南开大学
Nankai University

南　　开　　大　　学

计　算　机　学　院

算法导论课程大作业

基于网络流和基于聚类的图像分割问题

姓名：韩佳迅

学号：2012682

年级：2020 级

专业：计算机科学与技术

2022 年 6 月 5 日

摘要

在本实验中，我们分别实现了基于网络流建模和基于聚类的图像分割程序。在网络流方面，我们基于图像建模，并实现了两种算法——EK 算法和当前弧优化后的 Dinic 算法来分别进行编程。并且，讨论了交互式分割的子问题——将单像素点并入已完成分割后的图像前景。在聚类方面，我们实现了利用 K-means 方法进行多类别图像分割的问题。

关键字：网络流，EK，Dinic，K-means

目录

一、 引言

图像分割是计算机视觉研究中的一个经典难题。所谓图像分割是指根据灰度、彩色、几何形状等特征把图像划分成若干个互不相交的区域，使得这些特征在同一区域内表现出一致性或相似性，而在不同区域间表现出明显的不同。也就是在一副图像中，把目标从背景中分离出来。而在图像分割中需要考虑的最基本问题之一是前景/背景的分割：把图像中的每个像素按照属于该场景的前景还是背景进行标记。

在当前的图像处理领域里，深度学习模型在广泛的视觉应用中取得了巨大成功，已经有大量的工作致力于开发使用深度学习模型的图像分割方法。而在本文中，我们将要实现和研究的是深度学习大火之前人们利用数字图像处理、拓扑学、数学等方面的只是来进行图像分割的算法。当然现在随着算力的增加以及深度学习的不断发展，一些传统的分割方法在效果上已经不能与基于深度学习的分割方法相比较了，但其天才的思想非常值得我们去学习，并且迁移到其他领域继续开拓创新。

二、 基于网络流的图像分割

(一) 问题的提出与分析

问题的简单描述：对输入图像的每个像素点标记为前景或背景，据此实现分割。

一幅图像本质上是由若干像素点及其像素值组成。因此，我们可以用图论知识对图像建模，我们将像素点集合记为 V，构成点阵；相邻的像素点之间由边连通，记为集合 E；此时我们就得到了可以表示该图像的连通图 $G=(V,E)$。对于每个像素点，我们有属于前景的可能性 $a_i(>0)$ 和属于背景的可能性 $b_i(>0)$。从宏观上来看，我们定义它们只是为了描述像素点在前景或背景的合适程度，而并不与任何具体的物理意义挂钩。

对于每个像素点，我们定义：如果 $a_i > b_i$，我们更倾向于标记像素点 i 为前景，否则为背景。然而，考虑到每个像素点是前景或背景还会受到其相邻像素点的影响，为了使标记更“光滑”，我们引入分离罚分的概念：对于每对相邻的像素点 (i,j)，若其中一个点在前景，而另一个在背景，则它们会有一个分离罚分 $p_{i,j}(>0)$。

基于上述定义，我们提出**分割问题的数学描述**：定义一个 $quality(A,B)$，记为 $q(A,B)$，表示某种划分方式的“价值”，它由三部分组成：所有已被划分为前景 A 的像素点 i，其属于前景的可能性之和 $\sum_{i\in A} a_i$，所有已被划分为背景 B 的像素点 j，其属于背景的可能性之和 $\sum_{j\in B} b_j$，以及相邻像素被不同划分的分离罚分之和 $\sum_{\substack{(i,j)\in E\\ |A\cap(i,j)|=1}} p_{ij}$，而我们则需要找到一个把像素集合分成 A（前景）和 B（背景）的划分，使得

$$q(A,B)=\sum_{i\in A} a_i+\sum_{j\in B} b_j-\sum_{\substack{(i,j)\in E\\ |A\cap(i,j)|=1}} p_{ij} \tag{1}$$

达到最大，此时的划分就是一个最优的标记策略。

(二) 算法的设计

在数学描述的过程中，我们注意到，处理分割问题实际上也是一个在图 G 上求目标函数最值的问题，这与最大流有着相似之处。因此，我们考虑使用网络流建模，将分割问题抽象为网络流问题。[1] [2]

首先，为了将前/背景的可能性 (a_i/b_i) 与网络流容量结合，我们定义一个“源点” s 表示前景，“汇点” t 表示背景，并把 s 和 t 都分别与每个像素点联系起来，使用 a_i 和 b_i 分别定义像素点 i 与源点和汇点边上的容量。并且，定义有向边的方向分别为从 s 到 i，以及从 i 到 t（为了后面的“最小割”用）。

其次，考虑到相邻像素点之间的分离罚分 $(p_{i,j})$，我们把每对相邻像素点 (i,j) 用两条有向边 (i,j) 和 (j,i) 相连，并且赋其容量均为分离罚分 $p_{i,j}$。

最后，我们需要解决最值转换的问题。我们的目的是把割和目标函数结合起来，并最大化目标函数。我们已知网络流中有“最大流最小割”算法，但它是一个割最小化的问题，因此，我们需要对目标函数 $q(A,B)$ 进行调整:

我们定义 $Q=\sum_i(a_i+b_i)$，则有

$$\sum_{i\in A}a_i+\sum_{j\in B}b_j=Q-\sum_{i\in A}b_i-\sum_{j\in B}a_j \tag{2}$$

由 (1) 和 (2) 式，我们可以将 $q(A,B)$ 转换为

$$q(A,B)=Q-\sum_{i\in A}b_i-\sum_{j\in B}a_j-\sum_{\substack{(i,j)\in E\\ |A\cap(i,j\rangle|=1}}p_{ij} \tag{3}$$

要想让 (3) 式达到最大，等价于让下面的 $q'(A,B)$ 最小:

$$q'(A,B)=\sum_{i\in A}b_i+\sum_{j\in B}a_j+\sum_{\substack{(i,j)\in E\\ |A\cap(i,j\rangle|=1}}p_{ij} \tag{4}$$

因此，我们选择 $q'(A,B)$ 为新的目标函数进行建模。

基于上面提到的三点，我们建立模型如图9

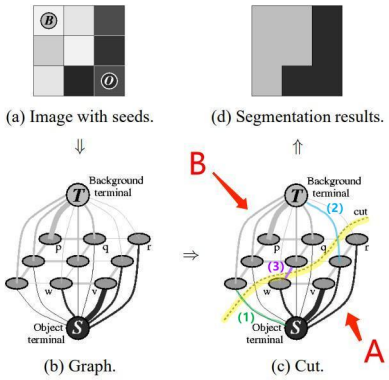

图 1

对于每一个 $s-t$ 割 (A,B)，它都将像素点分为 A、B 的一个划分，割的容量恰好由 (s,j)、(i,t)、(i,j) 三部分组成，分别对应 B 集合中的点属于前景的贡献 $\sum_{j\in B}a_j$、A 集合中的点属于背景的贡献 $\sum_{i\in A}b_i$、相邻节点被 AB 拆开的贡献 $\sum_{\substack{(i,j)\in E\\ |A\cap(i,j\rangle|=1}}p_{ij}$。（在图9中表示为 (1)(2)(3)）

所以我们得到 $s-t$ 割 (A,B) 的容量 $C(A,B)$ 满足 $C(A,B)=q'(A,B)$。此时，我们就成功地将分割问题转为最大流最小割问题。

(三) 算法实现与分析

1. 建模部分

首先，对于每幅输入的图像，我们在建模过程中需要解决以下问题：

1. 像素点的提取：对于彩色图像来说，它由 RGB 三色通道构成。在本实验中，为了简化问题，我们提取它的灰度图来进行计算。调用 C++ 中的 OpenCV 库，用其库函数先将图像转为灰度图，然后提取每个像素点的灰度值，依次存入 int 型数组 points 中。

2. a_i 和 b_i 的确定：之前我们提到，对于不同的图像来说，a_i 和 b_i 确定的方法并不唯一，而是由其属于前/背景的可能性来规定。因此，在本程序中，我们借鉴阈值分割的思想：使用灰度图像素值大小来估算其属于前/背景的可能性。(这只是为了简化问题难度，在具体的应用中，a_i 和 b_i 的确定通常是将像素 p 的灰度值与给定前景区域的灰度直方图进行比较来计算，获得概率值，再进行对数变换等。)

3. 相邻像素点的定义：在图像处理领域，相邻像素点有多种定义方式。在本实验中，我们采用 4 邻域的相邻定义，即对于任何一个不在图像边缘的像素点 (i, j) 来说，与它有边相连的四个点是 $(i-1, j)$、$(i, j-1)$、$(i+1, j)$、$(i, j+1)$。

4. 有向图的构建：我们将二维的坐标点转为一维，每个像素点用其对应在一维的下标来标识，再加上源点和汇点构成了所有点的 index。

(1) 邻接表的结构：构建结构体 edge，存储边的边尾（to)、边权（val)、兄弟边（net)。使用一个 head 数组来存储起点 u 的最后一个邻接边在 edge 型数组 e 中的下标，存到 head[u] 里。使用结构体 edge 中的 net 变量存储与边 e[tot] 共起点的上一条边。这样就把每个点的邻接表都用标记数组 head 和边数组 e 连了起来。

(2) 反向边的引入：由于网络流算法要涉及剩余图，因此每条边都会有对应的反向边。创新点：在存储时，我们设计将正向边和反向边“成对存储”，都打包放到 void add() 函数里，即每对边在数组 e 中都是相邻的。这样我们在更新边权时就会很方便，直接使用异或 1 的方式，即可将正反边转换（因为奇数异或 1 相当于-1，偶数异或 1 相当于 +1)。

图的构建

```
void add(int u, int v, int w) {
        e[++tot].to = v;
        e[tot].val = w;
        e[tot].net = head[u];
        head[u] = tot;

        e[++tot].to = u;
        e[tot].val = 0;
        e[tot].net = head[v];
        head[v] = tot;
}
```

2. EK 算法

在课程中，我们学习了 FF 方法，但仍存在比 FF 更好的强多项式算法来解决最大流问题。EK 算法就是利用 FF 思想优化的算法。

EK 算法流程：

1. 寻找最短路径：在剩余图上通过 BFS 拓展合法节点，并找出一条从源点到汇点的最短路径，记录每个节点的前面节点。若 BFS 能遍历到汇点，则算法继续；否则（即找不到增广路径)，

算法结束。

2. 最大流的更新：记录下路径中每条弧流量的最小值 min, 最大流 += min（否则就会超出某条边的限制流量）

（在程序中，我们使用 dis 数组来记录路径到达每个点时的“最细流”，当遍历到汇点 t 时，dis[t] 自然就是 min 了）

3. 更新剩余图: 通过 BFS 的记录, 从汇点 t 回溯回源点 s, 将所有经过的边的流量减去 min, 其反向边加上 min。

（在程序中，我们使用邻接表和结构体数组记录了每个点的正向和反向边，在回溯的时候，直接定位即可）

4. 重复上述步骤，直到找不到增广路径，算法结束。

通过上述步骤，我们能够得到建模网络中的最大流的值。此时，再调用一次 bfs，从源点出发遍历整张图，路径能达到的点就是集合 A 中的点，否则是集合 B 中的点。这样，我们就把所有像素点都分割开了。

Algorithm 1 *Edmonds−Karp* 算法伪代码

Input: *Graph* G，源点 s，汇点 t
Output: 最大流 *result*

```
function EK(G, s, t)
    result ← 0
    while there exists a BFS path from s to t do
        flow ← min(every edge in path)
        total ← total + flow
        update G according to flow
    end while
end function
```

复杂度分析：

相比我们学过的 FF 算法，EK 算法的优势在于它确定了在剩余图上寻找最短路径的方法，即 bfs，因为 bfs 的特性，遍历到终点的路径一定是最短路，能够大幅减少 FF 走过的“弯路”，提高收敛速度，更快地找到最大流。

从上面的伪代码中分析，算法的**时间复杂度**与增广次数和每次增广所用时间密切相关:

1. 一次增广的过程包括 bfs 找最短路径、找路径中的流选——最细流、回溯更新剩余图。每次 bfs 操作导致的边数增长不会使总边数超过原来的 2 倍，即算法的任意时刻总边数 $< 2|E|$。一次 BFS 的时间复杂为 $O(2|E|)$ 即 $O(|E|)$。又由于我们采用邻接表存储，并且在每次 bfs 的时候都用数组 $dis[]$ 记录了每个点的最细流，因此这部分的时间复杂度就是 $O(E)$。

2. 增广次数 =bfs 调用次数。而这一部分的复杂度是 $O(|V||E|)$。这是由于：EK 算法通过 bfs 寻找的增广路径长度非递减，所以 (u, v) 第二次成为饱和边时 s 到 u 的最短距离至少比前一次成为饱和边时大 2。而 s 到任意顶点的距离最多不超过 $|V| - 1$ 故 (u, v) 可以成为饱和边的次数最多为 $(|V| - 1)/2$。又由于每次增广至少有一条边成为饱和边，且 EK 算法过程中边数 $< 2|E|$，故考虑，所有边的总的增广次数必小于 $2|E| * (|V| - 1)/2$ ，则增广次数复杂度为 $O(|V||E|)$。

所以 EK 算法总的时间复杂度是 $O(|V||E|^2)$。

而对于**空间复杂度**来说，我们的图片大小是 $n\_size \times n\_size$ 的，定义问题规模 n 等于 $n\_size \times n\_size$。由于顶点数 V 就等于像素点个数 +2，即 $n\_points = n\_size \times n\_size + 2 = O(n)$，边数 E 即 $n\_edges = 6 \times n\_size \times n\_size2 - 4 \times n\_size = O(n)$。我们在图的建模存储阶段，使用 $O(n)$ 的 int 型数组 points 来存储像素值，edge 型数组 e 存储边，int 型数组 head

存储邻接表。在 bfs 阶段，还使用了 int 型数组 pre 存储前向边，int 型数组 dis 存储通过各点的最细流，队列 q 进行 bfs 遍历，且这些结构都声明为全局变量，故空间复杂度保持在 $O(n)$。

3. Dinic 算法与对其的优化

我们注意到，上述提到的 EK 算法的时间主要受限于边 E 的大小。虽然 EK 相比 FF 算法优化了，但具体到本实验中我们要处理的图像问题，主要还是针对稠密图，因此我们考虑对 EK 算法的进一步优化——Dinic 算法。相比 EK 算法以 BFS 方式实现 FF 方法中的增广操作，在 Dinic 算法中，采用 BFS 和 DFS 结合的方式增广，提高了增广的并发度。

首先我们引入**层次**和**分层图**的概念。BFS 算法从 s 到 t，标记 s 为第 1 层，s 的邻接顶点为第 2 层，以此类推，t 为最后一层。执行一次 BFS，赋予每个顶点层次信息，此时的图称作分层图。在此分层图内以 DFS 方式反复寻找增广路并执行增广操作 (找到饱和边，前向和反向流)，累计前向流。完成当前分层图的所有增广操作称作一个阶段。一个阶段结束后，重置高度标号信息，再次执行 BFS 得到新的分层图并重复 DFS 的增广操作，直到无法分层时说明 s 到 t 已无增广路径，算法结束，此时得到的前向流总和即为 s 到 t 的最大流。

Algorithm 2 *Dinic* 算法伪代码

Input: *Graph G*，源点 s，汇点 t
Output: 最大流 res

```
1: function DFS(x, sum)
2:     res, k ← 0
3:     if x == t then
4:         return sum
5:     end if
6:     for i = head[x]; i&&sum; i = next of e[i] do
7:         v ← 边 i 的尾节点
8:         if value of edge i > 0 and dis[v] == dis[x] + 1 then
9:             k ← DFS(v, min(sum, value of edge(i)))
10:            update value of both forward and reverse edge of i according to k
11:            res ← res + k
12:            sum ← sum − k
13:        end if
14:    end for
15:    return res
16: end function
17:
18: function Dinic
19:    while bfs() do
20:        ans ← ans+ DFS(s, inf)
21:    end while
22:    return ans
23: end function
```

然而，我们发现，在 dfs 寻找增广路径的过程中，如果找到了，那么会直接先朝增广路流过去，一直流到汇点。在这个过程中，必然有一条边达到了满流，即流不能再经过这条边，那么我

们找到的该条增广路径也就不能跑了，但是在下一次增广的过程中，如果再次到达这个点，而找到了不能增广的路进行增广，就会徒增很多的时间复杂度。

所以我们引入当前弧优化和剪枝操作，在本程序中定义一个数组 $now[]$，记录每个点下一次该走的边（因为一个点可能有多条边可走，而在前面的一些边可能已经把当前流量花光，而且那些边也不再能增广，流量花光就直接退出了循环，也就是对其进行剪枝操作），这样就能省下不少的时间复杂度。而代码的改进具体体现为在 bfs 中添加其初始化，在 dfs 中将 $now[]$ 作为 $head$，直接从 now 开始 dfs，并进行实时更新和剪枝。相关改进如下：

Algorithm 3 *Dinic* 优化算法伪代码

Input: *Graph* G，源点 s，汇点 t

Output: 最大流 res

```
function BFS(x, sum)
    //添加代码：在遍历过程中令 now[v] ← head[v];
end function

function DFS(x, sum)
    res, k ← 0
    if x == t then
        return sum
    end if
    for i = now[x]; i&&sum; i = next of e[i] do
        now[x] ← i //当前弧优化
        v ← end of edge i
        if value of edge i > 0 and dis[v] == dis[x] + 1 then
            k ← DFS(v, min(sum, value of edge(i)))
            if k == 0 then
                dis[v] ← inf //剪枝，去掉增广完毕的点
            end if
            update value of both forward and reverse edge of i according to k
            res ← res + k
            sum ← sum − k
        end if
    end for
    return res
end function
```

复杂度分析：

在空间复杂度上，EK 算法和 Dinic 算法没有太大区别，二者空间复杂度相同，不再赘述。而时间复杂度上，Dinic 算法由于引入了分层图，以及我们进行的剪枝、当前弧优化操作，使得 Dinic 算法要优于 EK 算法。

从伪代码分析，Dinic 算法的复杂度由分层图建立时间及其建立次数、每建立一次分层图后（每一阶段）的 DFS 增广时间和增广次数决定：

1. 总的 DFS 复杂度：

（1）一次 DFS 增广：$O(|V|)$

在分层图中，由于层标号加一的判断条件限制，DFS 只能沿着层高递增的方向从 s 到 t 推

进，因此单次 DFS 推进次数最多不超过 |V|-1，时间复杂度为 O(|V|)。

（2）一个阶段中 DFS 增广次数：$O(|E|)$

增广路径只能沿着层高递增的方向，每次 DFS 增广的结果使得一条层高递增方向上的饱和边 (u,v) 消失，而其反向边 (v,u) 增加，但 (v,u) 是层高递减的，因此 (u,v) 被删除后，无论是其正向边还是反向边都不会再次被 DFS 推进并删除，因此一个阶段内 DFS 的次数不会超过边数 |E|，即 $O(|E|)$。

2. 阶段数及其分层图建立时间

（1）BFS 建立分层图: $O(|E|)$

（2）阶段数 (分层图建立次数): $O(|V|)$

在层高递增条件的限制下，s−t 最短路径长度即 t 的层高，因此最短路径最大不超过 $|V|-1$，而每次分层图使得 s−t 最短路径长严格递增，则分层图建立次数的上限为 $|V|$，复杂度为 $O(|V|)$。

因此，每个阶段的复杂度是建立分层图的复杂度 $O(|E|)$ 加上在该分层图内执行的总 DFS 复杂度 $O(|V||E|)$，即 $O(|E|+|V||E|)$，而阶段数是 $O(|V|)$，因此，总的复杂度就等于这两者相乘，$O((|E|+|V||E|)\times|V|)$ 即 $O(|V|^2|E|)$。

因此，由于本实验的图像分割问题中的图是稠密图，采用 Dinic 算法可能会有更好的效果。

（四） 效果展示与分析

在附件中，我们分别提供了使用 EK 算法和优化版 Dinic 算法实现的图像分割代码，我们对本程序进行如下说明：

1. 由于读取像素点以及绘图操作需要调用 OpenCV 库，因此，需要在电脑上配置 OpenCV 环境才能正常运行。可以发现工程文件夹下有两张笔者已配置好的属性表（*OpenCV_Debug.props* 和 *OpenCV_Release.props*），分别为 debug 和 release 模式在 x64 下的 OpenCV 配置。但这两张表的 dll 文件路径等是按照笔者电脑配置的，因此，若需要运行程序，请先把更改属性表并配置系统环境。

2. 本程序代码只针对 1：1 的图像，即图像规模是 $n\_sizen \times \_size$。

3. 本程序可调整的参数有分离罚分 p 和每个像素点的 a_i、b_i。笔者在实现时，是根据像素值的大小确定的 a_i 和 b_i，事实证明，不同的 p、a_i、b_i 都会对分割结果产生影响。

4. 程序中 OpenCV 读取图片的路径是绝对路径，不同电脑运行时需修改路径。

由于 EK 算法和 Dinic 算法都是网络流算法，两者对图像分割的结果一致，均表现为如下情况：

示例一：实景图——鼠标

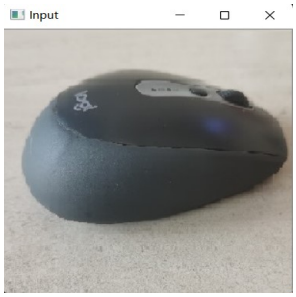

图 2: 原图

图 3: 灰度图

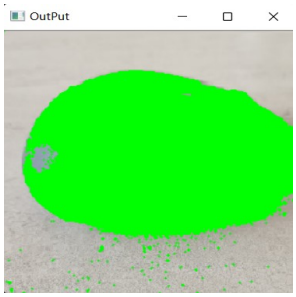

图 4: 分割后

在示例一中，我们调整像素点的灰度阈值来确定 a_i 和 b_i，最终得到将鼠标大致区分出来的效果。（图中绿色为标记的所有前景像素点）

示例二：卡通图

图 5: 原图

图 6: 灰度图

图 7: 分割图 1.0

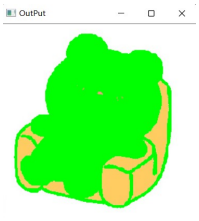

图 8: 分割图 2.0

可以看到，相比于示例一，示例二图像的难度要更大，因为它的色彩层次更多。在 1.0 版本中，我们只能分出目标的边缘。因此，我们调整像素点的 a_i 和 b_i，最终得到了 2.0 的版本，可以大致标记出前景。

通过上述示例，我们发现，我们程序的局限性在于对 a_i 和 b_i 的确定上。但这一部分的调整主观性较强，不作为本次大作业算法研究的重点。并且需要注意的是，a_i 和 b_i 是根据不同的场景决定的，在具体的应用中，应选择合适的方法来决定。

综上，经程序验证，我们的网络流模型可以起到图像前/背景分割的效果。

三、 拓展问题研究——交互式分割

在上面的讨论中，我们发现，上述建模过程是自动分割前景和背景。但在实际应用中，常常出现程序依据算法将某像素点标记为背景，但实际上用户知道它应该在前景中。比如，在对人的照片进行分割的时候，人手里拿着手机，用户知道手机应是前景的一部分，但程序却将手机标记为背景。因此，交互式分割被提出。它的适用场景主要是：在已有的一组图像分割结果中，某像素点被划分为背景，但根据语意，用户需要将此像素点放到前景里面。

分析上述问题，我们不难发现，要想将该点放到前景，最简单的方法就是重新调整 a_i 和 b_i，再跑一遍程序。但是，此时我们之前跑出来的分割结果将全部作废，会大大浪费时间，降低效率。因此，我们考虑一种更高效的方式来解决上述问题。

首先，我们需要探究如何将该问题抽象到我们的网络流模型里。将某个像素点指定到前景，实际上就是改变它在前景中的可能性 a_i 和背景中的可能性 b_i。我们考虑到，增大 a_i 实际上是增大边 (s,i) 的容量，假设 (s,i) 增加到某个足够大的值 λ，此时对于任意一个不过 (s,i) 的割 C 来说，C 的值一定小于 C'，C' 与 C 除 (s,i) 之外，割开的边都一样。因此，我们可以知道，只要我们把 a_i 增加到足够大，它就一定能被划分到前景。而对于这个足够大的值 λ 来说，我们规定所有 a_i、b_i、$p_{i,j}$ 的上限是 d，那么 $\lambda = 5d+1$ 即可满足条件。因为每个像素点最多有 5 条邻接边（4 个邻居 +t），因此 $\lambda = 5d+1$ 要大于所有从像素点 i 引出的边容量。

因此，我们抽象出的问题就是，如何在已有某个分割结果的前提下，再提高网络流中某边的容量，得到新的最大流最小割划分结果。

首先，我们考虑，若在原网络流的基础上，只给 a_i 增加 1 个单位，其余均不变的情况。此时，我们根据最大流最小割定理很自然地想到，若原最小割未跨过 a_i，则结果不变；否则，最多在原来的基础上加 1。因此，新网络流的最大流，要么等于原最大流，要么是原最大流的基础上加 1。

根据上述推测过程，我们提出：记原网络流被成功分割后的剩余图是 G_f，在剩余图 G_f 的基础上，给 a_i 对应的正向边和反向边分别增加和减少 1 个单位记为剩余图 G'_f。再在图 G'_f 上找一条从 s 到 t 的增广路径，若找不到，说明原最大流 f 就是新最大流；若找得到，则新路径的最大流的流量一定是 $f+1$，即为最终的结果。此时，我们只需要一次增广就可以完成任务，复杂度是 $O(|V|+|E|)$，也就是 $O|V|$。

其次，我们考虑把给 a_i 增加多个单位的情况：实际上我们需要进行 $(5d-1)-a_i$ 次的“+1”，

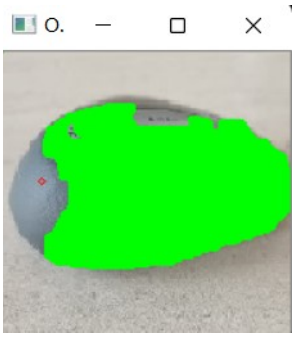

图 9: 初次划分的图

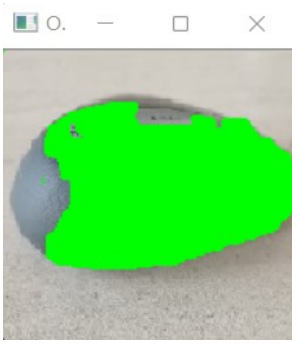

图 10: 指定像素点后划分的图

也就是 $O(d)$ 次的 $O(|V|)$，因此，总的复杂度是 $O(d|V|)$，当 d 较小时，要胜过重新计算的复杂度 $O(|V|^2|E|)$。

在具体的代码实现时，只需要在原来的分割结束之后添加 $(5d-1)-a_i$ 次的边 i 权值更新和基于新剩余图图的一次增广即可。

Algorithm 4 交互新增代码

1: **for** $j=0 \to 5d-a_i$ **do**
2: $value\ of\ edge\ (s,i)+=1$
3: $value\ of\ reverse\ edge\ (i,s)-=1$
4: **while** $bfs()$ **do**
5: $ans \leftarrow ans+ \text{DFS}(s,inf)$
6: **end while**
7: **end for**

实验结果如图9和10所示，其中图9用红色标记出了用户要指定的前景像素点，图10则是用交互程序跑出来的划分结果。可以看出，最后的划分结果中，该点确实被指定为前景了。

再看程序运行过程中，每次增广后的最大流变化，如图11。可以看到，前半段的单次增广过程中，最大流每次都会加 1，后半段，最大流保持不变，即使增加 a_i 也不会有新的增广路径了。

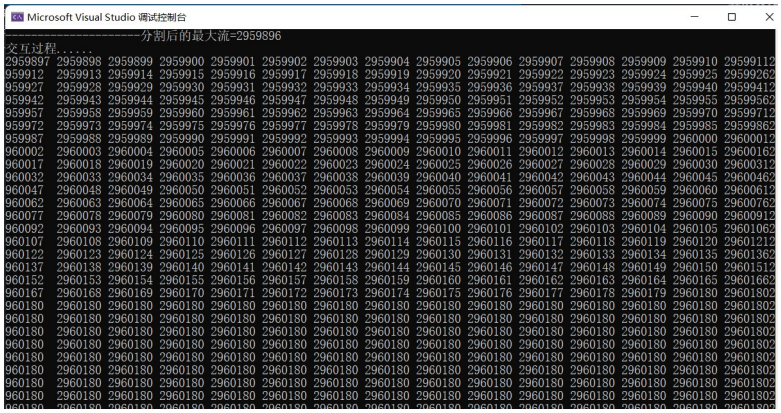

图 11: 增广过程

图 12: 原图

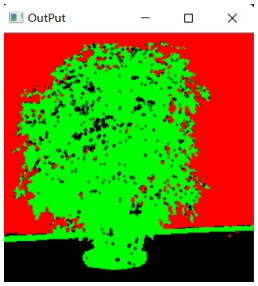

图 13: 分割图 1

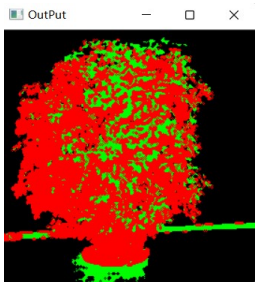

图 14: 分割图 2

四、 基于机器学习的图像分割

在上面的讨论中，我们认识到，这种最大流建模方式只能简单地分出前景和背景两类，要想分出三个以上的类别，还需要更复杂的网络流建模思想。

假设 λ 为类别数，每个像素点都有可能被分为 λ 个类别中的任何一个。问题的关键在于，最大流最小割算法只能将图像分割成两部分，但我们需要 λ 部分。因此原来的像素点与有向图顶点一一对应、边容量与 a_i、b_i 一一对应的模型不再适用。所以，我们考虑将每个像素点至少拆成 λ 个顶点，这样一个像素点至少对应 $\lambda+1$ 条边，每条边才能赋予不同类别的可能性。

从上面的假设中，我们发现，这样会将原本就不小的问题规模又扩大了至少 λ 倍，且随着图像越复杂，问题的难解度越高。实际上，多类别的图像分割也是一种分类问题，因此，我们可以使用 K-means 聚类的方法进行多类别分割。

K-means 聚类的算法流程如下：

Algorithm 5 K-means

选择 K 个点作为初始质心
repeat
    将每个点指派到最近的质心，形成 K 个簇;
    重新计算每个簇的质心;
until 簇不发生变化 **or** 达到最大迭代次数

为了将 K-means 应用到图像分割，我们做了如下调整：

1. 定义结构体 point 类型，用来存储各个像素点的横纵坐标和权值；

2. 将图像转为灰度图，读取各像素点的灰度值，将所有像素点存入结构体 point 类型的一维数组 points 里（定义权值 = 灰度值）；

3. 定义两点之间的距离等于它们灰度值之差的平方；

与网络流建模类似，在 K-means 聚类中，我们也是利用像素点灰度值进行分割，这一部分也可以有其他的定义方式（如 RGB 三值等），此时也要根据数据维度的不同，更换不同的距离函数。

对于复杂度来说，K-means 聚类的时间复杂度为 $O(m \times n \times k \times I)$，其中 I 为迭代次数，m 为点的个数，n 为特征的个数，k 为分类个数（簇的个数）；它的空间复杂度为 $O((m+k)n)$。对于聚类问题来说，它的时间复杂度很大程度上取决于算法收敛速度和初始质心的位置，而并不像最大流算法一样，能达到多项式级别的复杂度。

两组实验结果如图12、13、14；图15、16、17。

从实验结果我们可以看出，k-means 算法的受随机初始化质心的影响，每次执行出来的效果并不相同（比如图13和14，图15和16），结果的随机性强，稳定性低，并非每次都能达到好的效

图 15: 原图

图 16: 分割图 1

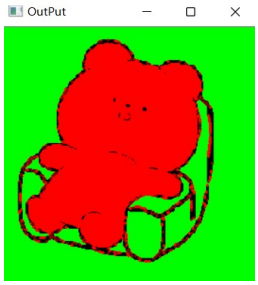

图 17: 分割图 2

果。但它的优点在于，分割的精度比网络流高，划分的类别也要比网络流多。

五、 总结

在本次实验中，我们实现了基于网络流建模的图像分割程序，并且用了两种算法——EK 算法和当前弧优化后的 Dinic 算法来分别实现编程。基于网络流模型，我们还讨论了交互式分割的子问题——如何不重跑程序地将单像素点并入已完成分割后的图像前景。并且，我们还讨论并实现了利用 K-means 聚类方法进行多类别图像分割的问题。

参考文献

[1] Yuri Y Boykov and M-P Jolly. Interactive graph cuts for optimal boundary & region segmentation of objects in nd images. In *Proceedings eighth IEEE international conference on computer vision. ICCV 2001*, volume 1, pages 105–112. IEEE, 2001.

[2] 张立昂 and 屈婉玲. **算法设计**. 清华大学出版社, 2007.