关于ClusteringGA算法的学习

(24.12.17)最近在学习ClusteringGA算法，在复现完整个代码后，在此记录自己对于聚类GA算法的体会和感悟。（说明：1、编程使用IntelliJ IDEA 2024.2.4，Java1.8，Maven3.8.1；2、本文采用的ClusteringGA算法是IEEE CEC2021中的竞赛算法；3、问题集是Competition on “Multi-modal Multi-objective Path Planning Optimization”；4、本文的AP聚类算法以算法所实现的AP聚类算法为准，与传统的AP聚类算法会有些许出入。）

1. 前言

ClusteringGA算法的整体流程与NSGAII相似，主要包括初始种群的产生、聚类、交叉、变异、非支配排序、选择等。其中聚类这个操作是ClusteringGA的主要创新点，而交叉变异也围绕聚类而进行，故本文将主要围绕聚类操作进行解读。

2. 聚类(Cluster)的介绍

聚类是一种机器学习技术，它涉及到数据点的分组。给定一组数据点，我们可以使用聚类算法将每个数据点划分为一个特定的组。理论上，同一聚类中的数据点应该具有相似的属性和/或特征，而不同聚类中的数据点应该具有高度不同的属性和/或特征。聚类是一种无监督学习的方法，是许多领域中常用的统计数据分析技术。

1. K-means聚类：最常用的聚类算法，通过指定K个中心点，将数据点分配到最近的中心点所在的簇。
1. AP聚类（Affinity Propagation）：通过在数据点之间传递消息来识别聚类中心，不需要预先指定聚类数量，能自动确定最优的聚类数量。
2. 层次聚类（Hierarchical Clustering）：通过构建树状层次结构来进行聚类，可以自底向上（凝聚）或自顶向下（分裂）进行。
3. DBSCAN（基于密度的空间聚类）：基于密度的聚类方法，可以发现任意形状的簇，对噪声数据具有较强的鲁棒性。
4. 期望最大化（EM）聚类：基于概率模型的聚类方法，通过迭代优化来估计模型参数。
5. 谱聚类（Spectral Clustering）：利用数据点之间的相似度矩阵的特征向量来降维，然后在低维空间进行聚类。

而在ClusteringGA算法中主要采用的则是AP聚类。

3. AP聚类

3.1 AP聚类的介绍

3.1.1 AP聚类的简介

AP聚类（Affinity Propagation）是一种基于消息传递的聚类算法，它将数据点视为网络中的节点，通过节点之间的信息交换来确定聚类中心和分类。算法的核心思想是让所有数据点之间通过传递消息来"协商"，从而选择出最佳的聚类中心（exemplar）。

在AP聚类中，每个数据点都可能成为聚类中心。算法通过迭代计算两种消息：

- 责任度（Responsibility）矩阵：表示数据点k作为数据点i的聚类中心的适合程度
- 可用度（Availability）矩阵：表示数据点i选择数据点k作为其聚类中心的适合程度

通过这两种消息的不断传递和更新，算法最终会收敛到一组稳定的聚类中心，从而完成数据的聚类过程。每个数据点都会被分配到与其最相似的聚类中心所在的类别中。

3.1.2 AP聚类的优点

AP聚类（Affinity Propagation）算法具有以下几个主要优点：

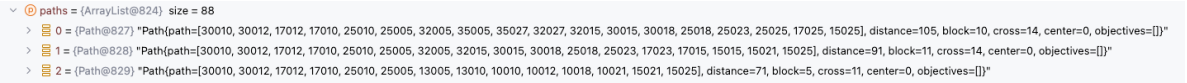
- 不需要预先指定聚类数量：算法能够自动确定最优的聚类数量，这在实际应用中非常有价值，特别是当我们不清楚数据应该分为多少类时
- 适用性广：可以处理相似度矩阵不对称的情况，也可以处理非欧氏距离的相似度度量
- 聚类效果好：通过消息传递机制，能够找到具有代表性的样本点作为聚类中心，聚类结果更加合理
- 收敛性好：算法具有良好的收敛性，且收敛速度相对较快
- 并行计算：算法的消息传递过程可以并行化，提高计算效率

3.2 AP聚类的流程

3.2.1 聚类前期准备工作

在进行AP聚类之前，我们需要进行一些准备操作。在这个基准问题中，决策变量是一条条路径，而AP聚类是基于适应度值进行操作的，因此我们需要对决策变量进行转换。在种群初始化时，我们根据预定的种群大小生成整个种群，并用数组列表表示，其中列表的每一行代表一个个体。在AP聚类中，我们使用种群中个体所在的行号作为该个体的标识。在计算相似度时，需要将每个个体的路径长度统一；对于长度不等的路径，较短的路径将在末尾补0以对齐长度。相似度值通过计算路径中对应位置的路径点之间的负欧式距离来表示。

下方图片中表示的则是种群以及种群中的部分个体



3.2.2 相似度/参考度矩阵(Similarity Matrix)

AP聚类的第一步则是生成相似度矩阵（以下将简称 s 矩阵），我们通常记为 s(i,k)，是指点k作为点i的聚类中心的相似度，数据点 i 和点 k 的相似度记为 s(i,k)，从点 i 发送至候选聚类中心点 k,反映了在考虑其他潜在聚类中心后，点 k 适合作为点i的聚类中心的程度。这里采用欧氏距离来计算，将点与点的相似度值全部取为负值，因此，s(i,k) 相似度值越大说明点 i 与点 *k* 的距离越近，AP算法中理解为数据点 k 作为数据点i的聚类中心的能力。

为了更加方便理解，以下提供一个简单的例子：假设点分布在实数轴上，坐标分别为 ：

A = 1，B = 2，C = 3，D = 5，E = 6

A　B　C　　D　E

0—1—2—3—4—5—6—7—8—9→

用两个点之间的距离的负数作为两个点之间的相似度，则s矩阵生成为：

s(i/k)	A	B	C	D	E
A	?	-1.0	-2.0	-4.0	-5.0
B	-1.0	?	-1.0	-3.0	-4.0
C	-2.0	-1.0	?	-2.0	-3.0
D	-4.0	-3.0	-2.0	?	-1.0
E	-5.0	-4.0	-3.0	-1.0	?

对角线 s(i,i) 表示自己与自己的相似度，AP聚类算法通常选择上述矩阵中元素的最小值或者中位数来填充对角线元素。接下来我们选择中位数得到完整的s矩阵：

s(i/k)	A	B	C	D	E
A	-3.0	-1.0	-2.0	-4.0	-5.0
B	-1.0	-3.0	-1.0	-3.0	-4.0
C	-2.0	-1.0	-3.0	-2.0	-3.0
D	-4.0	-3.0	-2.0	-3.0	-1.0
E	-5.0	-4.0	-3.0	-1.0	-3.0

以下是源码中 s 矩阵的实现片段：

```
// 生成相似度矩阵Similarity
double[][] s = InitialArray(clustering,paths,maxLength); // 生成s矩阵：每个个
double[][] S = new double[paths.size()][paths.size()]; // 填充s的对角线元素生
int l = 0;
for(int i=0;i<paths.size();i++){
    for(int j=0;j<paths.size();j++){
        if(i==j){S[i][j]=p;} // 对角线存相似度中位数
        else {S[i][j] = s[l][2];l++;} // 除对角线外元素存s矩阵第三列相似度值
    }
}

 // 创建初始化s数组
private static double[][] InitialArray(int[][] clustering, List<Path> path
    int N = paths.size();
    int M = N * (N - 1); // 排除对角线
    double[][] s = new double[M][3]; // 创建一个 M 行 3 列的矩阵，用于存放个体之
    int j = 0;

    for (int i = 0; i < N; i++) {
        for (int k = 0; k < N; k++) {
            if(i == k) continue; // 跳过自己
            double sum = 0; // 每对 (i, k) 重新计算 sum
            for (int l = 0; l < maxLength; l++) {
                double first = clustering[i][l];
                double second = clustering[k][l];
                if(first == 0 && second == 0) {continue;} // 若x、y都为0，则
```

```
                // 将路径点拆分为x、y坐标方式进行之后的欧式距离计算
                double first_x = (double) first / 1000;
                double first_y = (double) first % 1000;
                double second_x = (double) second / 1000;
                double second_y = (double) second % 1000;
                // 计算欧式距离
                double aum = Math.sqrt(Math.pow(first_x - second_x, 2) + M
                sum += aum * aum; // 欧式距离平方累加
            }
            s[j][0] = i+1;
            s[j][1] = k+1;
            s[j][2] = -sum;// 负欧式距离
            j++;
        }
    }
return s;
}
```

3.2.3 责任度/吸引度矩阵(Responsibility Matrix)

在完成好相似度矩阵后，便进入了主循环的迭代，所以AP聚类的第二步便是在主循环迭代中更新责任度矩阵（以下将简称 r 矩阵），r 用来描述**点 k 适合作为数据点 i 的聚类中心的程度**，上面的 s 矩阵虽然在一定程度上反映了程度，但是，不同行之间数据是不能进行比较的。例如，第1行第2列最大值是-1，意味着个体A会选个体B作为聚类中心。但是第2行的最大值-1有两个，这样不利于对聚类中心的选择，所以我们需要再创建一个r矩阵，来将这些数据就行优化利于聚类中心的选择。

r 矩阵按照以下数学公式来进行生成（在初始化时，a(i,k′) = 0）：

$$r(i,k) = s(i,k) - \max_{k' \neq k}\{s(i,k') + a(i,k')\}$$

让我们详细解读公式 r(i,k) = s(i,k) - max_{k' ≠ k}{s(i,k') + a(i,k')}：

- **r(i,k)**：表示点i选择点k作为其聚类中心的程度（责任度）
- **s(i,k)**：表示点i与点k之间的相似度
- **max_{k' ≠ k}{s(i,k') + a(i,k')}**：表示除了k以外，其他所有点作为i的聚类中心的竞争力最大值
  - k'表示除k以外的其他点
  - s(i,k')表示i与其他点k'的相似度
  - a(i,k')表示i选择k'作为聚类中心的适合程度（在初始时为0）

该公式的计算过程：将 s 矩阵中每一个元素与当前这一行中最大的元素做差。

这个公式的本质是：通过将点i和k的相似度，减去i与其他所有候选中心点的最大竞争力，来确定k作为i的聚类中心的程度。如果结果为正值，说明k比其他所有候选点更适合作为i的聚类中心。

根据以上描述，举例表格中的元素更新为：

r(i/k)	A	B	C	D	E
A	-2.0	1.0	-1.0	-3.0	-4.0
B	0.0	-2.0	0.0	-2.0	-3.0
C	-1.0	0.0	-2.0	-1.0	-2.0
D	-3.0	-2.0	-1.0	-2.0	1.0
E	-4.0	-3.0	-2.0	2.0	-2.0

为防止 r 矩阵的更新幅度过大造成结果不收敛后我们会生成 rold 矩阵来记录刚更新的 r 矩阵，并引入阻尼系数 λ，来控制更新幅度。一般取 λ = 0.5 ，更新公式如下：

$$r = (1-\lambda)r + \lambda \cdot r_{old}$$

对于举例表格而言，由于没有 rold，故我们直接将表格中的数据乘以0.5:

r(i/k)	A	B	C	D	E
A	-1.0	0.5	-0.5	-1.5	-2.0
B	0.0	-1.0	0.0	-1.0	-1.5
C	-0.5	0.0	-1.0	-0.5	-1.0
D	-1.5	-1.0	-0.5	-1.0	0.5
E	-2.0	-1.5	-1.0	1.0	-1.0

以下是源码中 r 矩阵的实现片段：

```
// 更新R责任度矩阵（将S中的每一个元素与该元素所处行最大值相减）
double[][] Xpro = new double[paths.size()][paths.size()]; // 将每行最大值填充
for (int i = 0; i < paths.size(); i++) {
    for (int j = 0; j < paths.size(); j++) {
        Xpro[i][j] = Y[i];
    }
}
R = calMetrices(S,Xpro,2); // 按公式计算R矩阵（calMertices实现了矩阵加减操作，后
// 采取手段使本来优秀的个体更加优秀
for(int k = 0;k<paths.size();k++){
    int num = I[k]; // 用来记录S中每一行最大值的位置（列索引）
    R[k][num] = S[k][num] - Y2[k]; // 将R中原来每一行最大值替换为最大值 - 第二大
}
```

以下是源码中阻尼系数实现片段：

```
R = addLam(R,Rold,0.5); // 加入阻尼系数(R=(1-lam)*R+lam*Rold)
//加入阻尼系数
private static double[][] addLam(double[][] metrix1,double[][] matrix2,dou
    int rows = metrix1.length;
    int cols = metrix1[0].length;
    double[][] result = new double[rows][cols]; // 创建存储结果的矩阵
```

```
    double[][] temp1 = new double[rows][cols]; // 新矩阵（更新后的矩阵）
    double[][] temp2 = new double[rows][cols]; // 旧矩阵（更新前的矩阵）
    for (int i = 0; i < metrix1.length; i++) { // 遍历行
        for (int j = 0; j < metrix1[i].length; j++) { // 遍历列
            temp1[i][j] = metrix1[i][j] * (1-lam); // 新矩阵元素乘以系数
        }
    }
    for (int i = 0; i < matrix2.length; i++) {
        for (int j = 0; j < matrix2[i].length; j++) {
            temp2[i][j] = matrix2[i][j] * lam; // 旧矩阵元素乘以系数
        }
    }
    result = calMetrices(temp1,temp2,1); // 将乘以系数后的两个新矩阵相加生成最终
    return result;
}
```

3.2.4 可用度/归属度矩阵(Availability Matrix)

AP聚类的第三步是在主循环迭代中更新可用度矩阵（以下将简称 a 矩阵）a(i,k) 用来描述**点 i 选择点 k 作为其聚类中心的适合程度**。从候选聚类中心点 k 发送至点 i，反映了在考虑其他点对点 k 成为聚类中心的支持后，点 i 选择点 k 作为聚类中心的合适程度，从候选聚类中心点 k 发送至点 i，反映了在考虑其他点对点 k 成为聚类中心的支持后，点 i 选择点 k 作为聚类中心的合适程度。简单来说就是根据目前 r 矩阵提供的支持度结果，做进一步的决策调整。

a 矩阵按照以下数学公式来进行生成：

$$a(i,k) = \min\{0, r(k,k) + \sum_{i' \neq i,k} \max\{0, r(i',k)\}\} \quad (i \neq k)$$

$$a(k,k) = \sum_{i' \neq k} \max\{0, r(i',k)\}$$

其中，第一个公式用于计算非对角线元素，第二个公式用于计算对角线元素。

让我们来详细解读这两个计算公式：

1. $a(i,k) = \min\{0, r(k,k) + \sum_{i' \neq i,k} \max\{0, r(i',k)\}\} \quad (i \neq k)$

- r(k,k)：表示候选聚类中心 k 对自身的责任度，即 k 作为聚类中心的可能性
- ∑max{0,r(i',k)}：对除了 i 和 k 之外的所有点 i'，计算它们对 k 作为聚类中心的正向支持度的总和
- min{0, ...}：将结果限制在负值范围内，这样可以防止可用度值过大

该公式的计算过程如下：

1. 首先计算r(k,k)值，即 k 点对自身的责任度
2. 对当前元素所在这一列中所有除了i和k之外的点i'，找出它们对 k 的责任度r(i',k)中的正值并求和
3. 将r(k,k)与上一步得到的和相加
4. 最后取这个结果与0比的较小值，确保结果不会是正数

这个公式的核心思想是：当计算点 i 选择 k 作为聚类中心的适合程度时，不仅要考虑 k 自身作为中心的倾向，还要考虑其他点对 k 的支持程度。如果其他点都强烈支持 k 作为中心（即有较多正的 r 值），那么 i 选择 k 作为中心也是较为合适的。本质上是在评估：如果其他点都认为 k 适合作为聚类中心（正的 r 值），那么 i 选择 k 作为其聚类中心也是合理的。

2. $a(k,k) = \sum_{i' \neq k} \max\{0, r(i',k)\}$

- ∑： 表示需要对所有除 k 以外的点进行累加运算
- i' ≠ k： 表示在累加过程中，排除当前考虑的点 k 自身
- max{0,r(i',k)}： 只考虑正的责任度值，如果责任度为负，则取0。

该公式的计算过程如下：

1. 遍历责任度矩阵 r 中第 k 列的所有元素（除了第 k 行）
2. 对每个元素，如果值为正数则保留，如果为负数则记为0
3. 将所有处理后的值相加，得到的结果就是a(k,k)

这个公式的核心思想是：它直接反映了其他点对 k 作为聚类中心的集体支持度，通过只考虑正值，消除了反对票的影响，使得聚类中心的选择更加稳健，对角线元素的值越大，表示该点越有可能成为最终的聚类中心。

根据非对角线元素更新公式的描述，举例表格中的元素更新为：

a(i/k)	A	B	C	D	E
A	-1.0	-1.0	-1.0	0.0	-0.5
B	-1.0	-1.0	-1.0	0.0	-0.5
C	-0.5	-0.5	-1.0	0.0	-0.5
D	-0.5	-0.5	-1.0	-1.0	-1.0
E	-0.5	-0.5	-1.0	-1.0	-1.0

进一步再根据对角线元素更新公式，举例表格中的元素更新为：

a(i/k)	A	B	C	D	E
A	0.0	-1.0	-1.0	0.0	-0.5
B	-1.0	0.5	-1.0	0.0	-0.5
C	-0.5	-0.5	0.0	0.0	-0.5
D	-0.5	-0.5	-1.0	1.0	-1.0
E	-0.5	-0.5	-1.0	-1.0	0.5

最后还是为防止 a 矩阵的更新幅度过大造成结果不收敛，生成 aold 矩阵来记录刚更新的 a 矩阵，并引入阻尼系数 λ，来控制更新幅度。取 λ = 0.5 ，举例表格中的元素更新为(aold取0)：

a(i/k)	A	B	C	D	E
A	0.0	-0.5	-0.5	0.0	-0.25
B	-0.5	0.25	-0.5	0.0	-0.25
C	-0.25	-0.25	0.0	0.0	-0.25
D	-0.25	-0.25	-0.5	0.5	-0.5

a(i/k)	A	B	C	D	E
E	-0.25	-0.25	-0.5	-0.5	0.25

以下是源码中 a 矩阵的实现片段：

```
// 更新A可用度矩阵
Aold = A; // 将更新前的A提前记录下来(为后续加入阻尼系数提供方便)
Rp = R; // Rp作为过渡矩阵
for(int i=0;i<paths.size();i++){
    for(int j=0;j<paths.size();j++){
        if(Rp[i][j] < 0 && i != j){
            Rp[i][j] = 0; // 除R(k,k)外，将R中的负数变为0，忽略不适合的点的不适合
        }
    }
}
for(int i=0;i<paths.size();i++){
    Rp[i][i] = R[i][i]; // Rp中对角线的值替换为之前R中的对角线的值
}
double[][] Ypro_add = new double[1][paths.size()]; // 按列求和
double[][] Ypro = new double[paths.size()][paths.size()]; // 将求和结果扩充成
for (int j = 0; j < paths.size(); j++) {
    double a = 0;
    for (int i = 0; i < paths.size(); i++) {
        a += Rp[i][j];
        Ypro_add[0][j] = a; // 求和
    }
}
for (int i = 0; i < paths.size(); i++) {
    for (int j = 0; j < paths.size(); j++) {
        Ypro[i][j] = Ypro_add[0][j]; // 扩充
    }
}
A = calMetrices(Ypro,Rp,2); // A=repmat(sum(Rp,1),[N,1])-Rp;按公式计算A矩阵
for(int i=0;i<paths.size();i++){
    dA[i][i] = A[i][i]; // 将A矩阵对角线元素保存
}
for(int i=0;i<paths.size();i++){
    for(int j=0;j<paths.size();j++){
        if(i != j && A[i][j] > 0){ A[i][j] = 0;} // 除A(k,k)外其他大于0的值置
    }
}
for(int i=0;i<paths.size();i++){
    A[i][i] = dA[i][i]; // 将之前保留的A对角线元素换到新的A中
}
A = addLam(A,Aold,0.5); // 加入阻尼系数A=(1-lam)*A+lam*Aold;
for(int i=0;i<paths.size();i++){
```

```
    dA[i][i] = A[i][i]; // 将A矩阵对角线元素保存
}
```

3.2.5 最终判断

在完成了上述 s 矩阵、r 矩阵与a 矩阵的生成后，我们设定 E 为一个逻辑向量，记录了哪些点被判定为簇中心也就是聚类中心，判定规则是 r(i,i)与a(i,i)依次相加(也就是 r 矩阵与 a 矩阵对角线元素之和)，若相加之和大于0，则令E(i) = 1；记录完后，将每一次的循环结果，依次循环存入矩阵 e 中，即第一次循环得出的 E 放到 e 矩阵的第一列，第51次的结果又放在第一列。再然后将 e 中元素按行求和放进 se 中，se 是一个列向量，用来记录每个点在过去 convits 次迭代中的簇中心状态之和。如果一个点在所有 convits 次迭代中都为 1 或都为 0，说明这个点已经收敛。通过sum 统计已经收敛的点的数量，如果所有点都收敛（即 sum == 初始生成种群大小），则说明算法整体已经收敛。最终如果所有点收敛且 K > 0（存在簇中心），或者迭代次数达到最大值 maxits，算法结束。很显然只有在所有点收敛时表示聚类成功，而如果仅仅只是迭代次数达到最大值则代表着聚类失败。

上述操作反应到举例表格中则是：

s(i/i)+a(i/i)	A	B	C	D	E
A	-1.0				
B		-0.75			
C			-1.0		
D				-0.5	
E					-0.75

很显然上述举例表格中的对角线元素都小于0，那么他们都无法成为簇1中心点，这时则说明没有聚类中心，那么将继续下轮的迭代。上述的迭代计算结果举例如下，供大家参考：

s(i/i)+a(i/i)	A	B	C	D	E
A	-2.75	0.5	-2.375	-2.75	-4.5
B	-1.625	-0.75	-1.625	-1.5	-3.25
C	-2.375	0.25	-2.75	-0.75	-2.5
D	-4.125	-1.25	-2.125	-1.0	-0.5
E	-5.125	-2.25	-3.125	0.5	-1.75

s(i/i)+a(i/i)	A	B	C	D	E
A	-1.71875	1.46875	-1.28125	-1.875	-3.4375
B	-0.9375	0.96875	-0.75	-0.1875	-1.75
C	1.59375	1.0625	-1.59375	0.0	-1.5625
D	-2.84375	-0.25	-0.90625	0.375	0.0
E	-3.84375	-1.25	-1.65625	0.875	-0.4375

可以看到在最后一张表中，已经选出了最终的聚类中心。

以下是源码中最终判断的实现片段：

```
int[] E = new int[paths.size()]; // E是一个逻辑向量，记录了哪些点被判定为簇中心
for(int i=0;i<paths.size();i++){
    if(dA[i][i] + dR[i][i] > 0){
        E[i] = 1;
    }
}
// 将循环计算结果列向量E放入矩阵e中，注意是循环存放结果
// 即第一次循环得出的E放到N*50的e矩阵的第一列，第51次的结果又放到第一列
double[][] e = new double[paths.size()][convits];
// 计算列索引
int colIndex = (counter - 1) % convits; // Java 的 % 操作符等效于 MATLAB 的
if (colIndex < 0) colIndex += convits; // 处理负数情况，确保结果在 [0, convits
colIndex++; // MATLAB 的索引从 1 开始，Java 的索引从 0 开始，所以加 1
// 将 E 的内容赋值到 e 的第 colIndex 列
for (int row = 0; row < E.length; row++) {
    e[row][colIndex - 1] = E[row]; // 注意 Java 的列索引从 0 开始
}

int K = 0;
for (int i = 0; i < paths.size(); i++) {
    if(E[i] == 1){K++;} // 计算 E 中值为 1 的数量，即当前被判定为簇中心的点的个数K
}

// 最终判断
if (counter >= convits || counter >= maxits) {
    // 将e中元素按行求和放进se中
    double[] se = new double[paths.size()]; // se是一个列向量，E的convits次迭
    for (int i = 0; i < paths.size(); i++) {
        double sumse = 0;
        for (int j = 0; j < convits; j++) {
            sumse += e[i][j];
        }
        se[i] = sumse;
    }
    boolean unconverged; // 用来判断收敛点的数量是否等于总数
    double sum = 0;
    for (int i = 0; i < paths.size(); i++) {
        if(se[i] == convits || se[i] == 0){
            sum++;
        }
    }
    unconverged = (sum != paths.size()); // 如果收敛的点数量不等于总点数N,说明有
    // 收敛
    if ((!unconverged && (K > 0)) || (counter == maxits)) {
        dn = true; // 如果条件成立，将 dn 设置为 true
```

```
    }
}
```

3.2.6 聚类后续完善

经过以上的整个操作，我们选出了聚类的中心点，最后我们需要按照我们所计算出的簇中心点，来将簇中心点以及每个所属的子个体放在同一个簇中，来方便后续的交叉变异操作，并且也需要将每一个个体还原成原来的路径表示。

该阶段的大致流程为：

1. 找出潜在的簇中心点
2. 分配点到最近的簇中心
3. 细化簇中心点
4. 计算聚类适合度
5. 去重和排序簇中心点
6. 构建簇并分配路径数据
7. 输出聚类结果

以下是源码中后续完善的实现片段：

```
// 经过上面的的循环，便确定好了哪些点可以作为簇中心点，找出簇1中心点
// int[] I = new int[paths.size()];
ArrayList<Integer> I = new ArrayList<>();
int K = 0; // 用来记录I中存放多少数据，及多少个簇中心点
int sum_k = 0; // 用来方便在I中按行记录簇中心点索引
for (int i = 0; i < paths.size(); i++) {
    // 将R与A矩阵对角线元素相加，若为正数则说明该点为簇中心点，且i反应了簇中心点的行列
    if(R[i][i] + A[i][i] > 0){
        //I[sum_k] = i;
        I.add(i);
        sum_k++;
        K++;
    }
}

double[] tmpidx = new double[paths.size()]; // 用来存放最后每个点的聚类中心
double tmpnetsim = 0; // 存放各个点到聚类中心的负欧式距离的和（衡量这次聚类的适合度
double tmpexpref = 0; // 存放所有被选为簇中心的点的适合度

if(K > 0){
    // [~,c]=max(S(:,I),[],2);
    // 将S中对应I的所有列取出来，例如I中是3，4，则将S的第3和第4列的所有元素取出来
    //  （判断除此簇中心点外，其他的点是否以此簇中心点为中心）
    // 对取出的这几列，按行查找最大值并记录这个最大值的列索引
    double[][] Sp = new double[paths.size()][paths.size()]; // 用来存S中对应
```

```
    // 将Sp内部填充负无穷大，方便后续查找最大值的列索引
    for (int i = 0; i < paths.size(); i++) {
        for (int j = 0; j < paths.size(); j++) {
            Sp[i][j] = Double.NEGATIVE_INFINITY;
        }
    }
    int[] c = new int[paths.size()]; // 存放列索引（存放的是每个点的簇中心点是哪
    // 取出S对应I中的所有列
    for (int k = 0; k < I.size(); k++) {
        int sum = 0;
        //sum = I[k];
        sum = I.get(k);
        for (int i = 0; i < paths.size(); i++) {
            Sp[i][sum] = S[i][sum];
        }
    }
    // 按行查找最大值,并记录最大值列索引
    for(int i=0;i<paths.size();i++){
        double max = Double.NEGATIVE_INFINITY; // 定义成负无穷大方便查找最大值
        int maxIndex = -1; // 记录列索引
        for(int j=0;j<paths.size();j++){
            if(Sp[i][j] > max){
                max = Sp[i][j];
                //maxIndex = j+1;
                maxIndex = j;
            }
        }
        c[i] = maxIndex;
    }
    // 将c数组内的值换成I中的序号(索引)
    for(int i=0;i<c.length;i++){
        int numc = c[i];
        for(int j=0;j<I.size();j++){
            if(I.get(j) == numc){
                c[i] = j;
            }
        }
    }
    // 将 1:K 的值赋给向量 c 的索引为 I 的位置（c(I)=1:K;）
    for (int i = 0; i < I.size(); i++) {
        c[I.get(i)] = i;
    }

    // 细化聚类中心
    // 当k=2时，在c中找到个体1、2、4都以2为聚类中心，提取S中1、2、4行和1、2、4列组成
    // 分别算出3列之和取最大值，y记录最大值，j记录最大值所在的行
```

```
    for(int k = 0; k < K; k++){
        ArrayList<Integer> ii = new ArrayList<>();
        int sum=0;
        // 首先找到同一聚类中心的每个个体
        for (int m = 0; m < c.length; m++) {
            if(c[m] == k){
                ii.add(m);
                sum++;
            }
        }
        // 提取这些个体在S中的行列，生成子矩阵
        double[][] jj = new double[ii.size()][ii.size()];
        for(int m = 0; m < ii.size(); m++){
            for(int n = 0; n < ii.size(); n++){
                jj[m][n] = S[ii.get(m)][ii.get(n)];
            }
        }
        // 并计算三列之和，记录最大值与最大值的列
        int maxCols = jj.length; // 由于不知道jj为多大，首先找到jj的大小
        int[] caltemp = new int[maxCols]; // 用来记录每列最大值，且最大值所在的
        int[] yCols = new int[1]; // 记录最大值所在的列
        double y = Double.NEGATIVE_INFINITY; // 记录最大值
        for(int m=0;m<maxCols;m++){
            for(int n=0;n<maxCols;n++){
                caltemp[m] += jj[m][n]; // 将jj一列的值相加存入caltemp数组
            }
        }
        for(int m=0;m<caltemp.length;m++){
            if(caltemp[m] > y){
                y = caltemp[m]; // 最大值
                yCols[0] = m; // 最大值所在的列
            }
        }
        I.set(k, ii.get(yCols[0])); // 将子矩阵中列和最大的列对应的点在原矩阵中的
    }
    // [tmp,c]=max(S(:,I),[],2);
    // 在矩阵S的指定列I中，逐行寻找最大值，并返回这些最大值及其对应的列索引
    double[] tmp = new double[paths.size()];
    for (int row = 0; row < S.length; row++) {
        double maxVal = Double.NEGATIVE_INFINITY;
        int maxIndex = -1;
        // 遍历 I 中的列索引
        for (int colIndex = 0; colIndex < I.size(); colIndex++) {
            int actualCol = I.get(colIndex); // 转换为实际列索引
            if (S[row][actualCol] > maxVal) {
                maxVal = S[row][actualCol];
```

```
                maxIndex = colIndex; // 局部索引
            }
        }
        // 保存最大值和对应列索引
        tmp[row] = maxVal;
        c[row] = maxIndex;
    }
    // c(I)=1:K;
    for (int i = 0; i < I.size(); i++) {
        c[I.get(i)] = i;
    }
    for(int i = 0; i < paths.size(); i++){
        tmpidx[i] = I.get(c[i]); // 每个点的聚类中心是谁
    }
    // 将各点到簇中心的欧式距离负值的和来衡量这次聚类的适合度
    for (int i = 0; i < tmpidx.length; i++) {
        //int row = (int)tmpidx[i] - 1; // 转换为 0-based 索引
        int row = (int)tmpidx[i];
        tmpnetsim += S[row][i]; // 累加第 row 行的第 i 列的元素
    }
    // 表示所有被选为簇中心的点的适合度之和
    for (int i = 0; i < I.size(); i++) {
        tmpexpref += dS[I.get(i)][I.get(i)];
    }
    //int qwe = 0; // 测试
}
    else{
        Arrays.fill(tmpidx, Double.NaN); // 用 Double.NaN 填充数组
        tmpnetsim = Double.NaN;
        tmpexpref = Double.NaN;
    }

    double netsim = tmpnetsim; // 反应这次聚类的适合度
    double expref = tmpexpref;
    double dpsim = tmpnetsim - tmpexpref;
    double[] idx = new double[tmpidx.length];
    idx = Arrays.copyOf(tmpidx, tmpidx.length);
    // 返回数组 idx 中的唯一元素（去重后的元素），并按升序排列
    // 使用 HashSet 去重
    Set<Double> uniqueSet = new HashSet<>();
    for (double element : idx) {
        uniqueSet.add(element);
    }
    // 将去重后的元素转换为数组
    double[] ans = uniqueSet.stream().mapToDouble(Double::doubleValue).toA
    // 对数组进行排序（升序）
```

```
    Arrays.sort(ans);

    // 根据聚类结果，将数据分配到各个簇中，并提取每个簇的中心点信息
    ArrayList<ArrayList<Path>> cluster = new ArrayList<>(); // 用来储存最后的
    int tt = 0; // 计数器，记录当前是哪个簇

    // 开始遍历所有的簇（idx中是所有个体归属于哪个聚类中心，ans是整个种群选出来的聚类中
    for(int k = 0; k < ans.length; k++){
        ArrayList<Integer> ii = new ArrayList<>();
        // 按照ans的顺序找到当前簇的所有点存入ii
        for(int i = 0; i < idx.length; i++){
            if(idx[i] == ans[k]){
                ii.add(i);
            }
        }
        // 将当前簇的簇中心点表示的路径存入center
        int aum = 0;
        aum = (int) ans[tt];
        paths.get(aum).center = 1; // 若该条路径是聚类中心点，则修改其center属性
        // 将当前簇的每个点所对应的路径存入cluster中
        ArrayList<Path> interRow = new ArrayList<>(); // 储存当前簇内部的路径
        for(int j = 0; j < ii.size(); j++){
            int num = ii.get(j);
            interRow.add(paths.get(num));
        }
        cluster.add(interRow);
        tt++;
    }

    //return result;//测试
    return cluster;
```

最后所生成的聚类为下图所示：

至此整个聚类操作完成。

4. 交叉变异操作

4.1 交叉操作

ClusteringGA的交叉操作，针对于每一个簇中的个体，分为三种情况

1、当簇中只有1个个体时：

不进行交叉操作

2、当簇中有且只有2个个体时：

父代个体1的前部与父代个体2的后部交叉生成子代个体1

父代个体2的前部与父代个体1的后部交叉生成子代个体2

3、当簇中个体大于等于3个个体时:

父代个体1的前部与父代个体2的后部交叉生成子代个体1

父代个体2的前部与父代个体3的后部交叉生成子代个体2

……

最后一个父代个体的前部与父代个体1的后部交叉生成最后一个子代个体

```
// 交叉
switch (ss) {
    //  case 1:若当前聚类只有一个个体则不进行交叉操作
    case 2: // 若当前聚类只有两个个体
        List<Integer> Ppath1 = APCluster.get(i).get(0).path;
        List<Integer> Ppath2 = APCluster.get(i).get(1).path;
        // 由第一个个体前部与第二个个体后部交叉形成第一个个体的子代
        APCluster.get(i).get(0).path = ClusteringGACrossover.Crossover(Ppa
        // 由第二个个体前部与第一个个体后部交叉形成第二个个体的子代
        APCluster.get(i).get(1).path = ClusteringGACrossover.Crossover(Ppa
        Non_repeating_sets.add(APCluster.get(i).get(0));
        Non_repeating_sets.add(APCluster.get(i).get(1));
        break;
    default: // 若当前聚类有三个及以上个体
        for (int j = 0; j < APCluster.get(i).size(); j++) {
            if (j != APCluster.get(i).size() - 1) { // 最后一个元素之前的元素
                List<Integer> Ppath_1 = APCluster.get(i).get(j).path;
                List<Integer> Ppath_2 = APCluster.get(i).get(j + 1).path;
                //第一个个体与第二个个体交叉生成第一个个体的子代
                APCluster.get(i).get(j).path = ClusteringGACrossover.Cross
                Non_repeating_sets.add(APCluster.get(i).get(j));
            } else { // 最后一个元素与第一个元素交叉
                List<Integer> Ppath_1 = APCluster.get(i).get(j).path; // 最
                List<Integer> Ppath_2 = APCluster.get(i).get(0).path; // 第
                APCluster.get(i).get(j).path = ClusteringGACrossover.Cross
                Non_repeating_sets.add(APCluster.get(i).get(j));
            }
        }
```

```
            break;
    }
```

4.2变异操作

变异操作按照正常GA的变异操作进行，在此不做过多描述。

5 快速非支配排序

5.1 拆解聚类

在进行快速非支配排序之前，首先拆除聚类，将种群换回最开始的矩阵列表

```
// 将聚类拆开形成原来的Path对象
public static List<Path> Discluster(ArrayList<ArrayList<Path>> APcluster){
    List<Path> paths = new ArrayList<>();
    int n = 0;
    for (int i = 0; i < APcluster.size(); i++) {
        for (int j = 0; j < APcluster.get(i).size(); j++) {
            paths.add(APcluster.get(i).get(j));
        }
    }
    return paths;
}
```

5.2 快速非支配排序

快速非支配排序按照正常GA的快速非支配排序进行，在此不做过多描述。

6 总结

至此，整个ClusteringGA便介绍完了，对于AP聚类算法，不可否认有很多的优点，但是或许对于多模态问题来说，并不是一个很有效的办法，存在许多的缺点：

1. AP算法需要事先计算每对数据对象之间的相似度，如果数据对象太多的话，AP算法的时间复杂度较高，一次迭代大概O(N^3)
2. 聚类的好坏受到参考度和阻尼系数的影响。
3. 对于ClusteringGA算法，普通的GA算法每个个体都可以进行交叉，而ClusteringGA的交叉仅仅只是针对每一个聚类中的个体，破坏了多样性，以至于其实看一些测试问题的结果ClusteringGA并不能找到所有的解。

7 参考

https://www.science.org/doi/10.1126/science.1136800

https://link.springer.com/chapter/10.1007/978-3-030-95391-1_36

https://blog.csdn.net/qq_38195197/article/details/78136669

https://blog.csdn.net/quicmous/article/details/129489166