《高级算法设计与分析》作业
基于最大流最小割思想的图像分割

班　　级： 计科学硕二班

学　　号： 23214368

姓　　名： 彭郑威

二〇二四年四月十一日

目录

1. 背景介绍

1.1 图像分割

图像分割是计算机视觉和图像处理领域的一项基本任务，其目的是将一幅图像划分成多个区域或对象，使得这些区域在某种意义上具有相似性，而与其他区域有所区别。简而言之，图像分割旨在简化或改变图像的表示形式，使其更容易分析和理解。图像分割的结果通常是一组区域（或“超像素”），每个区域代表图像中的一个对象或图像的一部分。

图像分割方法有很多，本实验使用的方法是基于图的分割，即将图像视为一个图，其中像素表示节点，像素之间的关系（例如颜色相似性或空间接近性）表示边，然后借助最大流最小割的思想来进行图像分割。

1.2 最大流最小割

最大流最小割定理是图论中的一个基本定理，广泛应用于网络流、图像分割、排程问题等多个领域。这个定理揭示了网络流中“最大流”和“最小割”之间的直接关系，是解决许多优化问题的关键。

对于给定的容量网络 $G(V, E, c, s, t)$，其中 V 表示顶点集合，E 是边集合，c 是每条边的容量，s, t 分别是源点和汇点。给定割集 $(A, \bar{A})$ 满足 $A \subset V$ 且 $s \in A, t \in \bar{A}$。割集的容量 $c(A, \bar{A})$ 定义为：

$$c(A, \bar{A}) = \{< i, j > \mid < i, j > \in E, \quad i \in A, j \in \bar{A}\} \tag{1}$$

并且具有以下定理：

定理 1（最大流最小割定理） 容量网络的最大流的流量等于最小割集的容量。

2. 实验方法

2.1 图像建模为容量网络

在网络流问题中，通常使用有向连通图作为要分析的数据结构，叫作容量网络：

$$N =< V, E, c, s, t > \tag{2}$$

其中，V 是所有的顶点集合，E 是所有的边集合，c 是每条边的容量，s, t 分别是源点和汇点。因此，面临的问题是如何将图像建模为上述的容量网络。

给定一张图像 $x \in R^{h \times w \times c}$，其中 h, w, c 分别是图像的高度、宽度和通道数（对于彩色图像为 3，灰度图像为 1）。为了将图像建模为容量网络，可以将图像中的每个像素都表示为图中的一个节点。此外，还需要添加两个特殊的节点：源点（source）和汇点（sink）。这样就构成了容量网络 N 中的顶点集合 V，并指定了 s, t，如下图 2-1 所示。

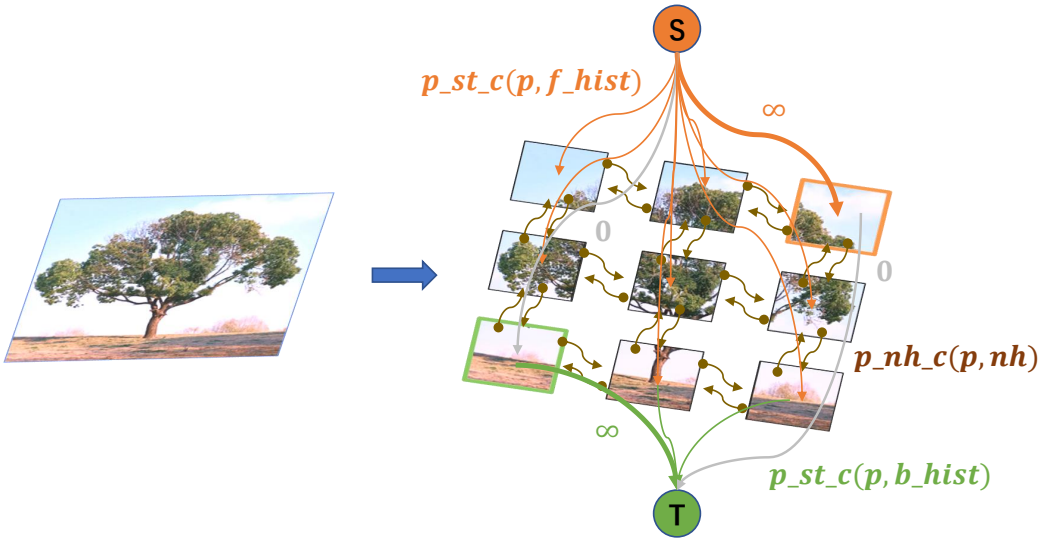

图 2-1 图像建模为容量网络，右图每个图元应看作一个像素

接下来面临的问题是边集合 E 的构建和边容量 c 的指定。在指定边时，我们分为两种情况讨论：

1. **邻域连接**：图像中每个普通像素节点与其邻近像素节点之间连接有边（如图 2-1 中的褐色线），这些边的权重（容量）反映了相邻像素之间的相似性，也表示将这两个像素划分成前景和后景的代价。

2. **连接到源点和汇点**：每个普通像素节点也会与源点和汇点相连（如图 2-1 中的

绿色和橙色线）。连接到源点的边权重代表将该像素归类为前景的“代价”，而连接到汇点的边权重代表将该像素归类为背景的“代价”。

在指定边的权重即容量网络的容量前，为了提高分割的效果，常常会要求用户提供前景区域和后景区域的一些点，这些点叫作种子（seed），如图 2-1 中加粗的图元为选中的前景（橙色）/后景（绿色）种子。除此之外，会计算这些种子的颜色直方图，其将 RGB 空间（$0 \sim 255$）分割成宽度为 10 的颜色段 $bins \in R^{26}$，统计种子区域中每个像素的灰度值落在每个颜色段的数量，像素 $p \in R^3$ 的灰度值由下式 3 给出：

$$p\_gray = 0.114 \times p^{(0)} + 0.587 \times p^{(1)} + 0.299 \times p^{(2)} \tag{3}$$

种子区域的颜色直方图的计算是：

$$hist[i] = bins[\lfloor \frac{i}{10} \rfloor]/n/w(i), \quad 0 \leq i \leq 256 \tag{4}$$

其中 n 表示前景/背景种子中的像素数，$w(i)$ 表示 RGB 空间的第 i 个通道所所在的颜色段的宽度，对于 $0 \leq w(i) < 250$，$w(i) = 10$，否则 $w(i) = 6$。对于前景种子和背景种子，各自计算其对应的颜色直方图 f_hist, b_hist。

对容量的指定，我们设前景种子像素、后景种子像素、普通像素分别为 f, b, p，每个像素的邻居像素为 nh。每个像素和源点、汇点的容量由式 5 给出，和邻居像素的容量由式 6 给出。

$$p\_st\_c(p, hist) := -\log(hist[p\_gray(p)]) \tag{5}$$

$$p\_nh\_c(p, nh) := \exp\left(-\sum \frac{\|p - nh\|^2}{2}\right) \tag{6}$$

具体来说就是：

1. **前景种子** f：和源节点 s 的容量设置为 ∞，和汇节点 t 的容量设置为 0，和邻居像素 nh 的容量设置为 $p\_nh\_c(f, nh)$；

2. **后景种子** b：和源节点 s 的容量设置为 0，和汇节点 t 的容量设置为 ∞，和邻居像素 nh 的容量设置为 $p\_nh\_c(b, nh)$

3. **普通种子** p：和源节点 s 的容量设置为 $p\_st\_c(p, f\_hist)$，和汇节点 t 的容量设置为 $p\_st\_c(p, b\_hist)$，和邻居像素 nh 的容量设置为 $p\_nh\_c(p, nh)$。

如图 2-1 所示，所有边的容量，可以表示为下表 2-1：

表 2-1 容量网络边的容量指定

头节点	尾节点	容量
f	s	∞
	t	0
	nh	$p\_nh\_c(f, nh)$
b	s	0
	t	∞
	nh	$p\_nh\_c(b, nh)$
p	s	$p\_st\_c(p, f\_hist)$
	t	$p\_st\_c(p, b\_hist)$
	nh	$p\_nh\_c(p, nh)$

这样，就完成了容量网络的构建。

2.2 图像分割

在将图像建模为如图 2-1 中的容量网络后，接下来就可以利用最大流最小割的思想来对容量网络进行分割。其旨在寻找图中的一个割集 $(A, \bar{A})$，满足 $s \in A, t \in \bar{A}$，并且这个割集的容量最小，那么这个就称为最小割，也就是图像分割的结果，集合 A 中的所有顶点构成前景，集合 $\bar{A}$ 中的所有顶点则构成后景。根据定理 2，网络中从源点到汇点的最大流的值等于切断源点和汇点的最小割的容量。因此，最大流算法执行完毕后，所有从源点可达的节点（即流量可以从源点流向这些节点的节点）构成 A，而其他节点构成 $\bar{A}$。实际上，这个分割就是源点和汇点之间的最小割。

2.2.1 整体框架

本实验实现了基于 Dinic 有效算法的最大流优化算法。算法的输入是 2.1 节构建好的容量网络 G，以及 G 上的初始可行流 $flow = 0$。这个算法的整体框架是：首先使用广度优先搜索（BFS）构建层次图，它从源点出发，对图进行广度优先搜索，并记

录到达每个节点的层级。搜索时，只考虑有剩余容量的边。当到达汇点时，记录汇点的层次并在 BFS 结束后返回。BFS 结束后，层次图构建完成，判断返回的汇点层次是否为 0，如果是，则说明无法再到达汇点，算法结束，当前的流量就是最大流。否则，在构建的层次图上执行深度优先搜索（DFS）。它从源点出发，尝试找到一条到汇点的路径，同时确保路径上的每一步都到达更高的层级。沿途，DFS 方法会尽可能地增加流量，并返回能够增加的流量数，直到找到一条路径无法再增加更多流量为止（即找到一个阻塞流）。DFS 算法结束后，$flow$ 加上增加的流量数，然后重复上述过程。图 2-2 和附录算法 1 展示了该算法的整体流程。有关该算法的详细细节我们后续展开。

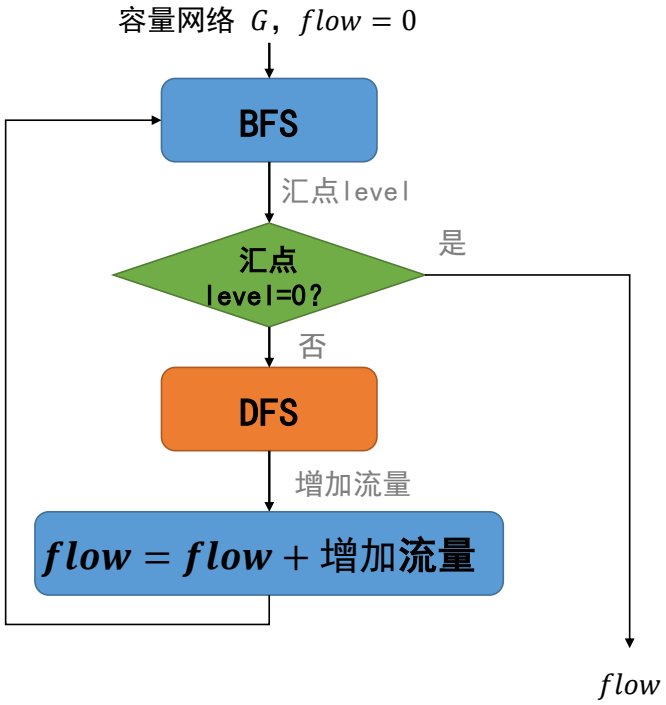

图 2-2 Dinic 最大流算法

2.2.2 广度优先搜索 BFS

算法中的广度优先搜索 BFS，其目的是构建层次图并确定是否存在从源点到汇点的路径。层次图中的每个节点都被分配了一个层级，这个层级表示从源点出发到达该节点的最短路径。

该算法的详细实现步骤如下，伪代码参见附录算法 2：

1. **初始化**

   - 首先，方法初始化所有节点的层级为 0，表示这些节点尚未被访问。仅源点的层级设置为 1，因为源点是搜索的起点。
   - 使用一个队列来进行宽度优先搜索，初始时只包含源点。
   - 定义了其他辅助数据结构，如 $level\_cnt$ 来记录每一层的节点数，$father$ 数组来记录到达每个节点的前一个节点（这在路径重构中很有用），以及 can_reach_sink 布尔数组来标记哪些节点可以到达汇点。

2. **BFS 循环**

   - 在队列不为空的情况下，从队列中取出（或称“弹出”）一个节点作为当前节点 cur。
   - 检查是否已经达到了汇点所在的层级（$break\_level$）。如果已到达，则终止搜索。这一步是 Dinic 算法的优化之一，避免了不必要的搜索。
   - 遍历当前节点 cur 的所有邻接节点 nei。对于每个邻接节点，只有当它尚未被访问（即 $levels[nei] = 0$）并且 cur 到 nei 有剩余容量时，才考虑它。
   - 如果满足上述条件，则将邻接节点 nei 的层级设置为当前节点层级加一（$levels[nei] = levels[cur] + 1$），并将 nei 加入队列中以供后续搜索。
   - 在遍历邻接节点的过程中，如果发现任何邻接节点是汇点，并且通过该节点可以增加流量，则更新 $break\_level$ 为汇点的层级，同时标记从源点到当前节点路径上的所有节点为可以到达汇点。

3. **更新和返回**

   - 搜索结束后，如果汇点的层级大于 0（即存在从源点到汇点的路径），则打印出当前流量、汇点层级以及可以到达汇点的路径数量等调试信息。
   - 返回汇点的层级，如果汇点层级为 0，则表示没有从源点到汇点的路径，最大流计算将终止。

BFS 通过构建层次图来优化后续的 DFS 过程，使得 DFS 只在这些层次图上寻找增广路径，从而提高了整体算法的效率。

2.2.3 深度优先搜索 DFS

算法中的深度优先搜索 DFS 目的是构建的层次图上寻找增广路径的。该方法通过深度优先搜索遍历层次图，尝试找到从源点到汇点的路径，并在这些路径上推送尽可能多的流量，直到无法增加更多流量为止，这样的路径被称为阻塞流。

该算法的详细实现步骤如下，伪代码参见附录算法 3：

1. **参数和初始化**

   - DFS 方法接收四个参数：当前节点 cur、当前节点的最大输入流量 $cur\_max\_inbound$、搜索路径 $path$（用于调试或记录路径）和汇点的层级 $sink\_level$。
   - 如果当前最大输入流量 $cur\_max\_inbound$ 为零或者当前节点就是汇点，则搜索终止。在后一种情况下，返回 $cur\_max\_inbound$ 作为通过这条路径的流量。

2. **遍历邻接节点**

   - 对于当前节点 cur，方法遍历所有邻接节点 nei。只有当邻接节点 nei 的层级正好比当前节点 cur 的层级高一级（即 $levels[nei] = levels[cur] + 1$）时，才考虑该邻接节点，这保证了搜索沿着层次图进行，避免回环。
   - 此外，还要求当前节点到邻接节点的边上有剩余容量（$cap > 0$），这表示该边可以用来推送流量。

3. **递归 DFS**

   - 对于每个符合条件的邻接节点 nei，DFS 方法递归地调用自身，尝试找到从 nei 到汇点的增广路径。
   - 递归调用中，更新当前最大输入流量 $cur\_max\_inbound$ 为原值减去已经通过当前节点 cur 推送的流量 $cur\_outbound$ 和当前边的容量 cap 中的较小值，这保证了不会推送超过边容量或节点容量的流量。
   - 如果递归调用返回的流量大于零，表示找到了一条增广路径，更新边的剩余容量，并且如果当前边不是直接连接到源点或汇点的边，则更新反向边的容量（增加相同的流量）。

4. **更新和返回**

- 累加所有通过当前节点 cur 的流量 $cur\_outbound$。如果已经不能从当前节点推送更多流量（即达到了最大输入流量 $cur\_max\_inbound$），则停止遍历邻接节点。
- 返回通过当前节点的总流量 $cur\_outbound$。

DFS 尝试在层次图上找到所有可能的增广路径，并在这些路径上推送尽可能多的流量，直到找到阻塞流或无法增加更多流量为止，通过在每次 BFS 构建的层次图上执行，大大提高了算法的效率和实际应用的性能。

3. 实验结果

实验选取了 6 张场景图片，背景既含有简单单色图像，又含有复杂场景图像。实验编写了一套基于 OpenCV 的交互式图像风格的系统，所有的代码位于 https://github.com/AaronPeng920/GraphCuts.git。在 6 张图像上的风格结果如图 3-3 所示。第二行控制种子中绿色线为前景种子，红色线为后景种子，分割结果显示前景。

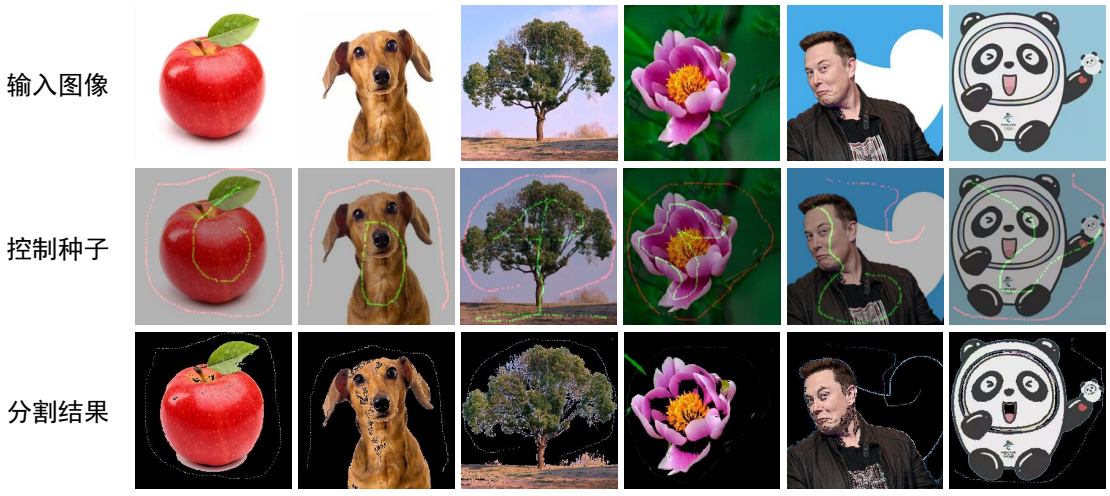

图 3-3 分割结果。

A 附录

A.1 Dinic 最大流算法

Algorithm 1 Dinic 最大流算法

Input: Graph G with source node s, sink node t and capacity table c

Initialization: $flow \leftarrow 0$
while True **do**
  $sinkLevel \leftarrow BFS(G)$
  if $sinkLevel = 0$ **then**
    break
  end if
  $path \leftarrow []$
  while True **do**
    $pathFlow \leftarrow DFS(source, \infty, path, sinkLevel)$
    if $pathFlow = 0$ **then**
      break
    end if
    $flow \leftarrow flow + pathFlow$
  end while
end while

Output: $flow$

A.2 构造层次网络的 BFS 算法

Algorithm 2 构造分层网络的 BFS 算法

Input: Graph G with source node s, sink node t, node count n and capacity table c

Initialization: $levels \leftarrow [0] \times n$, $queue \leftarrow EmptyQueue()$
$queue.append(s)$
$levels[s] \leftarrow 1$
while $queue$ **is not** empty **do**
    $current \leftarrow queue.popLeft()$
    for all $(neighbor, capacity)$ **in** $c[current]$ **do**
        if $capacity > 0$ **and** $levels[neighbor] = 0$ **then**
            $levels[neighbor] \leftarrow levels[current] + 1$
            $queue.append(neighbor)$
        end if
    end for
end while

Output: $levels[t]$

A.3 寻找阻塞流的 DFS 算法

Algorithm 3 寻找阻塞流的 DFS 算法

Input: Graph G with source node s, sink node t, node count n and capacity table c, current node max inbound $cur\_max\_inbound$, current node cur, level for each node $levels$ and sink node level $sink\_level$ obtained from algorithm 2, search path $path$

if $cur\_max\_inbound \leq 0$ **or** $cur = t$ **then**
  return $cur\_max\_inbound$
  $cur\_outbound \leftarrow 0$
  for all $(neighbor, capacity)$ **in** $c[cur]$ **do**
    if $levels[neighbor] = levels[cur] + 1$ **and** $capacity > 0$ **then**
      $new\_max\_inbound \leftarrow \min(cur\_max\_inbound - cur\_outbound, capacity)$
      $flow \leftarrow DFS(neighbor, new\_max\_inbound, path + [neighbor], sink\_level)$
      $c[cur][neighbor] \leftarrow c[cur][neighbor] - flow$
      if $neighbor \neq t$ **and** $cur \neq s$ **then**
        $c[neighbor][cur] \leftarrow c[neighbor][current] + flow$
      end if
      $cur\_outbound \leftarrow cur\_outbound + flow$
      $cur\_outbound = cur\_max\_inbound$
      break
    end if
  end if
end for

Output: $cur\_outbound$