信号与系统大作业实验报告

无42 2014011058 陈誉博

一. 作业要求

- 请将3186个图像文件分成两组，第一组是“播音员报道”，第二组是“李博士考察”；提交两个文本文件（A11.txt和A12.txt）,内容分别是这两组的图像文件名。
- 请将第一组图像文件重新排序，并组合成一个连续的图像序列（镜头）；提交一个文本文件（A2.txt）,内容是按正确顺序（或逆序）排列的图像文件名。
- 请将第二组组图像文件重新排序，并组合成一个连续的图像序列（内含 多个镜头）；提交一个文本文件（A3.txt），内容是按正确顺序（或逆 序）排列的图像文件名及其所在的镜头序号（从 1 开始）。

二. 语言与工具

由于我对C++比较熟悉，所以我选择用C++来编写这个程序。为了能够实现用C++读取图片并对图片进行操作，我使用了OpenCV的开源代码（版本号为2.4.13）。此外，程序中还使用了STL模板库中的部分数据结构。

注：OpenCV 2.4.13版本部分函数形式较OpenCV 1.0的形式有较大变化，因此下文中部分函数释义采用OpenCV 1.0版本形式，但实际程序中采用的函数均为OpenCV 2.4.13版本。

三. 编程思路与代码实现

1. 将图片分为“播音员报道”和“李博士考察”两组。

大概思路：将图片压缩至较小尺寸，之后将图片转为HSV空间后对整个图片的三个通道取均值，通过判断三个通道的值是否在某一阈值范围内来判断该图片是否属于“播音员报道”分组。

首先，为了降低复杂度，提高程序的运行速度，需要先将读入的图片进行压缩处理。此处使用了OpenCV中自带的库函数：pyrDown函数（在OpenCV 1.0中对应函数cvPyrDown）。使用pyrDown函压缩片之后，就可以在保证速度的基础上处理图片从而达到分组的目的。

> ***void cvPyrDown(const CvArr* src, CvArr* dst, int filter=CV_GAUSSIAN_5x5);***
> **src**
> 输入的8-bit,16-bit或 32-bit单倍精度浮点数影像。
> **dst**
> 输出的8-bit, 16-bit或 32-bit单倍精度浮点数影像, 宽度和高度应是输入图像的一半
> **filter**
> 卷积滤波器的类型，目前仅支持 CV_GAUSSIAN_5x5
> **功能**
> 函数 cvPyrDown 使用 Gaussian 金字塔分解对输入图像向下采样。它对输入图像用指定滤波器进行卷积，然后通过拒绝偶数的行与列来下采样图像

经观察发现，在所有的图片中，属于“播音员报道”场景的图片具有非常显著的特征，即图片中的大部分区域为蓝色。为了方便判断图片中像素点的颜色，此处使用了OpenCV中自带的库函数：cvtColor函数（在OpenCV1中对应函数cvCvtColor）。

> ***void cvCvtColor(const CvArr* src, CvArr* dst, int code);***
> **src**
> 输入的8-bit,16-bit或 32-bit单倍精度浮点数影像。
> **dst**
> 输出的8-bit, 16-bit或 32-bit单倍精度浮点数影像。
> **code**
> 色彩空间转换的模式，该code来实现不同类型的颜色空间转换。比如CV_BGR2GRAY表示转换为灰度图，CV_BGR2HSV将图片从RGB空间转换为HSV空间。其中当code选用CV_BGR2GRAY时，dst需要是单通道图片。当code选用CV_BGR2HSV时，对于8位图，需要将RGB值归一化到0-1之间。这样得到HSV图中的H范围才是0-360，S和V的范围是0-1。
> **功能**
> 函数cvCvtColor将某一色彩空间的图片转化到另一个色彩空间

此处使用的code参数为CV_BGR2HSV,即可将RGB三通道的图像转化为HSV三通道的图像。转化为HSV图像之后，对整幅图片的所有像素点取均值。之后通过判断HSV三个通道的均值是否取值在一定范围内就可以判别图像的平均颜色是否为蓝色。此处为了提高效率，没有采用遍历矩阵求均值的方法，而是使用了OpenCV中自带的库函数:mean函数。由于图像为HSV图像，所以mean函数的返回值为一个长度为3的向量，三个分量分别为H，S，V。

> ***cv::Scalar cv::mean(InputArray src, InputArray mask = noArray());***
> **src**
> 输入的 8-bit,16-bit或 32-bit单倍精度浮点数影像。
> **mask**
> 输入的 8-bit,16-bit或 32-bit单倍精度浮点数影像,代表感兴趣区域(ROI)。
> **功能**
> 函数mean计算并返回src中感兴趣区域mask(默认情况为整个src)所有取值的平均值

至此将图片分组的任务已经完成，计算部分代码如下（部分变量定义语句略）：

```
fstream file("all_image.txt");
fstream filea11("A11.txt",ios_base::out|ios_base::trunc);
fstream filea12("A12.txt",ios_base::out|ios_base::trunc);
char name[10]={0,0,0,0,0,0,0,0,0,0};
char name2[30]="unsorted_images/";
for(i=0;i<total_number;i++)
    file<<i+1<<".jpg"<<endl;
file.seekg(0,ios::beg);
filea11.seekg(0,ios::beg);
filea12.seekg(0,ios::beg);
for(i=0,talking_count=0,outdoor_count=0;i<total_number;i++)
{
    //loading images
    file.getline(name,10);
    for(j=16;j<26&&name[j-16]!='\0';j++)
        name2[j]=name[j-16];
    img[i]=imread(name2,1);
    //decrease dimensions to 1/64 of the original image
    //through Gaussian Pyramid algorithm
    pyrDown(img[i],temp);
    pyrDown(temp,temp);
    pyrDown(temp,temp);
    //convert RGB images to HSV images
    cvtColor(temp,temp,CV_BGR2HSV);
    //if the average of all the pixels is blue, writing to the talking file
    if((mean(temp).val[0]>105&&mean(temp).val[0]<135)
        &&(mean(temp).val[1]>50&&mean(temp).val[1]<255)
        &&(mean(temp).val[2]>50&&mean(temp).val[2]<255))
    {
        filea11<<i+1<<endl;
        cvtColor(img[i],img_gray[i],CV_RGB2GRAY);
        talking_img.push_back(img_gray[i]);
        wrongorder.push_back(temp_picture);
        wrongorder.back().original_talking_order=talking_count;
        wrongorder.back().number=i+1;
```

```
        talking_count++;
    }
    else
    {
        filea12<<i+1<<endl;
        outdoor_count++;
        cvtColor(img[i],img_gray[i],CV_RGB2GRAY);
        outdoor_img.push_back(img_gray[i]);
    }
}
```

2. 请将第一组图像文件重新排序，并组合成一个连续的图像序列（镜头）。

大概思路：首先任取一张第一组中的图片（不妨取第一张）作为正确顺序集合的初始图片。首先以正确顺序集合中的第一张图片为基准，在错误顺序集合中找到与基准图片最相似的图片并记录其编号A和与基准图片之间的误差error_A。然后以正确顺序集合中的最后一张图片为基准，在错误顺序集合中找到与基准图片最相似的图片并记录其编号B和与基准图片之间的误差error_B。找到两张备选图片A和B后，比较error_A和error_B的大小，若error_A小于error_B,则将A放在正确顺序集合第一张图片之前；否则将B放在正确顺序集合最后一张图片之后。将加入正确顺序集合的图片从错误顺序集合中删掉。不断重复上述过程直至所有图片全部位于正确顺序集合中。

为了提高效率并降低编程难度，提高代码可读性，此处正确顺序集合和错误顺序集合的数据结构均采用STL模板库中的vector。

接下来需要不断扩充正确顺序集合的规模，其中包含找到与基准图片最相似的一张图片的步骤。此处我采用的方法是将每张图片视为一个多维向量，用两个向量的差向量的长度来表示两张图片之间的误差。此处为了提高效率节省时间，计算差向量长度的运算使用了OpenCV自带的库函数：norm函数。此处使用了CV_L2，计算的是两个矩阵之间的二范数。

> ***double cv::norm(InputArray src1, InputArray src2, int normType, InputArray mask=noArray());***
> **src1**
> 输入的第一个8-bit,16-bit或 32-bit单倍精度浮点数影像。
> **src2**
> 输入的第二个8-bit,16-bit或 32-bit单倍精度浮点数影像。
> **normType**
> 范数的类型，normType=CV_INF时norm=max|src1-src2|; normType=CV_L1时normsrc1=sum{src1-src2}; normType=CV_L2时norm=sqrt(sum{(src1-src2)^2})
> 功能
> 函数norm根据normType的取值计算src1和src2的相对差分范数

接下来根据误差的大小将最相似的图片放在正确顺序集合中的某个位置。在这里最开始我采用了遍历错误顺序集合，每访问一张图片都计算它与基准图片的误差的方案。但是经过分析之后发现了这种方案存在一定的问题：每次计算图片之间的误差时都要以正确顺序集合第一张和最后一张为基准计算两遍，但是每次只将一张图片加入正确顺序集合。假如某次循环将一张图片放在了正确顺序集合的第一张，那么下次循环中以最后一张图片为基准的误差计算结果和本次的计算结果完全相同，带来了冗余的计算。为了提高效率，避免浪费时间，采用先计算误差后排序的方案。即在循环开始之前先声明一个二维数组，其第i行第j列元素的含义为错误顺序集合中第i张图片与第j张图片之间的误差。在循环中每次需要用到误差的值时不用重新计算，只需要访问数组中的值即可。此处还有一个小技巧：易知误差矩阵一定为对称阵，所以数组的ij元素和ji元素的值一定相同，计算完ij元素之后直接给ji元素赋值即可，不必再次计算。

至此将第一组图像文件重新排序的任务已经完成，计算部分代码如下（部分变量定义语句略）：

```
double **error_matrix=new double*[talking_count];
for(i=0;i<talking_count;i++)
    error_matrix[i]=new double[talking_count];
//calculate the distance from the ith image to the jth image
Mat temp_mat;
for(i=0;i<talking_count;i++)
{
    for(j=i;j<talking_count;j++)
    {
        if(i==j) error_matrix[i][j]=0;
        else
        {
            //temp_mat=talking_img[i]-talking_img[j];
            error_matrix[i][j]=norm(talking_img[i],talking_img[j],CV_L2);
            error_matrix[j][i]=error_matrix[i][j];
        }
    }
}
//search for the min_distance image for the front and back of the set of right order
//choose the smallest distance in order to decide
//whether put the new image in the front or the back
rightorder.push_back(wrongorder[0]);
wrongorder.erase(wrongorder.begin());
Mat error_mat_front,error_mat_back;
double temp_min_front=2147483647,temp_min_back=2147483647,
min_error_front=2147483647,min_error_back=2147483647;
int position_front=0,position_back=0;int iTemp_error_front=0,iTemp_error_back=0;
for(i=0;rightorder.size()<talking_count;i++)
{
    for(j=0,temp_min_front=2147483647,temp_min_back=2147483647,
    min_error_front=2147483647,min_error_back=2147483647;
    j<wrongorder.size();j++)
    {
        iTemp_error_front=error_matrix[rightorder.front().original_talking_order]
        [wrongorder[j].original_talking_order];
        if(iTemp_error_front<min_error_front)
        {
            min_error_front=iTemp_error_front;
            position_front=j;
        }
        iTemp_error_back=error_matrix[rightorder.back().original_talking_order]
        [wrongorder[j].original_talking_order];
        if(iTemp_error_back<min_error_back)
        {
            min_error_back=iTemp_error_back;
            position_back=j;
        }
    }
    if(min_error_front<min_error_back)
    {
        rightorder.insert(rightorder.begin(),wrongorder[position_front]);
        wrongorder.erase(wrongorder.begin()+position_front);
    }
    else
    {
        rightorder.insert(rightorder.end(),wrongorder[position_back]);
            wrongorder.erase(wrongorder.begin()+position_back);
    }
}
//write correct order to the file
fstream talking_writeorder("A2.txt",ios_base::out);
for(i=0;i<talking_count;i++)
    talking_writeorder<<rightorder[i].number<<endl;
```

3.请将第二组组图像文件重新排序，并组合成一个连续的图像序列（内含多个镜头）。

大概思路:首先，与前两问类似，先用pyrDown函数对图片进行降维压缩处理。压缩之后对所有图片进行聚类操作，把每个镜头中的所有图片归到一个集合中。聚类完成之后，对每个镜头中的图片进行排序操作。类间排序结束后再对所有类进行排序即可。

- 聚类

在实现聚类操作的时候，考虑到K-means算法中含有一定的随机性，而且需要将目标类的个数作为函数的参数，所以此处不采用K-means聚类算法。考虑到聚类的时候需要以不同图片之间的误差来判别是否处于一个集合中，所以此处选择用层次聚类算法。然而直接使用层次聚类算法的复杂度过高，需要的时间很长，所以此处对层次聚类算法做了一定的改进，具体算法如下：首先寻找出所有图片中误差最小的两张图片，将这两张图片放在一个类中。将这两张图片从未聚类的集合中去除。以该类的第一张图片为基准找到对应误差最小的图片A和误差error_A；以该类的最后一张图片为基准找到对应误差最小的图片B和误差error_B。若error_A<error_B，则将A放到该类的第一张图片的位置上，否则将B放到该类的最后一张图片的位置上。重复上述过程，直至未聚类集合中的图片到该类中图片的最小误差大于一定阈值，第一次聚类过程结束。不断重复上述聚类操作直至未聚类集合中无图片。经过上述操作，就可以得到聚类完毕而且每个类中的图片均大致有序的集合。

在聚类这一步中，最耗费时间的步骤即为求未聚类图片集合到已聚类的误差最小值。如果直接用for循环遍历的话复杂的将非常大。此处为了节省时间，使用了OpenCV中自带的库函数：minmanldx函数。该函数可以在O（1）的时间内返回最大值/最小值以及它们的位置。这为求全局最小值的计算节省了大量时间。

> *void minMaxIdx(InputArray src, double* minVal, double* maxVal, int* minIdx=0, int* maxIdx=0, InputArray mask=noArray());*
> **src**
> 输入的单通道矩阵。
> **minVal/maxVal**
> 指向最小值/最大值的指针。
> **minIdx/maxIdx**
> 最小值/最大值在输入矩阵中的位置。（注：由于矩阵的维度至少为2维，所以minIdx/maxIdx数组的长度至少为2.）
> **mask**
> 输入的单通道矩阵，代表感兴趣区域(ROI)，默认为整个矩阵。
> 功能
> 函数minmaxIdx求输入矩阵的感兴趣区域中最小值/最大值及其位置。

为了能够正确使用这个函数，在求全局最小值的时候才用之前提到过的先计算误差后算最小值的策略，即把所有图片之间的误差计算出来之后存到一个Mat结构中（第i行第j列元素代表第i张图片和第j张图片之间的误差），将此Mat作为输入即可求得最小值和位置和对应图片。同时在每次将一张图片加入已聚类时，为了在未聚类集合中将该图片去除，需要将加入图片对应的行向量和列向量的值均置为某一大数以防重复聚类。计算部分代码如下：

```
//the distance[i][j] stands for the distance
//between outdoor_img[i] and outdoor_img[j]
Mat distance(outdoor_count,outdoor_count,CV_64FC1);
vector <vector <Picture2>> sets;
vector <Picture2> temp_push,temp_vector;
for(i=0;i<outdoor_count;i++)
    for(j=i+1;j<outdoor_count;j++)
    {
        distance.at<double>(i,j)=norm(compressed_outdoor_img[i].image,
            compressed_outdoor_img[j].image,CV_L2);
        distance.at<double>(j,i)=distance.at<double>(i,j);
    }
for(i=0;i<outdoor_count;i++)
    distance.at<double>(i,i)=2147483647.0;
//fix_distance is used in the fixing process
Mat fix_distance=distance.clone();
int sum=outdoor_count;
while(sum>10)
{
    //here I use the optimized HierarchicalCluster method
    //to figure out different scenes
    double minVal=0.0,maxVal=0.0;
    int minIdx[2]={0,0},maxIdx[2]={0,0};
    //find the overall minium distance and the matrices
    minMaxIdx(distance,&minVal,&maxVal,minIdx,maxIdx);
    sets.push_back(temp_push);
    //use the matrices that have the smallest distance to gain a set
    sets.back().push_back(compressed_outdoor_img[minIdx[0]]);
    sets.back().push_back(compressed_outdoor_img[minIdx[1]]);
    int minIdx_front[2]={0,0},minIdx_back[2]={0,0};
    double minVal_front=0.0,minVal_back=0.0;
    while((int)sets.back().size()<outdoor_count)
    {
        //get the other pictures' distance to the front and back pictures
        temp_mat=distance.row(sets.back().front().outdoor_order).clone();
        temp_mat2=distance.row(sets.back().back().outdoor_order).clone();
        //set the distance of the front and back to maximum
        for(i=0;i<(int)sets.back().size();i++)
        {
            temp_mat.at<double>(0,sets.back()[i].outdoor_order)=2147483647;
            temp_mat2.at<double>(0,sets.back()[i].outdoor_order)=2147483647;
        }
        //temp_mat and temp_mat2 are all row vectors
        //so the number of the min_distance is minIdx[1]
        minMaxIdx(temp_mat,&minVal_front,&maxVal,minIdx_front,maxIdx);
        minMaxIdx(temp_mat2,&minVal_back,&maxVal,minIdx_back,maxIdx);
        //if the smallest distance is smaller than some value,
        //put it into the current set,else the picture belongs to a new set
        if(minVal_front<2200||minVal_back<2200)
        {
            if(minVal_front<minVal_back)
                sets.back().insert(sets.back().begin(),
                    compressed_outdoor_img[minIdx_front[1]]);
            else
                sets.back().insert(sets.back().end(),
                    compressed_outdoor_img[minIdx_back[1]]);
        }
        else break;
    }
    //set the distances to pictures in current sets to maximum
    //logically delete the known pictures from the unknown sets
    for(i=0;i<(int)sets.back().size();i++)
    {
        temp_mat=Mat::ones(1,outdoor_count,CV_64FC1)*2147483647.0;
        temp_mat2=Mat::ones(outdoor_count,1,CV_64FC1)*2147483647.0;
        temp_mat.copyTo(distance.row(sets.back()[i].outdoor_order));
        temp_mat2.copyTo(distance.col(sets.back()[i].outdoor_order));
    }
    sum-=(int)sets.back().size();
}
```

然而，在调试过程中发现，用此种聚类方法得到的结果总会有一张缺失。这是由于当只剩一张图片的时候，误差矩阵中元素已经全部变为最大值，所以这张图片无法加入之前的已聚类。为了将这张图片包括进已聚类，代码中增加了test机制，不仅仅是为了将最后一张图片包含进来，也是为了防止中间某些过程可能由于代码编写的疏忽导致某些图片没有进入已聚类。test部分完成之后，聚类的工作到此结束。test部分代码如下：

```
//the test array is used to examine
//whether there is any picture still unknown
bool *test=new bool[outdoor_count];
```

```
for(i=0;i<outdoor_count;i++)
    test[i]=false;
//if one picture is missed, the number of
//the picture in the test array stay false
for(i=0;i<(int)sets.size();i++)
    for(j=0;j<(int)sets[i].size();j++)
        test[sets[i][j].outdoor_order]=true;
int first_minIdx[2]={0,0},first_maxIdx[2]={0,0};
double first_minVal=0.0,first_maxVal=0.0;
//find the nearest picture to the missed picture
//to decide which set is the missed picture from
for(i=0;i<outdoor_count;i++)
{
    if(test[i]==false)
    {
        minMaxIdx(fix_distance.row(i),&first_minVal,&first_maxVal,
            first_minIdx,first_maxIdx);
        for(ii=0;ii<(int)sets.size();ii++)
        {
            for(jj=0;jj<(int)sets[ii].size();jj++)
            {
                if(sets[ii][jj].outdoor_order==first_minIdx[1])
                {
                    sets[ii].insert(sets[ii].begin()+sets[ii].size()/2,
                        compressed_outdoor_img[i]);
                    test[i]=true;
                    temp_mat=Mat::ones(1,outdoor_count,CV_64FC1)*2147483647.0;
                    temp_mat2=Mat::ones(outdoor_count,1,CV_64FC1)*2147483647.0;
                    temp_mat.copyTo(fix_distance.row(sets[ii].back().outdoor_order));
                    temp_mat2.copyTo(fix_distance.col(sets[ii].back().outdoor_order));
                    break;
                }
            }
            if(test[i]==true)
                break;
        }
    }
}
```

- 类内排序

在聚类工作结束之后，类内排序的复杂度较整个室外场景排序的复杂度就大大降低。观察到图片中有一个转动的地球可以作为排序的依据，所以此处使用了OpenCV中的ROI机制将小地球单独取出，排序时只看小地球即可。此处排序的基本思想和第二问相同。排序部分代码如下：

```
Picture2 *outdoor_mask=new Picture2[outdoor_count];
vector <vector <Picture2>> rightorder_sets;
for(i=0;i<outdoor_count;i++)
{
    outdoor_mask[i].number=compressed_outdoor_img[i].number;
    outdoor_mask[i].outdoor_order=compressed_outdoor_img[i].outdoor_order;
    outdoor_mask[i].image=outdoor_img[i].image(cv::Rect(55,398,65,37));
}
double temp_distance_front=0.0,temp_distance_back=0.0;
double min_distance_front=2147483647.0,min_distance_back=2147483647.0;
int outdoor_position_front=0,outdoor_position_back=0;
for(i=0;i<(int)sets.size();i++)
{
    vector <Picture2> wrongorder=sets[i];
    rightorder_sets.push_back(temp_push);
    rightorder_sets.back().push_back(outdoor_mask[sets[i][sets[i].size()/2].outdoor_order]);
    wrongorder.erase(wrongorder.begin()+sets[i].size()/2);
    while(wrongorder.size()>0)
    {
        for(ii=0,min_distance_front=2147483647.0,min_distance_back=2147483647.0;
            ii<(int)wrongorder.size();ii++)
        {
            temp_distance_front=norm(rightorder_sets.back().front().image,
                outdoor_mask[wrongorder[ii].outdoor_order].image,CV_L1);
            temp_distance_back=norm(rightorder_sets.back().back().image,
                outdoor_mask[wrongorder[ii].outdoor_order].image,CV_L1);
            if(temp_distance_front<min_distance_front)
            {
                min_distance_front=temp_distance_front;
                outdoor_position_front=ii;
            }
            if(temp_distance_back<min_distance_back)
            {
                min_distance_back=temp_distance_back;
                outdoor_position_back=ii;
            }
        }
        if(min_distance_front<min_distance_back)
        {
            rightorder_sets.back().insert(rightorder_sets.back().begin(),
                outdoor_mask[wrongorder[outdoor_position_front].outdoor_order]);
            wrongorder.erase(wrongorder.begin()+outdoor_position_front);
        }
        else
        {
            rightorder_sets.back().push_back(
                outdoor_mask[wrongorder[outdoor_position_back].outdoor_order]);
            wrongorder.erase(wrongorder.begin()+outdoor_position_back);
        }
    }
}
```

然而，经过调试发现，有个别集合出现了中间图片连续，两端图片跳变的情况。为了消除这种情况，保证最后一步类间排序的准确度，我加入了矫正端点跳变的措施，代码如下：

```
//check the last image of the vector
for(i=0;i<rightorder_sets.size();i++)
{
    j=rightorder_sets[i].size();
    double distance_behind=norm(outdoor_mask[rightorder_sets[i][j-1].outdoor_order].image,
        outdoor_mask[rightorder_sets[i][j-2].outdoor_order].image,CV_L1);
    double temp_distance=0.0,min_distance=2147483647.0;
    int position=0;
    double distance_forward=0.0,distance_next=0.0;
    if(distance_behind>50000)
    {
        temp=rightorder_sets[i][j-1];
        rightorder_sets[i].pop_back();
        for(jj=0,min_distance=2147483647.0;jj<rightorder_sets[i].size();jj++)
```

```
        {
            temp_distance=norm(outdoor_mask[temp.outdoor_order].image,
                    outdoor_mask[rightorder_sets[i][jj].outdoor_order].image,CV_L1);
            if(temp_distance<min_distance)
            {
                min_distance=temp_distance;
                position=jj;
            }
        }
        if(position>0&&position<rightorder_sets[i].size()-1)
        {
            distance_forward=norm(outdoor_mask[rightorder_sets[i][position].outdoor_order].image,
                outdoor_mask[rightorder_sets[i][position-1].outdoor_order].image,CV_L1);
            distance_next=norm(outdoor_mask[rightorder_sets[i][position].outdoor_order].image,
                outdoor_mask[rightorder_sets[i][position+1].outdoor_order].image,CV_L1);
            if(distance_forward<distance_next)
                rightorder_sets[i].insert(rightorder_sets[i].begin()+position,temp);
            else
                rightorder_sets[i].insert(rightorder_sets[i].begin()+position+1,temp);
        }
        else if(position==rightorder_sets[i].size()-1)
            rightorder_sets[i].insert(rightorder_sets[i].end()-1,temp);
        else
            rightorder_sets[i].insert(rightorder_sets[i].begin(),temp);
    }
    else continue;
}
//check the first image of the vector
for(i=0;i<rightorder_sets.size();i++)
{
    double distance_before=norm(outdoor_mask[rightorder_sets[i][0].outdoor_order].image,
        outdoor_mask[rightorder_sets[i][1].outdoor_order].image,CV_L1);
    double temp_distance=0.0,min_distance=2147483647.0;
    int position=0;
    double distance_forward=0.0,distance_next=0.0;
    if(distance_before>50000)
    {
        temp=rightorder_sets[i][0];
        rightorder_sets[i].erase(rightorder_sets[i].begin());
        for(jj=0,min_distance=2147483647.0;jj<rightorder_sets[i].size();jj++)
        {
            temp_distance=norm(outdoor_mask[temp.outdoor_order].image,
                outdoor_mask[rightorder_sets[i][jj].outdoor_order].image,CV_L1);
            if(temp_distance<min_distance)
            {
                min_distance=temp_distance;
                position=jj;
            }
        }
        if(position>0&&position<rightorder_sets[i].size()-1)
        {
            distance_forward=norm(outdoor_mask[rightorder_sets[i][position].outdoor_order].image,
                outdoor_mask[rightorder_sets[i][position-1].outdoor_order].image,CV_L1);
            distance_next=norm(outdoor_mask[rightorder_sets[i][position].outdoor_order].image,
                outdoor_mask[rightorder_sets[i][position+1].outdoor_order].image,CV_L1);
            if(distance_forward<distance_next)
                rightorder_sets[i].insert(rightorder_sets[i].begin()+position,temp);
            else
                rightorder_sets[i].insert(rightorder_sets[i].begin()+position+1,temp);
        }
        else if(position==rightorder_sets[i].size()-1)
            rightorder_sets[i].insert(rightorder_sets[i].end(),temp);
        else
            rightorder_sets[i].insert(rightorder_sets[i].begin()+1,temp);
    }
    else continue;
}
```

至此类内排序已经完成。

- 类间排序

类间排序的依据就是每张图片上面都有的小地球。类间排序的主要思路是先在所有排过序的类中按照头尾距离最短的原则进行连接。按照此种原则连接好之后，序列中可能会出现某两个镜头中地球转动方向不一致的情况。通过某种方法将所有镜头的转动方向调节为一致之后，再用类似第二问图像排序的思路将所有的镜头按照球的转动情况排好，即可得到最终的正确序列。

第一步是按照头尾距离最短的原则对所有集合进行连接。这一步中一共考虑四种情况：两个集合之间头头匹配，头尾匹配，尾头匹配，尾尾匹配，对应的四种操作为（以第一个集合为基准）：,<2,1>,<1,2>,<1,reverse(2)>。连接操作代码如下：

```
vector <vector <Picture2>> outdoor_rightorder;
outdoor_rightorder.push_back(rightorder_sets.front());
rightorder_sets.erase(rightorder_sets.begin());
double min_head_head=2147483647.0,temp_head_head=0.0;
double min_head_end=2147483647.0,temp_head_end=0.0;
double min_end_head=2147483647.0,temp_end_head=0.0;
double min_end_end=2147483647.0,temp_end_end=0.0;
int position_head_head=0;
int position_head_end=0;
int position_end_head=0;
int position_end_end=0;
while(rightorder_sets.size()>0)
{
    for(i=0,min_head_head=2147483647.0,min_head_end=2147483647.0,
        min_end_head=2147483647.0,min_end_end=2147483647.0;i<(int)rightorder_sets.size();i++)
    {
        temp_head_head=norm(outdoor_mask[outdoor_rightorder.front().front().outdoor_order].image,
            outdoor_mask[rightorder_sets[i].front().outdoor_order].image,CV_L1);
        temp_head_end=norm(outdoor_mask[outdoor_rightorder.front().front().outdoor_order].image,
            outdoor_mask[rightorder_sets[i].back().outdoor_order].image,CV_L1);
        temp_end_head=norm(outdoor_mask[outdoor_rightorder.back().back().outdoor_order].image,
            outdoor_mask[rightorder_sets[i].front().outdoor_order].image,CV_L1);
        temp_end_end=norm(outdoor_mask[outdoor_rightorder.back().back().outdoor_order].image,
            outdoor_mask[rightorder_sets[i].back().outdoor_order].image,CV_L1);
        if(temp_head_head<min_head_head)
        {
            min_head_head=temp_head_head;
            position_head_head=i;
        }
        if(temp_head_end<min_head_end)
        {
            min_head_end=temp_head_end;
            position_head_end=i;
        }
        if(temp_end_head<min_end_head)
```

```
        {
            min_end_head=temp_end_head;
            position_end_head=i;
        }
        if(temp_end_end<min_end_end)
        {
            min_end_end=temp_end_end;
            position_end_end=i;
        }
    }
    //the rightorder's head match some sets_vector's head
    if(min_head_head < min_head_end && min_head_head < min_end_head && min_head_head < min_end_end)
    {
        reverse(rightorder_sets[position_head_head].begin(),rightorder_sets[position_head_head].end());
        outdoor_rightorder.insert(outdoor_rightorder.begin(),rightorder_sets[position_head_head]);
        rightorder_sets.erase(rightorder_sets.begin()+position_head_head);
    }
    //the rightorder's head match some sets_vector's end
    if(min_head_end < min_head_head && min_head_end < min_end_head && min_head_end < min_end_end)
    {
        outdoor_rightorder.insert(outdoor_rightorder.begin(),rightorder_sets[position_head_end]);
        rightorder_sets.erase(rightorder_sets.begin()+position_head_end);
    }
    //the rightorder's end match some sets_vector's head
    if(min_end_head < min_head_head && min_end_head < min_head_end && min_end_head < min_end_end)
    {
        outdoor_rightorder.insert(outdoor_rightorder.end(),rightorder_sets[position_end_head]);
        rightorder_sets.erase(rightorder_sets.begin()+position_end_head);
    }
    //the rightorder's end match some sets_vector's end
    if(min_end_end < min_head_head && min_end_end < min_head_end && min_end_end < min_end_head)
    {
        reverse(rightorder_sets[position_end_end].begin(),rightorder_sets[position_end_end].end());
        outdoor_rightorder.insert(outdoor_rightorder.end(),rightorder_sets[position_end_end]);
        rightorder_sets.erase(rightorder_sets.begin()+position_end_end);
    }
}
rightorder_sets.clear();
```

第二步是将所有镜头的转动方向调整到一致。由于上一步已经用头尾相连的策略将各个集合连接在了一起，所以可以观察到，当相邻两个镜头的旋转方向不一致时，其图片中的地球区域在两次转动过程中具有对称性，而转动方向一致的两个镜头则没有这个特性，对应图片相差很大。所以只需要在每两个镜头的连接处向左/向右找第25张图片（为了提高区分度，保证结果的准确性，此处取25张图片而不是一张图片）并求出两张图片之间的距离。当求得的距离大于一定阈值的时候，将两个镜头合并，作为一组连续镜头，否则将该组镜头划分为一组新镜头。划分结束之后，显然每间隔一组镜头，地球的旋转方向相反，所以只需要每隔一组镜头将所有图片逆序排列。通过这种方法，可以将所有镜头中地球的转动方向调整至完全一致。调整代码如下：

```
rightorder_sets.push_back(outdoor_rightorder.front());
for(i=1;i<(int)outdoor_rightorder.size();i++)
{
    j=(int)outdoor_rightorder[i-1].size();
    double distance_between=norm(outdoor_mask[outdoor_rightorder[i][24].outdoor_order].image,
           outdoor_mask[outdoor_rightorder[i-1][j-25].outdoor_order].image,CV_L1);
    //if distance_between < 100000, it means that rightorder[i-1] and rightorder[i] twist in different orders
    if(distance_between>=100000.0)
    {
        rightorder_sets.back().insert(rightorder_sets.back().end(),
            outdoor_rightorder[i].begin(),outdoor_rightorder[i].end());
    }
    else
        rightorder_sets.push_back(outdoor_rightorder[i]);
}
for(i=1;i<(int)rightorder_sets.size();i=i+2)
    reverse(rightorder_sets[i].begin(),rightorder_sets[i].end());
```

最后一步是用类似第二问的方法将划分为组的镜头再次排序。原理不再赘述，代码如下：

```
outdoor_rightorder.clear();
outdoor_rightorder.push_back(rightorder_sets.front());
rightorder_sets.erase(rightorder_sets.begin());
//rightorder_sets is the known sequence and outdoor_rightorder is the output sequence
outdoor_position_front=0;outdoor_position_back=0;
while(rightorder_sets.size()>0)
{
    for(i=0,min_distance_front=2147483647.0,min_distance_back=2147483647.0;
        i<(int)rightorder_sets.size();i++)
    {
        //find the best match for front
        temp_distance_front=norm(outdoor_mask[outdoor_rightorder.front().front().outdoor_order].image,
            outdoor_mask[rightorder_sets[i].back().outdoor_order].image,CV_L1);
        //find the best match for back
        temp_distance_back=norm(outdoor_mask[outdoor_rightorder.back().back().outdoor_order].image,
            outdoor_mask[rightorder_sets[i].front().outdoor_order].image,CV_L1);
        if(temp_distance_front<min_distance_front)
        {
            min_distance_front=temp_distance_front;
            outdoor_position_front=i;
        }
        if(temp_distance_back<min_distance_back)
        {
            min_distance_back=temp_distance_back;
            outdoor_position_back=i;
        }
    }
    if(min_distance_front<min_distance_back)
    {
        outdoor_rightorder.insert(outdoor_rightorder.begin(),rightorder_sets[outdoor_position_front]);
        rightorder_sets.erase(rightorder_sets.begin()+outdoor_position_front);
    }
    else
    {
        outdoor_rightorder.push_back(rightorder_sets[outdoor_position_back]);
        rightorder_sets.erase(rightorder_sets.begin()+outdoor_position_back);
    }
}
fstream output3("A3.txt");
int previous_scene_number=2147483647;
int scene_number=0;
for(i=0,scene_number=0;i<(int)outdoor_rightorder.size();i++)
    for(j=0;j<(int)outdoor_rightorder[i].size();j++)
    {
        if(previous_scene_number!=outdoor_rightorder[i][j].set_order)
            scene_number++;
        output3<<outdoor_rightorder[i][j].number<<","<<scene_number<<endl;
        previous_scene_number=outdoor_rightorder[i][j].set_order;
```

```
    }
output3.clear();
output3.close();
```

至此所有计算已经全部完成。

四. 实验总结与感悟

此次大作业的内容在之前学过的课程中从未接触过。为了完成这次大作业，我查阅了很多资料，并且学会了OpenCV的使用，感觉收获非常大。尤其是对图片聚类的那一部分，在确定用优化过的层次聚类算法之前也尝试了很多不同的方法，比如K-means和传统层次聚类的算法。为了能够编写出可以执行的代码，我也查阅了很多与聚类算法有关的文献，学到了很多全新的知识。另外，在判断图片相似性的时候，也使用了很多种不同的判别方法。首先想到的是相关系数法。因为在信号与系统的课上讲过相关系数这一概念，所以首先想到了这一概念。但是由于相关系数涉及到积分运算，复杂度较高，所以在程序中没有使用。其次，我还想到了用sift算法做特征点匹配，但是虽然OpenCV提供了相应的库函数，其复杂度还是很高，所以这一方案也没有采用。最后采用了矩阵相减求范数的方法来比较相似度。同时在编写程序的过程中也遇到了很多细节上的问题，导致结果出现错误，也因此花费了很多时间寻找这些问题的解决办法。总之，这次大作业带给我的收获非常大，同时也激起了我对编程的热情和对计算机视觉领域的兴趣。非常感谢谷老师能够布置这次大作业，让我能够自主学习之前没有接触过的知识。

附：当前目录文件清单

当前目录下有一个code文件夹，存放的是四个程序的源代码、工程文件和运行结果。 code 目录下有四个子目录，分别对应ABCD四个测试样本。每个子目录下都有 SignalSystemHomework 和 SignalSystemHomework2 和 results 三个文件夹，分别对应前两问和第三问以及最终的运行结果。 以第一个文件夹为例：

- code/signalsystem/SignalSystemHomework 中：除 SignalSystemHomework 文件夹和 SignalSystemHomework.sln 文件之外均为无关文件。 SignalSystemHomework.sln 文件为 Visual Studio 2012 的工程文件，双击该文件即可加载配置，运行程序。 SignalSystemHomework 文件夹中为数据文件。
- code/signalsystem/SignalSystemHomework/SignalSystemHomework中， unsorted_images 为乱序图片所在文件夹， *11.txt 、 *12.txt 、 *2.txt 为前两问的运算结果。 all_image.txt 为中间数据文件，用于读取文件。 OpenCVProject.cpp 为前两问运算程序的源代码。其余均为无关文件。
- code/signalsystem/SignalSystemHomework2 中：除 SignalSystemHomework 文件夹和 SignalSystemHomework.sln 文件之外均为无关文件。 SignalSystemHomework.sln 文件为 Visual Studio 2012 的工程文件，双击该文件即可加载配置，运行程序。 SignalSystemHomework 文件夹中为数据文件。
- code/signalsystem/SignalSystemHomework2/SignalSystemHomework2中， unsorted_images 为乱序图片所在文件夹， *11.txt 、 *12.txt 、 *2.txt 为前两问的运算结果，*3.txt 为第三问的运行结果。 all_image.txt 为中间数据文件，用于读取文件。 OpenCVProject2.cpp 为第三问问运算程序的源代码。其余均为无关文件。
- code/signalsystem/results 为运行结果的四个txt文件所在文件夹，文件均按规定格式命名。

其余所有文件夹中的格式与第一个文件夹中一模一样。

另外当前目录下还有一个support文件夹，里面包含了一个说明文档，内容为OpenCV开源代码下载地址和安装说明。