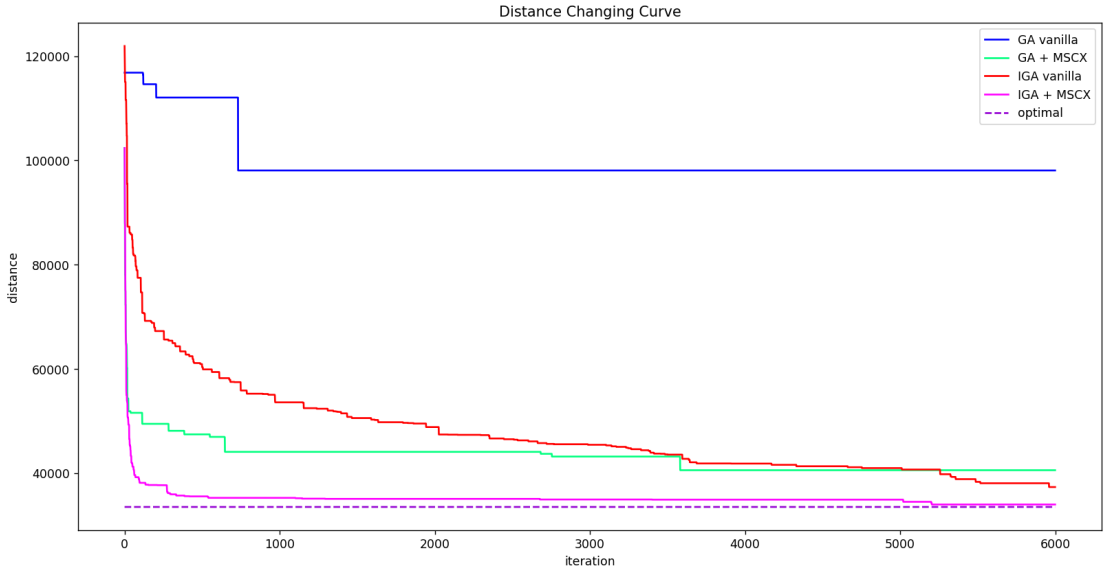
Node	1	2	3	4	5	6	7
1	999	75	99	9	35	63	8
2	51	999	86	46	88	29	20
3	100	5	999	16	28	35	28
4	20	45	11	999	59	53	49
5	86	63	33	65	999	76	72
6	36	53	89	31	21	999	52
7	58	31	43	67	52	60	999

- $p = 1$，$P1$中位于p后的基因为5而$P2$中为6。由于$C_{15} < C_{16}$，故将5加入到子代中，此时子代序列为$(1, 5)$。
- $p = 5$，$P1$和$P2$中位于p后的基因都为7，则将7加入到子代中，此时子代序列为$(1, 5, 7)$。
- $p = 7$，$P1$中位于p后的基因为3而$P2$中为空，则对于$P2$，继续向后搜索并找到的第一个合法基因为6。由于$C_{76} > C_{73}$，故将3加入到子代中，此时子代序列为$(1, 5, 7, 3)$
- ……

得到最终的子代基因序列为：

$$C \quad 1 \quad 5 \quad 7 \quad 3 \quad 2 \quad 6 \quad 4$$

实验结果如下图所示。



可以看到加入了MSCX的遗传算法（GA + MSCX）表现有很大提升，其收敛速度甚至快于免疫遗传算法；而使用了MSCX的免疫遗传算法（IGA + MSCX）也在收敛速度上进一步提升，其结果也非常逼近最优解了。

6 结论

遗传算法作为一种仿生的随机搜索算法，展现了其在求解某些难题时的有效性。但基本的遗传算法仅仅做到了大致模拟生物进化过程中的某些现象，未能更加精细地对实际求解的问题做出算法上的设计。同时，它不具备记忆功能，就不利于种群中某些数量较少的优秀个体发挥其优势；另一方面单纯基于适应度的选择机制不利于维持种群的多样性，导致算法容易“早熟”，即陷入局部最优。

免疫算法作为另一种仿生的随机搜索算法，具有和遗传算法相似的思路，且有记忆功能和更为合理的选择机制。将其与遗传算法结合得到的免疫遗传算法的性能要远远优于基本的遗传算法。另一方面，为了更好的解决旅行商问题，引入特别设计的算子能够进一步加速算法收敛和找到最优解。

透过本次实验，我加深了对遗传算法和免疫算法的理解，提高了文献阅读能力和编程能力。

7 参考文献

[2001.11590v1] New mechanism of combination crossover operators in genetic algorithm for solving the traveling salesman problem (arxiv.org)

Enhanced traveling salesman problem solving by genetic algorithm technique (TSPGA) | Hamza Ali - Academia.edu

Genetic Algorithm for the Traveling Salesman Problem using Sequential Constructive Crossover Operator | Zakir H. Ahmed - Academia.edu

8 附录

完整代码以开源至Github，此处展示主要代码。

8.1 主函数

```
# 初始化种群
cities, city_num = utils.get_cities(data_path)
distance_matrix = utils.get_distance_matrix(cities, city_num)

Population = models.Population(chro_num=population_size,
                               gene_num=city_num,
                               distance_matrix=distance_matrix,
                               alpha=alpha,
                               beta=beta,
                               delta=delta,
                               crossover_prob=crossover_prob,
                               mutation_prob=mutation_prob,
                               memory_size=memory_size,
                               radius=radius,
                               crossover_mode=crossover_mode)

Population.init_population()
distances = []  # 存储每代的最优结果，用于可视化

# 迭代
for i in range(iterations):
    # 选择
    Population.selection()

    # 交叉
    Population.crossover()

    # 变异
    Population.mutation()

    # 记忆
    Population.memorization()

    # 更新
    Population.update()
```

8.2 交叉算子

OX

```
def crossover_OX(self, chro):
    """
    Older Crossover
    1. 随机产生两个交叉点 (cross point)
    2. P1介于交叉点之间的基因片段直接复制到子代相同位置
    3. P2从第二个交叉点开始顺序遍历，将没有出现在子代的基因加入到子代中
    4. 认为self对应P1而chro对应P2
    """
    child = Chromosome(self.gene_num, self.distance_matrix)
    child_seq = np.zeros(self.gene_num, dtype=np.int32)  # 子代个体
```

```
    xp1 = random.randint(0, self.gene_num - 1)  # cross point 1
    xp2 = random.randint(0, self.gene_num - 1)  # cross point 2
    if xp1 > xp2:
        temp = xp1
        xp1 = xp2
        xp2 = temp

    # P1的片段直接复制到子代
    for i in range(xp1, xp2 + 1):
        child_seq[i] = self.gene_seq[i]

    # P2中不在child里出现的基因顺序复制到child
    i = j = (xp2 + 1) % self.gene_num  # i, j分别是P2和child的伪指针
    count = 0  # 记录child是否已经填满
    while True:
        if chro.gene_seq[i] not in child_seq:
            child_seq[j] = chro.gene_seq[i]
            i = (i + 1) % self.gene_num
            j = (j + 1) % self.gene_num
            count += 1
        else:
            i = (i + 1) % self.gene_num

        # child填满退出
        if count == self.gene_num - (xp2 - xp1 + 1):
            break

    child.gene_seq = child_seq
    child.compute_affinity()

    return child, xp1, xp2
```

MSCX

```
def crossover_MSCX(self, chro):
    """
    Modified Sequential Constructive Crossover
    1. 基因p = Parent1(1)，并将其作为子代第一个基因
    2. 按顺序搜索Parent1和Parent2中接在p后面的基因，即合法基因，并标记为visited
    如果没有合法基因，则循环从该Parent中按顺序找出第一个合法基因
    3. 从Parent1和Parent2中找到了a和b两个基因，然后计算Cost(p, a)和Cost(p, b)
    如果Cost(p, a) < Cost(p, b)，则将b接到子代中并把b赋给p
    4. 循环，直至子代拼成了完整的染色体
    5. 此处要建立基因值到索引的映射
    """

    parent1 = {}  # key为基因，value为索引
    parent2 = {}
    for i in range(self.gene_num):
        gene1 = self.gene_seq[i]
        gene2 = chro.gene_seq[i]
        parent1[gene1] = i
        parent2[gene2] = i
```

```
    idx_p = 0
    gene_p = self.gene_seq[idx_p]
    child = Chromosome(self.gene_num, self.distance_matrix)
    child_seq = np.zeros(self.gene_num, dtype=np.int32)
    child_seq[0] = self.gene_seq[0]
    count = 1  # 判断子代是否填满

    idx_a = (parent1[gene_p] + 1) % self.gene_num
    idx_b = (parent2[gene_p] + 1) % self.gene_num

    while count < self.gene_num:
        while True:
            gene_a = self.gene_seq[idx_a]
            if gene_a not in child_seq:
                break
            idx_a = (idx_a + 1) % self.gene_num

        while True:
            gene_b = self.gene_seq[idx_b]
            if gene_b not in child_seq:
                break
            idx_b = (idx_b + 1) % self.gene_num

        distance_a = self.distance_matrix[gene_p - 1][gene_a - 1]
        distance_b = self.distance_matrix[gene_p - 1][gene_b - 1]

        if distance_a < distance_b:
            child_seq[count] = gene_a
            gene_p = gene_a
            idx_a = (parent1[gene_p] + 1) % self.gene_num
        else:
            child_seq[count] = gene_b
            gene_p = gene_b
            idx_b = (parent2[gene_p] + 1) % self.gene_num

        count += 1

    child.gene_seq = child_seq
    child.compute_affinity()

    return child
```

8.3 亲和度、浓度和激励度

亲和度

```
def compute_affinity_probs(self):
    """
    计算亲和度概率
    """
    for i in range(self.chro_num):
        affinity = self.chromosomes[i].get_affinity()
        affinity_prob = affinity / self.total_affinity
        self.affinity_probs.append(affinity_prob)
```

浓度

```
def compute_density_probs(self):
    """
    计算浓度概率，用两个个体的亲和度概率作差，绝对值小于阈值则认为是同一类
    """
    density_counts = []  # 记录每个个体的同类数目
    for i in range(self.chro_num):
        density_count = 0
        for j in range(self.chro_num):
            affinity_diff = self.affinity_probs[i] - self.affinity_probs[j]
            if np.absolute(affinity_diff) <= self.delta:
                density_count += 1
        density_counts.append(density_count)

    # 计算浓度概率
    for i in range(self.chro_num):
        density = density_counts[i]
        density_prob = density / self.chro_num
        self.density_probs.append(density_prob)
```

激励度

```
def compute_simulation_probs(self):
    """
    计算激励度概率，它是最终用于选择个体的标准
    1. alpha x affinity_prob - beta x density prob
    2. 线性标准化
    """
    simulations = []
    for i in range(self.chro_num):
        affinity_prob = self.affinity_probs[i]
        density_prob = self.density_probs[i]
        simulation = self.alpha * affinity_prob - self.beta * density_prob
        simulations.append(simulation)

    # 线性标准化
    max_simulation = max(simulations)
    min_simulation = min(simulations)
    for i in range(self.chro_num):
        simulation_prob = (simulations[i] - min_simulation) / (max_simulation -
min_simulation)
```

```
        self.simulation_probs.append(simulation_prob)
```

8.4 种群的选择、交叉、变异、记忆和更新

选择

```
def selection(self):
    """
    种群选择，基于激励度，采用轮盘赌。
    1. 根据每个个体的simulation probability计算累计概率
    2. 从[0, 1)中随机采样一个数来决定哪个个体存活
    3. 共进行chro_num次选择，以保证种群大小不变
    """

    new_chromosomes = []  # 存活个体
    selected = np.zeros(self.chro_num, dtype=np.int32)  # 被选中染色体的索引
    cumulative_probs = np.zeros(self.chro_num)  # 累计概率

    for i in range(self.chro_num):
        new_chromosome = Chromosome()
        new_chromosomes.append(new_chromosome)

    # 计算累计概率
    cumulative_probs[0] = self.simulation_probs[0]
    for i in range(1, self.chro_num):
        cumulative_probs[i] = cumulative_probs[i - 1] + self.simulation_probs[i]

    # 轮盘赌 ROULETTE
    for i in range(self.chro_num):
        rand = random.random()  # [0,1)
        if rand <= cumulative_probs[0]:
            selected[i] = 0
        else:
            # 循环找出rand介于哪两个概率之间
            for j in range(self.chro_num - 1):
                if cumulative_probs[j] < rand <= cumulative_probs[j + 1]:
                    selected[i] = j

    # 更新种群
    for i in range(self.chro_num):
        new_chromosomes[i].duplication(self.chromosomes[i])
    self.chromosomes = new_chromosomes
```

交叉

```
def crossover(self):
    """
    种群中个体之间的交叉
    1. 随机选出两个个体进行交叉
    2. 两个个体依据概率是否交叉
    3. 直到子代个体数达到chro_num
```

```
    4. 自己不和自己交叉
    """
    count = 0  # 记录子代个体的数目
    child_chromosomes = []

    while count < self.chro_num:
        rand = random.random()  # 依概率发生交叉
        if rand <= self.crossover_prob:
            chro_idx1 = random.randint(0, self.chro_num - 1)
            chro_idx2 = random.randint(0, self.chro_num - 1)
            # 自己不和自己交叉
            if chro_idx1 == chro_idx2:
                offset = random.randint(1, self.chro_num - 1)
                chro_idx2 = (chro_idx1 + offset) % self.chro_num

            chro1 = self.chromosomes[chro_idx1]
            chro2 = self.chromosomes[chro_idx2]

            # 选择crossover的方式
            if self.crossover_mode == 'OX':
                child, _, _ = chro1.crossover_OX(chro2)
            elif self.crossover_mode == 'MOX':
                child = chro1.crossover_MOX(chro2)
            elif self.crossover_mode == 'SCX':
                child = chro1.crossover_SCX(chro2)
            elif self.crossover_mode == 'MSCX':
                continue
            elif self.crossover_mode == 'MSCX_Radius':
                continue
            else:
                raise ValueError('Not supported mode!')

            child_chromosomes.append(child)

            count += 1

    self.chromosomes = child_chromosomes
```

变异

```
def mutation(self):
    """
    种群中个体的变异
    1. 遍历每个个体
    2. 依概率发生变异
    """
    for i in range(self.chro_num):
        rand = random.random()
        if rand <= self.mutation_prob:
            self.chromosomes[i].mutation()
```

记忆

```
def memorization(self):
    """
    将亲和度最好的几个个体存储到记忆库中，用于之后更新种群
    """
    # 当前种群中的个体
    sorted_chromosomes = []
    for i in range(self.chro_num):
        sorted_chromosomes.append(self.chromosomes[i])

    # 记忆库中保存的个体
    for i in range(self.memory_size):
        sorted_chromosomes.append(self.memory[i])

    sorted_chromosomes.sort(key=self.take_affinity, reverse=True)

    for i in range(self.memory_size):
        self.memory[i] = sorted_chromosomes[i]

@staticmethod
def take_affinity(chro):
    return chro.get_affinity()
```

更新

```
def update(self):
    """
    种群更新
    1. 记忆库个体加入
    2. 总亲和度
    3. 亲和度概率、浓度概率和激励度概率
    4. 最好个体
    """
    self.generation += 1
    total_affinity = 0.0

    # 记忆库个体加入
    self.chromosomes.sort(key=self.take_affinity, reverse=False)  # 替换亲和度小的
个体
    for i in range(self.memory_size):
        new_chro = Chromosome(self.gene_num, self.distance_matrix)
        new_chro.duplication(self.memory[i])
        self.chromosomes[i] = new_chro

    # 更新总亲和度
    for i in range(self.chro_num):
        total_affinity += self.chromosomes[i].get_affinity()

    self.total_affinity = total_affinity

    # 更新亲和度概率、浓度概率和激励度概率
    self.compute_affinity_probs()
```

```
        self.compute_density_probs()
        self.compute_density_probs()

        # 更新最好个体
        self.find_best()
```