华中科技大学数据结构实验报告

姓名：夜吼者

学号：

班级：

院系：人工智能与自动化学院

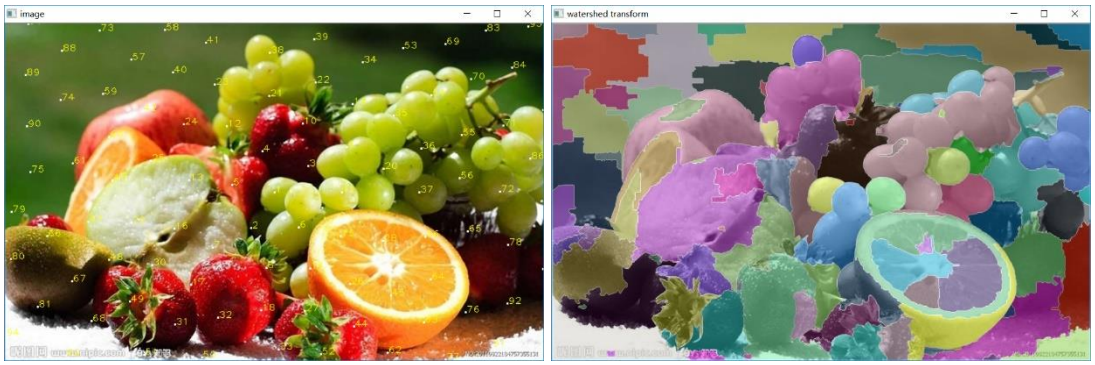
指导老师：韩守东

一、 问题描述

1 实验任务

1.1 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。



2 规范要求

输入要求：输入一张 M*N 的彩色图片。

其他要求：

1. 学习使用 STL 标准模板库和 OpenCV 基本数据格式；
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，确保稳定性及低时空复杂度。

3 编程环境

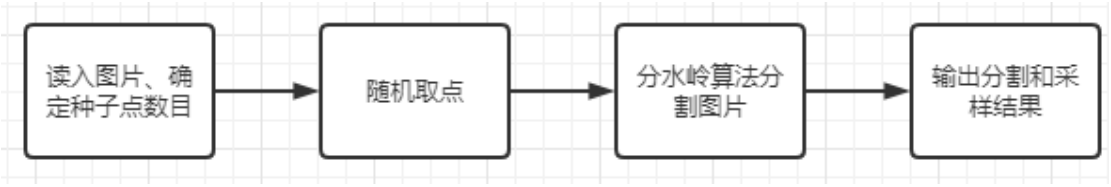
使用 Widows10 系统下的 VS2019 编译器，使用 Opencv4.6，编程语言为 C/C++。

二、 算法设计与分析

1 随机算法+分水岭算法

1.1 任务整体框架设计

该任务主要分为三个部分：准备工作、随机取点和分水岭划分图像，框架如下：



1.2 算法分析

1.2.1 分水岭（watershed）算法

分水岭算法以 opencv 库的函数 watershed()为基础，其函数原型为：

```
void watershed( InputArray image, InputOutputArray markers )
```

其中：第一个参数 image 是一个 8bit 3 通道彩色图像矩阵序列，即彩色图片；第二个参数 markers 是一个 32bit 矩阵序列，作为划分边界的种子，即注水点。任务一的核心是随机在图片上取点，并以这些点为注水点划分图像。

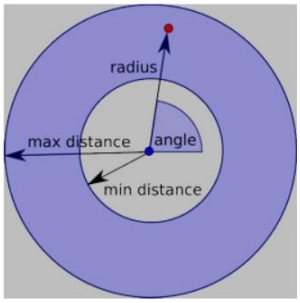
1.2.2 快速泊松圆盘采样算法（Fast-Poisson-Disc-Sampling）

快速泊松圆盘采样算法是对传统泊松圆盘采样算法的改良，降低了时间复杂度（$O(N^2)$变成$O(N)$），加快了采样速度。

泊松圆盘采样算法的基本步骤是：

1）.从激活点集中随机选出一个点 x（激活点集是已采样点集的子集）；

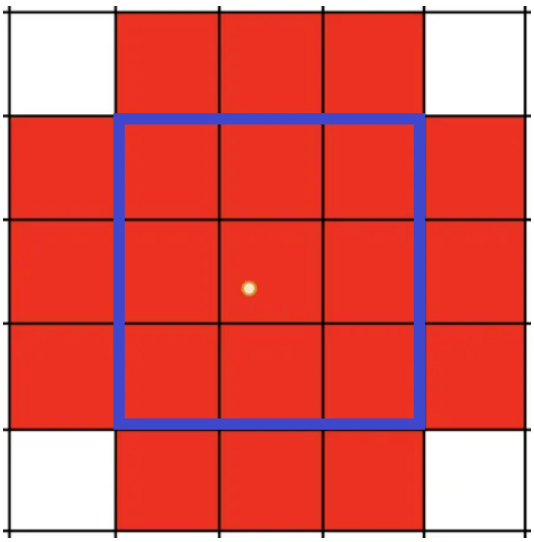
2）.在以该点为中心，内半径为 r/外半径为 2r 的圆环上随机生成 k 个临时点；



3）.对 k 个临时点进行逐一检查，如果某个临时点与所有已经采样的点距离都小于 r，则将该点放入激活集和采样点集；一旦有一个点成功，立即结束该轮检查，如果都没成功，将 x 移除激活点集。

整个算法的时间复杂度主要集中在第三步中对 k 个临时点进行注意验证上，该步骤的时间复杂度为$O(N^2)$，为降低该复杂度，快速泊松圆盘算法改进了该部分的算法。改进方法如下：

首先，对于 d 维采样空间，若样本间距离不小于 r，则该空间中，边长为$\frac{r}{\sqrt{d}}$的超立方体的对角线长度即为 r，所以每个超立方体中至多存在一个样本。基于此理论，在二维空间中，将空间划分成若干相同的正方形，则在检查临时点的时候，只需要检查周围若干方格，即如图红色方格，中是否存在种子点。图中蓝色框的正方形边长为



$\frac{r}{\sqrt{2}}$。

1.3 数据结构设计

在实现该任务时，对数据结构的使用主要集中在随机取点环节。在该环节，需要存储的数据有激活点列表、种子点列表；同时为了实现快速泊松圆盘采样算法，我们

还需要将图片分割成若干相同正方形组成的网格，所以还需要存储该网格以及各网格中点信息。

激活点和种子点列表使用 C++STL 标准模板库中的容器 vector 进行存储，列表中元素的数据类型为 Point，是一个 opencv 库中的类类型。使用 vector 不仅可以动态分配储存空间，还可以方便地进行元素的插入、删除、访问、遍历以及查找。

空间网格使用数组存储，即 vector<vector>，由若干个同构的 vector 容器组成。容器中元素的数据类型为自建类 grid_cell,其定义如下：

```
class grid_cell
{
public:
    Point pos;//若该 cell 中存在种子点，则 pos 为种子点坐标
    int flag;//1：存在种子点，0：不存在种子点

    //grid_cell 初始化
    void Init_grid_cell()
    {
        flag = 0;
        pos.x = -1;
        pos.y = -1;
    }

    //向 gird 中添加点
    void Add_NewPoint(Point NewPoint)
    {

        flag = 1;
        this->pos.x = NewPoint.x;
        this->pos.y = NewPoint.y;

    }

    //删除 grid_cell 中的点
    void Delete_Point()
    {
        this->pos.x = -1;
        this->pos.y = -1;
        flag = 0;
    }
};
```

2 四原色填色

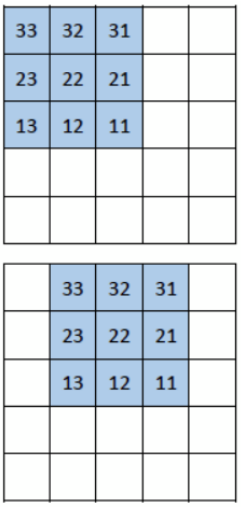
2.1 任务整体框架设计

2.2 算法设计

任务二的核心算法是利用回溯法进行填色。回溯法的基本思想就是当我们的颜色赋值 1,2,3,4 时，判断是否有一条通路是可以实现的，如果不能就回退到上一个走另一条分支。实现最好的方法就是利用堆栈。

2.2.1 创建邻接表

邻接表的创建基于任务一生成的轮廓图。任务一将轮廓图中不同封闭区域的像素点打上了序号，判断像素点序号即可判断是否是同一区域。为了生成邻接关系，使用 3×3 矩阵遍历轮廓图，如图：



判断中心点和边缘点的序号，即可得出邻接关系。

2.2.2 开辟最佳路径

本次实验中，填色顺序的基本逻辑是先难后易，即先填涂拥有最多未填涂邻近区域的区域，同时填涂该区域的邻接区域。不断重复上述步骤，开辟最佳填色路径。

2.2.3 回溯法填色

回溯法的核心思想很简单，如果某一区域填色失败，就回退到上一个区域并将该区域重新填色，随后按原路线继续尝试。填色结果用堆栈储存，其先进后出的特性方便回退；退出的元素用队列储存，先进先出的特性可以保存先后顺序。

2.3 数据结构设计

2.3.1 邻接表

```
typedef struct ArcNode//边表结点
{
    int adjvex;//顶点序号,存储邻接顶点对应的下标
    int degree;//顶点的度
    int area;//区块的面积
    struct ArcNode* nextarc;
}ArcNode;

typedef struct VertexPoint//顶点表结点
{
    int adjvex;//顶点序号，亦为区块编号
    int Block_No;
    Point gross;
    ArcNode* firstarc;//指向第一条依附于该顶点的弧
}VertexPoint,AdjList[Max_Vertex_Num];

typedef struct//邻接表
{
    AdjList vertices;
    int vexnum;//图的顶点数
}AdjGraphList;
```

邻接表的关键属性是其储存的变量，在本此实验中，邻接表中的顶点需要储存其身份值 adjvex 和区域编号 Block_No；边表结点需要储存身份值 adjvex、该结点的度 degree 以及该节点对应区域的面积 area。

3 堆排序+折半查找

3.1 算法分析

3.1.1 堆排序

将无序的面积序列使用传统堆排序法进行排序。通过建立大顶堆，降序排列顺序表。

3.1.2 折半查找

任务三使用折半查找的目的是查找某一范围内所有的元素，为了简化过程，我们只需要查找上界和下界的位置即可。同时，为了提高筛选能力，当输入的上下界本身不存在有序顺序表中，不会影响查找。

三、 调试分析

1 随机算法+分水岭算法

1.1 调试与优化

完成采样算法框架搭建后，开始对采样效果进行调试。

第一阶段调试，逐一对点数为 100、500 和 1000 的输入进行测试，发现实际生成点大致为输入值的 60%。分析输出图像，发现部分点距离远大于 r，对空间造成了浪费。为解决该问题，增加空间利用率，我对算法的两个部分进行了优化：

1） 原算法中，在检查激活点生成的临时点时，若存在一个临时点不满足距离要求，就会认定该激活点不合格，这显然是不合适的。所以我增大了临时点容错次数，即当一定数量的临时点不合格时，才会判断激活点不合格。

2） 在一次采样结束后，在种子列表中随机选点进行多轮采样。

第一阶段优化后，实际采样点数显著增加，最终结果已经可以达到目标值的 95%。

第二阶段调试，该阶段优化主要针对边缘空间利用率。优化策略如下：

1） 在采样开始时，将四角的点加入种子点列表，从四角开始采样；

2） 在生成新的激活点时，若激活点超出图像范围，不再将其舍弃，而是放在边缘。

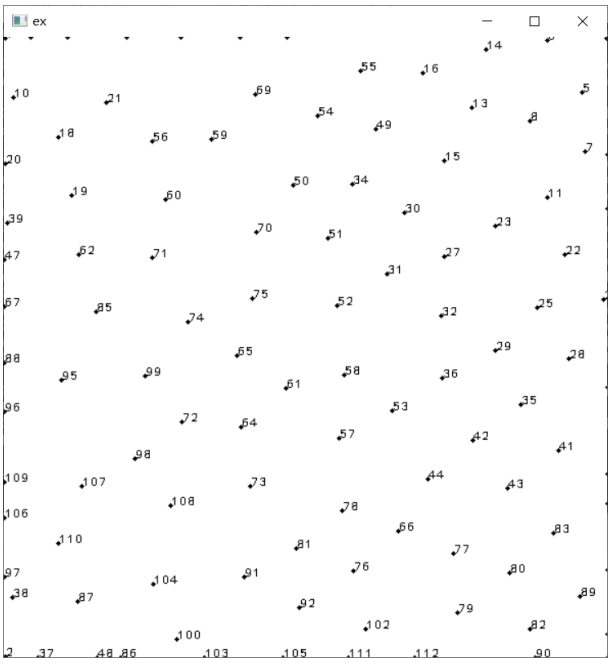
第二阶段优化完成后，实际采样点数进一步增加，在点数较低时，已经可以达到目标点数。

第三阶段调试，对所有可调参数进行调试。可调参数有 cell_size、临时点数 k，不断调试，最终达到目标。

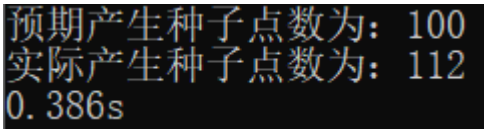
1.2 输出结果

输入图像大小：591×609。

预期种子点数：100


ex

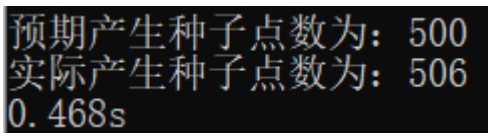

watershed


预期产生种子点数为： 100
实际产生种子点数为： 112
0. 386s

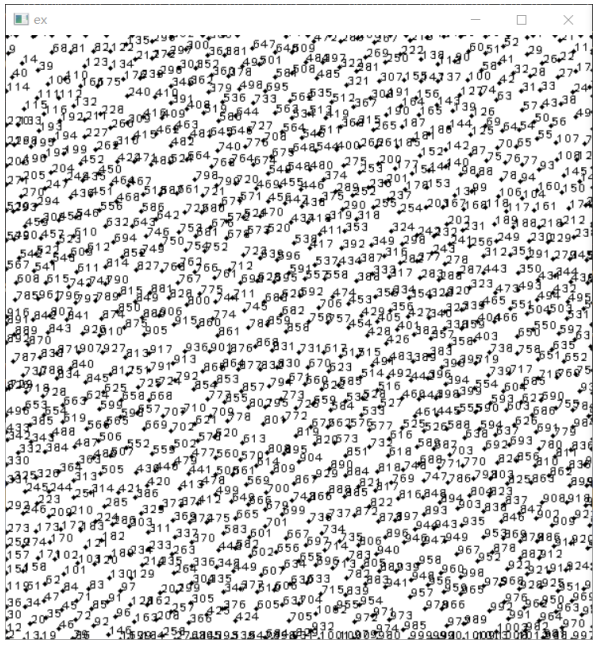
预期种子点数：500

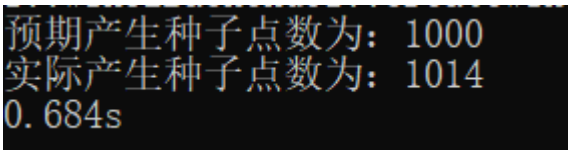

ex


watershed

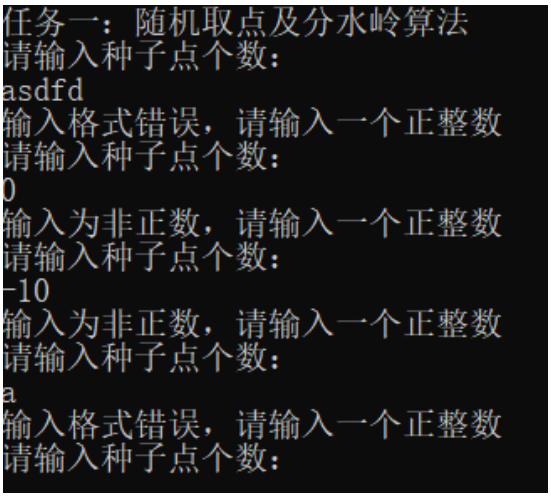

预期产生种子点数为： 500
实际产生种子点数为： 506
0. 468s

预期种子数 1000





1.3 合法性测试



2 四原色填色

2.1 调试与优化

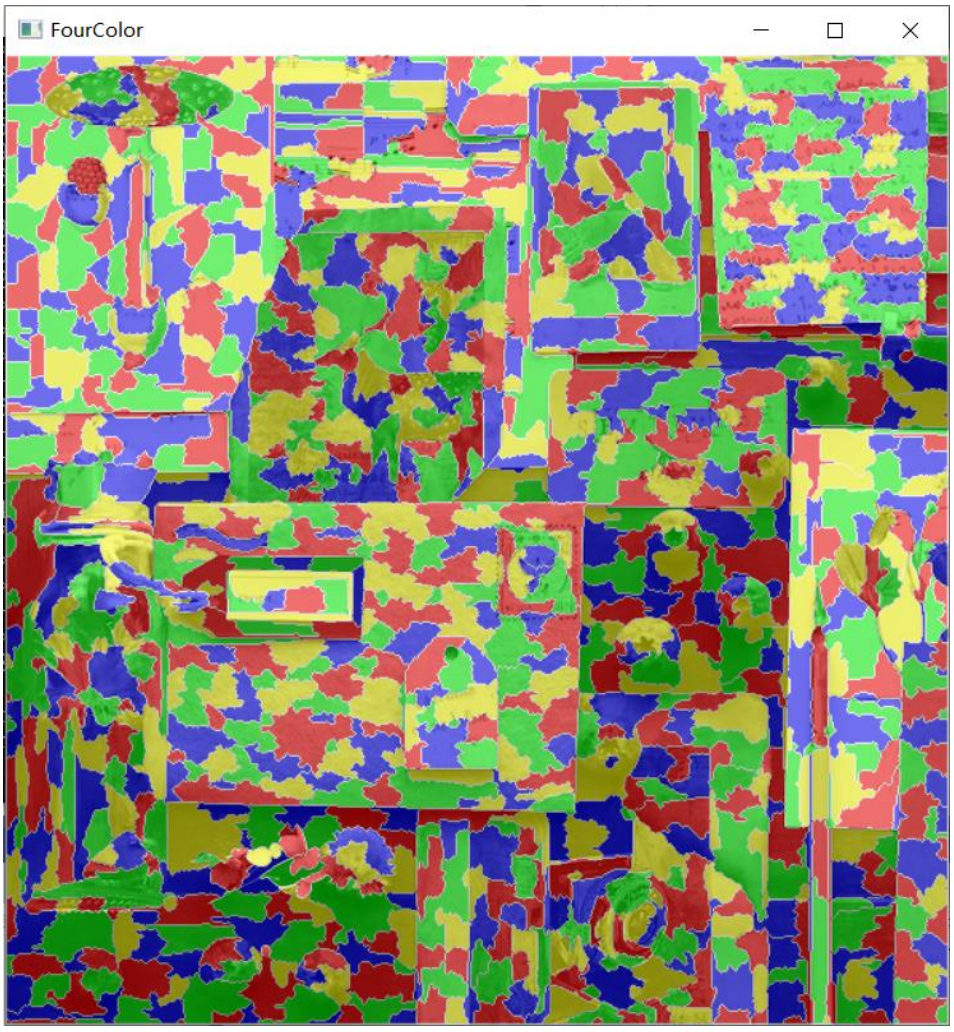
初始算法使用广度优先遍历建立填色路径、深度优先遍历进行填色，此时无论做什么优化，填色效果都很差，只能勉强运行 100 个随机点。

更新算法后，有时会出现陷入深度过大无法有效回退的情况，更新修复算法逻辑为：记录结点回退次数，当回退次数达到某一阈值，认为陷入死去，则回退 50 个结点，退出死区。回退代码如下：

```
// 如果涂色过程一直卡在某个地方，既不能继续往前涂，又不能往后退，说明陷入了死胡同
// 那么就往回退 50 个试试重新制定涂色方案，此过程非常重要
for (int i = 0; i < Block_Graph.vexnum; i++)
{
    // 如果某个结点频繁回退，即达到阈值，就直接回退 50 个结点
    if (pop_record[i] >= 250)
    {
        pop_record[i] = 0;
        for (int j = 0; j < 50; j++)
        {
            int temp_area_number;
            if (EmptyStack(stack) == TRUE) break;
            StackPop(stack, temp_area_number);
            // 先入中转站
            InQueue(transfer_station, temp_area_number);
            //StackPush(Buffer, temp_area_number, NoColor);
            while (EmptyQueue(buffer) == FALSE)
            {
                int transfer;
                DeQueue(buffer, transfer);
                InQueue(transfer_station, transfer);
            }
            while (EmptyQueue(transfer_station) == FALSE)
            {
                int transfer;
                DeQueue(transfer_station, transfer);
                InQueue(buffer, transfer);
            }
        }
    }
}
```

2.2 输出结果

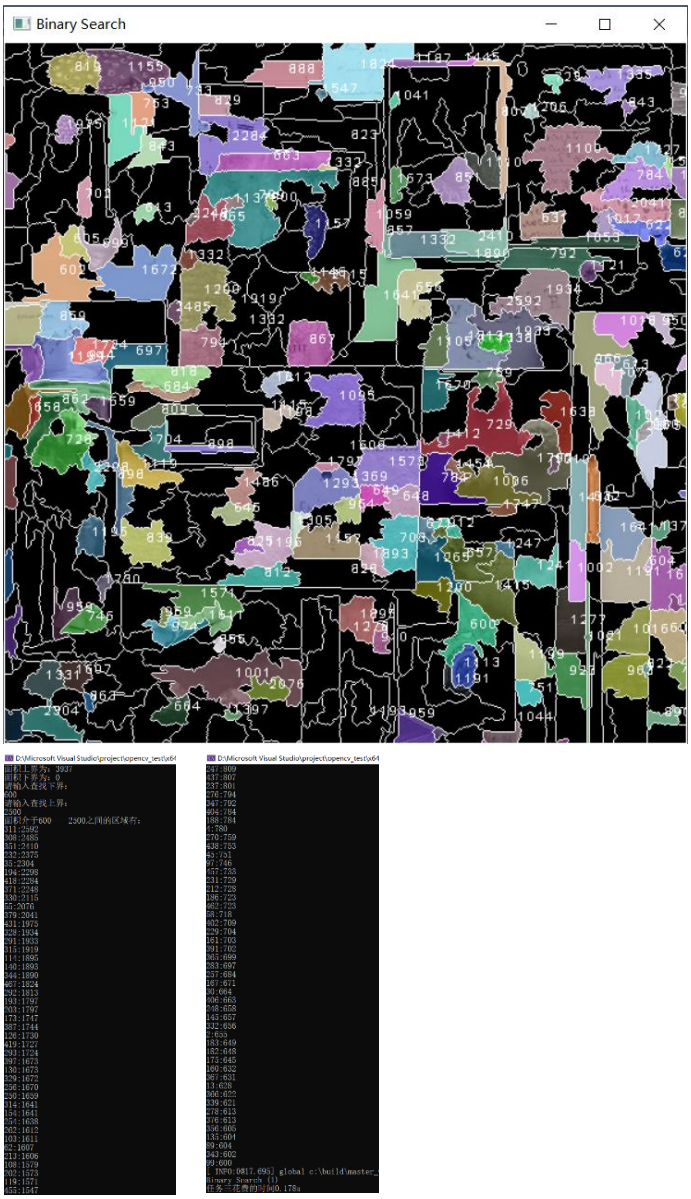
输入点数为 1000：



3 堆排序+折半查找

3.1 输出结果

输入点数为：500



3.2 合法性测试

```
面积上界为：3485
面积下界为：0
请输入查找下界：
30098
查找下界过大
请输入查找下界：
20
请输入查找上界：
-90
查找上界过小
请输入查找上界：
6
查找上界小于下界
请输入查找上界：
5202
```

四、 总结展望

本次数据结构上机实验的重点有两个：算法和数据结构。算法是完成特定任务的更优方案，数据结构则是算法的砖瓦，搭建起整个代码。

在编写本次上级实验任务的过程中，我第一次体验了算法设计和优化，其中给我影响最深刻的是算法优化的过程。在任务一中，为了满足最后几个点数，不仅需要分享当前算法的不足和优化点，还需要结合实际输出结果研究优化方向。

在设计和优化算法的时候，需要考虑设计和使用什么样子的数据结构，才开始意识到我们学习数据结构的意义——在于实现某种功能、支撑代码框架。不同的算法对数据结构提出了不同的要求，更好的数据结构也可以增强算法的性能。

附录：主要代码

```
int main()
{
    cout << "任务一：随机取点及分水岭算法" << endl;
    vector<Point> SeedPointList;//种子点列表
    int K = 0;//种子点数
    string Temp;
    while (1)
    {
        try
        {
            cout << "请输入种子点个数：" << endl;
            cin >> Temp;
            K = stoi(Temp);
        }
        catch (exception& invalid_argument)
        {
            cout << "输入格式错误，请输入一个正整数" << endl;
            continue;
        }
        if (K <= 0) cout << "输入为非正数，请输入一个正整数" << endl;
        else if (K > 5000) cout << "输入值过大，请减小输入值" << endl;
        else break;
    }
    clock_t start, end;//用于计算时间
    start = clock();
    Mat img = imread("C:\\Users\\Dragon\\Desktop\\1.jpg");
```

```
    Mat img_gray;
    Mat marker_mask;
    Mat Edge_img = Mat::zeros(img.size(), CV_32S);//绘制轮廓
    //用于储存轮廓
    vector<vector<Point>> Edge;
    vector<Vec4i> Hier;
    cout << img.size() << endl;
    //图像预处理
    cvtColor(img, marker_mask, COLOR_BGR2GRAY);
    cvtColor(marker_mask, img_gray, COLOR_GRAY2BGR);
    marker_mask = Scalar::all(0);


    /* preparation */
    /* 快速泊松圆盘算法采集随机点 */
    if (FastPossionDisc(K, &SeedPointList) == OK)
    {
        cout << "实际产生种子点数为: " << SeedPointList.size() << endl;
        draw_Point(SeedPointList, K, marker_mask);

    }
    else cout << "ERROR" << endl;

    /* mission1 */
    /* 分水岭算法 */
    //寻找轮廓
    cout << "任务一: 分水岭算法" << endl;
    findContours(marker_mask, Edge, Hier, RETR_CCOMP,
CHAIN_APPROX_SIMPLE);
    if (Edge.empty()) cout << "ERROR" << endl;
    int j=0;
    for (int i = 0; i >=0; i=Hier[i][0],j++)
    {
        //对marks进行标记，对不同区域的轮廓进行编号，相当于设置注水点，有多少
轮廓，就有多少注水点
        drawContours(Edge_img, Edge, i, Scalar::all(j + 1), -1, 8, Hier,
INT_MAX);
    }
    //设置随机颜色盘
    vector<Vec3b> colorTab;
    for (int i = 0; i < j; i++)
    {
        int b = theRNG().uniform(0, 255);
        int g = theRNG().uniform(0, 255);
        int radius = theRNG().uniform(0, 255);
```

```
        colorTab.push_back(Vec3b((uchar)b, (uchar)g, (uchar)radius));
    }
    //绘制分水岭图样
    watershed(img, Edge_img);
    Mat WaterShed_img = Mat(Edge_img.size(), CV_8UC3);
    for (int i = 0; i < Edge_img.rows; i++)
    {
        for (int k = 0; k < Edge_img.cols; k++)
        {
            int index = Edge_img.at<int>(i, k);
            if (index == -1)
                WaterShed_img.at<Vec3b>(i, k) = Vec3b(255, 255, 255);
            else if (index <= 0 || index > j)
                WaterShed_img.at<Vec3b>(i, k) = Vec3b(0, 0, 0);
            else
                WaterShed_img.at<Vec3b>(i, k) = colorTab[index - 1];
        }
    }
    WaterShed_img = WaterShed_img * 0.5 + img_gray * 0.5;

    float Rmin = sqrt((M * N) / (float)K);

    imshow("watershed", WaterShed_img);
    cout << "预期产生种子点数为: " << K << endl;
    cout << "实际产生种子点数为: " << SeedPointList.size() << endl;
    end = clock();
    cout << "任务一花费的时间" << (float)(end - start) / CLOCKS_PER_SEC
<< "s" << endl;
    cout << endl;
    cout << "按任意键进行下一个任务" << endl;
    waitKey(0);
    destroyAllWindows();

    int mission = 0;
    while (1)
    {
        cout << "输入 1 执行任务 2——四原色填色" << endl;
        cout << "输入 2 执行任务 3——堆排序和折半查找" << endl;
        cout << "输入 3 退出" << endl;
        cin >> Temp;
        if (Temp == "1")
        {
            mission = 1;
            break;
```

```
        }
        else if (Temp == "2")
        {
            mission = 2;
            break;
        }
        else if (Temp == "3") exit(0);
        else continue;
    }

    AdjGraphList Block_Graph;
    //创建邻接表
    Creat_AdjList(Block_Graph, Edge_img, SeedPointList.size());

    vector<AreaNode> AreaList;
    Init_AreaList(AreaList, Block_Graph.vexnum);
    Area_Measure(Block_Graph, Edge_img, AreaList);

    /* mission2 */
    /* 四原色填色*/
    if (mission == 1)
    {
        start = clock();
        Mat ColorImg = Mat(Edge_img.size(), CV_8UC3);


        //ShowBlock_Graph(Block_Graph);
        //创建队列，准备广度优先遍历
        LinkQueue ColorFill_Path;
        ColorFill_Path.front = NULL;
        ColorFill_Path.rear = NULL;
        InitQueue(ColorFill_Path);
        if (SortPath(Block_Graph, ColorFill_Path) == ERROR)
        {
            return 0;
        }

        DFS_FullColor(Block_Graph, ColorFill_Path, Edge_img, ColorImg);

        end = clock();
        cout << "任务二花费的时间" << (float)(end - start) /
CLOCKS_PER_SEC << "s" << endl;
    }
```

```
    /***任务三***/
    else if (mission == 2)
    {
        //堆排序，递减排序
        start = clock();
        HeapSort(AreaList);
        end = clock();
        clock_t time_ = end - start;

        //折半查找
        cout << "面积上界为: " << AreaList[0].area << endl;
        cout << "面积下界为: " << AreaList[AreaList.size() - 1].area <<
endl;
        //cout<<"请输入要"
        int highNo, lowNo;
        if (Binary_Search(AreaList, Block_Graph.vexnum, highNo, lowNo,
time_) == OK)
        {
            HighLight(AreaList, highNo, lowNo, Block_Graph.vertices,
WaterShed_img, Edge_img, time_);
        }
        cout << "任务三花费的时间" << (float)(time_) / CLOCKS_PER_SEC <<
"s" << endl;
    }
    waitKey(0);
    destroyAllWindows();
    return 0;
}
```

```
/*************************************************
功能：快速泊松圆盘采样
返回值：OK：种子采集完毕
        ERROR：出错
*************************************************/
int FastPossionDisc(int K, vector<Point>* SeedPointList)
{
    vector<Point> ActPointList;
    float R_min2 = (M * N) / (float)K;
    int max_retry = 120;//在活动点周围生成的尝试点的个数，一般取 30
    int cell_size = int(sqrt(R_min2 / 16));//区块大小
    int col_x = int(N / cell_size) + 1;//+1 防止溢出
    int row_y = int(M / cell_size) + 1;

    vector<vector<grid_cell>> grids;//区块矩阵，row_y 行，col_x 列
```

```
    grids.resize(row_y);
    for (vector<vector<grid_cell>>::iterator it = grids.begin(); it !=
grids.end(); it++)
    {
        it->resize(col_x);
        for (vector<grid_cell>::iterator jt = it->begin(); jt <
it->end(); jt++) {
            jt->Init_grid_cell();
        }
    }
    RNG rng((unsigned)time(NULL));//用于生成随机数

    //从四个顶点开始采集，有效利用边缘空间
    vector<Point> p0;
    p0.push_back(Point(1, 1));
    p0.push_back(Point(1, M - 1));
    p0.push_back(Point(N - 1, 1));
    p0.push_back(Point(N - 1, -1));
    for (int i = 0; i < 4; i++)
    {
        ActPointList.push_back(p0[i]);
        SeedPointList->push_back(p0[i]);
        col_x = int(p0[i].x / cell_size);
        row_y = int(p0[i].y / cell_size);
        grids[row_y][col_x].Add_NewPoint(p0[i]);//表示该区块已经有点
    }
    int try_again = 3;//当Act Point List为空时，重试的次数
    while (!ActPointList.empty() && try_again > 0)
    {
        int If_Found = FALSE;
        //从活动点列表中随机取一个点，在其周围生成新采样点，并对检验
        int ActPointNum;//活动点序号
        //获取活动点序号
        if (ActPointList.size() == 1) ActPointNum = 0;
        else if (ActPointList.size() == 0 && try_again == 3)
        {
            cout << "ActPoint Get ERROR" << endl;
            return ERROR;
        }
        else if (try_again < 3) ActPointNum = floor((rng.uniform(0,
SeedPointList->size())));
        else ActPointNum = floor((rng.uniform(0, ActPointList.size())));
        if (ActPointNum > ActPointList.size() || ActPointNum < 0)
        {
```

```
            cout << "ActPointNum out of range" << endl;
            return ERROR;
        }

        Point ActPoint;
        if (try_again == 3) ActPoint = ActPointList[ActPointNum];
        else //如果是再次尝试，从 SeedPointList 中取点
            ActPoint = (*SeedPointList)[ActPointNum];
        //在快速泊松采样中，如果生成的新点与周围的点距离不满足要求，立即退出，这样会导致空间存在浪费，error_count 提供了几个容错次数
        int error_count = 0;
        //在活动点周围生成 max_retry 个点
        for (int i = 0; i < max_retry; i++)
        {
            Point NewPoint;
            int NewR = int(sqrt(R_min2)) * rng.uniform(1, 2);
            float NewAngle = rng.uniform(0., 2 * PI);
            //计算采样点坐标
            NewPoint.x = ActPoint.x + int(NewR * cos(NewAngle));
            NewPoint.y = ActPoint.y + int(NewR * sin(NewAngle));

            //验证新点
            //如果新点超出范围，将其放在边缘
            if (NewPoint.x<0 || NewPoint.x>N ||
                NewPoint.y<0 || NewPoint.y>M)
            {
                if (NewPoint.x < 0) NewPoint.x = 0;
                else if (NewPoint.x > N) NewPoint.x = N;
                if (NewPoint.y < 0) NewPoint.y = 0;
                else if (NewPoint.y > M) NewPoint.y = M;
            }
            //计算采样点的对应的区块
            col_x = floor(NewPoint.x / cell_size);
            row_y = floor(NewPoint.y / cell_size);
            if (grids[row_y][col_x].flag == 1)//该区块已经存在采样点
                continue;
            int flag = OK;
            //检查周围区块是否有小于最小距离的点
            //在这些区块之外的点距离一定满足，无需检验
            int min_x = floor((NewPoint.x - sqrt(R_min2)) / cell_size);
            int max_x = floor((NewPoint.x + sqrt(R_min2)) / cell_size);
            int min_y = floor((NewPoint.y - sqrt(R_min2)) / cell_size);
            int max_y = floor((NewPoint.y + sqrt(R_min2)) / cell_size);
```

```
            for (int i = min_x; i <= max_x; i++)
            {
                if (i<0 || i>floor(N / cell_size)) continue;
                for (int j = min_y; j <= max_y; j++)
                {
                    if (j<0 || j>floor(M / cell_size)) continue;
                    if (grids[j][i].flag == 1)//该区块存在活动点
                    {
                        if (j == int(ActPoint.y / cell_size) && i ==
int(ActPoint.x / cell_size))continue;
                        if (Cal_Dis2(grids[j][i].pos, NewPoint) < R_min2)
                        {
                            flag = ERROR;
                            break;
                        }
                    }
                }
            }

            //新采样点符合要求
            if (flag == OK)
            {
                ActPointList.push_back(NewPoint);
                SeedPointList->push_back(NewPoint);
                col_x = floor(NewPoint.x / cell_size);
                row_y = floor(NewPoint.y / cell_size);
                grids[row_y][col_x].Add_NewPoint(NewPoint);
                If_Found = TRUE;
                continue;
            }

            //新采样点不符合要求
            if (If_Found == FALSE)
            {
                if (error_count == 12 || error_count + i == max_retry)
                {
                    vector<Point>::iterator it = ActPointList.begin() +
ActPointNum;
                    ActPointList.erase(it);
                    if (ActPointList.size() == 0)
                        try_again--;
                    break;
                }
                else error_count++;
```

```
            }
        }
    }
    return OK;
}

/*************************************************
功能：计算两点之间距离的平方
返回值：R：两点距离的平方
*************************************************/
float Cal_Dis2(Point P1, Point P2)
{
    float R = pow((float)(P1.x - P2.x), 2.0) + pow((float)(P1.y - P2.y),
2.0);//两点距离
    return R;
}

void draw_Point(vector<Point> SeedPointList, int K, Mat img_gray)
{
    Mat img(M, N, CV_8UC3, Scalar(255, 255, 255));
    float Rmin = sqrt((M * N) / (float)K);
    int i = 0;
    for (vector<Point>::iterator it = SeedPointList.begin(); it !=
SeedPointList.end(); it++, i++)
    {
        circle(img_gray, *it, 2, Scalar(255, 255, 255), -1);
        circle(img, *it, 2, Scalar(0, 0, 0), -1);
        string Num = to_string(i + 1);
        putText(img, Num, *it, FONT_HERSHEY_PLAIN, 0.8f, Scalar(0, 0,
0));
    }
    imshow("ex", img);
}


/*************************************************
功能：根据轮廓创建邻接表
参数：Block_Graph：为图片中区块创建的邻接表
      Edge_img：轮廓图
      SeedPoint_Num：实际产生的种子点数
*************************************************/
int Creat_AdjList(AdjGraphList& Block_Graph,Mat Edge_img,int
SeedPoint_num)
{
    Block_Graph.vexnum = SeedPoint_num;//顶点数等于种子点数
```

```
for (int i = 0; i < Block_Graph.vexnum; i++)
{
    Block_Graph.vertices[i].adjvex = i + 1;
    Block_Graph.vertices[i].firstarc = NULL;

}
int this_block = 1, neighbor_block = 1;
int neighbor_block_dot[8];

//遍历像素点
for (int i = 1; i < Edge_img.rows - 1; i++)
{
    int* PixelPoint_col_1 = Edge_img.ptr<int>(i - 1);
    int* PixelPoint_col_2 = Edge_img.ptr<int>(i);//第 i 行像素点头指针
    int* PixelPoint_col_3 = Edge_img.ptr<int>(i + 1);
    for (int j = 1; j < Edge_img.cols - 1; j++)
    {
        int this_dot = PixelPoint_col_2[j];
        neighbor_block_dot[0] = PixelPoint_col_1[j-1];
        neighbor_block_dot[1] = PixelPoint_col_1[j];
        neighbor_block_dot[2] = PixelPoint_col_1[j+1];
        neighbor_block_dot[3] = PixelPoint_col_2[j - 1];
        neighbor_block_dot[4] = PixelPoint_col_2[j + 1];
        neighbor_block_dot[5] = PixelPoint_col_3[j - 1];
        neighbor_block_dot[6] = PixelPoint_col_3[j];
        neighbor_block_dot[7] = PixelPoint_col_3[j + 1];
        // nd  nd  nd
        // nd  td  nd
        // nd  nd  nd
        if (this_dot == -1)//如果该点在区域边缘
        {
            for (int k = 0; k < 8; k++)
            {
                if (neighbor_block_dot[k] > 0)
                {
                    this_block = neighbor_block_dot[k];
                    break;
                }
            }
            for (int k = 0; k < 8; k++)
            {
                if (neighbor_block_dot[k] > 0 &&
                    neighbor_block_dot[k] != this_block)
                {
```

```
                        neighbor_block = neighbor_block_dot[k];
                        break;
                    }
                }

                //如果this_block区块尚未加入图的顶点列表
                if (Block_Graph.vertices[this_block - 1].firstarc ==
NULL)
                {
                    ArcNode* p = NULL;
                    if ((p = (ArcNode*)malloc(sizeof(ArcNode))) == NULL)
                    {
                        cout << "ArcNode Failed to Request memory" <<
endl;
                        exit(1);
                    }
                    //将该区块加入顶点列表
                    p->nextarc = NULL;
                    p->adjvex = neighbor_block;//弧指向的顶点的序号
                    Block_Graph.vertices[this_block - 1].firstarc = p;
                }
                //如果this_block区块已经加入图的顶点列表
                else if (Block_Graph.vertices[this_block -
1].firstarc != NULL)
                {
                    int IsDuplicate = FALSE;//是否重复的标识量
                    ArcNode* p = NULL;
                    ArcNode* temp = Block_Graph.vertices[this_block -
1].firstarc;
                    //遍历查重
                    while (temp != NULL)
                    {
                        if (temp->adjvex != neighbor_block &&
temp->nextarc != NULL)
                        {
                            temp = temp->nextarc;
                        }
                        else if (temp->adjvex == neighbor_block ||
                            Block_Graph.vertices[this_block - 1].adjvex
== neighbor_block)
                        {
                            IsDuplicate = TRUE;
                            break;
                        }
```

```
                    else break;
                }
                //若没有重名
                if (IsDuplicate == FALSE)
                {
                    ArcNode* p = NULL;
                    if ((p = (ArcNode*)malloc(sizeof(ArcNode))) ==
NULL)
                    {
                        cout << "ArcNode Failed to Request memory" <<
endl;
                        exit(1);
                    }
                    p->adjvex = neighbor_block;
                    p->nextarc = NULL;
                    temp->nextarc = p;
                }
            }

            //如果neighbor_block 区块尚未加入图的顶点列表
            if (Block_Graph.vertices[neighbor_block - 1].firstarc ==
NULL)
            {
                ArcNode* p = NULL;
                if ((p = (ArcNode*)malloc(sizeof(ArcNode))) == NULL)
                {
                    cout << "ArcNode Failed to Request memory" <<
endl;
                    exit(1);
                }
                //将该区块加入顶点列表
                p->nextarc = NULL;
                p->adjvex = this_block;//弧指向的顶点的序号
                Block_Graph.vertices[neighbor_block - 1].firstarc =
p;
            }
            //如果neighbor_block 区块已经加入图的顶点列表
            else if (Block_Graph.vertices[neighbor_block -
1].firstarc != NULL)
            {
                int IsDuplicate = FALSE;//是否重复的标识量
                ArcNode* p = NULL;
                ArcNode* temp = Block_Graph.vertices[neighbor_block
- 1].firstarc;
```

```
                    //遍历查重
                    while (temp != NULL)
                    {
                        if (temp->adjvex != this_block &&
temp->nextarc != NULL)
                        {
                            temp = temp->nextarc;
                        }
                        else if (temp->adjvex == this_block ||
                            Block_Graph.vertices[neighbor_block -
1].adjvex == neighbor_block)
                        {
                            IsDuplicate = TRUE;
                            break;
                        }
                        else break;
                    }
                    //若没有重名
                    if (IsDuplicate==FALSE)
                    {
                        ArcNode* p = NULL;
                        if ((p = (ArcNode*)malloc(sizeof(ArcNode))) ==
NULL)
                        {
                            cout << "ArcNode Failed to Request memory" <<
endl;
                            exit(1);
                        }
                        p->adjvex = this_block;
                        p->nextarc = NULL;
                        temp->nextarc = p;

                    }
                }
            }
        }
    }

    if (AdjList_Sort(Block_Graph) != OK) return ERROR;
    return OK;
}

/************************************************
功能：优化建立的邻接表，根据顶点的度降序重排邻接链表
```

```
参数：Block_Graph：为图片中区块创建的邻接表
**********************************************/
int AdjList_Sort(AdjGraphList& Block_Graph)
{
    int AdjList_len = 0;
    int Block_No_List[Max_Vertex_Num];
    for (int i = 0; i < Max_Vertex_Num; i++) Block_No_List[i] = 0;
    //计算各邻接表链表长度
    for (int i = 0; i < Block_Graph.vexnum; i++)
    {
        ArcNode* temp = Block_Graph.vertices[i].firstarc;
        while (temp != NULL)
        {
            AdjList_len++;
            temp = temp->nextarc;
        }
        Block_No_List[i] = AdjList_len;
        AdjList_len = 0;
    }
    for (int i = 0; i < Block_Graph.vexnum; i++)
    {
        Block_Graph.vertices[i].Block_No = Block_No_List[i];
        ArcNode* temp = Block_Graph.vertices[i].firstarc;
        while (temp != NULL)
        {
            temp->degree = Block_No_List[temp->adjvex - 1];
            temp = temp->nextarc;
        }
    }

    for (int i = 0; i < Block_Graph.vexnum; i++)
    {
        for (int j = 0; j < Block_No_List[i] - 1; j++)
        {
            ArcNode* temp = Block_Graph.vertices[i].firstarc;
            ArcNode* shadow = NULL;//影子结点
            for (int k = 0; k < Block_No_List[i] - j - 1; k++)
            {
                //若需要调整顺序
                if (temp->degree < temp->nextarc->degree)
                {
                    //若第一个顶点序号比下一个序号小，交换
                    if (Block_Graph.vertices[i].firstarc == temp)
                    {
```

```
                        ArcNode* NewFirst = temp->nextarc;
                        temp->nextarc = temp->nextarc->nextarc;
                        NewFirst->nextarc = temp;
                        Block_Graph.vertices[i].firstarc = NewFirst;
                        shadow = NewFirst;
                    }
                    else
                    {
                        ArcNode* TempVex = temp->nextarc;
                        temp->nextarc = temp->nextarc->nextarc;
                        TempVex->nextarc = temp;
                        shadow->nextarc = TempVex;//***
                        shadow = TempVex;
                    }
                }
                //若不需要调整顺序，顺序到下一个顶点
                else
                {
                    shadow = temp;
                    temp = temp->nextarc;
                }
            }
        }
    }
    return OK;
}

/**
 * @brief  利用队列梳理最佳涂色路径
 * @param  Block_Graph: 邻接表
 * @param  Path: 涂色路径队列
 * @return 函数执行状态
 */
int SortPath(AdjGraphList& Block_Graph, LinkQueue& Path)
{
    LinkQueue Q;
    int* visited;
    if ((visited = (int*)malloc(Block_Graph.vexnum * sizeof(int))) ==
NULL)
    {
        cout << "Visited Failed to Request Memory" << endl;
        exit(1);
    }
    for (int i = 0; i < Block_Graph.vexnum; i++)
```

```
{
    visited[i] = FALSE;
    if (Block_Graph.vertices[i].firstarc == NULL)
        visited[i] = TRUE;
}
InitQueue(Q);

while (Is_AllVisited(Block_Graph, visited) == FALSE)
{
    int BlockNo;
    BlockNo = BestColorBlock(Block_Graph, visited);
    if (BlockNo == ERROR)
    {
        cout << "ERROR" << endl;
        return ERROR;
    }
    if (visited[BlockNo - 1] == FALSE)
    {
        visited[BlockNo - 1] = TRUE;

        InQueue(Path, BlockNo);
        InQueue(Q, BlockNo);
        while (EmptyQueue(Q) == FALSE)
        {
            int e;
            DeQueue(Q, e);
            ArcNode* temp = Block_Graph.vertices[e - 1].firstarc;
            if (temp == NULL) break;
            while (temp != NULL)
            {
                if (visited[temp->adjvex - 1] == FALSE)
                {
                    visited[temp->adjvex - 1] = TRUE;
                    InQueue(Q, temp->adjvex);
                    InQueue(Path, temp->adjvex);
                }
                temp = temp->nextarc;
            }
        }
    }
}

cout << "最佳填色路径为: " << endl;
LinkQueue temp = Path;
```

```
    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;
}

/************************************************
功能：根据深度优先算法，进行填色
参数：Block_Graph：为图片中区块创建的邻接表
      Path：填色路径，列表
      EdgeImg：分水岭图样
      ColorImg：重新填色图样
************************************************/
void DFS_FullColor(AdjGraphList& Block_Graph, LinkQueue Path, Mat
EdgeImg, Mat ColorImg)
{
    int pop_record[Max_Vertex_Num] = { 0 };
    // 初始化一个记录颜色是否被用的记录表
    int colors_used_matrix[Max_Vertex_Num][4] = { 0 };
    // 创建一个缓冲队列和一个中转站，前者用于存储回退出来的区域，后者用来调整
缓冲队列的顺序
    LinkQueue buffer;
    LinkQueue transfer_station;
    LinkStack Buffer;
    InitStack(Buffer);
    InitQueue(buffer);
    InitQueue(transfer_station);
    // 创建堆栈
    LinkStack stack = NULL;
    InitStack(stack);
    int e;
    // 开始涂色，要求尽可能先用同一种颜色去涂，实在冲突再换颜色
    // 如果遇到无法涂色，则回退上一步，换颜色再尝试
    // 如果涂色队列非空，即未完成涂色，则继续尝试
    while (EmptyQueue(Path) == FALSE || EmptyQueue(buffer) == FALSE)
    {
        // 如果涂色过程一直卡在某个地方，既不能继续往前涂，又不能往后退，说明
陷入了死胡同
        // 那么就往回退 50 个试试重新制定涂色方案，此过程非常重要
        for (int i = 0; i < Block_Graph.vexnum; i++)
```

```
        {
            // 如果某个结点频繁回退，即达到阈值，就直接回退50个结点
            if (pop_record[i] >= 250)
            {
                pop_record[i] = 0;
                for (int j = 0; j < 50; j++)
                {
                    int temp_area_number;
                    if (EmptyStack(stack) == TRUE) break;
                    StackPop(stack, temp_area_number);
                    // 先入中转站
                    InQueue(transfer_station, temp_area_number);
                    //StackPush(Buffer, temp_area_number, NoColor);

                    while (EmptyQueue(buffer) == FALSE)
                    {
                        int transfer;
                        DeQueue(buffer, transfer);
                        InQueue(transfer_station, transfer);
                    }

                    while (EmptyQueue(transfer_station) == FALSE)
                    {
                        int transfer;
                        DeQueue(transfer_station, transfer);
                        InQueue(buffer, transfer);
                    }
                }
            }
        }
        // 检查buffer是否为空，因为要优先处理buffer中的结点，当buffer中无结点，再从Path中取出待涂色结点
        if (EmptyQueue(buffer) == FALSE)
            //if(EmptyStack(Buffer) == FALSE)
        {
            bool left_colors_table[4] = { 0 };
            DeQueue(buffer, e);
            //StackPop(Buffer, e);
            // 只有当栈的长度大于0才去检查
            if (EmptyStack(stack) == FALSE)
            {
                // 检查此时将要涂色的区域的邻接区域的颜色是否冲突
                // 遍历现在的栈，看看是否有相邻的区域
                LinkStack search_stack = stack;
```

```
                while (search_stack != NULL)
                {
                    ArcNode* temp = Block_Graph.vertices[e -
1].firstarc;
                    // 开始从自己的邻接区域去搜索是否含有这个元素
                    while (temp != NULL)
                    {
                        if (temp->adjvex == search_stack->adjvex)
                            left_colors_table[search_stack->color] = 1;
                        temp = temp->nextarc;
                    }
                    search_stack = search_stack->next;
                }
                // 搜寻结束，现在看看还剩下什么颜色可以涂
                int tinted = FALSE;
                // 如果有颜色剩下，则进行入栈等操作
                for (int i = 0; i < 4; i++)
                {
                    if (left_colors_table[i] != 1 &&
colors_used_matrix[e - 1][i] == 0)
                    {
                        StackPush(stack, e, i);
                        colors_used_matrix[e - 1][i] = 1;
                        tinted = TRUE;
                        break;
                    }
                }
                // 如果还没有颜色剩下，证明再前面的颜色涂的也有问题，又需要回
退
                if (tinted == FALSE)
                {
                    for (int i = 0; i < 4; i++)
                        colors_used_matrix[e - 1][i] = 0;


                    int temp_area_number;
                    StackPop(stack, temp_area_number);
                    pop_record[temp_area_number - 1]++;

                    /*StackPush(Buffer, temp_area_number, NoColor);
                    StackPush(Buffer, e, NoColor);*/

                    // 先入中转站
                    InQueue(transfer_station, temp_area_number);
                    InQueue(transfer_station, e);
```

```
                while (EmptyQueue(buffer) == FALSE)
                {
                    int transfer;
                    DeQueue(buffer, transfer);
                    InQueue(transfer_station, transfer);
                }

                while (EmptyQueue(transfer_station))
                {
                    int transfer;
                    DeQueue(transfer_station, transfer);
                    InQueue(buffer, transfer);
                }
            }
        }
        // 如果栈为空
        else
        {
            int flag = FALSE;
            // 说明是第一个区域
            for (int i = 0; i < 4; i++)
            {
                if (colors_used_matrix[e - 1][i] != 1)
                {
                    StackPush(stack, e, i);
                    colors_used_matrix[e - 1][i] = 1;
                    flag = TRUE;
                    break;
                }
            }
            if (flag == FALSE)
                cout << "涂色失败! " << endl;
        }
    }


    else if (EmptyQueue(buffer) == TRUE)
        //else if(EmptyStack(Buffer) == TRUE)
    {
        // 一个涂色区域序号出队列
        DeQueue(Path, e);
        // 存放 RGBY 的使用情况，0 表示未使用，1 表示已使用
        bool left_colors_table[4] = { 0 };
        // 只有当栈的长度大于 0 才去检查
```

```
            if (EmptyStack(stack) == FALSE)
            {
                // 检查此时将要涂色的区域的邻接区域的颜色是否冲突
                // 遍历现在的栈，看看是否有相邻的区域
                LinkStack search_stack = stack;
                while (search_stack != NULL)
                {
                    ArcNode* temp = Block_Graph.vertices[e -
1].firstarc;
                    // 开始从自己的邻接区域去搜索是否含有这个元素
                    while (temp != NULL)
                    {
                        if (temp->adjvex == search_stack->adjvex)
                            left_colors_table[search_stack->color] = 1;
                        temp = temp->nextarc;
                    }
                    search_stack = search_stack->next;
                }
                // 搜寻结束，现在看看还剩下什么颜色可以涂
                int tinted = FALSE;
                // 如果有颜色剩下，则进行入栈等操作
                for (int i = 0; i < 4; i++)
                {
                    if (left_colors_table[i] != 1)
                    {
                        StackPush(stack, e, i);
                        colors_used_matrix[e - 1][i] = 1;
                        tinted = TRUE;
                        break;
                    }
                }
                // 如果没有颜色剩下，证明前面颜色涂的有问题，需要回退
                if (tinted == FALSE)
                {
                    int temp_area_number;
                    StackPop(stack, temp_area_number);
                    pop_record[temp_area_number]++;
                    InQueue(buffer, temp_area_number);
                    InQueue(buffer, e);
                    /*StackPush(Buffer, temp_area_number, NoColor);
                    StackPush(Buffer, e, NoColor);*/
                }
            }
            // 如果栈为空
```

```
            else
            {
                // 说明是第一个区域，无须考察是否冲突，直接入栈
                StackPush(stack, e, Red);
                colors_used_matrix[e - 1][Red] = 1;
            }
        }
    }


    //涂色
    Mat Img = imread("C:\\Users\\Dragon\\Desktop\\1.jpg");
    Mat Img_Gray;
    cvtColor(Img, Img_Gray, COLOR_BGR2GRAY);
    cvtColor(Img_Gray, Img_Gray, COLOR_GRAY2BGR);
    Mat FourColor_Img = Mat(EdgeImg.size(), CV_8UC3);
    for (int i = 0; i < EdgeImg.rows; i++)
    {
        int* pixel_row = EdgeImg.ptr<int>(i);
        for (int j = 0; j < EdgeImg.cols; j++)
        {
            //int pixel_data = EdgeImg.at<int>(i, j);
            //int pixel_data = EdgeImg.ptr<int>(i)[j];
            int pixel_data = pixel_row[j];
            SNode* Stemp = stack;
            if (pixel_data > 0)
            {
                while (Stemp->next != NULL)
                {
                    if (Stemp->adjvex == pixel_data)
                    {
                        switch ((Stemp->color + 1))
                        {
                        case Red:
                            FourColor_Img.at<Vec3b>(i, j) = Vec3b(0, 0,
255);
                            ColorImg.at<Vec3b>(i, j) = Vec3b(0, 0, 255);
                            break;
                        case Green:
                            FourColor_Img.at<Vec3b>(i, j) = Vec3b(0, 255,
0);
                            ColorImg.at<Vec3b>(i, j) = Vec3b(0, 255, 0);
                            break;
                        case Blue:
```

```
                            FourColor_Img.at<Vec3b>(i, j) = Vec3b(255, 0,
0);
                            ColorImg.at<Vec3b>(i, j) = Vec3b(255, 0, 0);
                            break;
                        case Yellow:
                            FourColor_Img.at<Vec3b>(i, j) = Vec3b(0, 255,
255);
                            ColorImg.at<Vec3b>(i, j) = Vec3b(0, 255,
255);
                            break;
                        default:
                            break;
                        }
                    }
                    Stemp = Stemp->next;
                }
            }
            else if (pixel_data == -1)
            {
                FourColor_Img.at<Vec3b>(i, j) = Vec3b(255, 255, 255);
                ColorImg.at<Vec3b>(i, j) = Vec3b(255, 255, 255);
            }
        }
    }
    FourColor_Img = 0.5 * FourColor_Img + 0.5 * Img_Gray;
    imshow("FourColor", FourColor_Img);
}

void Init_AreaList(vector<AreaNode>& AreaList, int len)
{
    for (int i = 0; i < len; i++)
    {
        AreaNode temp;
        temp.area = 0;
        temp.Block_No = i + 1;
        AreaList.push_back(temp);
    }
}

/************************************************
功能：计算各区块的面积
参数：Block_Graph：为图片中区块创建的邻接表
      AreaList：面积列表
************************************************/
```

```
void Area_Measure(AdjGraphList& Block_Graph, Mat EdgeImg,
vector<AreaNode>& AreaList)
{
    for (int i = 0; i < EdgeImg.rows-  1; i++)
    {
        int* data = EdgeImg.ptr<int>(i);
        for (int j = 0; j < EdgeImg.cols - 1; j++)
        {
            if (data[j] != -1 && data[j] > 0)
            {
                AreaList[data[j]].area++;
            }
        }
    }
    for (int i = 0; i < Block_Graph.vexnum; i++)
    {
        ArcNode* temp = Block_Graph.vertices[i].firstarc;
        while (temp != NULL)
        {
            temp->area = AreaList[temp->adjvex - 1].area;
            temp = temp->nextarc;
        }
    }
}

int SelectColor(AdjGraphList& Block_Graph,int *Block_Color,int
Color_BlockNo)
{
    if (Block_Color[Color_BlockNo - 1] < 0) Block_Color[Color_BlockNo -
1] = 0;
    int color0 = Block_Color[Color_BlockNo - 1];
    int Is_Settled = TRUE;
    int count = 0;
    do
    {
        count++;
        if (count == 4) cout << count;
        Is_Settled = TRUE;
        if (Block_Color[Color_BlockNo - 1] != Yellow)
        {
            Block_Color[Color_BlockNo - 1]++;
        }
        else Block_Color[Color_BlockNo - 1] = Red;
```

```
        ArcNode* temp = Block_Graph.vertices[Color_BlockNo -
1].firstarc;
        while (temp->nextarc != NULL && temp != NULL)
        {
            if (Block_Color[Color_BlockNo - 1] ==
Block_Color[temp->adjvex - 1])
            {
                //Block_Color[Color_BlockNo - 1] = color0;
                //return ERROR;//相邻区域有相同填色
                Is_Settled = FALSE;
                break;
            }
            temp = temp->nextarc;
            //else return OK;
        }
        if (Block_Color[Color_BlockNo - 1] == color0) cout <<
"InsideOut" << endl;
        if (Is_Settled == TRUE)
        {

            return OK;
        }
    } while (Block_Color[Color_BlockNo - 1] != color0);//&&
Block_Color[Color_BlockNo - 1] != Yellow);
    if (count == 4) cout << "Out" << endl;
    Block_Color[Color_BlockNo - 1] = color0;
    return ERROR;
}

int SelectColor(AdjGraphList& Block_Graph, int Color_BlockNo,int
&color, LinkStack stack,int (*color_table)[4])
{
    int left_color[4] = { 0 };
    StackNode* search_stack = stack;
    //筛选可用填色
    while (search_stack != NULL)// && search_stack->next != NULL)
    {
        ArcNode* temp = Block_Graph.vertices[Color_BlockNo -
1].firstarc;
        while (temp != NULL)// && temp->nextarc != NULL)
        {
            if (temp->adjvex == search_stack->adjvex)
            {
                left_color[search_stack->color - 1] = 1;
```

```
            }
            temp = temp->nextarc;
        }
        search_stack = search_stack->next;
    }

    for (int i = 0; i < 4; i++)
    {
        if (left_color[i] != 1 && color_table[Color_BlockNo - 1][i] ==
0)
        {
            color = i + 1;
            return OK;
        }
    }
    return ERROR;
}

void ShowBlock_Graph(AdjGraphList &Block_Graph)
{
    cout << "区块邻接表为: " << endl;
    for (int i = 0; i < Block_Graph.vexnum; i++)
    {
        cout << Block_Graph.vertices[i].adjvex << " - ";
        ArcNode* temp = Block_Graph.vertices[i].firstarc;
        while (temp != NULL && temp->nextarc != NULL)
        {
            cout << temp->adjvex << "," << temp->degree << " - ";
            temp = temp->nextarc;
        }
        if (temp != NULL && temp->nextarc == NULL)
        {
            cout << temp->adjvex << "," << temp->degree << endl;
        }
        else if (temp == NULL) cout << "NULL" << endl;
    }
}

/**
 * @brief  判断是否所有的结点都被访问过
 * @param  Block_Graph: 邻接表
 * @param  visited: 记录是否访问过的标志数组
 * @return 函数执行状态
 */
```

```
int Is_AllVisited(AdjGraphList& BG, int* visited)
{
    for (int i = 0; i < BG.vexnum; i++)
    {
        if (visited[i] == FALSE)
            return FALSE;
    }
    return TRUE;
}

/**
 * @brief  选取最优的涂色区域
 * @param  BG: 邻接表
 * @param  visited: 记录是否访问过的标志数组
 * @return 最优的涂色区域的编号，若出现故障，返回ERROR
 */
int BestColorBlock(AdjGraphList &BG,int *visited)
{
    int BlockNo = -1;
    int length = 0;

    for (int i = 0; i < BG.vexnum; i++)
    {
        if (visited[i] == FALSE)
        {
            int lenTemp = UnvistedNum_Count(BG, i + 1, visited);
            if (lenTemp > length)
            {
                BlockNo = i + 1;
                length = lenTemp;
            }
        }
    }
    if (BlockNo == -1) return ERROR;
    else return BlockNo;
}

/**
 * @brief  得到邻接表中某一链表没有被访问结点的个数
 * @param  Block_Graph: 邻接表
 * @param  BlockNo: 结点序号
 * @param  visited: 记录是否访问过的标志数组
 * @return 链表中没有被访问结点的个数
 */
```

```
int UnvistedNum_Count(AdjGraphList& BG, int BlockNo, int* visited)
{
    int length = 0;
    if (visited[BG.vertices[BlockNo - 1].adjvex] == FALSE)
        length++;
    ArcNode* temp = BG.vertices[BlockNo - 1].firstarc;
    if (temp == NULL) return length;
    while (temp != NULL)
    {
        if (visited[temp->adjvex - 1] == FALSE)
            length++;
        temp = temp->nextarc;
    }
    return length;
}
```

```
/************************************************
功能：建成大顶堆
参数：H：无序堆
************************************************/
void HeapAdjust(vector<AreaNode>& H,int s,int m)
{
    AreaNode temp = H[s - 1];
    for (int i = 2 * s; i <= m ; i *= 2)
    {
        if (i<m && H[i - 1].area>=H[i - 1 + 1].area)
            i++;
        if (temp.area < H[i-1].area) break;
        H[s-1] = H[i-1];
        s = i;
    }
    H[s-1] = temp;
}

/************************************************
功能：堆排序
参数：H：无序堆
************************************************/
void HeapSort(vector<AreaNode>& H)
{
    for (int i = H.size() / 2; i > 0; i--)//把H[1...H.len]建成大顶堆
    {
        HeapAdjust(H, i, H.size());
```

```
    }
    for (int i = H.size(); i > 1; i--)
    {
        AreaNode temp = H[0];
        H[0] = H[i - 1];
        H[i - 1] = temp;

        HeapAdjust(H, 1, i - 1);
    }
}

void HeapSort_Result(AdjGraphList &Block_Graph,Mat
EdgeImg)//,vector<AreaNode>& H)
{
    for (int i = 0; i < Block_Graph.vexnum; i++)
    {
        int center_x = 0, center_y = 0, pixel_number = 0;
        for (int j = 1; j < EdgeImg.cols - 1; j++)
        {
            int* temp = EdgeImg.ptr<int>(j);
            for (int k = 1; k < EdgeImg.rows - 1; k++)
            {
                if (temp[k] == i + 1)
                {
                    center_x += k;
                    center_y += j;
                    pixel_number++;
                }
            }
        }
        if (pixel_number != 0)
        {
            Block_Graph.vertices[i].gross.x = center_x / pixel_number;
            Block_Graph.vertices[i].gross.y = center_y / pixel_number;
        }
        else
        {
            Block_Graph.vertices[i].gross.x = center_x;
            Block_Graph.vertices[i].gross.y = center_y;
        }
    }
}

/*********************************************
```

```
功能：折半查找
参数：highNo：key 值的序号（数值更大更小）
      lowNo：key 值的序号（数值更小更大）
************************************************/
int Binary_Search(vector<AreaNode> AreaList,int vexnum,int &highNo,int
&lowNo,clock_t &time_)
{
    int max, min;
    string input;
    while (1)
    {
        try
        {
            cout << "请输入查找下界：" << endl;
            cin >> input;
            min = stof(input);
        }
        catch (exception& invalid_argument)
        {
            cout << "输入格式错误，请输入一个数" << endl;
            continue;
        }
        if (min >= AreaList[0].area) cout << "查找下界过大" << endl;
        else break;
    }
    while (1)
    {
        try
        {
            cout << "请输入查找上界：" << endl;
            cin >> input;
            max = stof(input);
        }
        catch (exception& invalid_argument)
        {
            cout << "输入格式错误，请输入一个数" << endl;
            continue;
        }
        if (max <= AreaList[AreaList.size() - 1].area) cout << "查找上界
过小" << endl;
        else if (max < min) cout << "查找上界小于下界" << endl;
        else break;
    }
    clock_t start, end;//用于计算时间
```

```
    start = clock();
    int low = 1, high = vexnum, mid;
    //先搜索 lowNo
    while (low <= high)
    {
        mid = (low + high) / 2;
        if (AreaList[mid - 1].area == max)
        {
            highNo = mid;
            break;
        }
        else if (AreaList[mid - 1].area < max)
        {
            high = mid - 1;
            highNo = low;
        }
        else
            low = mid + 1;
    }
    //再搜索 highNo
    high = vexnum;
    while (low <= high)
    {
        mid = (low + high) / 2;
        if (AreaList[mid - 1].area == min)
        {
            lowNo = mid;
            break;
        }
        else if (AreaList[mid - 1].area > min)
            low = mid + 1;
        else
        {
            high = mid - 1;
            lowNo = high;
        }
    }
    end = clock();
    time_ += end - start;

    if (highNo > lowNo)
    {
        cout << "无区域面积介于" << min << "——" << max << "之间" << endl;
        return ERROR;
```

```
    }
    else
    {
        cout << "面积介于" << min << "——" << max << "之间的区域有：" <<
endl;
        for (int i = highNo; i <= lowNo; i++)
        {
            cout << AreaList[i - 1].Block_No << ":" << AreaList[i -
1].area << endl;
        }
    }
    return OK;
}

void HighLight(vector<AreaNode> AreaList, int highNo, int lowNo,
AdjList vertices,Mat watershed_img,Mat EdgeImg, clock_t& time_)
{
    Mat img = imread("C:\\Users\\Dragon\\Desktop\\1.jpg");
    Mat img_gray;
    cvtColor(img, img_gray, COLOR_BGR2GRAY);
    cvtColor(img_gray, img_gray, COLOR_GRAY2BGR);
    Mat temp;
    clock_t start, end;//用于计算时间
    start = clock();
    watershed_img.copyTo(temp);
    for (int i = 0; i < EdgeImg.rows; i++)
    {
        int* pixel = EdgeImg.ptr<int>(i);
        for (int j = 0; j < EdgeImg.cols; j++)
        {
            int pixel_data = pixel[j];
            if (pixel_data > 0)
            {
                int flag = 0;
                for (int k = highNo; k <= lowNo; k++)
                {
                    if (pixel_data == AreaList[k - 1].Block_No)
                    {
                        flag = 1;
                        break;
                    }
                }
                if (flag == 0)
                    temp.at<Vec3b>(i, j) = Vec3b(0, 0, 0);
```

```
            }
            else if(pixel_data==-1)
                temp.at<Vec3b>(i, j) = Vec3b(255, 255, 255);
        }
    }
    for (int i = highNo; i <= lowNo; i++)
    {
        char str[6];
        sprintf_s(str, "%d", AreaList[i - 1].area);
        putText(temp, str, vertices[AreaList[i - 1].Block_No - 1].gross,
FONT_HERSHEY_PLAIN, 0.8, Vec3b(255, 255, 255));
    }
    end = clock();
    time_ += end - start;
    imshow("Binary Search", temp);
}
```