

华中科技大学人工智能与自动化学院数据结构课程设计

**上**

**机**

**报**

**告**

自动化2003班

杨淳如 U202090056

指导老师：韩守东

上交时间：2021年12月11日

目录

[1、问题描述 2](#_Toc89951137)

[1.1程序要求： 2](#_Toc89951138)

[1.2实验题目： 2](#_Toc89951139)

[1.3需求分析 3](#_Toc89951140)

[问题一： 3](#_Toc89951141)

[问题二： 4](#_Toc89951142)

[问题三： 4](#_Toc89951143)

[1.4运行环境 4](#_Toc89951144)

[2、算法设计 4](#_Toc89951145)

[2.1随即打点+分水岭算法 4](#_Toc89951146)

[2.2四色法填充分水岭区域 5](#_Toc89951147)

[2.3折半查找 5](#_Toc89951148)

[3、程序源码 6](#_Toc89951149)

[3.1宏定义 6](#_Toc89951150)

[3.2储存结构 7](#_Toc89951151)

[3.3主函数 10](#_Toc89951152)

[3.4问题一：随即打点+分水岭算法 15](#_Toc89951153)

[3.5问题二：四原色 20](#_Toc89951154)

[3.6问题三：折半查找 36](#_Toc89951155)

[4、调试分析 42](#_Toc89951156)

[4.1常规运行 42](#_Toc89951157)

[4.2容错处理 46](#_Toc89951158)

[4.3调试 47](#_Toc89951159)

[5、上机总结 49](#_Toc89951160)

# 1、问题描述

## 1.1程序要求：

输入：任意一张M\*N的彩色图片

工具：VC++6.0、VS或者Mac/Linux平台下的其他编译器

OpenCV 1.0、2.x或者3.x

纯C语言或者C/C++语言

具体的编写要求：1.不允许使用STL标准模板库。

2.少用静态数组，多用指针和动态内存分配/释放。

3.对源码中的文件、函数进行合理划分，保证模块独立性。

4.函数、变量、常量等命名规范（去汉语拼音）。

5.文件、函数宏观注释，核心变量、程序段微观注释。

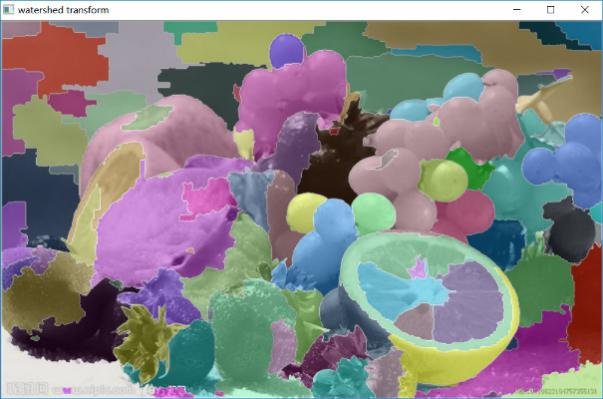
6.对输入进行合法性校验和功能、容错提示。

7.尽量优化算法，确保低时空复杂度。

## 1.2实验题目：

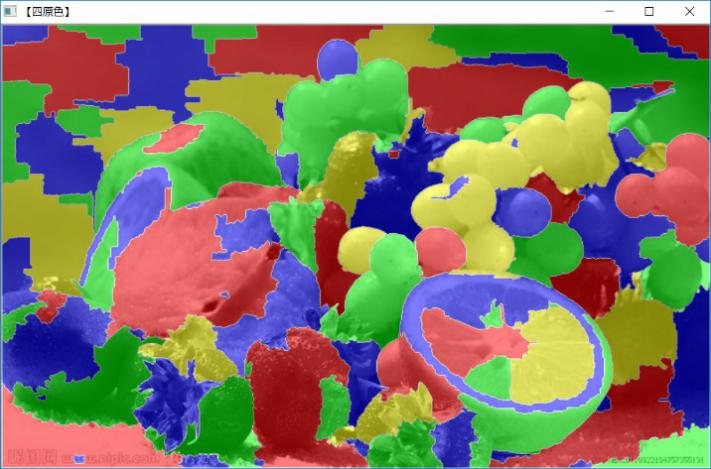
问题一:随机算法

使用基于种子标记的分水岭算法（cv::watershed）对输入图像进行过分割。用户输入图像和整数K，要求程序自动计算K个随机种子点，确保各种子点之间的距离均>(M\*N/K)0.5（参考泊松圆盘随机取样算法），然后程序在原图中使用3\*3小方块+区域编号[1,K]标出各种子点，并采用半透明+随机区域着色方式给出分水岭算法的可视化结果。



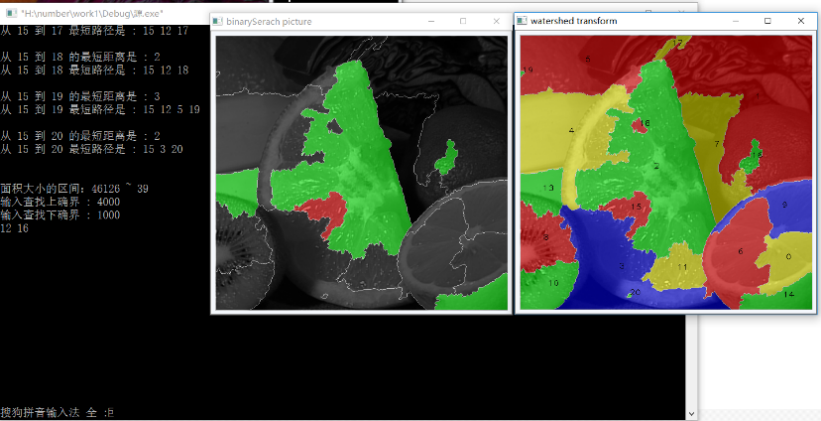
问题二:四原色

使用邻接表统计分水岭结果中各区域的邻接关系，并采用四原色法（合理选择初始着色区域，并基于图的广度优先遍历，采用队列对其他待着色区域进行着色顺利梳理，加速全图着色过程），对分水岭结果重新着色（使用堆栈，优化回退率）。



问题三:折半查找

根据分水岭结果中各区域面积大小的排序结果，提示最大和最小面积，使用折半查找，用户输入查找范围（面积下界和上界），程序对所有符合要求的分水岭结果进行突出显示。



## 1.3需求分析

### 问题一：

核心问题：尽可能多得生成符合距离要求的随机点，并用分水岭算法对其进行着色

数据模型：结构，存储种子点的各种信息

存储结构：自定义结构

核心算法：泊松圆盘算法，分水岭算法

输入数据：目标对应的生成种子点的个数

### 问题二：

核心问题：四色法填充分水岭法划分的区域

数据模型：邻接表，包括头节点HeadNode以及表节点Node，表结点来表示与头节点相邻的区域

存储结构：邻接表

核心算法：邻接表的遍历以及堆栈的入栈和出栈

输入数据：种子点信息以及分水岭算法的图像信息

### 问题三：

核心问题：使用折半查找查找用户的查找范围（面积下界和上界）内的区域

数据模型：一维整形数组，存储种子点区域的面积

存储结构：数组

核心算法：折半查找算法

输入数据：各区域面积

## 1.4运行环境

硬件开发环境：ASUS X540UP

软件开发环境：Visual Studio 2019

操作系统环境：Windows 10

# 2、算法设计

## 2.1随即打点+分水岭算法

2.1.1算法简介

泊松圆盘是指以泊松分布原理生成满足距离要求的随机点的算法。

2.1.2算法实现

具体算法描述：

固定一个位于画面中央的点（这样开始时的寻找空间更大，可以使找到的点数更多，生成速度更快），给其一个随机的角度，按规定的半径R向外延伸，若其与其他点之间的距离能够满足泊松圆盘的条件，则将该点放入点阵中，若不行，则重新生成随机角度，再次进行寻找，并对查找次数上限进行宏定义（GetNewNearPointListNodeIterateLimit），便于在调试过程中修改更优的查找次数上限。

## 2.2四色法填充分水岭区域

2.2.1算法简介

将种子点信息以顺序结构存储在邻接表中，按照邻接块数从大到小的顺序遍历后生成新的涂色顺序表数组，然后以堆栈的方式进行涂色。

2.2.2算法描述和实现

具体算法描述：

从所有邻接表头节点中找出链接表节点数最大的那个头节点作为顺序表的第一个；

遍历此头节点所有表节点，表节点依次存放到顺序表中，当所有表节点遍历完后，选择最后一个表节点作为新的头节点，重复上述操作，已经存入顺序表中的节点就无需二次存入，直到头节点中所有的表节点都已经存入顺序表中，生成涂色顺序表数组；

创建一个涂色信息数组，以涂色顺序数组储存的种子点区域特征值为下标，进行如下操作：

按上述下标给涂色信息数组赋颜色值（1.2.3.4），每赋值一次，遍历之前所有区域，判断是否有相邻区域涂相同颜色。如果没有，此颜色值入栈，继续赋值；如果有，颜色值增加，再次判断。如果四种颜色值均不满足要求，出栈，重新赋值，直到涂色完成。

## 2.3折半查找

2.3.1算法简介

在采用顺序存储结构的有序表存储的条件下每次将待查记录所在区间缩小一半的查找算法。

2.3.2算法描述和实现

具体算法描述：

折半查找所有符合要求的区域，在堆排序完成的基础上，本任务也很容易实现，可以用两种方法实现，一种找到区域中满足要求的最大区域，和最小区域。另外一种解决方案是，对上确界和下确界中间的每一个值进行折半查找。本实验采用的第一种方式，其具体算法在此不做赘述。

# 3、程序源码

## 3.1宏定义

**Defs.hpp**

#pragma once

#include <opencv2/opencv.hpp>

using namespace cv;

using namespace std;

#define ERROR 0

#define OK 1

#define EXIT 2

#define NONE -1

#define FALSE 0

#define TRUE 1

#define RED 0

#define GREEN 1

#define BLUE 2

#define YELLOW 3

#define PI 3.1415926

#define GetNewNearPointListNodeIterateLimit 20

#define INF 100000000

#define MAX\_RIGION\_NUM 1500

#define POP\_RECORDS\_LIMIT 250

#define M 1024

#define N 664

#define IMGPATH ("D:\\default.jpeg")

#define ORIGINPOINT (M / 2, N / 2)

#define STEP3 //用于控制是否进行四原色填充并输出可视化结果（由于四原色填充过程耗时较长，跳过该步骤可节省延时时间）

## 3.2储存结构

**ColorStack.hpp**

#pragma once

#include "Defs.hpp"

typedef struct ColorStackNode

{

    int regionID;

    int color;

    struct ColorStackNode \*next;

} ColorStackNode;

int ColorStackInit(ColorStackNode \*(&s));

int GetColorStackLen(ColorStackNode \*s);

int ColorStackPush(ColorStackNode \*(&s), int regionID, int color);

int ColorStackPop(ColorStackNode \*(&s), int \*regionID);

**ColorStack.cpp**

#include "ColorStack.hpp"

int ColorStackInit(ColorStackNode \*(&s))

{

    if ((s = (ColorStackNode \*)malloc(sizeof(ColorStackNode))) == NULL)

        return ERROR;

    s->next = NULL;

    s->regionID = -2;

    return OK;

}

int GetColorStackLen(ColorStackNode \*s)

{

    int length = 0;

    ColorStackNode \*temp = s->next;

    while (temp != NULL)

    {

        length++;

        temp = temp->next;

    }

    return length;

}

int ColorStackPush(ColorStackNode \*(&s), int regionID, int color)

{

    ColorStackNode \*p;

    if ((p = (ColorStackNode \*)malloc(sizeof(ColorStackNode))) == NULL)

        return ERROR;

    p->regionID = regionID;

    p->color = color;

    p->next = s->next;

    s->next = p;

    return OK;

}

int ColorStackPop(ColorStackNode \*(&s), int \*regionID)

{

    ColorStackNode \*p;

    if (s->next != NULL)

    {

        p = s->next;

        \*regionID = s->next->regionID;

        s->next = s->next->next;

        free(p);

        p = NULL;

        return OK;

    }

    else

        return ERROR;

}

**Queue.hpp**

#pragma once

#include "Defs.hpp"

typedef struct QNode

{

    int data;

    struct QNode \*next;

} QNode, \*QueuePtr;

typedef struct Queue

{

    QueuePtr front;

    QueuePtr rear;

} Queue;

int QueueInit(Queue &Q);

int QueuePush(Queue &Q, int e);

int QueueIsEmpty(Queue Q);

int QueuePop(Queue &Q, int &e);

**Queue.cpp**

#include "Queue.hpp"

int QueueInit(Queue &Q)

{

    Q.front = Q.rear = (QueuePtr)malloc(sizeof(QNode));

    if (!Q.front)

        return ERROR;

    Q.front->next = NULL;

    return OK;

}

int QueuePush(Queue &Q, int e)

{

    QueuePtr s;

    if ((s = (QueuePtr)malloc(sizeof(QNode))) == NULL)

        return ERROR;

    s->data = e;

    s->next = NULL;

    Q.rear->next = s;

    Q.rear = s;

    return OK;

}

int QueueIsEmpty(Queue Q)

{

    if (Q.front->next == NULL)

        return TRUE;

    else

        return FALSE;

}

int QueuePop(Queue &Q, int &e)

{

    QueuePtr p;

    if (Q.front == Q.rear)

        return ERROR;

    p = Q.front->next;

    e = p->data;

    Q.front->next = p->next;

    if (p == Q.rear)

        Q.rear = Q.front;

    free(p);

    p = NULL;

    return OK;

}

## 3.3主函数

**Main.hpp**

#pragma once

#include <iostream>

#include <cmath>

#include "Algorithm.hpp"

#include "PointList.hpp"

#include "Grid.hpp"

int main(void);

**Main.cpp**

#include "Main.hpp"

int main()

{

    //初始化图像

    Mat img3C = imread(IMGPATH);                 //三通道原图

    Mat emptyImg1C;                              //单通道空白图

    Mat grayImg3C;                               //三通道灰度图

    Mat img2watershed;                           //三通道图，用于分水岭算法

    Mat imgMask(img3C.size(), CV\_32S);           //分割掩码图

    Mat highlightedImg(imgMask.size(), CV\_8UC3); //高亮图

    img3C.copyTo(img2watershed);

    cvtColor(img3C, emptyImg1C, COLOR\_BGR2GRAY);

    cvtColor(emptyImg1C, grayImg3C, COLOR\_GRAY2BGR);

    emptyImg1C = Scalar::all(0);

    //初始化其他变量

    int numSampledPoints = 0; //实际生成的样本点数

    int numGridCols;          //取样网格列数

    int numGridRows;          //取样网格行数

    double numSeeds;          //理论取样点数

    int gridRadius;           //取样网格半径

    double gridScaleFactor;   //取样网格缩放因子

    //MISSION 1：随机打点+分水岭

    cout << "\n[MISSION 1]" << endl;

    while (1)

    {

        cout << "Number of seed points:";

        cin >> numSeeds;

        string num\_input\_string;

        while (cin.fail())

        {

            cin.clear();

            cin >> num\_input\_string;

            cout << "Please input a number:";

            cin >> numSeeds;

        }

        if (numSeeds != floor(numSeeds))

            cout << "Input error.(please input an integer)" << endl;

        else if (numSeeds <= 1 || numSeeds > 1200)

            cout << "Input error.(between 1~1200)" << endl;

        else

            break;

    }

    //计算取样网格参数

    gridRadius = (int)sqrt(M \* N / numSeeds);

    gridScaleFactor = gridRadius / sqrt(2);

    numGridCols = (int)(M / gridScaleFactor) + 1;

    numGridRows = (int)(N / gridScaleFactor) + 1;

    Mat grid(numGridRows, numGridCols, CV\_16SC3, Scalar(NONE, NONE, NONE)); //取样网格

    RNG rng((unsigned)time(NULL)); // C++随机数生成器

    double t1 = (double)getTickCount();

    PointListNode \*pointListNodePtr = NULL; //取样参考点链表指针

    if (!InitPointListNode(pointListNodePtr) || !GridInit(grid, gridScaleFactor, numSampledPoints, pointListNodePtr, gridRadius))

    {

        cout << "Error: run out of memory." << endl;

        exit(1);

    }

    //生成随机取样点链表

    while (GetPointListLen(pointListNodePtr))

    {

        PointListNode randPointNode; //随机的取样参考点链表

        Point newRandNearPoint;      //随机取样点

        GetRandPointListNode(pointListNodePtr, randPointNode);

        if (GetNewNearPointListNode(grid, randPointNode, newRandNearPoint, gridRadius, gridScaleFactor))

        {

            Vec2b posInGrid;

            GetPosInGrid(newRandNearPoint, posInGrid, gridScaleFactor);

            AddPoint2Grid(grid, newRandNearPoint, posInGrid, numSampledPoints);

            AddPointListNode(pointListNodePtr, newRandNearPoint, numSampledPoints);

            numSampledPoints++;

        }

        else

        {

            circle(img3C, randPointNode.point, 2, Scalar(255, 255, 255), -1);

            circle(emptyImg1C, randPointNode.point, 2, Scalar(255, 255, 255), -1);

            DelPointListNode(pointListNodePtr, randPointNode.sampleIndex);

        }

    }

    double t2 = ((double)getTickCount() - t1) / getTickFrequency();

    cout << "Number of seed points actually generated:" << numSampledPoints + 1 << endl;

    cout << "Elapsed time:" << t2 << endl;

    imshow("MISSION 1 RESULT", img3C);

    Mat watershedImg(imgMask.size(), CV\_8UC3);

    //获得图像分界轮廓

    while (1)

    {

        vector<vector<Point>> contours;

        vector<Vec4i> Hier;

        int fillColorIndexMax = 0;

        findContours(emptyImg1C, contours, Hier, RETR\_CCOMP, CHAIN\_APPROX\_SIMPLE);

        if (contours.empty())

            continue;

        imgMask = Scalar::all(0);

        for (int contourIndex = 0; contourIndex >= 0; contourIndex = Hier[contourIndex][0], fillColorIndexMax++)

            drawContours(imgMask, contours, contourIndex, Scalar::all((double)fillColorIndexMax + 1), -1, 8, Hier, INT\_MAX);

        if (fillColorIndexMax == 0)

            continue;

        vector<Vec3b> colorTable;

        for (int i = 0; i < fillColorIndexMax; i++)

        {

            uchar b = theRNG().uniform(0, 255);

            uchar g = theRNG().uniform(0, 255);

            uchar r = theRNG().uniform(0, 255);

            colorTable.push\_back(Vec3b((uchar)b, (uchar)g, (uchar)r));

        }

        watershed(img2watershed, imgMask);

        for (int i = 0; i < imgMask.rows; i++)

            for (int j = 0; j < imgMask.cols; j++)

            {

                int fillColorIndex = imgMask.at<int>(i, j);

                if (fillColorIndex == -1)

                    watershedImg.at<Vec3b>(i, j) = Vec3b(255, 255, 255); //轮廓着白色

                else if (fillColorIndex <= 0 || fillColorIndex > fillColorIndexMax)

                    watershedImg.at<Vec3b>(i, j) = Vec3b(0, 0, 0); //啥玩意

                else

                    watershedImg.at<Vec3b>(i, j) = colorTable[(double)fillColorIndex - 1]; //区域着色

            }

        watershedImg = watershedImg \* 0.5 + grayImg3C \* 0.5;

        imshow("watershed RESULT", watershedImg);

#ifdef STEP3

        cout << "\nClick [MISSION 1 RESULT], press enter to get in MISSION 2." << endl;

#endif

        cout << "\nClick [MISSION 1 RESULT], press enter to get in MISSION 3." << endl;

        waitKey(0);

        break;

    }

    //MISSION 2：四原色填充

    AdjacentTable adjacentTable; //邻接表

#ifdef STEP3

    cout << "\n[MISSION 2]" << endl;

#endif

    //生成邻接表

    t1 = (double)getTickCount();

    GenerateAdjacentTable(adjacentTable, imgMask, numSampledPoints);

#ifdef STEP3

    //展示邻接表

    //ShowAdjacencyList(adjacentTable);

    Queue pathQueue; //生成路径队列

    pathQueue.front = NULL;

    pathQueue.rear = NULL;

    ChoosePath(adjacentTable, pathQueue); //生成生成路径队列

    t2 = ((double)getTickCount() - t1) / getTickFrequency();

    cout << "Elapsed time:" << t2 << endl;

    //生成四原色图

    GetFourColorsImg(adjacentTable, pathQueue, imgMask, highlightedImg);

    cout << "\nClick [MISSION 1 RESULT], press enter to get in MISSION 2." << endl;

    waitKey(0);

#endif

    //MISSION 3：折半查找

    cout << "\n[MISSION 3]" << endl;

    //按区域面积大小堆排序

    t1 = (double)getTickCount();

    HeapNode heap[MAX\_RIGION\_NUM + 1];

    HeapInit(heap);

    CaluHeapArea(imgMask, heap);

    HeapSortAsc(heap, numSampledPoints);

    t2 = ((double)getTickCount() - t1) / getTickFrequency();

    watershedImg = (watershedImg - grayImg3C \* 0.5) \* 2;

    while (1)

    {

        if (BinarySearch(heap, numSampledPoints, imgMask, watershedImg, adjacentTable, t2) == EXIT)

            break;

    }

    cout << "\nEnd" << endl;

    return 0;

}

## 3.4问题一：随即打点+分水岭算法

**Grid.hpp**

#pragma once

#include "Defs.hpp"

#include "PointList.hpp"

int GridInit(Mat &Wgarray, double stepk, int &samnumsjud, PointListNode \*acts, int r);

void GetPosInGrid(Point pointer1, Vec2b &ceposi, double stepk);

int GetNearGrid(Mat Wgarray, Vec2b ceposi, Mat &nearGrid);

void AddPoint2Grid(Mat &Wgarray, Point result\_pt, Vec2b ceposi, int sampleIndex);

**Grid.cpp**

#include "Grid.hpp"

int GridInit(Mat &grid, double gridScaleFactor, int &numSampledPoints, PointListNode \*pointListNodePtr, int gridRadius)

{

    Point originPoint(ORIGINPOINT);

    Vec2b posInGrid;

    GetPosInGrid(originPoint, posInGrid, gridScaleFactor);

    short \*rowInGrid = grid.ptr<short>(posInGrid[1]);

    rowInGrid[posInGrid[0] \* grid.channels()] = originPoint.x;

    rowInGrid[posInGrid[0] \* grid.channels() + 1] = originPoint.y;

    rowInGrid[posInGrid[0] \* grid.channels() + 2] = 0;

    numSampledPoints += 1;

    AddPointListNode(pointListNodePtr, originPoint, 0);

    return OK;

}

void GetPosInGrid(Point point, Vec2b &posInGrid, double gridScaleFactor)

{

    posInGrid[0] = (int)(point.x / gridScaleFactor);

    posInGrid[1] = (int)(point.y / gridScaleFactor);

}

int GetNearGrid(Mat grid, Vec2b posInGrid, Mat &nearGrid)

{

    int j = 0;

    int offsetInGrid[25][2]{{-2, -2}, {-1, -2}, {0, -2},

        {1, -2}, {2, -2}, {-2, -1}, {-1, -1}, {0, -1},

        {1, -1}, {2, -1}, {-2, 0}, {-1, 0}, {0, 0},

        {1, 0}, {2, 0}, {-2, 1}, {-1, 1}, {0, 1},

        {1, 1}, {2, 1}, {-2, 2}, {-1, 2}, {0, 2},

        {1, 2}, {2, 2}};

    for (int i = 0; i < 25; i++)

    {

        if ((posInGrid[0] + offsetInGrid[i][0]) > -3 && (posInGrid[0] + offsetInGrid[i][0]) < grid.cols \* grid.channels() && (posInGrid[1] + offsetInGrid[i][1] >= 0 && (posInGrid[1] + offsetInGrid[i][1]) < grid.rows))

        {

            short \*rowInGrid = grid.ptr<short>(posInGrid[1] + offsetInGrid[i][1]);

            if (i % 5 == 0 && i != 0)

                j++;

            short \*rowInNearGrid = nearGrid.ptr<short>(j);

            rowInNearGrid[(i % 5) \* nearGrid.channels()] = rowInGrid[(posInGrid[0] + offsetInGrid[i][0]) \* grid.channels()];

            rowInNearGrid[(i % 5) \* nearGrid.channels() + 1] = rowInGrid[(posInGrid[0] + offsetInGrid[i][0]) \* grid.channels() + 1];

            rowInNearGrid[(i % 5) \* nearGrid.channels() + 2] = rowInGrid[(posInGrid[0] + offsetInGrid[i][0]) \* grid.channels() + 2];

        }

        else

            continue;

    }

    return OK;

}

void AddPoint2Grid(Mat &grid, Point newRandNearPoint, Vec2b posInGrid, int sampleIndex)

{

    short \*rowInGrid = grid.ptr<short>(posInGrid[1]);

    rowInGrid[posInGrid[0] \* grid.channels()] = newRandNearPoint.x;

    rowInGrid[posInGrid[0] \* grid.channels() + 1] = newRandNearPoint.y;

    rowInGrid[posInGrid[0] \* grid.channels() + 2] = sampleIndex;

}

**PointList.hpp**

#pragma once

#include "Defs.hpp"

typedef struct PointListNode

{

    Point point;

    int sampleIndex;

    struct PointListNode \*next;

} PointListNode;

int InitPointListNode(PointListNode \*&acts);

int AddPointListNode(PointListNode \*p, Point pointer1, int sampleIndex);

int DelPointListNode(PointListNode \*p, int sampleIndex);

int GetNewNearPointListNode(Mat Wgarray, PointListNode ref\_pt, Point &result\_pt, int r, double stepk);

int JudgeNearPoint(Point pointer1, int r, double stepk, Mat Wgarray);

int GetPointListLen(PointListNode \*p);

void GetRandPointListNode(PointListNode \*acts, PointListNode &ref\_pt);

**PointList.cpp**

#include "PointList.hpp"

#include "Grid.hpp"

int InitPointListNode(PointListNode \*&pointListNodePtr)

{

    if ((pointListNodePtr = (PointListNode \*)malloc(sizeof(PointListNode))) == NULL)

        return ERROR;

    pointListNodePtr->next = NULL;

    return OK;

}

int AddPointListNode(PointListNode \*p, Point newPoint, int sampleIndex)

{

    PointListNode \*temp = NULL;

    while (p->next != NULL)

        p = p->next;

    if ((temp = (PointListNode \*)malloc(sizeof(PointListNode))) == NULL)

        return ERROR;

    p->next = temp;

    temp->next = NULL;

    temp->point = newPoint;

    temp->sampleIndex = sampleIndex;

    return OK;

}

int DelPointListNode(PointListNode \*p, int sampleIndex)

{

    PointListNode \*temp = p->next;

    if (p->next == NULL)

        return ERROR;

    while (temp->next != NULL && temp->sampleIndex != sampleIndex)

    {

        p = temp;

        temp = temp->next;

    }

    if (temp->sampleIndex == sampleIndex)

    {

        p->next = temp->next;

        free(temp);

        temp = NULL;

    }

    return OK;

}

int GetNewNearPointListNode(Mat grid, PointListNode randPointNode, Point &newRandNearPoint, int gridRadius, double gridScaleFactor)

{

    int i = 0;

    double randAngle = 0, randRadius;

    RNG rng((unsigned)time(NULL));

    Point nearPoint;

    while (i < GetNewNearPointListNodeIterateLimit)

    {

        randAngle = rng.uniform((double)0, (double)(2 \* PI));

        randRadius = (double)gridRadius + 1;

        nearPoint.x = randPointNode.point.x + (int)(randRadius \* cos(randAngle));

        nearPoint.y = randPointNode.point.y + (int)(randRadius \* sin(randAngle));

        if (nearPoint.x < 0 || nearPoint.x > M || nearPoint.y < 0 || nearPoint.y > N)

            continue;

        if (JudgeNearPoint(nearPoint, gridRadius, gridScaleFactor, grid))

        {

            newRandNearPoint.x = nearPoint.x;

            newRandNearPoint.y = nearPoint.y;

            return OK;

        }

        i++;

    }

    return ERROR;

}

int JudgeNearPoint(Point nearPoint, int gridRadius, double gridScaleFactor, Mat grid)

{

    Mat nearGrid(5, 5, CV\_16SC3, Scalar(NONE, NONE, NONE));

    Vec2b posInGrid;

    GetPosInGrid(nearPoint, posInGrid, gridScaleFactor);

    GetNearGrid(grid, posInGrid, nearGrid);

    for (int i = 0; i < nearGrid.rows; i++)

    {

        for (int j = 0; j < nearGrid.cols; j++)

        {

            short \*rowInNearGrid = nearGrid.ptr<short>(i);

            if (rowInNearGrid[j \* nearGrid.channels()] == NONE && rowInNearGrid[j \* nearGrid.channels() + 1] == NONE && rowInNearGrid[j \* nearGrid.channels() + 2] == NONE)

                continue;

            double distanceSquared = pow((rowInNearGrid[j \* nearGrid.channels()] - nearPoint.x), 2) + pow((rowInNearGrid[j \* nearGrid.channels() + 1] - nearPoint.y), 2);

            if (distanceSquared < pow(gridRadius, 2))

                return FALSE;

        }

    }

    return TRUE;

}

int GetPointListLen(PointListNode \*p)

{

    int length = 0;

    while (p->next != NULL)

    {

        length++;

        p = p->next;

    }

    return length;

}

void GetRandPointListNode(PointListNode \*pointListNodePtr, PointListNode &randPointNode)

{

    RNG rng((unsigned)time(NULL));

    int Rinds = rng.uniform(0, GetPointListLen(pointListNodePtr));

    for (int i = 0; i < Rinds; i++)

        pointListNodePtr = pointListNodePtr->next;

    randPointNode.point = pointListNodePtr->next->point;

    randPointNode.sampleIndex = pointListNodePtr->next->sampleIndex;

}

## 3.5问题二：四原色

**AdjacentTable.hpp**

#pragma once

#include "Defs.hpp"

#include "Queue.hpp"

typedef struct AdjacentListNode

{

    int regionID;

    int numAdjacentRegions;

    struct AdjacentListNode \*next;

} AdjacentListNode;

typedef struct Region

{

    int regionID;

    Point centerPoint;

    AdjacentListNode \*adjacentListHead;

} Region, RegionArray[MAX\_RIGION\_NUM];

typedef struct AdjacentTable

{

    RegionArray regions;

    int numRegions;

} AdjacentTable;

int ChoosePath(AdjacentTable allim, Queue &tinting\_queue);

int IsAllVisited(AdjacentTable allim, const bool \*visited);

void ShowAdjacencyList(AdjacentTable allim);

void CaluCenterPoint(AdjacentTable &allim, Mat maskImage, Mat fourColorsImg);              //

int GetNumUnvisitedAdjacentRegion(AdjacentTable allim, int regionID, const bool \*visited); //

int GenerateAdjacentTable(AdjacentTable &allim, Mat maskImage, int kcount);                //

int SortAdjacentLists(AdjacentTable &allim);

int GetRegionIDWithMaxUnvisitedAdjacentRegion(AdjacentTable allim, const bool \*visited);

**AdjacentTable.cpp**

#include "AdjacentTable.hpp"

int ChoosePath(AdjacentTable adjacentTable, Queue &pathQueue)

{

    Queue tempQueue;

    bool visited[MAX\_RIGION\_NUM] = {FALSE};

    int tempInt, thisRegionID;

    for (int i = 0; i < adjacentTable.numRegions; i++)

    {

        AdjacentListNode \*temp = adjacentTable.regions[i].adjacentListHead;

        if (temp == NULL)

            visited[i] = TRUE;

    }

    QueueInit(pathQueue);

    QueueInit(tempQueue);

    while (!IsAllVisited(adjacentTable, visited))

    {

        thisRegionID = GetRegionIDWithMaxUnvisitedAdjacentRegion(adjacentTable, visited);

        if (!visited[thisRegionID])

        {

            visited[thisRegionID] = TRUE;

            QueuePush(pathQueue, thisRegionID + 1);

            QueuePush(tempQueue, thisRegionID);

            while (!QueueIsEmpty(tempQueue))

            {

                QueuePop(tempQueue, tempInt);

                AdjacentListNode \*temp = adjacentTable.regions[tempInt].adjacentListHead;

                if (temp == NULL)

                    break;

                while (temp != NULL)

                {

                    if (!visited[(temp->regionID) - 1])

                    {

                        visited[(temp->regionID) - 1] = TRUE;

                        QueuePush(pathQueue, temp->regionID);

                        QueuePush(tempQueue, (temp->regionID) - 1);

                    }

                    temp = temp->next;

                }

            }

        }

    }

    //输出路线

    cout << "\nThe path is:" << endl;

    Queue temp = pathQueue;

    temp.front = temp.front->next;

    while (temp.front != temp.rear)

    {

        cout << temp.front->data << "->";

        temp.front = temp.front->next;

    }

    cout << temp.front->data << endl;

    return OK;

}

int IsAllVisited(AdjacentTable adjacentTable, const bool \*visited)

{

    for (int i = 0; i < adjacentTable.numRegions; i++)

    {

        if (visited[i] == FALSE)

            return FALSE;

    }

    return TRUE;

}

void ShowAdjacencyList(AdjacentTable adjacentTable)

{

    cout << "[Adjacent List]" << endl;

    for (int i = 0; i < adjacentTable.numRegions; i++)

    {

        AdjacentListNode \*temp = adjacentTable.regions[i].adjacentListHead;

        if (temp == NULL)

            continue;

        cout << adjacentTable.regions[i].regionID << "-";

        while (temp != NULL && temp->next != NULL)

        {

            cout << temp->regionID << "-";

            temp = temp->next;

        }

        if (temp != NULL && temp->next == NULL)

            cout << temp->regionID << endl;

    }

}

void CaluCenterPoint(AdjacentTable &adjacentTable, Mat imgMask, Mat fourColorsImg)

{

    for (int i = 0; i < adjacentTable.numRegions; i++)

    {

        int centerX = 0, centerY = 0, numPoints = 0;

        for (int j = 1; j < imgMask.rows - 1; j++)

        {

            int \*rowInImgMask = imgMask.ptr<int>(j);

            for (int k = 1; k < imgMask.cols - 1; k++)

            {

                if (rowInImgMask[k] == i + 1)

                {

                    centerX += k;

                    centerY += j;

                    numPoints++;

                }

            }

        }

        if (!numPoints)

            numPoints = 1;

        adjacentTable.regions[i].centerPoint.x = centerX / numPoints;

        adjacentTable.regions[i].centerPoint.y = centerY / numPoints;

    }

    for (int i = 0; i < adjacentTable.numRegions; i++)

    {

        char str[10];

        sprintf\_s(str, "%d", i + 1);

        putText(fourColorsImg, str, adjacentTable.regions[i].centerPoint, FONT\_HERSHEY\_PLAIN, 0.8, Scalar(255, 255, 255));

    }

}

int GetNumUnvisitedAdjacentRegion(AdjacentTable adjacentTable, int regionID, const bool \*visited)

{

    int length = 0;

    if (visited[adjacentTable.regions[regionID - 1].regionID] == FALSE)

        length++;

    AdjacentListNode \*temp = adjacentTable.regions[regionID - 1].adjacentListHead;

    if (temp == NULL)

        return length;

    while (temp != NULL)

    {

        if (visited[temp->regionID - 1] == FALSE)

            length++;

        temp = temp->next;

    }

    return length;

}

//分配各区域颜色

int GetRegionIDWithMaxUnvisitedAdjacentRegion(AdjacentTable adjacentTable, const bool \*visited)

{

    int maxRegionID = NONE, maxNumUnvisited = 0, numUnvisited;

    for (int i = 0; i < adjacentTable.numRegions; i++)

    {

        if (visited[i] == FALSE)

        {

            numUnvisited = GetNumUnvisitedAdjacentRegion(adjacentTable, i + 1, visited);

            if (numUnvisited > maxNumUnvisited)

            {

                maxRegionID = i;

                maxNumUnvisited = numUnvisited;

            }

        }

    }

    if (maxRegionID == NONE)

        return ERROR;

    else

        return maxRegionID;

}

int GenerateAdjacentTable(AdjacentTable &adjacentTable, Mat imgMask, int numSampledPoints)

{

    int regionID1, regionID2, edge[8], ref;

    adjacentTable.numRegions = numSampledPoints;

    for (int i = 0; i < numSampledPoints; i++) //分配区域编号

    {

        adjacentTable.regions[i].regionID = i + 1;

        adjacentTable.regions[i].adjacentListHead = NULL;

    }

    for (int i = 1; i < imgMask.rows - 1; i++)

    {

        int \*rowInImgMask1 = imgMask.ptr<int>(i - 1);

        int \*rowInImgMask2 = imgMask.ptr<int>(i);

        int \*rowInImgMask3 = imgMask.ptr<int>(i + 1);

        for (int j = 1; j < imgMask.cols - 1; j++)

        {

            ref = rowInImgMask2[j];

            edge[0] = rowInImgMask1[j - 1];

            edge[1] = rowInImgMask1[j + 1];

            edge[2] = rowInImgMask1[j + 1];

            edge[3] = rowInImgMask2[j - 1];

            edge[4] = rowInImgMask2[j + 1];

            edge[5] = rowInImgMask3[j - 1];

            edge[6] = rowInImgMask3[j];

            edge[7] = rowInImgMask3[j + 1];

            if (ref == -1)

            {

                for (int k = 0; k < 8; k++)

                {

                    if (edge[k] > 0)

                    {

                        regionID1 = edge[k];

                        break;

                    }

                }

                for (int k = 0; k < 8; k++)

                {

                    if (edge[k] > 0 && edge[k] != regionID1)

                    {

                        regionID2 = edge[k];

                        break;

                    }

                }

                if (adjacentTable.regions[regionID1 - 1].adjacentListHead == NULL)

                {

                    AdjacentListNode \*tempNode = NULL;

                    if ((tempNode = (AdjacentListNode \*)malloc(sizeof(AdjacentListNode))) == NULL)

                    {

                        cout << "Error: insufficient memory..." << endl;

                        exit(1);

                    }

                    tempNode->next = NULL;

                    tempNode->regionID = regionID2;

                    adjacentTable.regions[regionID1 - 1].adjacentListHead = tempNode;

                }

                else

                {

                    int isDuplicate = FALSE;

                    AdjacentListNode \*temp = adjacentTable.regions[regionID1 - 1].adjacentListHead;

                    while (temp != NULL)

                    {

                        if (temp->regionID != regionID2 && temp->next != NULL)

                            temp = temp->next;

                        else if (temp->regionID == regionID2 || regionID2 == adjacentTable.regions[regionID1 - 1].regionID)

                        {

                            isDuplicate = TRUE;

                            break;

                        }

                        else

                            break;

                    }

                    if (!isDuplicate)

                    {

                        AdjacentListNode \*tempNode = NULL;

                        if ((tempNode = (AdjacentListNode \*)malloc(sizeof(AdjacentListNode))) == NULL)

                        {

                            cout << "Error: insufficient memory..." << endl;

                            exit(1);

                        }

                        tempNode->regionID = regionID2;

                        tempNode->next = NULL;

                        temp->next = tempNode;

                    }

                }

                if (adjacentTable.regions[regionID2 - 1].adjacentListHead == NULL)

                {

                    AdjacentListNode \*tempNode = NULL;

                    if ((tempNode = (AdjacentListNode \*)malloc(sizeof(AdjacentListNode))) == NULL)

                    {

                        cout << "Error: insufficient memory..." << endl;

                        exit(1);

                    }

                    tempNode->next = NULL;

                    tempNode->regionID = regionID1;

                    adjacentTable.regions[regionID2 - 1].adjacentListHead = tempNode;

                }

                else

                {

                    int isDuplicate = FALSE;

                    AdjacentListNode \*temp = adjacentTable.regions[regionID2 - 1].adjacentListHead;

                    while (temp != NULL)

                    {

                        if (temp->regionID != regionID1 && temp->next != NULL)

                            temp = temp->next;

                        else if (temp->regionID == regionID1 || regionID1 == adjacentTable.regions[regionID2 - 1].regionID)

                        {

                            isDuplicate = TRUE;

                            break;

                        }

                        else

                            break;

                    }

                    if (!isDuplicate)

                    {

                        AdjacentListNode \*tempNode = NULL;

                        if ((tempNode = (AdjacentListNode \*)malloc(sizeof(AdjacentListNode))) == NULL)

                        {

                            cout << "Error: insufficient memory..." << endl;

                            exit(1);

                        }

                        tempNode->regionID = regionID1;

                        tempNode->next = NULL;

                        temp->next = tempNode;

                    }

                }

            }

        }

    }

    SortAdjacentLists(adjacentTable);

    return OK;

}

int SortAdjacentLists(AdjacentTable &adjacentTable)

{

    int length = 1;

    int adjacentListsLen[MAX\_RIGION\_NUM] = {0};

    for (int i = 0; i < adjacentTable.numRegions; i++)

    {

        AdjacentListNode \*temp = adjacentTable.regions[i].adjacentListHead;

        while (temp != NULL)

        {

            length++;

            temp = temp->next;

        }

        adjacentListsLen[i] = length;

        length = 1;

    }

    for (int i = 0; i < adjacentTable.numRegions; i++)

    {

        AdjacentListNode \*temp = adjacentTable.regions[i].adjacentListHead;

        while (temp != NULL)

        {

            temp->numAdjacentRegions = adjacentListsLen[temp->regionID - 1];

            temp = temp->next;

        }

    }

    for (int i = 0; i < adjacentTable.numRegions; i++)

    {

        for (int j = 0; j < adjacentListsLen[i] - 2; j++)

        {

            AdjacentListNode \*temp = adjacentTable.regions[i].adjacentListHead;

            AdjacentListNode \*shadow = NULL;

            for (int k = 0; k < adjacentListsLen[i] - j - 2; k++)

            {

                if (temp->numAdjacentRegions < temp->next->numAdjacentRegions)

                {

                    if (adjacentTable.regions[i].adjacentListHead == temp)

                    {

                        AdjacentListNode \*current = temp->next;

                        temp->next = temp->next->next;

                        current->next = temp;

                        adjacentTable.regions[i].adjacentListHead = current;

                        shadow = current;

                    }

                    else

                    {

                        AdjacentListNode \*current = temp->next;

                        temp->next = temp->next->next;

                        current->next = temp;

                        shadow->next = current;

                        shadow = current;

                    }

                }

                else

                {

                    shadow = temp;

                    temp = temp->next;

                }

            }

        }

    }

    return OK;

}

**Algorithm.hpp**

#pragma once

#include "Defs.hpp"

#include "ColorStack.hpp"

#include "AdjacentTable.hpp"

#include "Heap.hpp"

#include "Queue.hpp"

int GetFourColorsImg(AdjacentTable &allim, Queue tinting\_queue, Mat maskImage, Mat &hlightd);

**Algorithm.cpp**

#include "Algorithm.hpp"

int GetFourColorsImg(AdjacentTable &adjacentTable, Queue pathQueue, Mat imgMask, Mat &highlightedImg)

{

    int popRecords[MAX\_RIGION\_NUM] = {0};

    bool colorUsedFlags[MAX\_RIGION\_NUM][4] = {0};

    Queue bufferQueue;

    Queue transferStationQueue;

    QueueInit(bufferQueue);

    QueueInit(transferStationQueue);

    ColorStackNode \*colorStack = NULL;

    ColorStackInit(colorStack);

    int thisRegionID;

    while (!QueueIsEmpty(pathQueue) || !QueueIsEmpty(bufferQueue))

    {

        for (int i = 0; i < adjacentTable.numRegions; i++)

        {

            if (popRecords[i] >= POP\_RECORDS\_LIMIT)

            {

                popRecords[i] = 0;

                for (int j = 0; j < 20; j++)

                {

                    int temp\_area\_number;

                    ColorStackPop(colorStack, &temp\_area\_number);

                    QueuePush(transferStationQueue, temp\_area\_number);

                    while (!QueueIsEmpty(bufferQueue))

                    {

                        int transfer;

                        QueuePop(bufferQueue, transfer);

                        QueuePush(transferStationQueue, transfer);

                    }

                    while (!QueueIsEmpty(transferStationQueue))

                    {

                        int transfer;

                        QueuePop(transferStationQueue, transfer);

                        QueuePush(bufferQueue, transfer);

                    }

                }

            }

        }

        if (!QueueIsEmpty(bufferQueue))

        {

            QueuePop(bufferQueue, thisRegionID);

            bool left\_colors\_table[4] = {0};

            if (GetColorStackLen(colorStack))

            {

                ColorStackNode \*search\_stack = colorStack->next;

                while (search\_stack != NULL)

                {

                    AdjacentListNode \*temp = adjacentTable.regions[thisRegionID - 1].adjacentListHead;

                    while (temp != NULL)

                    {

                        if (temp->regionID == search\_stack->regionID)

                            left\_colors\_table[search\_stack->color] = 1;

                        temp = temp->next;

                    }

                    search\_stack = search\_stack->next;

                }

                int tinted = FALSE;

                for (int i = 0; i < 4; i++)

                {

                    if (left\_colors\_table[i] != 1 && colorUsedFlags[thisRegionID - 1][i] == 0)

                    {

                        ColorStackPush(colorStack, thisRegionID, i);

                        colorUsedFlags[thisRegionID - 1][i] = 1;

                        tinted = TRUE;

                        break;

                    }

                }

                if (tinted == FALSE)

                {

                    for (int i = 0; i < 4; i++)

                        colorUsedFlags[thisRegionID - 1][i] = 0;

                    int temp\_area\_number;

                    ColorStackPop(colorStack, &temp\_area\_number);

                    popRecords[temp\_area\_number - 1]++;

                    QueuePush(transferStationQueue, temp\_area\_number);

                    QueuePush(transferStationQueue, thisRegionID);

                    while (!QueueIsEmpty(bufferQueue))

                    {

                        int transfer;

                        QueuePop(bufferQueue, transfer);

                        QueuePush(transferStationQueue, transfer);

                    }

                    while (!QueueIsEmpty(transferStationQueue))

                    {

                        int transfer;

                        QueuePop(transferStationQueue, transfer);

                        QueuePush(bufferQueue, transfer);

                    }

                }

            }

            else

            {

                int flag = FALSE;

                for (int i = 0; i < 4; i++)

                {

                    if (colorUsedFlags[thisRegionID - 1][i] != 1)

                    {

                        ColorStackPush(colorStack, thisRegionID, i);

                        colorUsedFlags[thisRegionID - 1][i] = 1;

                        flag = TRUE;

                        break;

                    }

                }

                if (flag == FALSE)

                    cout << "Coloring failure!" << endl;

            }

        }

        else if (QueueIsEmpty(bufferQueue))

        {

            QueuePop(pathQueue, thisRegionID);

            bool left\_colors\_table[4] = {0};

            if (GetColorStackLen(colorStack))

            {

                ColorStackNode \*search\_stack = colorStack->next;

                while (search\_stack != NULL)

                {

                    AdjacentListNode \*temp = adjacentTable.regions[thisRegionID - 1].adjacentListHead;

                    while (temp != NULL)

                    {

                        if (temp->regionID == search\_stack->regionID)

                            left\_colors\_table[search\_stack->color] = 1;

                        temp = temp->next;

                    }

                    search\_stack = search\_stack->next;

                }

                int tinted = FALSE;

                for (int i = 0; i < 4; i++)

                {

                    if (left\_colors\_table[i] != 1)

                    {

                        ColorStackPush(colorStack, thisRegionID, i);

                        colorUsedFlags[thisRegionID - 1][i] = 1;

                        tinted = TRUE;

                        break;

                    }

                }

                if (tinted == FALSE)

                {

                    int temp\_area\_number;

                    ColorStackPop(colorStack, &temp\_area\_number);

                    popRecords[temp\_area\_number]++;

                    QueuePush(bufferQueue, temp\_area\_number);

                    QueuePush(bufferQueue, thisRegionID);

                }

            }

            else

            {

                ColorStackPush(colorStack, thisRegionID, RED);

                colorUsedFlags[thisRegionID - 1][RED] = 1;

            }

        }

    }

    Mat img3C = imread(IMGPATH), grayImg3C;

    cvtColor(img3C, grayImg3C, COLOR\_BGR2GRAY);

    cvtColor(grayImg3C, grayImg3C, COLOR\_GRAY2BGR);

    Mat fourColorsImg(imgMask.size(), CV\_8UC3);

    for (int i = 0; i < imgMask.rows; i++)

        for (int j = 0; j < imgMask.cols; j++)

        {

            int regionID = imgMask.at<int>(i, j);

            ColorStackNode \*temp = colorStack->next;

            if (regionID > 0)

            {

                while (temp != NULL)

                {

                    if (temp->regionID == regionID)

                    {

                        switch (temp->color)

                        {

                        case RED:

                            fourColorsImg.at<Vec3b>(i, j) = Vec3b(0, 0, 255);

                            highlightedImg.at<Vec3b>(i, j) = Vec3b(0, 0, 255);

                            break;

                        case GREEN:

                            fourColorsImg.at<Vec3b>(i, j) = Vec3b(0, 255, 0);

                            highlightedImg.at<Vec3b>(i, j) = Vec3b(0, 255, 0);

                            break;

                        case BLUE:

                            fourColorsImg.at<Vec3b>(i, j) = Vec3b(255, 0, 0);

                            highlightedImg.at<Vec3b>(i, j) = Vec3b(255, 0, 0);

                            break;

                        case YELLOW:

                            fourColorsImg.at<Vec3b>(i, j) = Vec3b(0, 255, 255);

                            highlightedImg.at<Vec3b>(i, j) = Vec3b(0, 255, 255);

                            break;

                        }

                    }

                    temp = temp->next;

                }

            }

            else if (regionID == -1)

            {

                fourColorsImg.at<Vec3b>(i, j) = Vec3b(255, 255, 255);

                highlightedImg.at<Vec3b>(i, j) = Vec3b(255, 255, 255);

            }

        }

    fourColorsImg = fourColorsImg \* 0.5 + grayImg3C \* 0.5;

    CaluCenterPoint(adjacentTable, imgMask, fourColorsImg);

    imshow("[four primary colour]", fourColorsImg);

    return OK;

}

## 3.6问题三：折半查找

**Algorithm.hpp**

#pragma once

#include "Defs.hpp"

#include "ColorStack.hpp"

#include "AdjacentTable.hpp"

#include "Heap.hpp"

#include "Queue.hpp"

int BinarySearch(HeapNode \*heap, int kcount, Mat maskImage, Mat hlightd, AdjacentTable allim, double t0);

void ShowHighLightedImg(HeapNode \*heap, unsigned int lowerIndex, unsigned int upperIndex, Mat maskImage, Mat hlightd, AdjacentTable allim);

**Algorithm.cpp**

void ShowHighLightedImg(HeapNode \*heap, unsigned int lowerIndex, unsigned int upperIndex, Mat imgMask, Mat highlightedImg, AdjacentTable adjacentTable)

{

    Mat img3C = imread(IMGPATH), grayImg3C, highlightedImgCopy;

    cvtColor(img3C, grayImg3C, COLOR\_BGR2GRAY);

    cvtColor(grayImg3C, grayImg3C, COLOR\_GRAY2BGR);

    highlightedImg.copyTo(highlightedImgCopy);

    int index, is\_target = FALSE;

    char str[10];

    //查询区域面积是否在范围内，在则涂白，不在则涂黑（gaoguang1\_copy）

    for (int i = 0; i < imgMask.rows; i++)

        for (int j = 0; j < imgMask.cols; j++)

        {

            index = imgMask.at<int>(i, j);

            if (index > 0)

            {

                is\_target = FALSE;

                for (int k = lowerIndex; k <= upperIndex; k++)

                {

                    if (index == heap[k].id)

                    {

                        is\_target = TRUE;

                        break;

                    }

                }

                if (!is\_target)

                    highlightedImgCopy.at<Vec3b>(i, j) = Vec3b(0, 0, 0);

            }

            else if (index == -1)

                highlightedImgCopy.at<Vec3b>(i, j) = Vec3b(255, 255, 255);

        }

    highlightedImgCopy = highlightedImgCopy \* 0.5 + grayImg3C \* 0.5;

    //输出区域编号

    for (int i = lowerIndex; i <= upperIndex; i++)

    {

        sprintf\_s(str, "%d", heap[i].id);

        putText(highlightedImgCopy, str, adjacentTable.regions[heap[i].id - 1].centerPoint, FONT\_HERSHEY\_PLAIN, 0.8, Scalar(255, 255, 255));

    }

    imshow("Binary Search", highlightedImgCopy);

}

int BinarySearch(HeapNode \*heap, int numAreas, Mat imgMask, Mat highlightedImg, AdjacentTable adjacentTable,double t0)

{

    unsigned int lowerBound, upperBound, lowerIndex = 0, upperIndex = 0;

    unsigned int low = 1, high = numAreas, mid;

    int temp, count = 0;

    //输入

    while (1)

    {

        cout << "the area range:" << heap[1].area << "~" << heap[numAreas].area << endl;

        cout << "lower bound(-1 for quit):";

        cin >> temp;

        string num\_input\_string;

        while (cin.fail() || (temp < (int)heap[1].area && temp!=-1))

        {

            if (cin.fail())

            {

                cin.clear();

                cin >> num\_input\_string;

                cout << "Please input a number:";

            }

            else

            {

                if(temp < 0)

                    cout << "Please input a positive number." << endl;

                else

                    cout << "the lower bound must >=" << heap[1].area << "." << endl;

                cout << "lower bound(-1 for quit):";

            }

            cin >> temp;

        }

        if (temp == -1)

            return EXIT;

        lowerBound = (unsigned int)temp;

        cout << "upper bound(-1 for quit):";

        cin >> temp;

        while (cin.fail() || ((temp <= (int)lowerBound || temp > (int)heap[numAreas].area) && temp!=-1))

        {

            if (cin.fail())

            {

                cin.clear();

                cin >> num\_input\_string;

                cout << "Please input a number:";

            }

            else

            {

                cout << "the upper bound must >=" << lowerBound << " and <=" << heap[numAreas].area << "." << endl;

                cout << "upper bound(-1 for quit):";

            }

            cin >> temp;

        }

        if (temp == -1)

            return EXIT;

        upperBound = (unsigned int)temp;

        break;

    }

    //查找下界

    while (low <= high)

    {

        mid = (low + high) / 2;

        if (heap[mid].area == lowerBound)

        {

            while (mid)

            {

                if (heap[mid - 1].area == lowerBound)

                    mid--;

                else

                    break;

            }

            lowerIndex = mid;

            break;

        }

        else if (heap[mid].area < lowerBound)

        {

            low = mid + 1;

            lowerIndex = low;

        }

        else

            high = mid - 1;

    }

    //查找上界

    high = numAreas;

    while (low <= high)

    {

        mid = (low + high) / 2;

        if (heap[mid].area == upperBound)

        {

            while (mid<numAreas-1)

            {

                if (heap[mid + 1].area == upperBound)

                    mid++;

                else

                    break;

            }

            upperIndex = mid;

            break;

        }

        else if (heap[mid].area < upperBound)

            low = mid + 1;

        else

        {

            high = mid - 1;

            upperIndex = high;

        }

    }

    if (lowerIndex > upperIndex)

        cout << "No region area between" << lowerBound << "~" << upperBound << "." << endl;

    else

    {

        count = 0;

        cout << "regions the area between" << lowerBound << "~" << upperBound << "are:" << endl;

        for (unsigned int i = lowerIndex; i <= upperIndex; i++)

        {

            cout << heap[i].id << ": " << heap[i].area << endl;

            count++;

        }

        cout << count << " areas in total." << endl;

        cout << "Elapsed time:" << t0 << endl;

    }

    ShowHighLightedImg(heap, lowerIndex, upperIndex, imgMask, highlightedImg, adjacentTable);

    cout << "\nClick [MISSION 1 RESULT] for search again." << endl;

    waitKey(0);

    return OK;

}

**Heap.hpp**

#pragma once

#include "Defs.hpp"

typedef struct HeapNode

{

    unsigned int area;

    int id;

} HeapNode;

int HeapInit(HeapNode \*heap);

void CaluHeapArea(Mat maskImage, HeapNode \*heap);

void HeapSortAsc(HeapNode \*heap, unsigned int heapSortLimit);

void MaximizeHeap(HeapNode \*heap, unsigned int heapSortLimit);

void MaximizeHeapOnce(HeapNode \*heap, int specific, unsigned int heapSortLimit);

**Heap.cpp**

#include "Heap.hpp"

int HeapInit(HeapNode \*heap)

{

    for (int i = 0; i <= MAX\_RIGION\_NUM; i++)

    {

        heap[i].area = 0;

        heap[i].id = i;

    }

    return OK;

}

void CaluHeapArea(Mat imgMask, HeapNode \*heap)

{

    for (int i = 1; i < imgMask.rows - 1; i++)

    {

        int \*rowInImgMask = imgMask.ptr<int>(i);

        for (int j = 1; j < imgMask.cols - 1; j++)

        {

            if (rowInImgMask[j] != -1 && rowInImgMask[j] > 0)

                heap[rowInImgMask[j]].area++;

        }

    }

}

void MaximizeHeapOnce(HeapNode \*heap, int specific, unsigned int heapSortLimit)

{

    unsigned int issue = heap[specific].area;

    unsigned int index = heap[specific].id;

    unsigned int j = 2 \* specific;

    while (j <= heapSortLimit)

    {

        if (j < heapSortLimit && heap[j].area < heap[j + 1].area)

            j = j + 1;

        if (issue >= heap[j].area)

            break;

        heap[j / 2] = heap[j];

        j \*= 2;

    }

    heap[j / 2].area = issue;

    heap[j / 2].id = index;

}

void MaximizeHeap(HeapNode \*heap, unsigned int heapSortLimit)

{

    for (int i = heapSortLimit / 2; i >= 1; i--)

        MaximizeHeapOnce(heap, i, heapSortLimit);

}

void HeapSortAsc(HeapNode \*heap, unsigned int heapSortLimit)

{

    MaximizeHeap(heap, heapSortLimit);

    for (int i = heapSortLimit; i >= 2; i--)

    {

        HeapNode temp = heap[1];

        heap[1] = heap[i];

        heap[i] = temp;

        MaximizeHeapOnce(heap, 1, i - 1);

    }

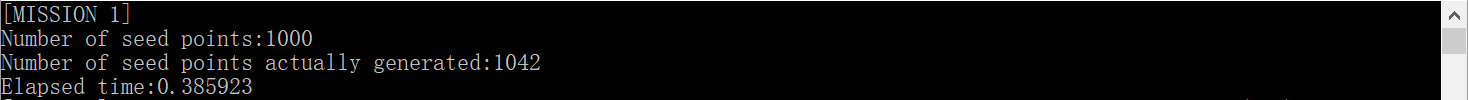
}

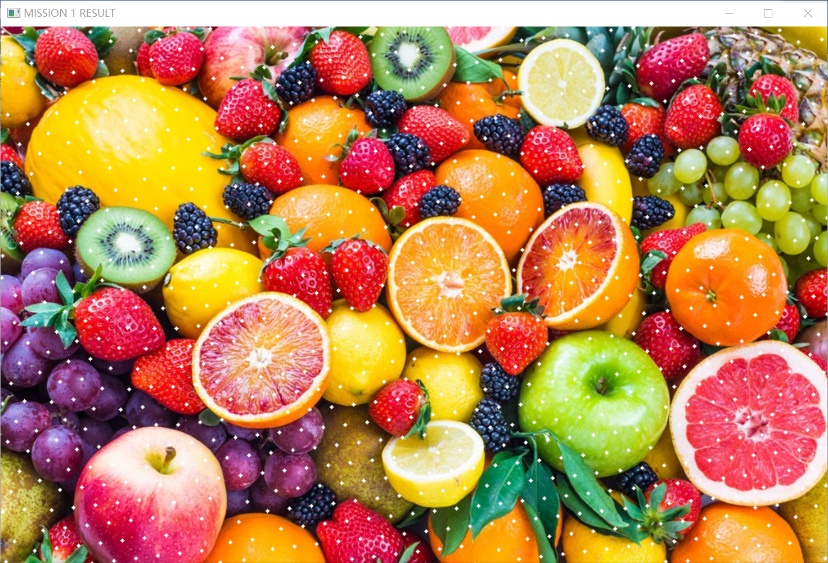
# 4、调试分析

## 4.1常规运行

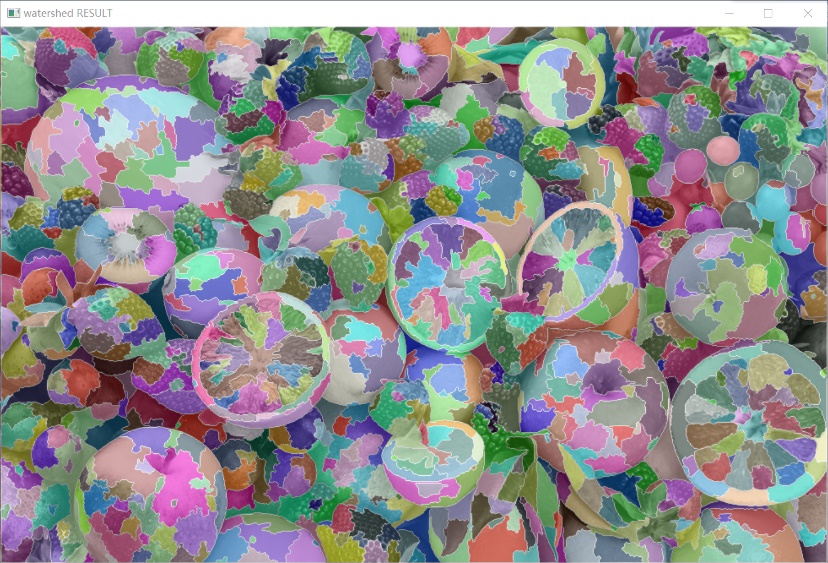
4.1.1问题一：随机打点+分水岭算法

输入1000后程序运行0.39秒得到1042个点。结果如下：





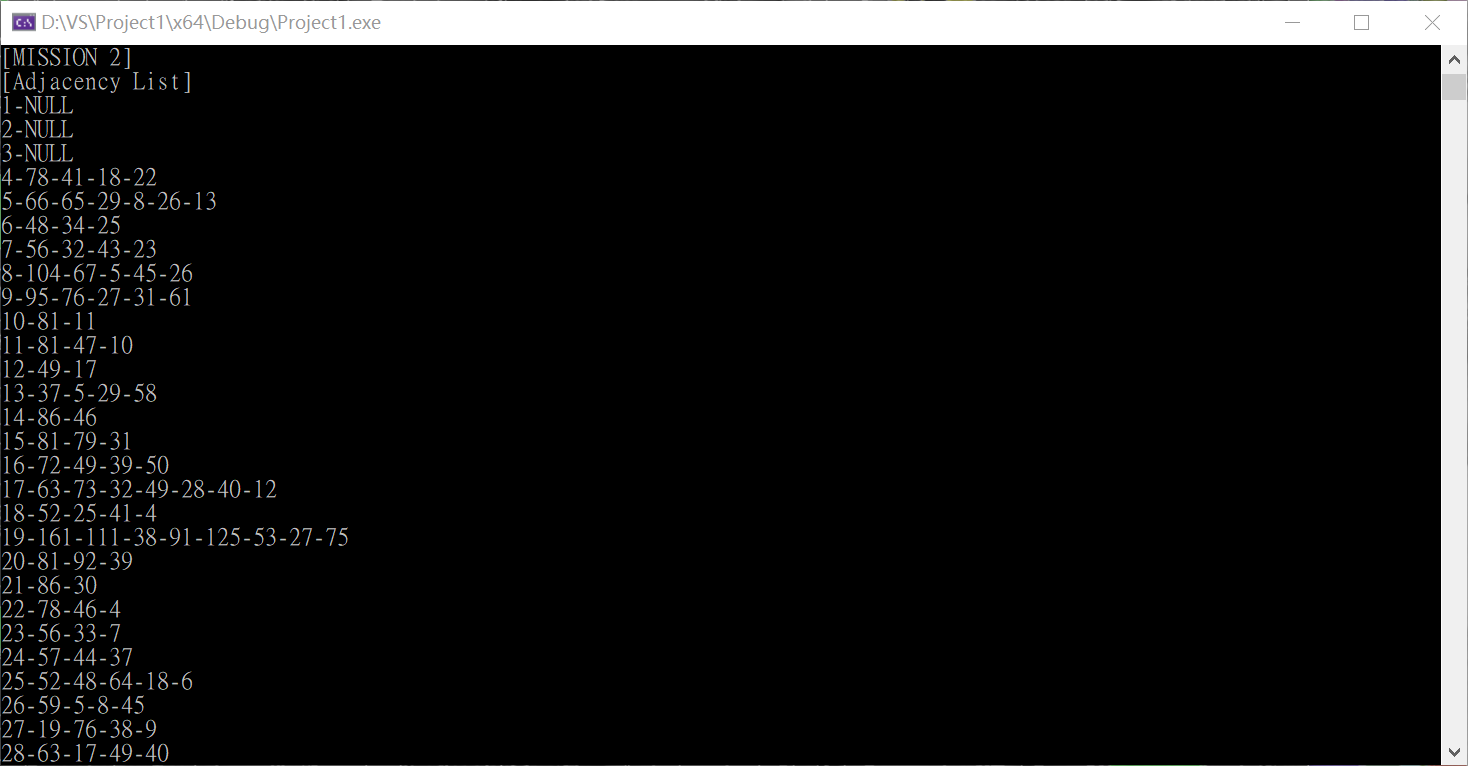
随即打点结果

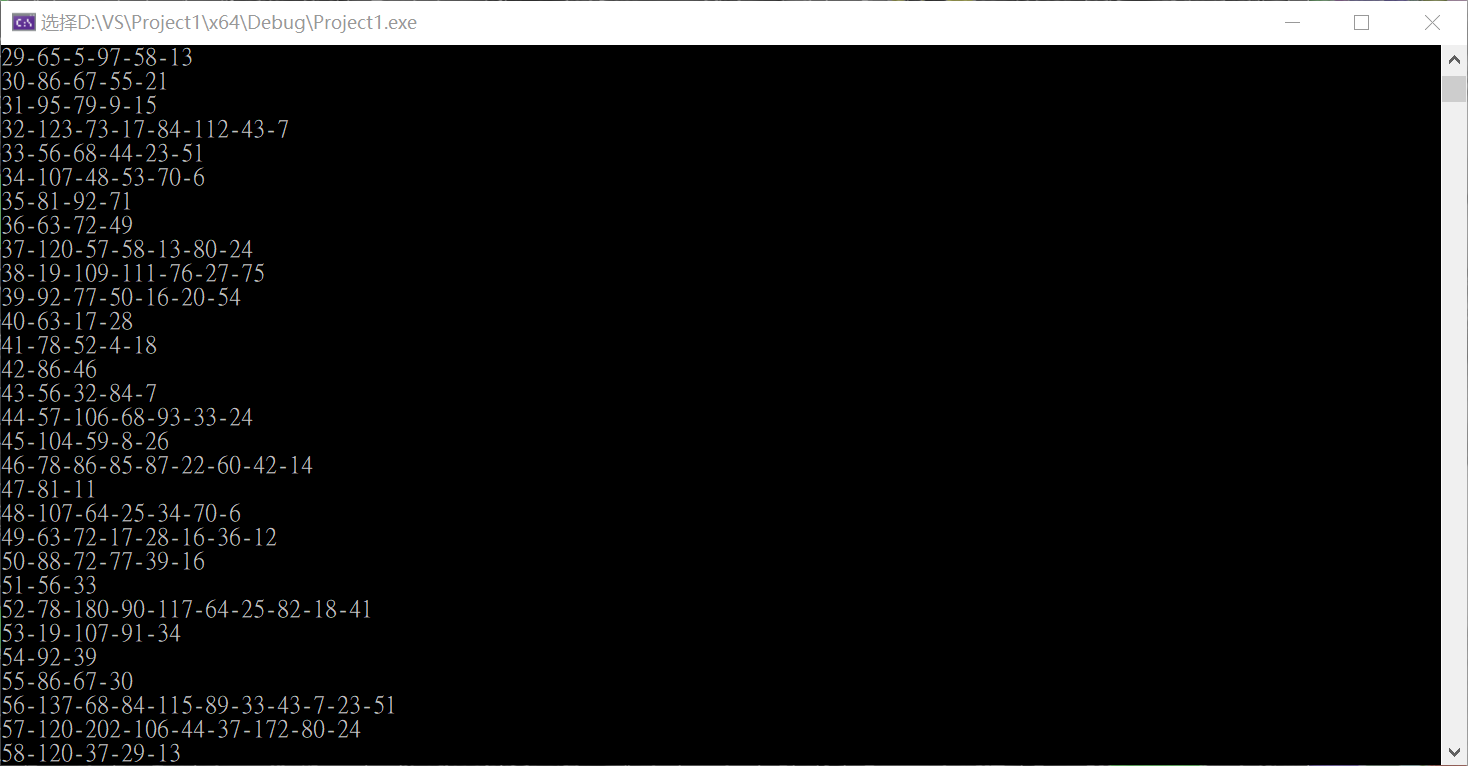


分水岭算法可视化结果

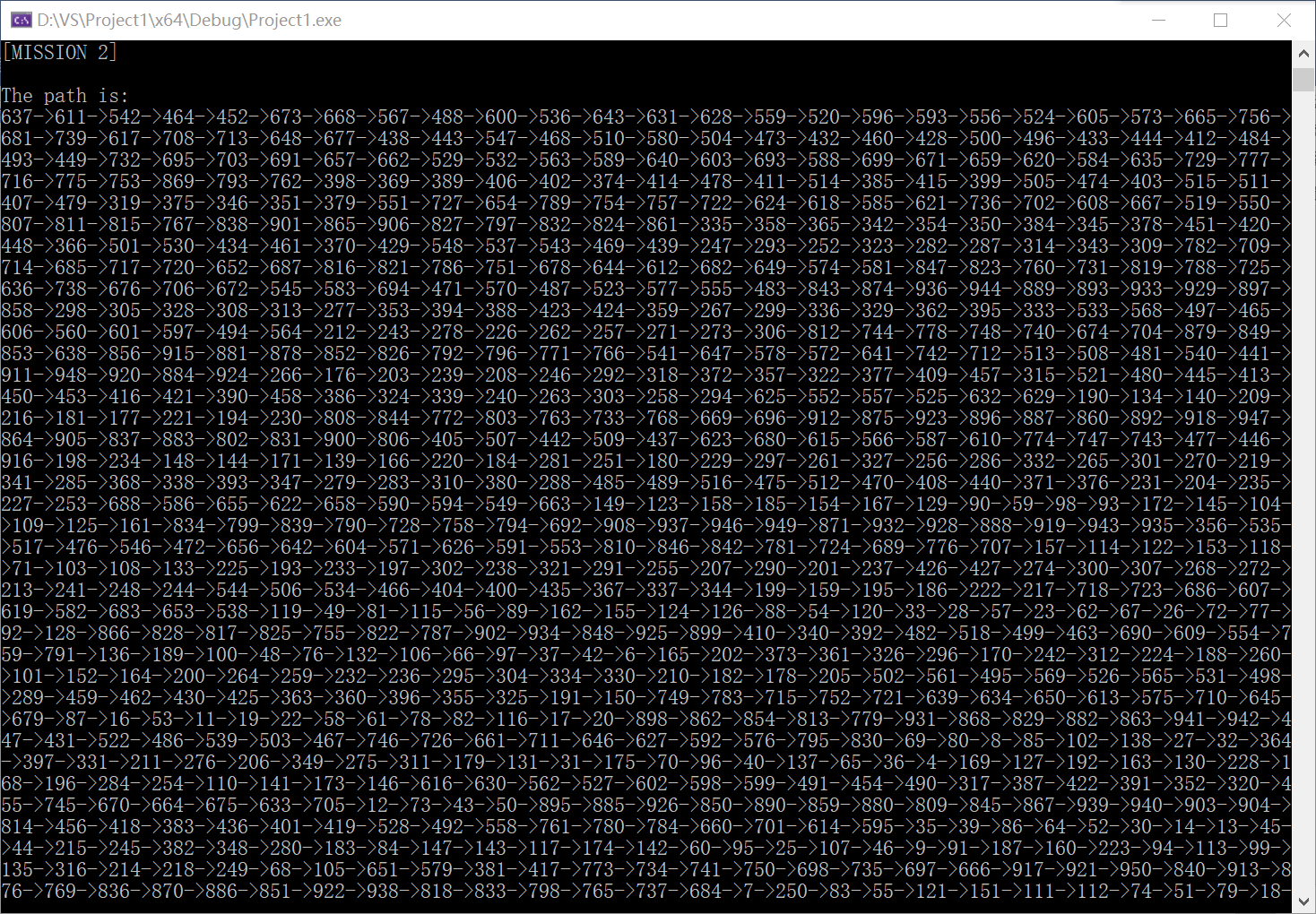
4.1.2问题二：四原色法填充分水岭区域

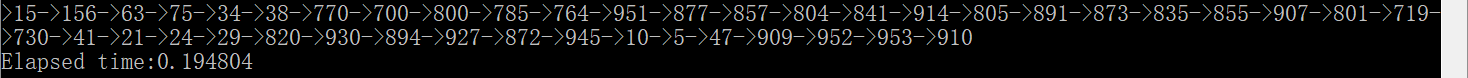
在问题一运行结果的基础上，单击回车，运行0.19秒后得到问题二运行结果如下：



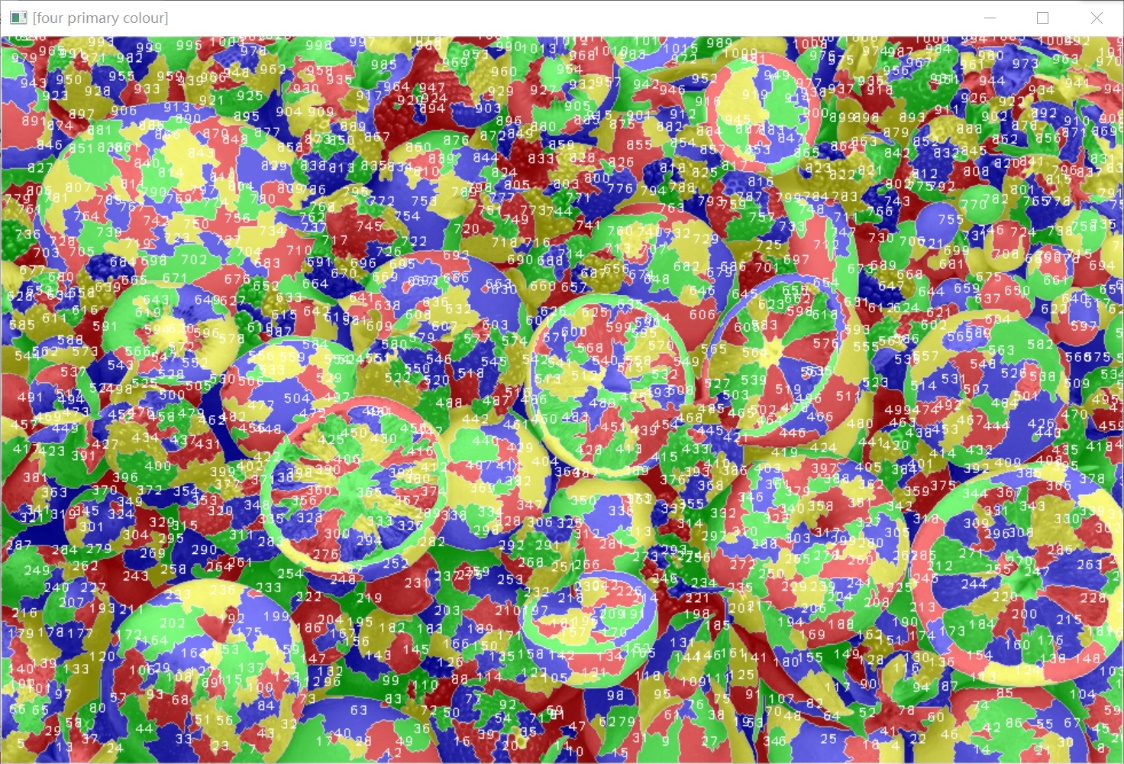


邻接表（部分）





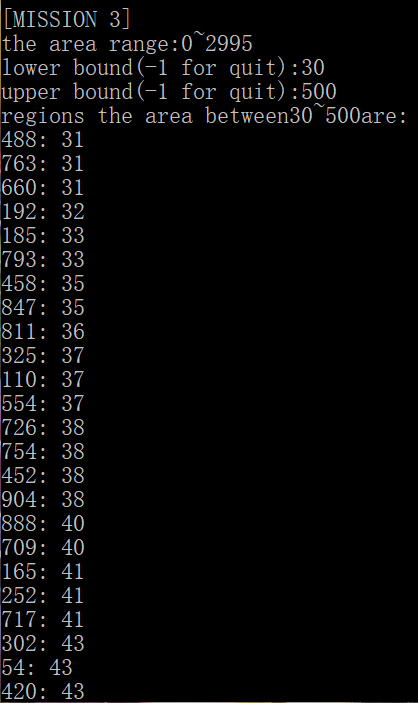
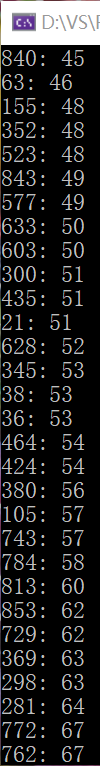
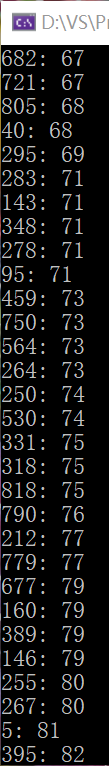
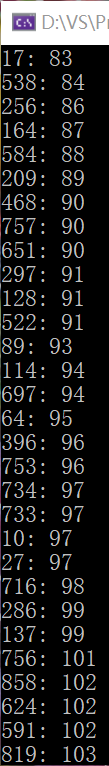
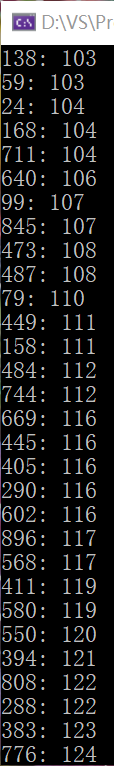
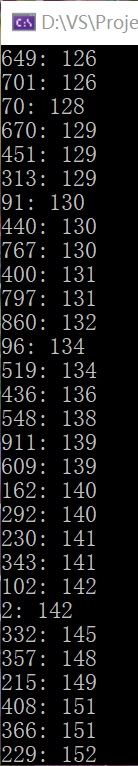
涂色路径



四原色涂色可视化结果

4.1.3问题三：折半查找

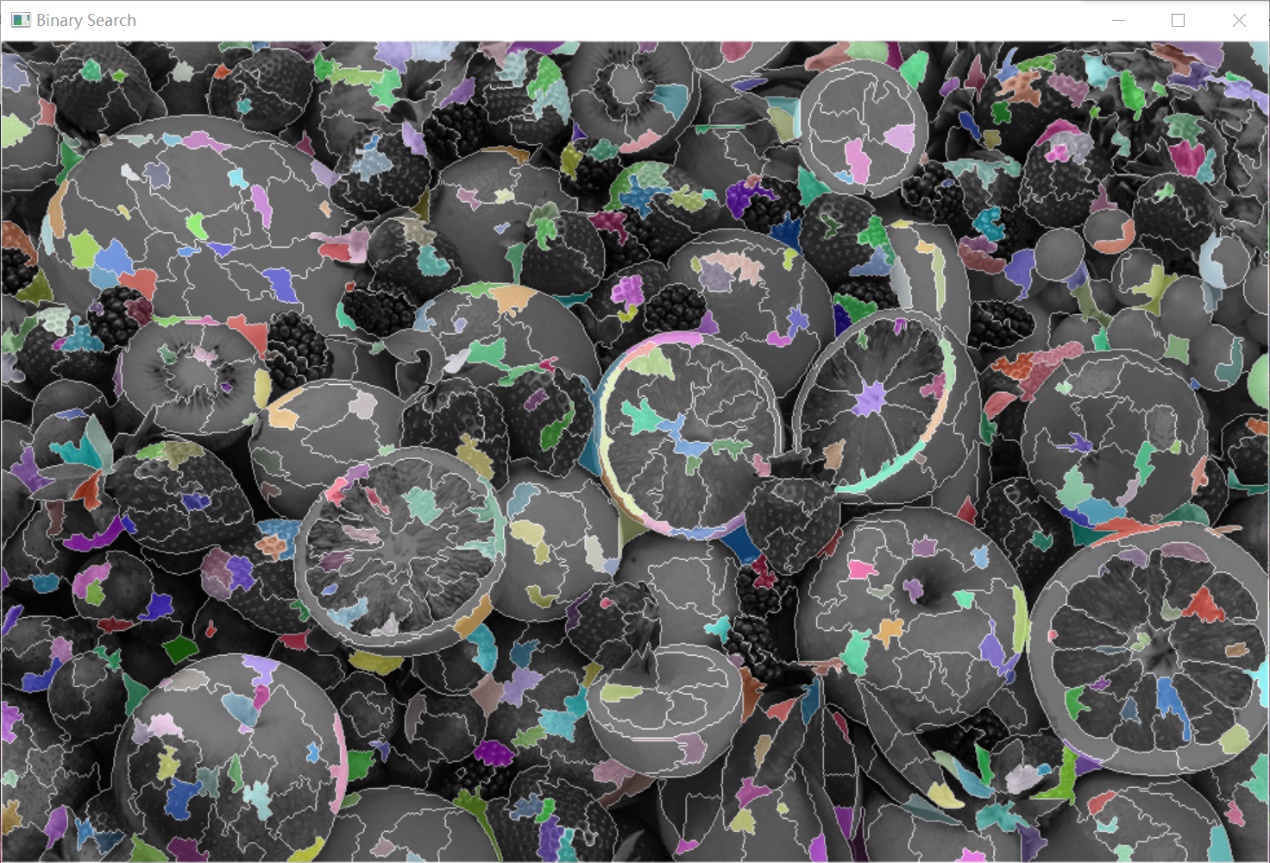
在问题二运行结果的基础上，先对区域面积进行堆排序，得到区域面积范围0~2995，输入查找范围30~500，运行0.003秒后共找到440个区域，得到结果如下：

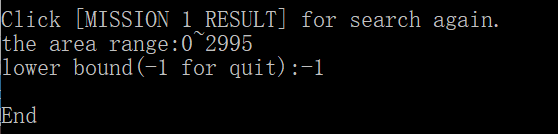
按面积大小输出在查找范围内的区域编号及相应面积（图为部分）



输出区域数量及所用时间



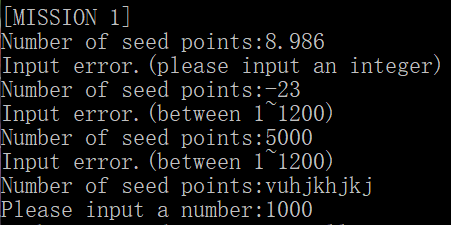
折半查找可视化结果



可重复查找，输入-1退出

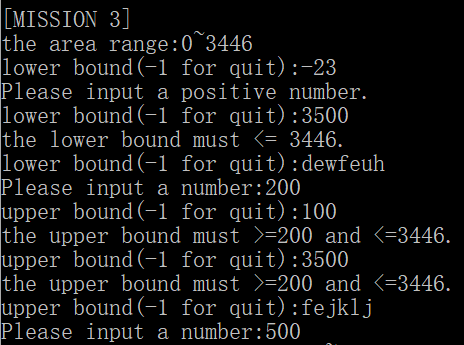
## 4.2容错处理

4.2.1问题一的容错处理



输入为负、输入过大（保护程序，避免卡死）、输入非数字或非整数都会有相应提示。

4.2.2问题三的容错处理



输入为负（非-1）、输入值不在范围内、不符合逻辑(上限小于下限)、输入非数字都会有相应提示。

## 4.3调试

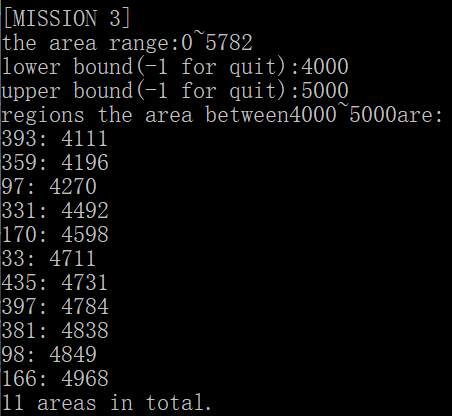
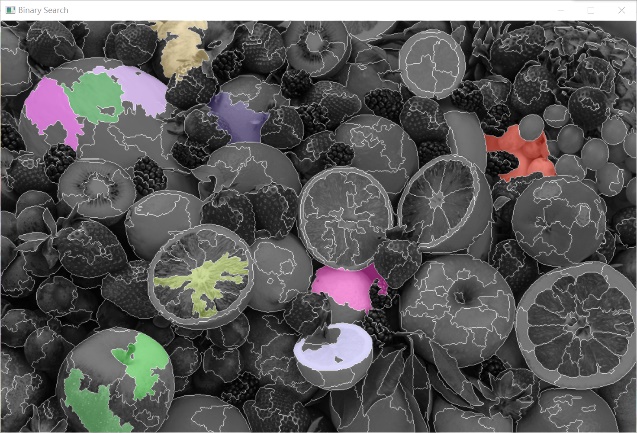
4.3.1测试最佳查找次数上限（GetNewNearPointListNodeIterateLimit）

从10开始逐渐增大GetNewNearPointListNodeIterateLimit，每次加十，发现当取值为40时成功率即达到90%以上，停止调试。

4.3.2测试折半查找查找结果是否完整

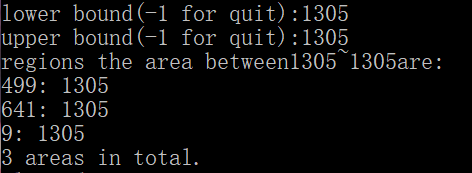
调试目标：输入区域面积上限和下限进行查找，若查找结果区域数与问题一中输出结果相同，则说明算法无误。

一开始调试时折半查找的输出结果小于最终打点个数，但可视化结果显示区域全亮，我首先将查找区域范围改小，使范围内的区域数控制在方便计数的范围，发现调试窗口的输出结果个数与点亮区域数相同，因此排除计数代码所在循环不当的错误；

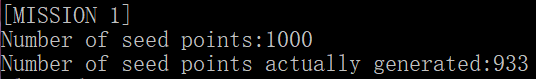
共输出11个区域编号及相应面积，计数结果为11，点亮区域数也为11

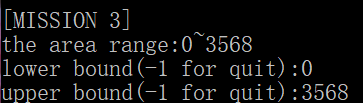
接着我输入相同的上限和下限，发现点亮区域数多于一个且计数结果不为1，排除相同面积区域只计了一次的错误；

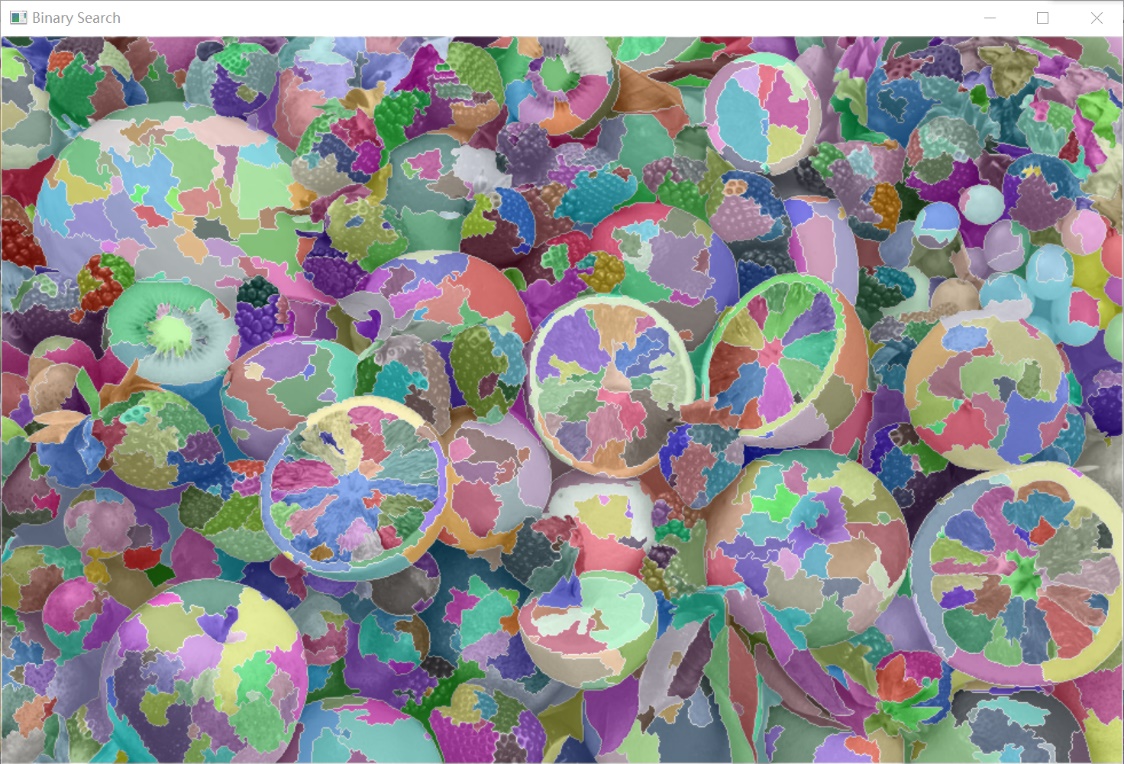
共输出3个区域编号及相应面积，计数结果为3，点亮区域数也为3

经过一段时间的思考及求助学长（主要是求助学长）后发现，由于随机取点过程中的缺陷：有部分点恰好位于经分水岭算法后的边界上，成为边界后该区域面积即为0，加上折半查找的特性，一开始的算法仅判断查找结果与期望值是否相同，而并未判断该区域是否为队列中的第一个，例如共有15个面积为0的区域，编号0~14，折半查找时该次查找的下标为7，返回为真，停止查找，输出结果，因此导致了编号0~6的区域符合要求却未被找到。改进后的算法对此进行了优化，当查找下标对应的区域面积符合要求时并不立即停止查找，而是会再往前/往后逐个查询区域面积直到面积不满足要求或已到队首/队尾，改进后发现查找结果区域数与问题一中输出结果相同，说明算法无误。









问题一输出结果933，折半查找计数结果933，可视化结果区域全亮

# 5、上机总结

当我听说数据结构还有一次课设且我们班的课设难度还大于其它班时，我的内心是崩溃的，但是在学习的过程中，我逐渐发现了乐趣。从一开始调配一个环境都让人心烦，到后来慢慢熟悉了opencv库的用法，当我的电脑开始第一次跑出了分水岭算法，第一次跑出了输入为1000时的四原色涂色结果的时候，我都感受到了由衷的开心。

虽然我自己并没有做很多很多东西，但是在这个过程中，我将数据结构我们正在学习的几个结构全部理解消化，还提前学习了堆排序，这都令我成就满满。

在课设中，我学习了很多，收获了很多。感谢韩守东老师一学期的任教，您是一位良师，也希望您能将这样的精神一直延续下去！