

使用最大流最小割思想分割图像

一、 算法概述

对于一个带边权的有向图 $G = \langle V, E \rangle$ (如图 1 所示), 如果存在一个边集 E_1 , 从 G 中删去 E_1 中所有边能使 G 不再连通 (即分成两个子图), 则 E_1 称为图的割 (cut)。 G 的所有割中边权重和最小的一个割称为 G 的最小割。比如图 1 中, $\{s \rightarrow a, b \rightarrow t\}$ 即为 G 的最小割。

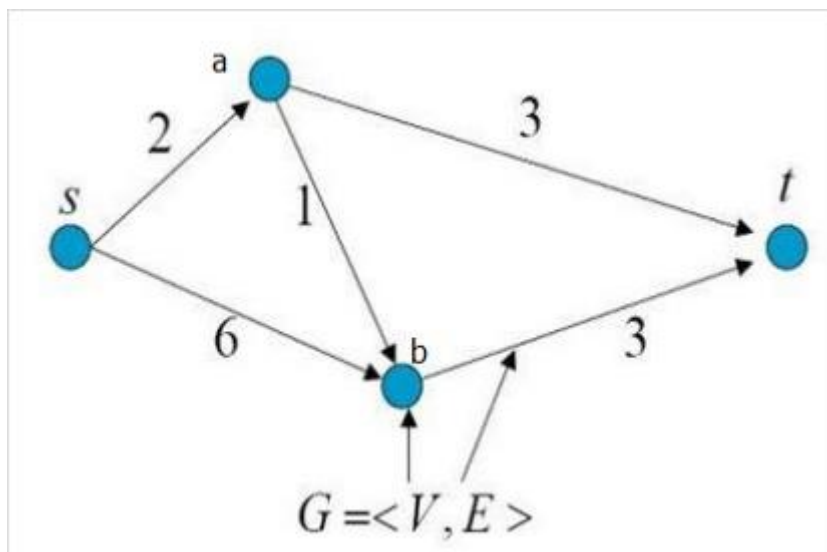


图 1 一个有向图的例子

对于一幅图像 M , 可以将其转化为类似图 1 的一个有向图 G , G 的节点对应 M 中的像素点, G 中连接相邻节点的边的权重则根据 M 中相邻像素的差异来赋值。

当我们想分割一幅图像 (这里只讨论将图像分成两部分: 前景和背景) 时, 我们可以将该图像转为一个图, **相邻像素差距越大, 则转化到图上相邻节点的边权重就越小**。然后运用最小割算法对图进行分割。这样, 最小割会切断图中权重较小的边, 也即切断原图像中差异较大的邻近像素, 即切断原图像中的边缘。最后, 我们只需从图的源点进行广度搜索, 得到切割后仍与源点相连的点, 这些点对应的像素点集合就组成了原图像切割后得到的前景。

可以证明, 一个图的最大流和最小割结果是等价的。因此, 我们可以通过最大流算法来求最小割。本次作业主要完成了以下工作:

1. 通过最大流最小割思想对图像进行分割。
2. 阅读 Graphcut [1] 论文, 参考了论文中的思想, 包括种子节点的选取, 以及边权值设定方法。
3. 实现了 Dinic 算法作为最大流算法。
4. 复用了 github 上一个项目 [2] 的 UI 界面, 用于鼠标取点和图像显示。

二、 算法流程

算法首先将图像转化为有向图，然后在有向图上运行最大流算法得到最小割，最后根据有向图的分割结果转化为输出图像。接下来分别介绍这几个部分。

图像转图

主要包含三个步骤：

1. 添加节点到图中

```
1. g = myMaxflow()
2. g.set_nodes(node_num)
```

2. 计算每个像素点与源点和汇点的边权重，并添加边到图中

首先分别通过前景标记点和后景标记点计算各自的颜色直方图（可以先转灰度），进而得到概率密度函数 $H_F(x)$ 和 $H_B(x)$ 。

如果像素点被标记为背景，则将其与源点的边权重设为 0，与汇点的边权重设为无穷大。如果像素点被标记为前景，则将其与源点的边权重设为无穷大，与汇点的边权重设为 0。如果像素点没有被标记，则将其与源点的边权重设为 $-\ln(H_B(x_0))$ ，与汇点的边权重设为 $-\ln(H_F(x_0))$ ，其中 x_0 为像素值。

伪代码如下：

```
1. H_B, H_F = cal_hist(marks)
2. for pixel in image:
3.     if pixel belongs to background: # 前景标记点
4.         g.add_tedge(pixel, 0, float('inf'))
5.     else if pixel belongs to foreground: # 后景标记点
6.         g.add_tedge(pixel, float('inf'), 0)
7.     else: # 非标记点
8.         g.add_tedge(pixel, -ln(H_B(pixel)), -ln(H_F(pixel)))
```

3. 计算每个像素点与相邻像素点的边权重，并添加边到图中

对于图中每个节点，计算其与右方节点和下方节点的相连边权重，计算方法如 (1) 式所示。

$$B_{\{p,q\}} \propto \exp\left(-\frac{(I_p - I_q)^2}{2\sigma^2}\right) \cdot \frac{1}{dist(p,q)}. \quad (1)$$

简化代码如下：

```

1. for (i,j), value in image:
2.     weight = exp(-1 * (image[i,j]-image[i,j+1])^2) / 2)
3.     g.add_edge((i,j), (i,j+1), weight)
4.     weight = exp(-1 * (image[i,j]-image[i+1,j])^2) / 2)
5.     g.add_edge((i,j), (i+1), weight)

```

需要注意的是，add_edge 和 add_tedge 的不同在于，前者添加的边是双向的，且两个方向的权重相等。

最大流

经过以上步骤，我们得到了一个有向图，接下来介绍在该图上运行的 Dinic 算法。

Dinic 算法基于 Ford-Fulkerson 算法，能够更快地完成计算。其改进点是将图中节点按照离源点的路径长度（每条边的长度为 1）分层，并且在找最大流增广路径的时候规定：一个节点只会搜索下一层的节点，即使与其他层的节点有边连接，也不会搜索其它层的节点。图 2 展示了对图 1 进行分层的结果。

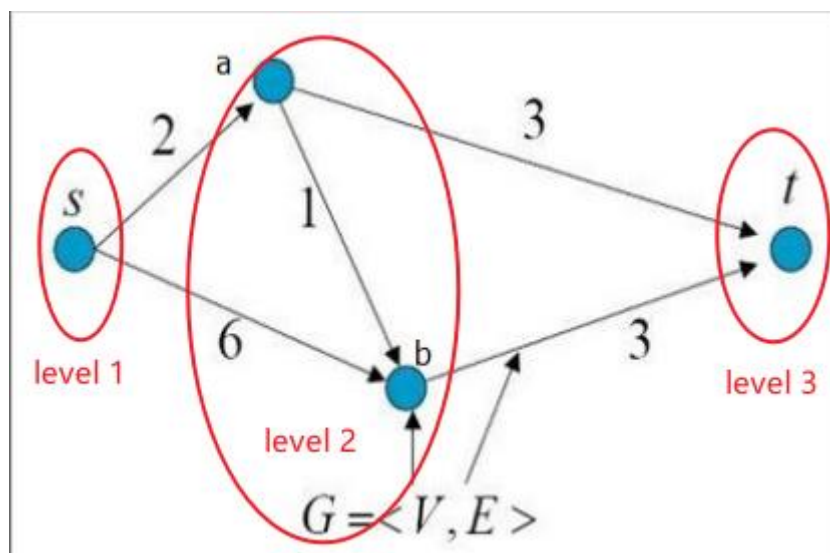


图 2 运用 Dinic 算法对图 1 的节点进行一次层次划分的结果

当从源点搜索到 a 时，由于 b 的深度与 a 的深度相同，所以 a 不会走 a -> b 这条边。

通过这样设计，Dinic 总是可以先找到那些最短的增广路径。如果较短的路径无法再增加流，Dinic 则重新进行一次广度优先搜索，寻找较长的增广路径。最后，当再也无法找到一条从源点到汇点的增广路径时，算法结束。

先走较短路径的好处是，需要搜索的深度较小，可以更快地把一些边容量耗尽。在之后搜索较长路径时，需要走的分支就会少一些。

Dinic 简化代码如下：

```

1. def maxflow():
2.     total = 0
3.     while(True):
4.         sink_level = bfs()
5.         if sink_level > 0:
6.             flow = dfs(source, float('inf'))
7.             total += flow
8.     return total
9.
10. def bfs():
11.     # 每次广度优先搜索重新为节点赋深度值
12.     levels = [0] * node_num
13.     queue = [source]
14.     while queue:
15.         cur = queue.pop()
16.         for nei in neis[cur]: # 遍历相邻节点
17.             if levels[nei] == 0 and capacity[cur][nei] > 0:
18.                 # cur -> nei 有剩余容量, 该路径才可行, 否则忽略
19.                 levels[nei] = levels[cur] + 1
20.                 queue.append(nei)
21.                 if nei == sink:
22.                     return levels[sink]
23.     return 0
24.
25. def dfs(cur, inbound):
26.     # 根据 bfs 划分的层次寻找增广路径
27.     if cur == sink:
28.         return inbound
29.     outbound = 0
30.     for nei in neis[cur]:
31.         if levels[nei] == levels[cur]+1 and capacity[cur][nei]>0:
32.             # 路径上的最小边的容量即为该路径本次能通过的最大 flow
33.             f = dfs(nei, min(inbound-outbound, capacity[cur][nei]))
34.             outbound += f
35.     return outbound

```

最小割

得到最大流后，只需从源点进行一次广度优先搜索，即可得到与源点连接的点集。

简化代码如下：

```

1. queue = [source]
2. is_foreground = [False] * node_num
3. while queue:
4.     cur = queue.pop()
5.     is_foreground[cur] = True
6.     for nei in neis[cur]:
7.         if not is_foreground[nei] and capacity[cur][nei] > 0:
8.             queue.append(nei)

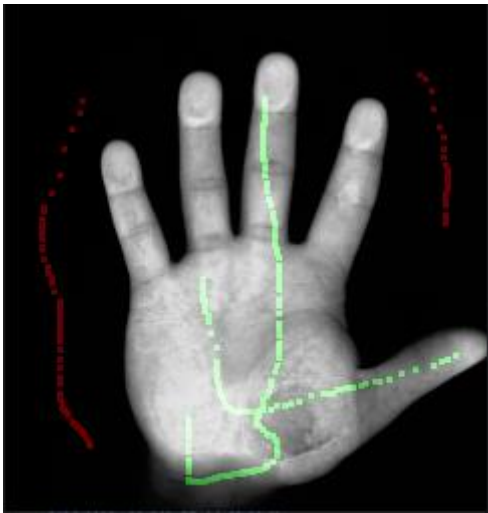

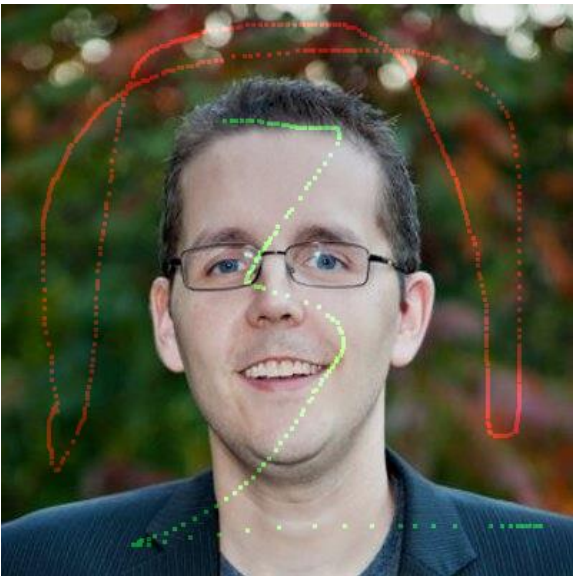



```

图转图像

最后，只需将与源点连接的点集映射到原图像上，即可得到前景点，剩下的就是背景点了。

三、 运行结果

表 1 图像分割的效果展示

原图像+标记	输出图像
 A grayscale image of a hand with segmentation masks. A red dashed line outlines the hand's perimeter, and a green dashed line outlines a specific region on the palm.	 The output image showing the hand with the segmentation masks applied, resulting in a high-contrast, binary-like appearance where the hand is white against a black background.
 A color image of a man's face with segmentation masks. A red dashed line outlines the face, and a green dashed line outlines a specific region on the forehead.	 The output image showing the face with the segmentation masks applied, resulting in a high-contrast, binary-like appearance where the face is white against a black background.
 A color image of four hats with segmentation masks. A red dashed line outlines the hats, and a green dashed line outlines a specific region on one of the hats.	 The output image showing the hats with the segmentation masks applied, resulting in a high-contrast, binary-like appearance where the hats are white against a black background.

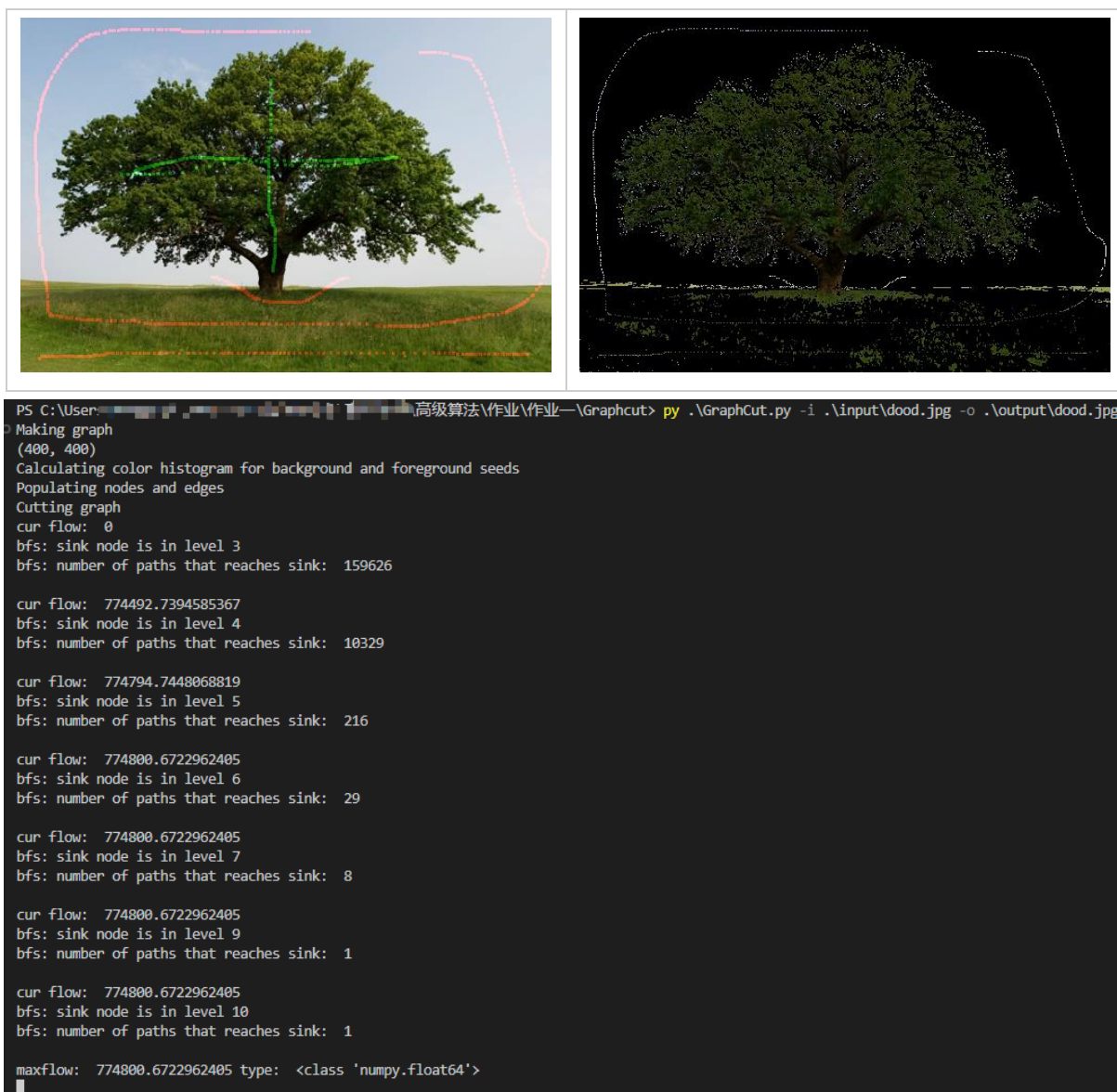


图 3 控制台运行例子

四、总结

通过结合 Graphcut 中的边权设置方法和 Dinic 最大流算法，该项目能够在较短时间内（大部分情况下 < 10s）完成 400*400 大小的图像的分割。如果分割后效果不好，可以尝试多取标记点，以及在边界附近取边界点，一般情况下都会有所改进。

五、困难与解决

一开始并没有想到采用 Graphcut 的边权设置方法，仅仅是将标记点与源点或汇点相连，其他点都默认与源点和汇点没有连接。这会导致搜索过程中的路径非常长，最短路径要达到几十甚

至上百（对比图 3，最长路径才 10）。一开始以为是广度和深度搜索的实现有问题，尝试了一些优化，比如记录广度搜索过程中搜到汇点的路径，在深搜时只走这些路径。但改进不大，直到将所有点都与源点和汇点建立连接，算法才终于快起来了。

另外，当标记点较少时，目前的算法运行效果可能不佳，这也是接下来要尝试去解决的一个问题。可以尝试将标记点附近的点也选为标记点。

参考

[1] Boykov, Yuri Y., and M-P. Jolly. "Interactive graph cuts for optimal boundary & region segmentation of objects in ND images." Proceedings eighth IEEE international conference on computer vision. ICCV 2001. Vol. 1. IEEE, 2001.

[2] <https://github.com/cm-jsw/GraphCut>