**《人工智能》大作业**

**项目题目： 基于遗传算法和改良K-means算法的图像配准和差距度量**

**姓 名： 黄一骏 学 号： 1191170222**

**班 级： 数字媒体技术1702**

**评语：**

签名：

**成 绩：**

**江南大学数字媒体学院**

**2020年1月**

# 正文

1. **总体设计方案**

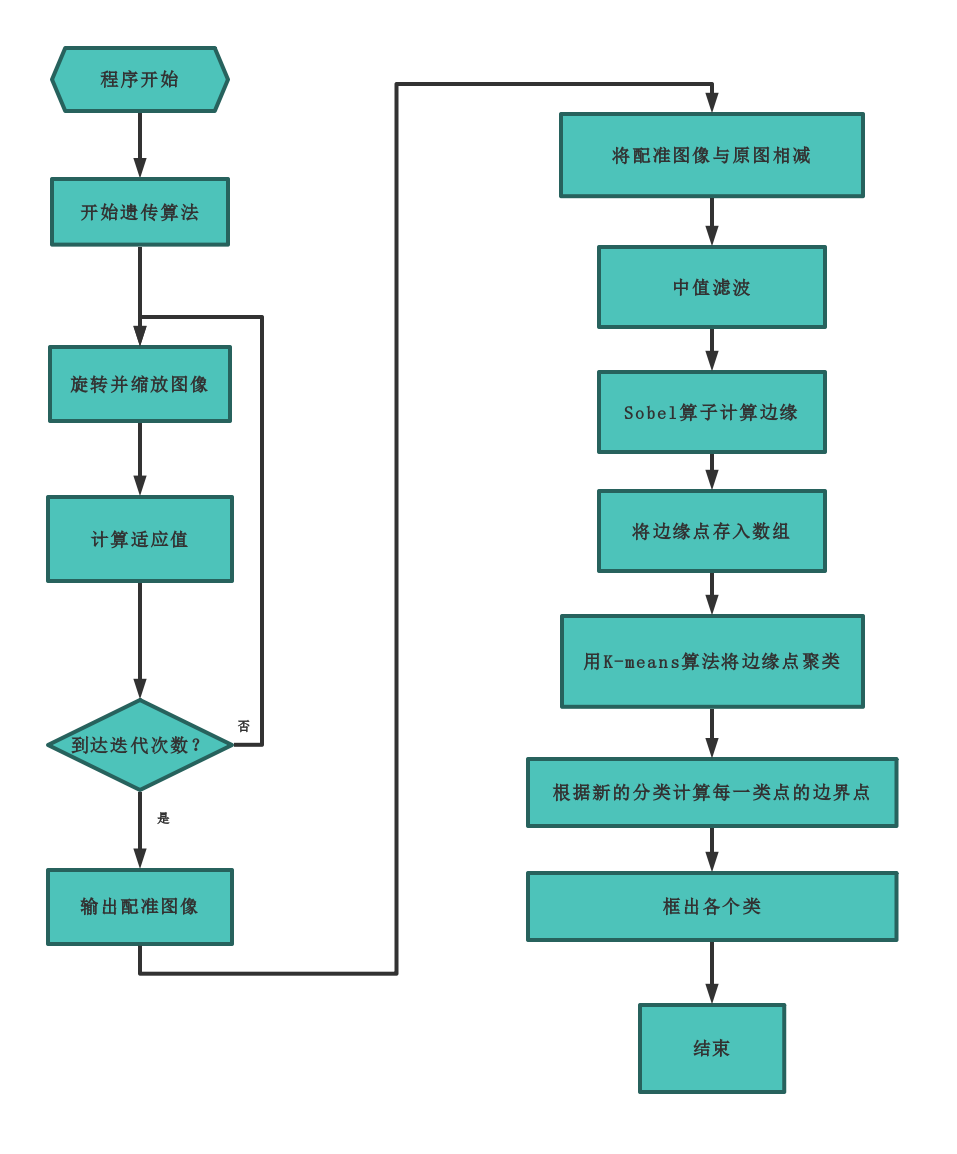
本次大作业我实现了一个具有图像自动配准、自动寻找两张图不同处，以及可以进行交互式找不同的软件。

在本实验中，我使用了遗传算法来进行图像配准，其中，通过均值哈希算法和汉明距离法求解配准图像与基准图像之间的差距值，并以此作为自变量编写了遗传算法的适应度函数。完成配准后，我对配准后的图像与基准图像进行了差值计算、中值滤波、边缘提取等处理，得到一系列离散的点集。紧接着，我利用K-means算法对点集进行了聚类，并通过轮廓系数来评估不同K取值下k-means聚类的效果，以此为依据来选择最佳的K值进行聚类。随后，通过对每一类的点坐标求最值来得出每一类点的范围，从而得出两张图的不同之处。

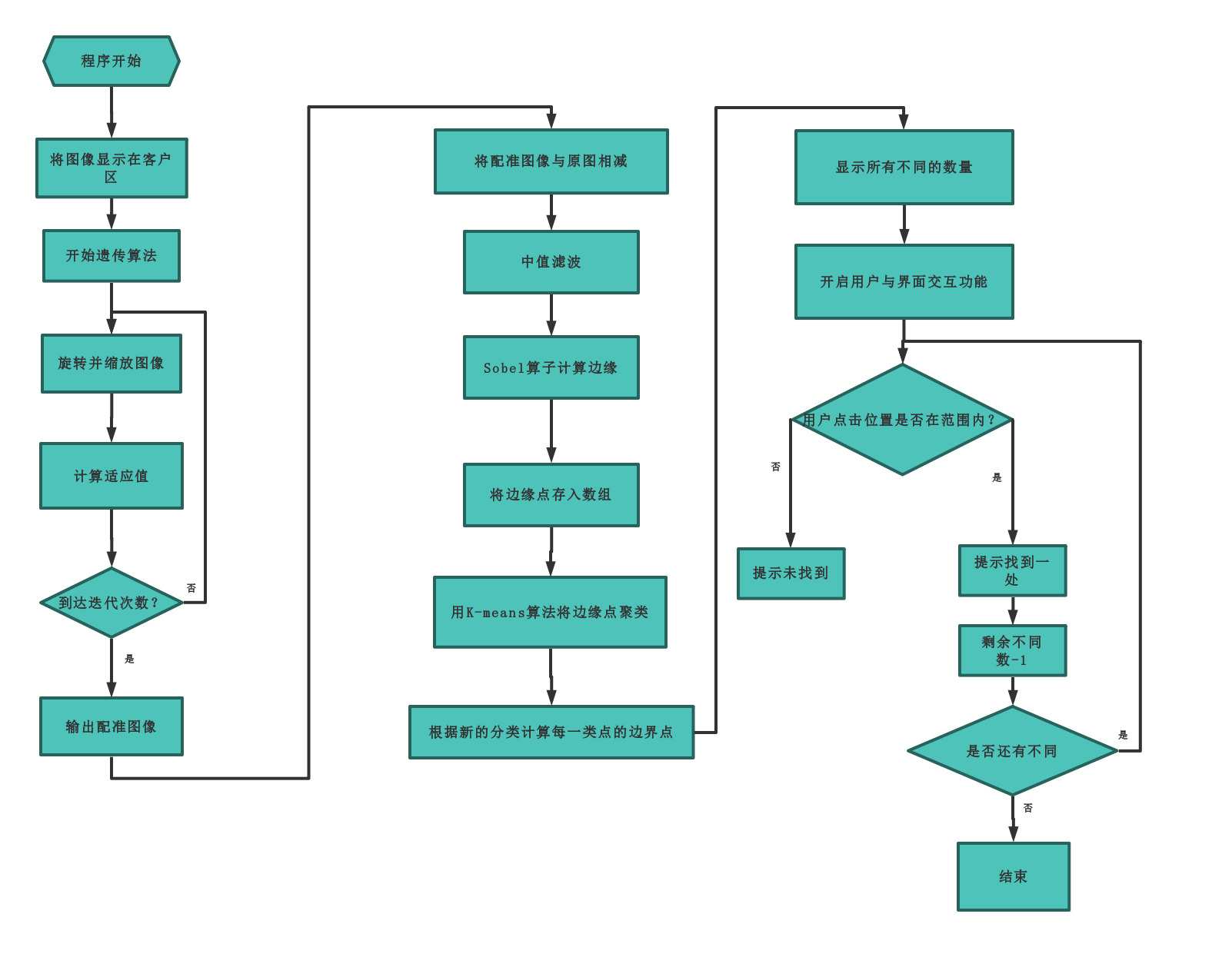
此外，我还通过对Sobel算法进行改进，使其更适应我的算法。并编写了一个用户交互预处理图像的系统，使迭代次数减少，算法准度进一步提升。另外，我还制作了一个交互性的“找不同”游戏，让用户自己找出两图中的不同并给予反馈。

具体的程序流程图如下：

功能一：机器图像配准+自动找不同

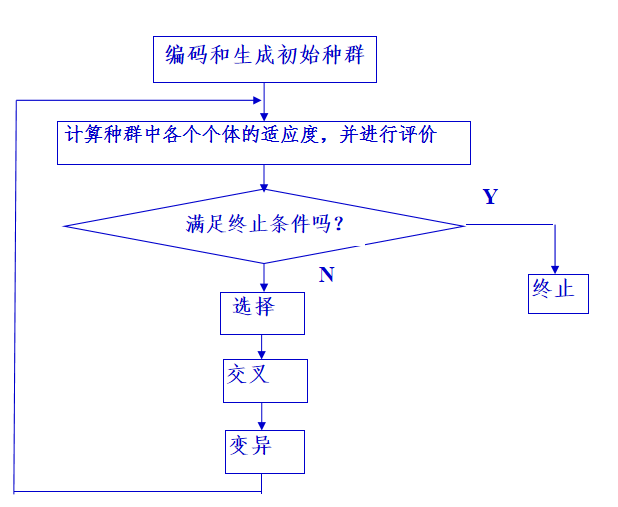


功能二：机器图像配准+用户找不同



1. **各功能实现方法，设计的算法、技术与原理描述**
2. 图像配准功能

为实现图像配准的功能，我使用了遗传算法。遗传算法的基本思想是从初始种群出发，采用优胜劣汰、适者生存的自然法则选择个体，并通过杂交、变异来产生新一代种群，如此逐代进化，直到满足目标为止。遗传算法的具体流程图如下：

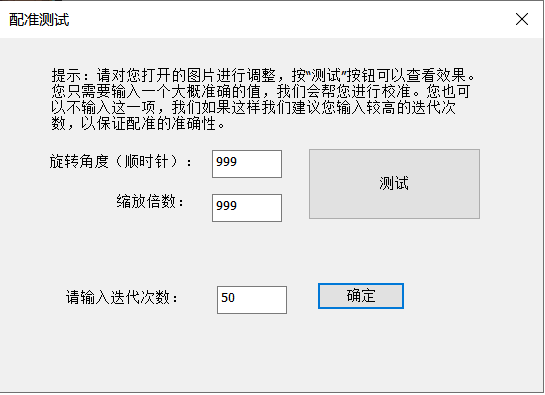


之所以使用遗传算法，是因为在图像配准的过程中有两个变量，一个是旋转角度，一个是缩放比例，如果用不同的方法，很难做到令两个变量同时变化并取最优值，而在遗传算法中，可以将旋转角度和缩放比例当做两个基因。同时，遗传算法容易实现且稳定。

使用遗传算法的难点是编写适应度函数。显而易见的，配准的目的是让配准图像与基准图像尽可能相似，那么需要评估的指标就是“两张图的相似度”，但是相似度是个很模糊的概念，若想评估两幅图像的相似度，有很多种方法可以实现。在本次实验中我采用了均值哈希算法对两幅图进行编码，然后使用汉明距离法对两幅图的编码计算距离，得出一个正数，即为当前个体的适应度值。

其中需要提到的是，我对均值哈希算法也进行了一定的改写与改进。常规的均值哈希算法是将原图压缩成一个8\*8像素的图然后再进行编码。但这样做无疑会导致精度的下降。于是我改写了均值哈希算法，额外传入了图像的长和宽两个新参数，并保持图像不变，再进行编码，这样提高了适应度函数的准确性和合理性。

在测试过程中，出现迭代次数一旦增大运行速度就会非常缓慢这一问题。针对这个问题，我编写了一个用户交互预处理图像的系统，如下图所示：



用户可以根据自己的感觉输入旋转角度和缩放比例，点击测试按钮即可预览配准结果，虽然无法得到十分精准的旋转数值，但是系统会根据用户输入的值取一个范围来包容用户视觉产生的误差，在这个范围内进行遗传算法的选择、交叉、变异操作。经过对多位用户测试，得出了用户视觉比较敏感的范围。

其中，用户对两幅旋转角度相差±3°以上的图比较敏感，也就是说，用户可以轻易分辨出两幅角度相差3°以上的图。

在缩放方面，用户的敏感度降低，但是如果缩放比例比实际的比例小，图像四周会出现黑边，比较容易分辨，但是如果缩放比例比实际的比例大，用户很难对此进行分辨。

综上所述，我对用户输入的值进行了以下的处理：

设用户输入旋转角度为，旋转角度的搜索范围就是。

设用户输入的缩放比例是，缩放比例的搜索范围就是。

经过测试，在这范围内进行遗传算法，可以进一步提升配准的精度，在多数情况下， 5次迭代内即可得到相对优秀的配准结果。

如果用户不愿意进行测试，也可以直接进行配准，不过推荐的迭代次数在50次以上，搜索范围如下：

角度：；缩放比例：。

1. 自动找不同功能

自动找不同功能分为预处理、边缘检测、边缘点聚类、框出范围四部分。

首先，在预处理方面，对两张图进行差值运算是对两幅图像不同点寻找最直接的办法，但是经过配准后的图像与基准图像在清晰度，角度等方面都有一定的差距，直接求差值的效果不佳。会有很多边缘部分也具有很高的亮度。所以紧接着我进行了几次中值滤波。之所以用中值滤波而不用均值滤波，是为了让不同处的点之外的点经过处理后尽量为0，这样可以进一步增加精度。

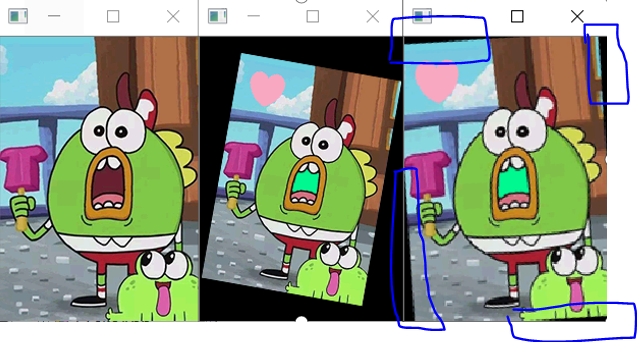
接着，进行边缘检测，之所以进行边缘检测，是因为即使进行了数次的滤波，整体差值图像还是具有很多我们不需要的信息，如下图所示：



可以看到右下角还是有一些白色的点，即使经过五次模板大小为11的中值滤波也无法去除。同时，我们也需要获取左上角的红色块的坐标和中间部分的绿色块的坐标。这些是我们真正要找的不同。

为此，我改写了Sobel算子的算法。传统的Sobel算子是通过一个模板来计算某个点是不是边缘点，如果如果是边缘点，就将其高亮。在本次实验中，我将这些点存入一个Vector数组中，以便后续的处理。同时，我将阈值设置为80，该阈值是在经过许多次测试后的最理想的值。

此外，由于旋转配准难免会遇到一些误差，且如果旋转的角度的误差会导致的直接结果就是配准后的图像四个角都会有一点空白，下图所示是一种配准误差较大的情况：



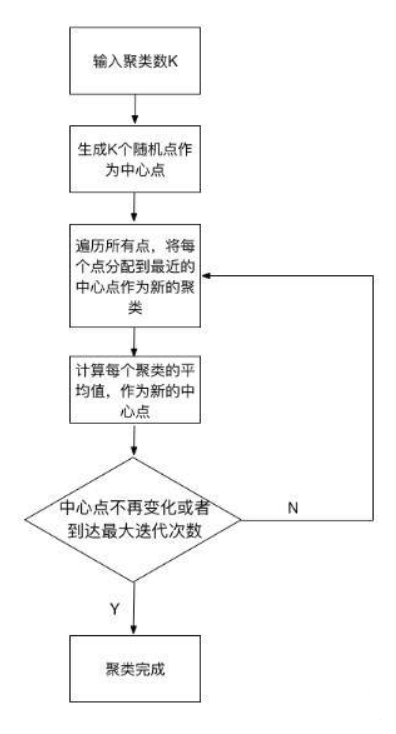
若出现这种情况，这四个角的差值是与真正的不同点一样难以通过滤波而消去的，这也就导致了配准的结果与找不同的正确性息息相关。

为此，我改变了Sobel算子边缘提取算法的初值条件，从第15个像素开始计算，到倒数第15个像素为止。一般来说不同的地方不会在很边缘的地方，所以这也做不会影响正确性，同时，也大大提高了容错率。

在提取了这些边缘点之后，我们得到了一个边缘点的点集，此时，所有不同处的边缘点都存在这个点集中，于是我采用了K-means算法将它们进行聚类。

k均值聚类算法是一种迭代求解的聚类分析算法，其步骤是随机选取K个对象作为初始的聚类中心，然后计算每个对象与各个种子聚类中心之间的距离，把每个对象分配给距离它最近的聚类中心。K-means的最大特点就是，它是一种无监督学习算法。在缺少训练集的情况下，本次实验无疑需要一个无监督学习算法，类似K近邻这样的算法是行不通的。

1. means的算法流程图如下：



1. means可以将一整个点集分成k个点集，即，将各个不同处的边缘点分开。这样一来我们就得到了几个数组，分别存储着各个不同处的点集。

但是现在的问题是，K-means算法必须指定聚类的数量K才能进行聚类，但是我们一般来说并不能事先知道两幅图之间到底有几处不同。所以起初我对K的取值是固定的，可是这样能实现的功能就非常有限。后来我查阅资料和论文，找到了合适的方法——通过轮廓系数来评估不同的K的取值下聚类的效果。从而找出聚类效果最佳的K，并将这个值作为两张图片的不同数量，再用K-means算法进行聚类。

接下来要做的事情就很简单了，在每一个点集内，找出最边缘的四个点，分别作为这个区域的左上，左下，右上，右下角坐标，然后将该区域框出即可。需要注意的是，这里为了防止还有噪声存在于我们的点集，我都选择了第三大，第三小的点，防止极端坐标值（噪声）对结果造成影响。

1. 交互式找不同游戏

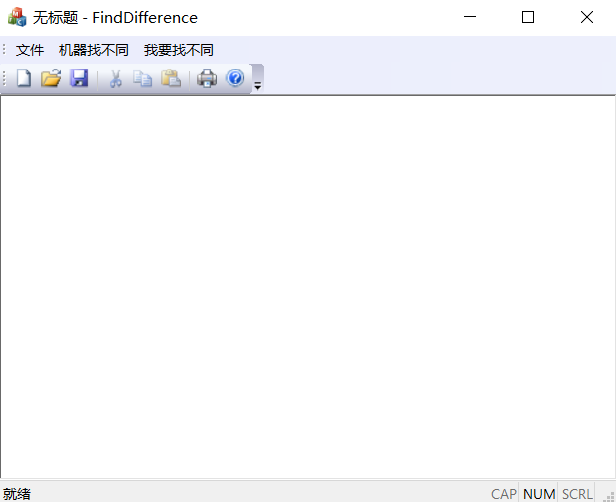
为了丰富功能，我又制作了交互式找不同游戏。该模块的实现方法相对简单，主要是利用了opencv将读取的图片在MFC单文档的客户区显示，然后检测鼠标的点击操作，判断用户点击的位置是否和机器查找的位置相同，并给予反馈。此间诸多细节，就不一一陈述了。

1. **环境配置和操作说明**

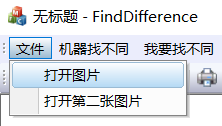
**环境配置**：本次实验利用了opencv4.0.1的库。

**操作说明：**

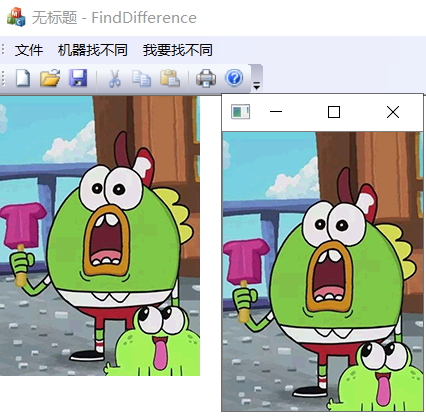
软件主界面如下：



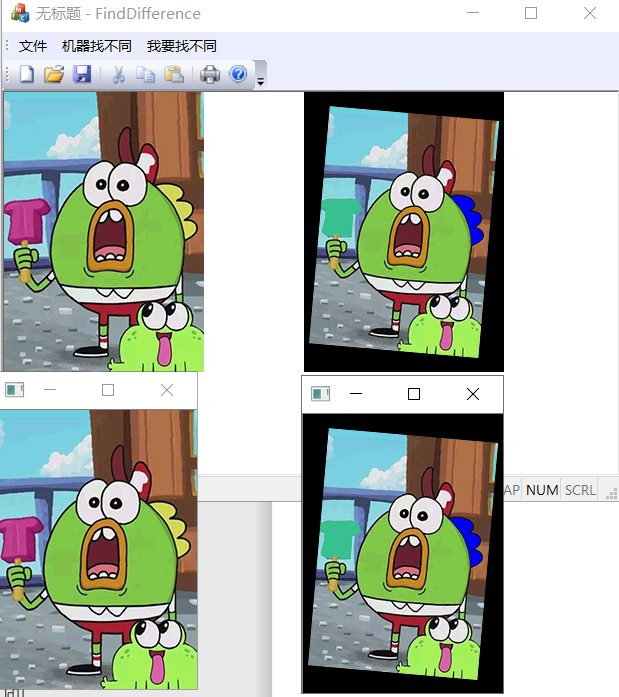
点击菜单“文件”——打开图片，选择基准图片打开。



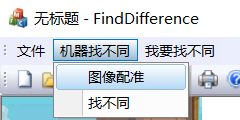
图像会同时在窗口和客户区显示：



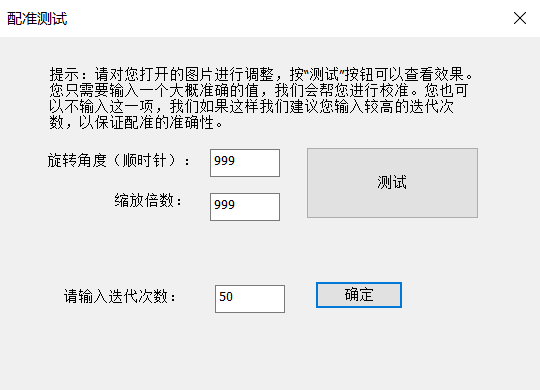
点击菜单“文件”——打开第二张图片，选择待配准的图片打开。



点击菜单——“机器找不同”——“图像配准”

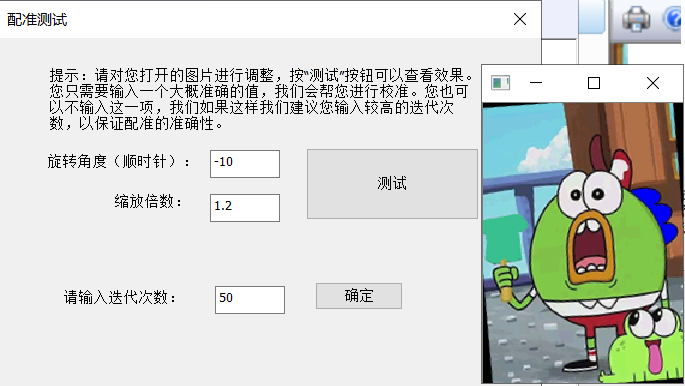


会弹出一个对话框：

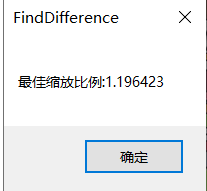
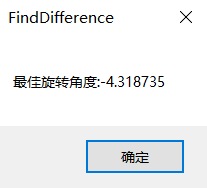
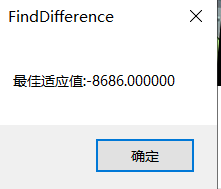


在这个对话框中，可以不管上方的内容，直接输入迭代次数，点击确定开始配准（不推荐）。

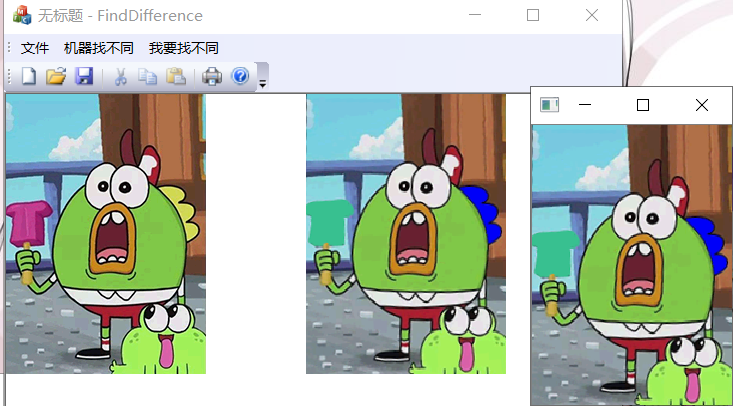
也可以输入旋转角度和缩放倍数，点击测试按钮，会出现按照用户输入的值进行旋转和缩放的图像：



可以进行多次测试，当觉得差不多的时候，输入迭代次数（一般来说5次以内即可），点击确定，程序会自动进行图像配准。配准完毕后，会显示最优解的适应值、旋转角度以及缩放比例：

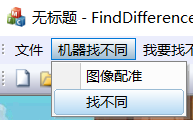


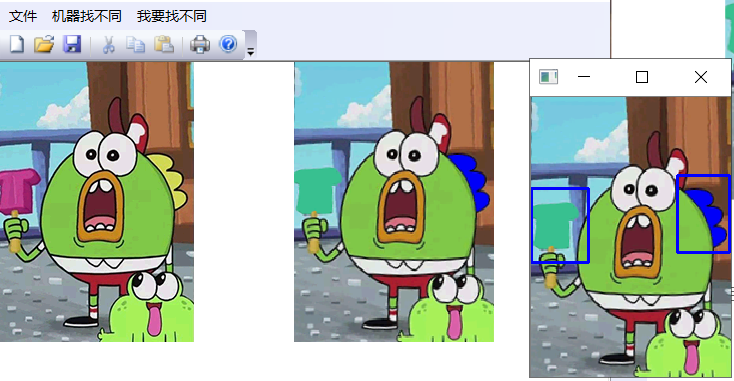
紧接着，会弹出窗口并在客户区第二张图的位置显示配准后的图像：



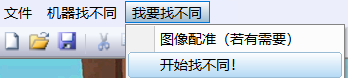
至此，图像配准结束。

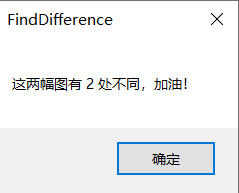
接着，点击菜单——“机器找不同”——“找不同”，机器会自动开始找不同：





点击菜单——“我要找不同”——“开始找不同！”





接下来在屏幕上进行点击，如果选中正确的地方，就会显示找出不同并框出，若点的地方不对则会提示不对：



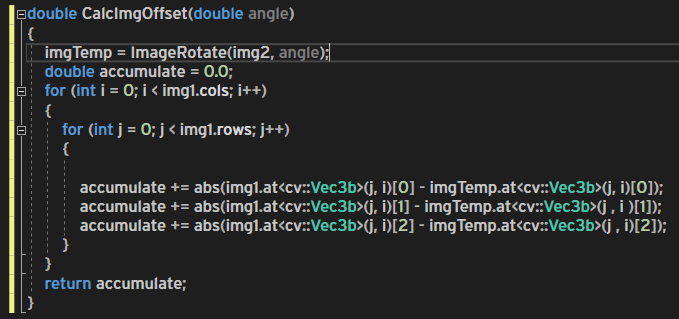


1. **实验难点和解决困难的方法以及收获**

在本次选题中，最大的难点无疑是图像的配准，配准的精度和迭代次数息息相关，但是如果迭代次数高了就会大大减缓程序的运行速度。这个问题贯穿始终，我不断在想，该如何在不牺牲准度的情况下尽可能的减少运算的时间。但是这本质上是个Trade（取舍），为此，我付出了很多努力，这些在后面进行介绍。

先来讲一下进行配准时遗传算法的适应度函数编写问题，众所周知遗传算法的关键是适应度函数，通过评估适应度函数的值来选出最优的群体，那么在进行配准的时候，适应度函数自然也就是两张图的相似程度。

刚开始的时候，我使用的方法是最简单粗暴的逐一像素比对的算法，如下图所示：



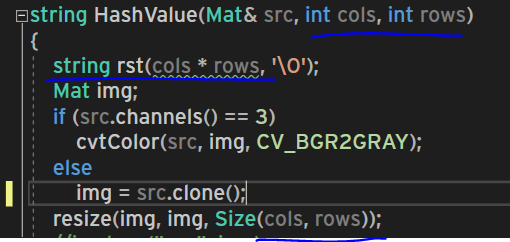
可以看到，这个方法就是逐一比对三个通道，然后加三个通道像素的差值的绝对值累加起来，存入accumulate变量中，这种算法简单，但是结果是不对的。通过这个方法做出来的配准结果错误，会有可能在非最优位置计算出最高的适应值。所以我开始寻找另一种方法。

通过查阅资料，我找到了几个衡量两张图片相似程度的方式。如：感知哈希、平均哈希、差异值哈希等。

其中，差异值哈希是根据计算相邻像素的灰度值差异；平均哈希是根据图像像素的平均值将图像二值化，然后生成哈希编码；而感知哈希是将图像进行离散余弦变换，转到频域上之后，再进行类似平均哈希的变换。为了方便实现，我在本次实验中采用了平均哈希算法。

得出两个图像的哈希编码后，下一步就是计算两张图的“差距”了。计算距离也有很多方法，如，欧氏距离、汉明距离等，本次实验中我采用的是汉明距离法。即可得出两张图的差距值，结果是一个整数，这个数越大，代表两张图差距也就越大。

这里需要提一下，传统的平均哈希算法是将图像压缩成一个8\*8的新图，然后进行编码的，得出的哈希编码长度是64位。但是我认为这样误差过大，会导致很多信息的损失，影响准度。于是我对平均哈希进行了改写，传入待配准图的长和宽，动态生成长度，如下图所示：



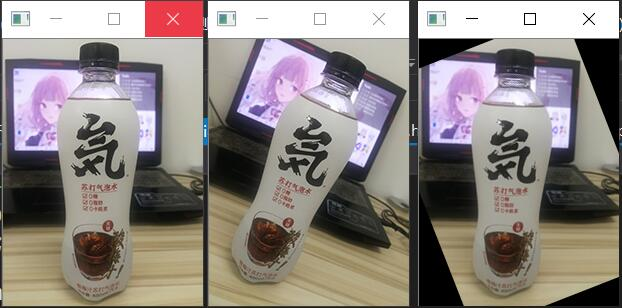
这样一来，就可以获取完整的差距值，使得配准更加准确。

接下来我遇到了一个比较大的问题。由于一般的遗传算法是求适应值最大的点，也就是取最大。但在本次实验中，我要找出的是两幅图差距最小的时候，也就是适应值最小的时候。我的思路是这样的：假设我计算出的两图的距离值为x，那么适应度函数，这样一来，当-x取到最大值时，x就取到了最小值。但是无论我怎么编写适应度函数，最终输出的结果还是最大值。

经过仔细的断点调试我发现，我发现存储最优解的位置始终为0，这也就导致了后面的连锁反应导致结果不对。经过观察和思考我发现了问题所在。在我的算法中，每个个体在迭代中，判断是否大于最优解位置的值，如果大于则替换。但由于我的适应值都是负数，而最优解位置初始化为0，始终无法大于它，所以产生了错误，解决方法很简单，将那个点的初值设为一个很小的负值就可以了：

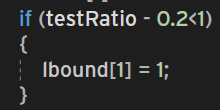


解决了这个问题后，配准的工作基本上就完成了，唯一的问题就是迭代次数高了就会很慢。可以看到，即使是对比较具体的物体，配准的效果依旧不错：



上面两幅图，图一为基准图像，图二为待配准图像，图三为配准结果。

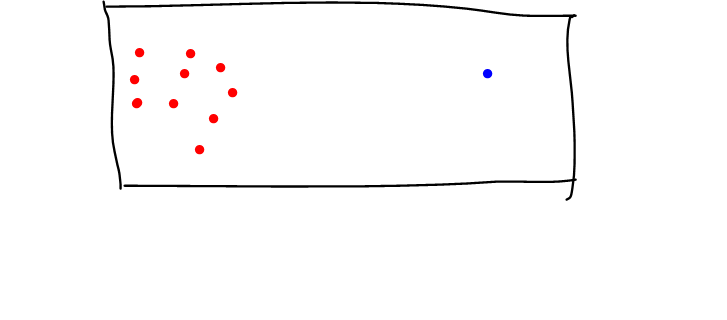
接下来需要解决的问题就是运行时间的问题了，怎么样在不影响精度的情况下提高精度呢？在经过很长时间的思考后，想起老师上课说的可以让用户进行预处理，于是我决定做一个用户预处理的界面。这一部分的功能我在报告的第二节中已经进行了介绍。这里又有一个小bug（展示课上才发现的），在缩放比例上我取得下届是用户输入的值-0.2，这样可能会导致越界，比如用户输入了1.15，可能会搜索到小于1的值（小于1的缩放由于我认为应用的意义不大，就没有做），导致bug，于是我加上了这一行代码，即若下届小于1，将其置为1，以此保证比例在正确范围内：



到此配准的任务就完成了，接下来是找不同的问题了。

找不同方面的流程我也在上面介绍过了，最主要遇到的难题就是配准准度不够导致的连锁反应，解决的方法上面也已经进行了陈述，主要是通过多次的中值滤波，以及对Sobel算子算法的改进，去除边缘，提高精度。

最后我又进行了最后一步的优化，即为了防止还有噪声存在于我们的点集，我都选择了第三大，第三小的点，防止极端坐标值对结果造成影响。这一点我在第二节中也有提到。原本我采用的方法非常简单，就是遍历每个聚类中所有的点，找出这个聚类的左上中下边界，然后将边界稍微扩大，再框出。但是我发现这样很有可能会导致边界扩大，经过我仔细思考，我考虑到了如果有一个点在中值滤波中逃过一劫，在边缘提取中又一次逃过一劫，这样一来就会导致这个点无论在哪个聚类中，都会是最极端的值，从而影响整个聚类的范围判断。



如上图所示，红色点是不同处的点集，也就是我们需要的点集，蓝色点是没有被处理掉的噪声点，它被聚类到了红色点集那一类中，黑色的框就是作为结果框出的范围。可以看到，这样的结果显然是不对的，虽然经过数次滤波和边缘检测，蓝色点出现的几率其实并不大，但是一旦出现就会导致结果出错，这是我不想看到的。

我原本的算法如下所示：

1. **for** (**int** i = 0; i < Diff1.size(); i++)
2. {
3. **if** (Diff1[i].x > Diff1\_X\_Max) Diff1\_X\_Max = Diff1[i].x;
4. **if** (Diff1[i].x < Diff1\_X\_Min) Diff1\_X\_Min = Diff1[i].x;
5. **if** (Diff1[i].y > Diff1\_Y\_Max) Diff1\_Y\_Max = Diff1[i].y;
6. **if** (Diff1[i].y < Diff1\_Y\_Min) Diff1\_Y\_Min = Diff1[i].y;
7. }

可以看到，最后算出的是整个点集中的最值。

改进后的算法如下：

1. sort(Diff1.begin(), Diff1.end(), sortFun\_x);
2. Diff1\_X\_Min = Diff1[3].x;
3. Diff1\_X\_Max = Diff1[Diff1.size()-3].x;
4. sort(Diff1.begin(), Diff1.end(), sortFun\_y);
5. Diff1\_Y\_Min = Diff1[3].y;
6. Diff1\_Y\_Max = Diff1[Diff1.size() - 3].y;

其中，sortFun\_x为：

1. **bool** sortFun\_x(**const** CPoint& p1, **const** CPoint& p2)
2. {
3. **return** p1.x < p2.x;//升序排列
4. }

其中，sortFun\_y为：

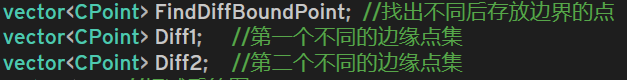
1. **bool** sortFun\_y(**const** CPoint& p1, **const** CPoint& p2)
2. {
3. **return** p1.y < p2.y;//升序排列
4. }

可以看到，我先是对Diff1点集根据x坐标进行升序排列，然后取得是第三个数。也就是说，Diff1\_X\_Min即最左边的点的x坐标，我取得是第三小的点。后面同理，这样一来就可以有效的排除那些极端值的干扰了。接着对y坐标进行升序排列，同理。

经过测试发现，上述的bug几乎不会再出现了。

接下来是一个比较大的问题，就是在给老师进行演示的时候老师提出的问题——由于K-means是需要指定聚类的数量K之后才能进行聚类的，这也就导致了算法不灵活性。由于我们很难事先知道两幅图像到底有几处不同。在我之前的实验作品中，是默认图片都有两个不同点，这显然是不合理的，为此我进行了大刀阔斧的改进。

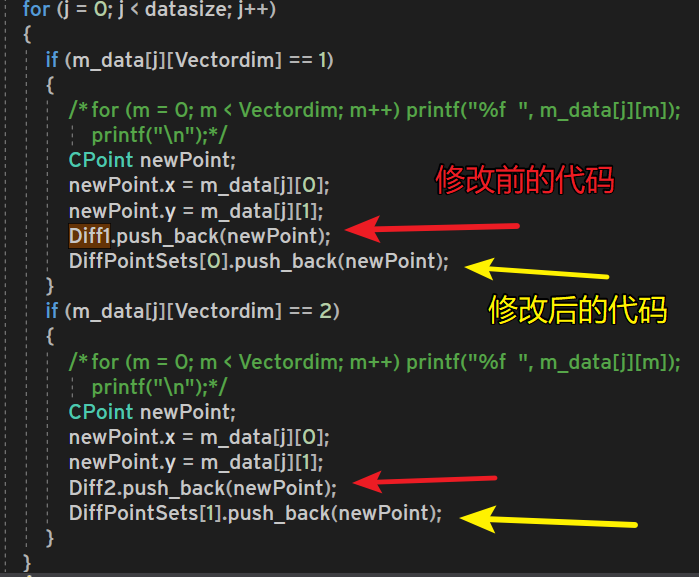
针对K的取值问题，我原本是建立了两个vector列表，分别用来存储两个不同的点集：



之后为了能存储任意个不同的点，我申请了一个二维的vector数组：

vector<vector<CPoint>>DiffPointSets;

随后，在k-means算法中将存入点的方式进行了修改：



做完这些基本的工作后，我开始编写不同K下的聚类评估函数。首先我想到的方法是最直接的一种方法——聚类完成后将所有点和他们的聚类中心点的距离累加起来，因为我想如果这个距离越小，说明所有点和聚类中心的平均距离越近，聚类效果的就好。于是我开始了编写：

1. //主要用这个函数，输入是聚类后得出的中心矩阵和K，输出是累加的总距离
2. **float** evaluate\_K\_distance(Mat centers,**int** K)
3. {
4. DiffPointSets.clear();//清空之前的数据
5. **for** (**int** i = 0; i < K; i++)
6. {
7. vector<CPoint> newDiffSet;
8. DiffPointSets.push\_back(newDiffSet);//开辟新的空间
9. }
10. **float** evaluatedDistance = 0.0;//总距离（用来评估K的取值，函数返回这个值）
11. **for** (**int** i = 0; i < FindDiffBoundPoint.size(); i++)
12. {
13. **float** cloestDistance=999999;//与该点距离最近的中心的距离
14. **int** cloestGroup;//与该点距离最近的中心的编号
15. **for** (**int** j = 0; j < K; j++)
16. {
17. **float** tempDistanceX = FindDiffBoundPoint[i].x - centers.at<Point2f>(j, 0).x;
18. **float** tempDistanceY = FindDiffBoundPoint[i].y - centers.at<Point2f>(j, 0).y;
19. **float** euclideanDistance = sqrt(tempDistanceX \* tempDistanceX + tempDistanceY \* tempDistanceY);//计算欧氏距离
20. **if** (euclideanDistance < cloestDistance)
21. {
22. cloestDistance = euclideanDistance;//更新最短距离
23. cloestGroup = j;//更新最近组
24. }
25. }
26. DiffPointSets[cloestGroup].push\_back(FindDiffBoundPoint[i]);//将该点存入相应的组中
27. evaluatedDistance += cloestDistance;
28. }
29. **return** evaluatedDistance;
30. }

可以看到evaluate\_K\_distance这个函数的作用就是根据聚类中心，将所有点集里的点存入不同的vector数组中，同时计算每一个点和其聚类中心点的欧式距离。

然后在主函数中，我遍历了K从2到5的所有取值，分别计算出在这些取值下evaluate\_K\_distance函数的返回值，即可以体现聚类效果的平均距离：

1. **int** bestK = 2;
2. **float** bestValue = 999999;
3. **for** (**int** k = 2; k < 5; k++)//测试不同的K的效果，kmeans的更改第二个参数
4. {
5. Mat centers(k, 1, InputData.type());    //用来存储聚类后的中心点
6. kmeans(InputData, k, labels, TermCriteria(CV\_TERMCRIT\_EPS + CV\_TERMCRIT\_ITER, 10, 1.0), 3, KMEANS\_PP\_CENTERS, centers);
7. **float** valueTemp=evaluate\_K\_distance(centers, k);
8. **if** (valueTemp < bestValue)
9. {
10. bestValue = valueTemp;
11. bestK = k;
12. }
13. }

但是结果并不尽如人意。对两张一共有三个不同点的图来说，运行结果如下：

K=2时：



K=3时：



K=4时：

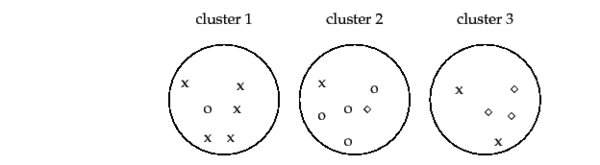


评估的值并没有在K=3处取到最小值，所以这种方法是行不通的。后来我经过思考分析，这种方法从原理上也是错的——因为如果聚类的数量足够大，和所有点的数量相同，也就是每个点都独自归为一个类，那么每个点离他们聚类中心的距离就是0，那么这个评估值就为0，这显然是错误的。而且经过我的分析，平均距离的大小应该是随着K值的增加而单调递减的。所以之前做的都白费功夫了，为此，我决定静下心来寻找相关的资料。

在阅读了很多篇论文之后我发现，对于评估k-means聚类的结果这一问题，当今学术界已经有很多人提出了各种各样的方法。因为必须指定K的值是k-means算法的一大弊端，所以学术界有很多人都想试图解决这一问题。下面我就列举一些我通过论文找到的，且认为还算不错的方法：

1. Purity指数

Purity指数是斯坦福大学一位教授提出的，主要思想如下图所示：

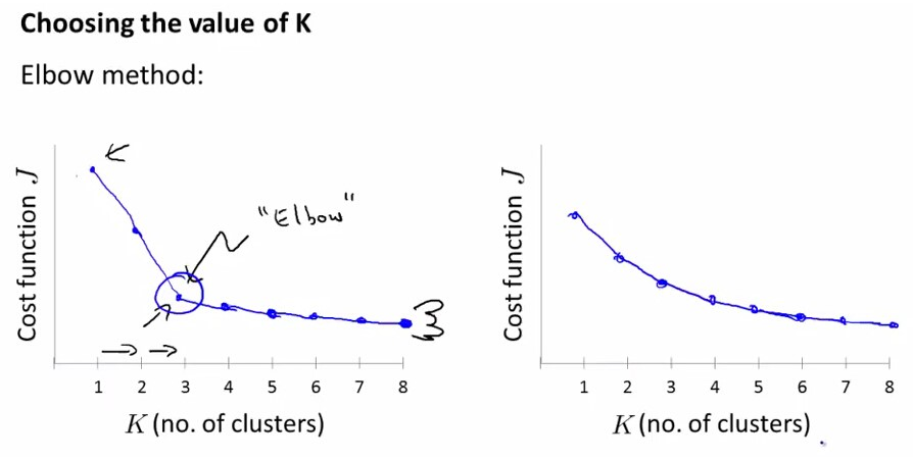


如图，有聚类算法把样本分为3个簇，cluster1,2,3。其中cluster1中x最多，把cluster1看作是x的簇。cluster2中o最多，就看做是o的簇，以此。cluster1中x有5个，cluster2中o有4个，cluster3中◇有3个，总样本数是17个。那么，此次划分Purity指数就是（5+4+3）/17=0.71。

Purity指数可以有效的反应一个聚类的评估效果，但是并不适用于我的实验。因为我的所有点都是一样的，我并不知道他们应该属于哪一类，所以无法使用。

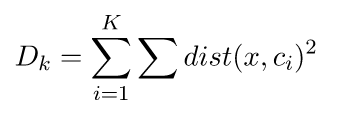
1. 肘部法则（Elbow Method）

Elbow Method ：Elbow意思是手肘，如下图左所示，此种方法适用于 K 值相对较小的情况，当选择的k值小于真正的时，k每增加1，cost值就会大幅的减小；当选择的k值大于真正的K时， k每增加1，cost值的变化就不会那么明显。这样，正确的k值就会在这个转折点，类似elbow的地方。 如下图：



通过画K与cost function的关系曲线图，如左图所示，肘部的值(cost function开始时下降很快，在肘部开始平缓了)做为K值，K=3。并不是所有的问题都可以通过画肘部图来解决，有的问题如右边的那个图，肘点位置不明显（肘点可以是3，4，5），这时就无法确定K值了。故肘部图是可以尝试的一种方法，但是并不是对所有的问题都能画出如左边那么好的图来确定K值。

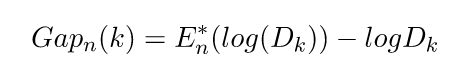
肘部法则的公式如下：



1. 间隔统计量（Gap Statistic）

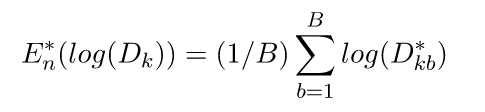
根据肘部法则选择最合适的K值有事并不是那么清晰，因此斯坦福大学的Robert等教授提出了Gap Statistic方法。

这里我们要继续使用上面的Dk。Gap Statistic的定义为:

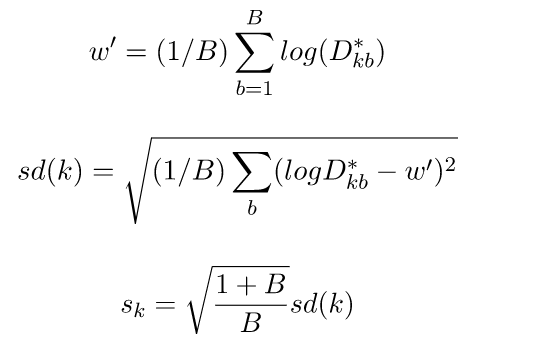


这里指的是的期望。这个数值通常通过蒙特卡洛模拟产生，我们在样本里所在的矩形区域中（高维的话就是立方体区域）按照均匀分布随机地产生和原始样本数一样多的随机样本，并对这个随机样本做K-Means，从而得到一个Dk。如此往复多次，通常20次，我们可以得到20个。对这20个数值求平均值，就得到了的近似值。最终可以计算Gap Statisitc。而Gap statistic取得最大值所对应的K就是最佳的K。

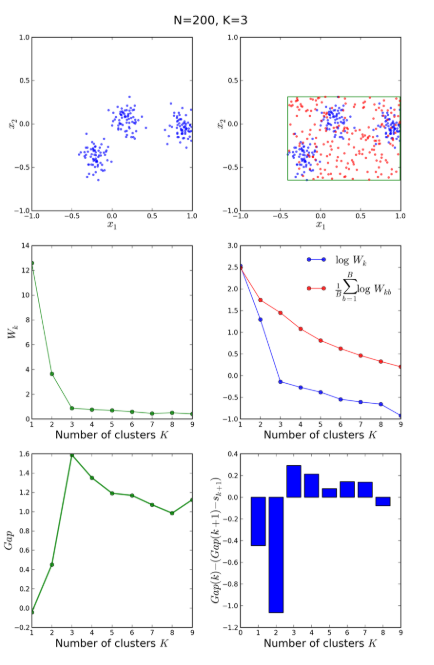
Gap Statistic的基本思路是：引入参考的测值，这个参考值可以有Monte Carlo采样的方法获得。



B是sampling的次数。为了修正MC带来的误差，我们计算sk也即标准差来矫正Gap Statistic。



选择满足的最小的k作为最优的聚类个数。下图阐释了Gap Statistic的实现：



1. 轮廓系数（Silhouette Coefficient）

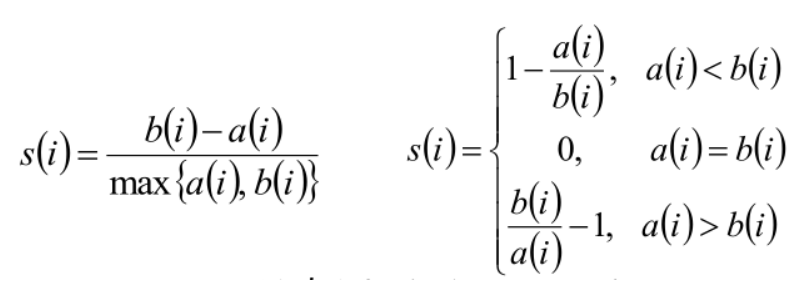
Silhouette method 会衡量对象和所属簇之间的相似度——即内聚性（cohesion）。当把它与其他簇做比较，就称为分离性（separation）。该对比通过 silhouette 值来实现，后者在 [-1, 1] 范围内。Silhouette 值接近 1，说明对象与所属簇之间有密切联系；反之则接近 -1。若某模型中的一个数据簇，生成的基本是比较高的 silhouette 值，说明该模型是合适、可接受的。

具体算法如下：

1）计算样本i到同簇其他样本的平均距离a(i)。a(i)越小，说明样本i越应该被聚类到该簇。将a(i)称为样本i的簇内不相似度。簇C中所有样本的a(i)均值称为簇C的簇不相似度。

2）计算样本i到其他某簇C(j)的所有样本的平均距离b(ij)，称为样本i与簇C(j)的不相似度。定义为样本i的簇间不相似度：b(i) =min{bi1, bi2, …, bik}，b(i)越大，说明样本i越不属于其他簇。

3）根据样本i的簇内不相似度a i 和簇间不相似度b i ，定义样本i的轮廓系数：



4）判断：

s(i)接近1，则说明样本i聚类合理

s(i)接近-1，则说明样本i更应该分类到另外的簇

若s(i) 近似为0，则说明样本i在两个簇的边界上

所有样本的s(i )的均值称为聚类结果的轮廓系数，是该聚类是否合理、有效的度量。但是，其缺陷是计算复杂度为O(n^2)，需要计算距离矩阵，那么当数据量上到百万，甚至千万级别时，计算开销会非常巨大。

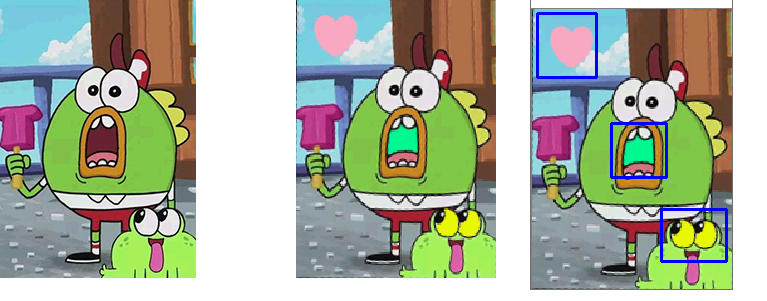
但是由于我找不同得到的差异点并不是特别多，所以完全可以用这种方法。轮廓系数法和我之前编写的直接比较法相比，优点就在于不仅考虑了每一簇中所有点与聚类中心的距离——即聚合度，还考虑了每一簇中所有点与其他簇的点的距离——即分离度。是个比较全面的评估方法，所以，我决定使用轮廓系数来对不同K取值下的K均值聚类的结果进行评估。

编写完函数，发现了一个Bug。对于轮廓系数来说，有一个需要注意的易错点，那就是对于聚合度ai，计算的是所有点在同一簇内其他点距离的平均值。而分离度bi则计算的是每个点到离他最近的那一簇的所有点的平均值（听起来比较绕），是一个先求平均，再求最小的过程。最终的函数如下：

1. **float** Evaluate\_K\_Silhouette\_Coefficient(**int** K)//根据轮廓系数
2. {
3. **float** siTotal = 0.0;//每个点的si累加值
4. **for** (**int** i = 0; i < DiffPointSets.size(); i++)
5. {
6. **for** (**int** j = 0; j < DiffPointSets[i].size(); j++)
7. {
8. //对每个点集中的每个点操作
9. //首先计算a(i)
10. **float** ai = 0.0;
11. **for** (**int** k = 0; k < DiffPointSets[i].size();k++ )
12. {
13. **float** tempDistanceX = (**float**)DiffPointSets[i][j].x - (**float**)DiffPointSets[i][k].x;
14. **float** tempDistanceY = (**float**)DiffPointSets[i][j].y - (**float**)DiffPointSets[i][k].y;
15. **float** euclideanDistance = sqrt(tempDistanceX \* tempDistanceX + tempDistanceY \* tempDistanceY);//计算欧氏距离
16. ai += euclideanDistance;
17. }
18. ai /= DiffPointSets[i].size();//ai是取的平均值
20. //接下来计算bi
21. **float** bi = 999999.9;
22. **for** (**int** m = 0; m < DiffPointSets.size(); m++)//对于每一个点集
23. {
24. **float** biThisSet = 0.0;
25. **if** (m != i)//排除自身所在的那个点集
26. {
27. **for** (**int** n = 0; n < DiffPointSets[m].size(); n++)
28. {
29. **float** tempDistanceX = (**float**)DiffPointSets[i][j].x - (**float**)DiffPointSets[m][n].x;
30. **float** tempDistanceY = (**float**)DiffPointSets[i][j].y - (**float**)DiffPointSets[m][n].y;
31. **float** euclideanDistance = sqrt(tempDistanceX \* tempDistanceX + tempDistanceY \* tempDistanceY);//计算欧氏距离
32. biThisSet += euclideanDistance;
33. }
34. biThisSet /= DiffPointSets[m].size();
35. **if** (biThisSet < bi) bi = biThisSet;
36. }
37. }
39. //通过ai和bi计算si
40. **float** si = (bi - ai) / max(ai, bi);
41. siTotal += si;
42. }
43. }
44. **return** siTotal/FindDiffBoundPoint.size();
45. }

这样一来，程序就可以对任意多个不同的两幅图进行查找了。最后，我修改了其他的一些细节，并修改了找不同的功能。寻找了两张有三处不同的图进行了测试：





此外，本软件对于一些假冒伪劣商标的识别方面也有着值得期待的应用潜力：





最后做一个总结。本次实验中用到了很多本学期学到的人工智能方面的算法，涉及到进化计算，遗传算法，机器学习等各种知识。同时也用到了很多数字图像处理方面的知识，如差距度量，边缘检测，空域滤波等等。总的来说本次实验在结合多学科知识方面做得还算不错，且实现了一定的功能。但是在交互性，算法的效率上还需要更进一步的完善。