中山大學

SUN YAT-SEN UNIVERSITY

实验作业报告

实验：	算法设计与分析
学号：	23320093
姓名：	林宏宇
专业：	计算机科学与技术
班级：	计科1班
指导教师：	戴智明
实验日期：	2025年12月23日

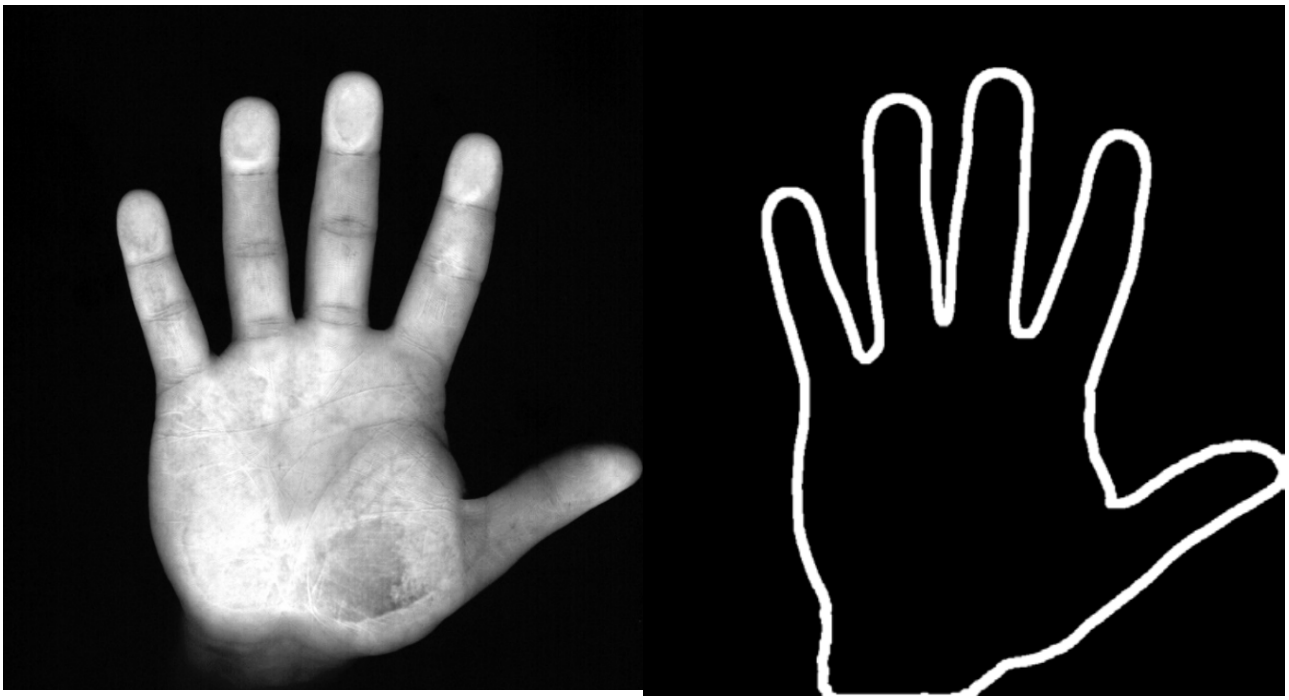
算法设计与分析

✏️ 实验目的

上机作业

应用最大流最小割集思想对图像进行分割。

- 数据输入：一幅图像（起码3个实例）
- 结果输出：分割后图像
- 例子：



输入图像　　输出图像

提交要求

- 提交要求：介绍算法实现思想（流程图、公式辅助说明）、输出截图、附上源代码，提交pdf文档。
- 提交方式：65959505@qq.com
- 文件命名格式：学号_姓名_第几次作业. 如：203838_张三_03
- 请各位同学12月25日18点前交作业，逾期无效

📋 算法原理

最大流最小割 (Max-Flow Min-Cut)

基于最大流最小割定理的图像分割是一种基于图论的方法。其核心思想是将图像分割问题转化为一个图的最小割问题，而最小割问题又可以通过求解最大流问题来解决。

1. 图像到图的映射 (Graph Construction)

首先，我们需要将输入的图像构建成一个加权无向图 $\mathrm{G} = (\mathrm{V}, \mathrm{E})$。

节点 (Nodes/Vertices, V)：

- 图像中的每一个像素点对应图中的一个普通节点。
- 此外，引入两个特殊的终端节点（Terminal Nodes）：
  - **源点 (Source, S)**：通常代表“前景”或“目标物体”。
  - **汇点 (Sink, T)**：通常代表“背景”。

边 (Edges, E)：

图中的边分为两类：

- **n-links (Neighborhood links)**：连接相邻像素点的边。
  - 这些边表示像素之间的连续性或相似性。
  - **权重**：通常基于像素颜色的相似度。如果两个相邻像素颜色非常接近，边的权重应该很大（表示它们很可能属于同一区域，不应该被切断）；如果颜色差异大，权重应该很小（容易被切断）。
  - 公式示例：$w(u, v) \propto \exp(-\frac{|I(u)-I(v)|^2}{2\sigma^2})$，其中 $I(u)$ 是像素 u 的强度或颜色。
- **t-links (Terminal links)**：连接像素点与终端节点（S 或 T）的边。
  - 每个像素点 p 都有两条 t-links：一条连接到 S，一条连接到 T。
  - **权重**：表示该像素属于前景或背景的概率（似然度）。
  - 连接 S 的权重：基于该像素属于前景的概率。如果用户标记该点为前景，权重设为无穷大。
  - 连接 T 的权重：基于该像素属于背景的概率。如果用户标记该点为背景，权重设为无穷大。

2. 最大流算法（Dinic）

Dinic算法是一种高效的最大流算法，适用于求解有向图中从源点S到汇点T的最大流问题。其核心思想是通过分层图和多路增广，显著提升了增广路径的查找效率。

算法流程

1. **分层图构建（BFS）**
   - 利用广度优先搜索（BFS）在残余网络中为每个节点分配层次$level(u)$，使得$level(S) = 0$，每条边(u, v)满足$level(v) = level(u) + 1$。
   - 只保留满足上述条件且残余容量$c_f(u, v) > 0$的边，形成分层图G_L。
2. **多路增广（DFS）**
   - 在分层图G_L上，使用深度优先搜索（DFS）从S到T寻找所有可行的增广路径。
   - 每次沿着分层图中的路径增广流量，直到无法再从S到T找到增广路径为止。
3. **阻塞流与迭代**
   - 一轮DFS结束后，若T不可达，则本轮分层图下的阻塞流已达到最大。
   - 重新用BFS构建新的分层图，重复上述过程，直到T不可达。

数学描述

设$G = (V, E)$为有向图，$c(u, v)$为边(u, v)的容量，$f(u, v)$为当前流量，残余容量为$c_f(u, v) = c(u, v) - f(u, v)$。

分层图构建：

$$\text{level}(S) = 0 \; \text{level}(v) = \text{level}(u) + 1, \;\; \forall (u, v) \in E, \;\; c_f(u, v) > 0$$

增广路径与流量更新： 每次沿着分层图中的路径P，增广流量：

$$\Delta = \min_{(u,v) \in P} c_f(u, v)$$

并更新：

$$f(u, v) \leftarrow f(u, v) + \Delta \; f(v, u) \leftarrow f(v, u) - \Delta$$

算法终止条件： 当BFS无法从S到T构建分层图时，算法终止，此时的总流量即为最大流：

$$\max \sum_{(S,v) \in E} f(S, v)$$

算法复杂度

Dinic算法的时间复杂度为$O(V^2E)$，在单位容量网络中可达$O(E \min(V^{2/3}, E^{1/2}))$，远优于朴素的Ford-Fulkerson方法。

应用说明

在本实验中，Dinic算法用于求解图像分割中像素图的最大流/最小割问题，通过高效的分层增广机制，能够快速处理大规模像素网络，实现高质量的分割效果。

3. 最大流最小割定理 (Max-Flow Min-Cut Theorem)

定理指出：**在一个网络流图中，从源点到汇点的最大流量等于该图的最小割容量。**

这意味着我们不需要去遍历所有可能的分割情况（这是指数级复杂度的），而是可以通过求解最大流问题来找到最小割。

4. 算法流程总结

1. **构建图**：根据图像像素建立节点，添加源点 S 和汇点 T。计算并设置 n-links（像素间相似度）和 t-links（前景/背景似然度）的权重。
2. **计算最大流**：使用标准的最大流算法（如 Ford-Fulkerson 或更高效的 Dinic 算法 ）计算从 S 到 T 的最大流。
3. **确定分割**：
    - 算法结束后，图中的节点会被分为两部分。
    - 从源点 S 出发，沿着剩余容量（Residual Capacity）大于 0 的边进行遍历（如 BFS 或 DFS）。
    - 所有能访问到的节点属于前景（S 集合）。
    - 无法访问到的节点属于背景（T 集合）。
4. **输出结果**：根据节点的归属，生成二值化掩膜或分割后的图像。

👨‍💻 重点代码说明

1. 图的压缩 `get_resized_image` 函数

为了演示算法，通过 `get_resized_image` 函数将图像缩小，否则构建图和计算最大流会非常慢。

- 将图像宽度调整为指定值 width
- 高度按比例缩放，保持宽高比不变
- 使用 OpenCV 的 `cv2.resize` 函数进行图像缩放

```
def get_resized_image(img, width=60):
    h, w = img.shape[:2]
    r = width / float(w)
    dim = (width, int(h * r))
    resized = cv2.resize(img, dim, interpolation=cv2.INTER_AREA)
    return resized
```

2. 图的构建 `build_graph` 函数原理与代码解读

`build_graph` 函数负责将图像像素信息转化为图结构，为后续最大流/最小割分割做准备。其设计思想和原理如下：

2.1. 节点设计

- 每个像素点 (y, x) 都是一个节点。
- 额外添加两个特殊节点：源点 S（source，前景）、汇点 T（sink，背景）。

2.2. 灰度化与预处理

- 若为彩色图像，先转为灰度图，便于亮度阈值判断。
- 高斯模糊降噪，减少误判。

2.3. t-links（终端边）设计

- 每个像素都与 S 和 T 相连。
- 连接权重（capacity）由像素亮度决定：
  - 亮度高于 FG_THRESH（如100）：极大概率为前景，S→像素的 capacity 设为极大（如10000），像素→T为0，强制归前景。
  - 亮度低于 BG_THRESH（如40）：极大概率为背景，像素→T的 capacity 设为极大，S→像素为0，强制归背景。
  - 介于两者之间：不确定，S和T都连，capacity设为较小值（如10），让算法根据整体能量自动决定归属。
- **原理**：高容量代表“强约束”，低容量代表“弱约束”，体现了先验知识和自动分割的结合。

2.4. n-links（邻域边）设计

- 每个像素与其右、下邻居相连（4邻域）。
- 权重由像素颜色差异决定，采用高斯函数 $\text{weight} = 80 \cdot \exp\left(-\frac{(\text{dist})^2}{2\sigma^2}\right)$，其中 dist 是像素颜色差异。
- 颜色越接近，weight越大，算法越不愿意割断这条边（鼓励同类像素归为一类）。
- 颜色差异大，weight变小，容易被割断（允许分割边界）。
- **原理**：weight体现了“平滑项”能量，保证分割结果边界尽量落在像素差异大的地方。

2.5. 返回图结构

- 返回构建好的图 G、源点 S、汇点 T。

2.6. 代码实现

```
def build_graph(img):
    """
    构建图:
    1. 节点：每个像素是一个节点，加上源点 'S' 和汇点 'T'。
    2. n-links (邻域边)：连接相邻像素，权重基于颜色相似度。
    3. t-links (终端边)：连接像素与S/T，权重基于亮度阈值（针对手掌图优化）。
    """
    # 图像像素与图结构的映射
    h, w = img.shape[:2] # 图像高度和宽度
    G = nx.DiGraph() # 有向图初始化

    # 转换为灰度图用于简单的亮度阈值判断
    if len(img.shape) == 3:
        gray = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
    else:
        gray = img

    # 高斯模糊减少噪声
    gray = cv2.GaussianBlur(gray, (5, 5), 0)

    # 定义源点和汇点
    S = 'source'
    T = 'sink'

    # 阈值与参数设定
    FG_THRESH = 100  # 前景亮度阈值
    BG_THRESH = 40   # 背景亮度阈值
    sigma = 8.0      # n-link 权重高斯参数

    # 边的容量越大，算法越不愿意割断这条边；容量越小，越容易被割断。
    FG_Capacity = 10000         # t-link 硬约束大权重
    Edge_Capacity = 10      # 模糊区域 t-link 权重


    # 遍历所有像素
    for y in range(h):
        for x in range(w):
            node_id = (y, x)
            pixel_color = img[y, x]
            gray_val = gray[y, x]

            # 添加 t-links (Terminal links)（像素与 S/T 的边）

            # 优化：不再仅依赖几何中心，而是依赖亮度
            if gray_val > FG_THRESH:
                # 亮区域 -> 前景种子
                G.add_edge(S, node_id, capacity=FG_Capacity)
                G.add_edge(node_id, T, capacity=0) # 容量0 “绝不可能”被分到背景（T），即使割断这条边也没有任何代价
```

```
            elif gray_val < BG_THRESH:
                # 暗区域 -> 背景种子
                G.add_edge(S, node_id, capacity=0)
                G.add_edge(node_id, T, capacity=FG_Capacity)
            else:
                # 模糊区域（边缘）-> 由最小割决定
                # 给予较小的均匀权重，让 n-links 起主导作用
                G.add_edge(S, node_id, capacity=Edge_Capacity)
                G.add_edge(node_id, T, capacity=Edge_Capacity)

            # 添加 n-links (Neighbor links)（像素与邻居的边）

            neighbors = []
            if x < w - 1: neighbors.append(((y, x+1), img[y, x+1])) # 右
            if y < h - 1: neighbors.append(((y+1, x), img[y+1, x])) # 下

            for n_id, n_color in neighbors:
                # 计算颜色差异
                dist = np.linalg.norm(pixel_color - n_color)
                # 高斯函数权重公式 颜色越接近，weight越大，算法越不愿意割断这条边
                weight = 80 * np.exp(- (dist**2) / (2 * sigma**2))

                G.add_edge(node_id, n_id, capacity=weight)
                G.add_edge(n_id, node_id, capacity=weight)

    return G, S, T, gray
```

3. 最大流最小割计算 `Dinic` 函数

Dinic 算法是解决最大流问题的高效算法，其时间复杂度为 $O(V^2E)$，在单位容量网络中甚至可达 $O(E\min(V^{2/3}, E^{1/2}))$。相比于 Edmonds-Karp 算法的 $O(VE^2)$，Dinic 算法通过引入**分层图 (Level Graph)** 和**阻塞流 (Blocking Flow)** 的概念，大大减少了寻找增广路径的次数。

3.1 算法核心原理

Dinic 算法的核心思想是维护一个分层图，并在分层图上寻找多条增广路径，直到无法再找到为止。

1. **分层图构建 (BFS)**：
    - 利用广度优先搜索 (BFS) 对残余网络中的节点进行分层。
    - 定义 $\text{level}(u)$ 为源点 S 到节点 u 的最短边数。
    - 分层图 G_L 仅包含满足 $\text{level}(v) = \text{level}(u) + 1$ 且残余容量 $c_f(u, v) > 0$ 的边 (u, v)。
    - 这保证了我们在寻找增广路径时，总是沿着最短路径前进，避免了绕圈。
2. **寻找阻塞流 (DFS)**：
    - 在分层图 G_L 上使用深度优先搜索 (DFS) 寻找增广路径。
    - **多路增广**：一次 DFS 可以找到多条路径，直到源点到汇点的流量“饱和”，即找不到新的路径为止。这些流量的总和称为一个“阻塞流”。
    - **当前弧优化 (Current Arc Optimization)**：为了避免重复遍历已经饱和的边，我们记录每个节点当前处理到的邻居索引 `ptr[u]`。下次访问该节点时，直接从 `ptr[u]` 开始，不再从头遍历。

3. **迭代过程**：

    - 重复上述两步：构建分层图 -> 寻找阻塞流 -> 更新残余网络。
    - 直到 BFS 无法到达汇点 T（即源点和汇点不连通），算法结束。

3.2 最小割的提取

当 Dinic 算法结束时，残余网络中不存在从源点 S 到汇点 T 的路径。此时，我们可以通过一次 BFS 找到最小割：

1. **定义集合** S_{cut}：从源点 S 出发，在残余网络中沿着剩余容量 $c_f(u, v) > 0$ 的边进行遍历（BFS/DFS），所有能访问到的节点集合即为 S_{cut}。
2. **定义集合** T_{cut}：图中剩余的所有节点 $V - S_{cut}$ 即为 T_{cut}。
3. **割边**：所有连接 S_{cut} 中节点到 T_{cut} 中节点的边，构成了最小割。

在图像分割中：

- S_{cut} 中的像素点被标记为 **前景**。
- T_{cut} 中的像素点被标记为 **背景**。

代码中最后一部分正是实现了这一逻辑，返回的 `reachable` 集合即为前景像素集合。

3.3 代码实现

```
def Dinic(G, source, sink):
    """
    使用 Dinic 算法计算最大流和最小割
    """
    print("正在使用自定义 Dinic 算法计算最大流...")
    sys.setrecursionlimit(200000) # 增加递归深度以防止深层图遍历溢出

    # 1. 初始化残余图
    residual = defaultdict(dict)
    for u, v, data in G.edges(data=True):
        residual[u][v] = data['capacity']
        if u not in residual[v]:
            residual[v][u] = 0

    # 预处理邻接表（因为 residual 的结构不会变，只有权值变）
    adj = {u: list(residual[u].keys()) for u in residual}

    max_flow = 0
    level = {}

    def bfs():
        """构建分层图"""
        level.clear()
        level[source] = 0
        queue = deque([source])
        while queue:
            u = queue.popleft()
            if u == sink:
                return True
```

```
            for v in adj[u]:
                cap = residual[u][v]
                if cap > 0 and v not in level:
                    level[v] = level[u] + 1
                    queue.append(v)
        return False

    def dfs(u, pushed, ptr):
        """在分层图中寻找增广路径 (多路增广)"""
        if pushed == 0 or u == sink:
            return pushed

        neighbors = adj[u]
        for i in range(ptr[u], len(neighbors)):
            ptr[u] = i # 当前弧优化
            v = neighbors[i]
            cap = residual[u][v]

            if v in level and level[v] == level[u] + 1 and cap > 0:
                tr = dfs(v, min(pushed, cap), ptr)
                if tr == 0:
                    continue

                residual[u][v] -= tr
                residual[v][u] += tr
                return tr
        return 0

    while bfs():
        ptr = {u: 0 for u in residual} # 每次构建分层图后重置当前弧
        while True:
            pushed = dfs(source, float('inf'), ptr)
            if pushed == 0:
                break
            max_flow += pushed

    # 5. 寻找最小割划分 (S集合: 从源点在残余图中可达的节点)
    reachable = set()
    queue = deque([source])
    visited = {source}
    while queue:
        u = queue.popleft()
        reachable.add(u)
        for v, cap in residual[u].items():
            if v not in visited and cap > 0:
                visited.add(v)
                queue.append(v)

    non_reachable = set(G.nodes()) - reachable

    return max_flow, (reachable, non_reachable)
```

4. 分割结果生成与可视化

4.1 从最小割集到轮廓图的转化分析

最大流最小割算法将图像像素节点划分为前景集合（S集）和背景集合（T集），但分割的本质结果是每个像素的归属标签。为了直观展示分割边界，需要将这种“集合划分”进一步转化为图像上的轮廓。

具体流程如下：

1. **掩膜生成**：根据最小割结果，将所有属于前景集合的像素在掩膜中赋值为1，背景为0，得到一张二值分割掩膜。
2. **掩膜放大**：由于分割是在低分辨率图像上完成的，需用插值方法（如`cv2.INTER_NEAREST`）将掩膜还原到原图大小，保证轮廓与原图对齐。
3. **轮廓提取**：对放大后的掩膜进行二值化处理，再用`cv2.findContours`函数检测所有连通区域的边界。这样可以自动提取出前景与背景的分割轮廓。
4. **轮廓筛选与可视化**：过滤掉面积过大的无效轮廓（如整图边框），只保留有效目标区域。最后在原图或黑色背景上绘制轮廓，实现分割结果的可视化。

这种转化方式将抽象的最小割集合划分，变成了直观的图像边界线，使分割结果更易于理解和展示。

4.2 代码实现

```
print("第四步：生成分割")
mask = np.zeros((h, w), dtype=np.uint8)

for y in range(h):
    for x in range(w):
        node_id = (y, x)
        if node_id in reachable:
            # 属于源点集合（前景）
            mask[y, x] = 1
        else:
            # 属于汇点集合（背景）
            mask[y, x] = 0
# 将掩膜放大回原图尺寸
original_h, original_w = img.shape[:2]
full_mask = cv2.resize(mask, (original_w, original_h),
interpolation=cv2.INTER_NEAREST)
# 提取轮廓
mask_uint8 = (full_mask * 255).astype(np.uint8)
# 改用 RETR_TREE 以便检测所有轮廓，防止因背景被识别为前景（掩膜反转）导致只检测到最外层边框
contours, _ = cv2.findContours(mask_uint8, cv2.RETR_TREE,
cv2.CHAIN_APPROX_SIMPLE)

# 在原图上绘制轮廓，而不是全黑背景
contour_img = img_rgb.copy()
# 另外创建一个纯黑背景用于单独展示轮廓
contour_only = np.zeros_like(img_rgb)

valid_contours = []
```

```
    img_area = original_w * original_h
    for cnt in contours:
        x, y, w_box, h_box = cv2.boundingRect(cnt)
        # 如果轮廓面积接近全图面积，说明是图像边框（通常发生在背景被误判为前景时），将其忽略
        if w_box * h_box > 0.90 * img_area:
            continue
        valid_contours.append(cnt)

    # 使用绿色 (0, 255, 0) 绘制轮廓，线条加粗
    cv2.drawContours(contour_img, valid_contours, -1, (0, 255, 0), 2)
    # 在纯黑背景上绘制白色轮廓
    cv2.drawContours(contour_only, valid_contours, -1, (255, 255, 255), 2)

    # 保存轮廓和对比结果
    plt.imsave(img_name + '_step4_contour.png', contour_only)
    plt.imsave(img_name + '_step5_compare.png', contour_img)
```

👣 实验流程

0. 输入图像读取

输入图像：

1. 压缩图像以加快处理速度

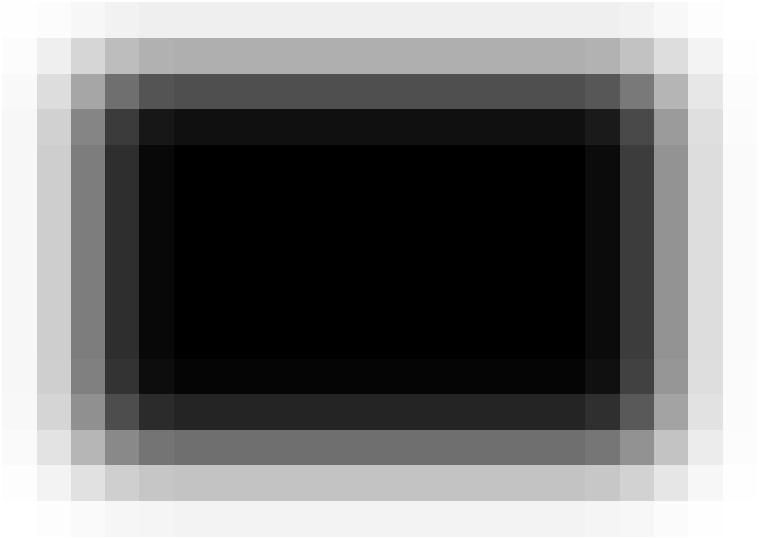
- 图像压缩成了宽度为30的超低分辨率图像(30*19)
- 更快速的构建图和计算最大流/最小割
- 更直观的展示每一步处理效果
- 对应30*19的图结构、最小割结果

压缩后的结果：

2.1. 转化为灰度图并高斯模糊降噪

- 灰度化简化像素信息
- 高斯模糊减少噪声影响，提升分割效果

灰度化并模糊后的图像：



2.2. 构建图结构

说明

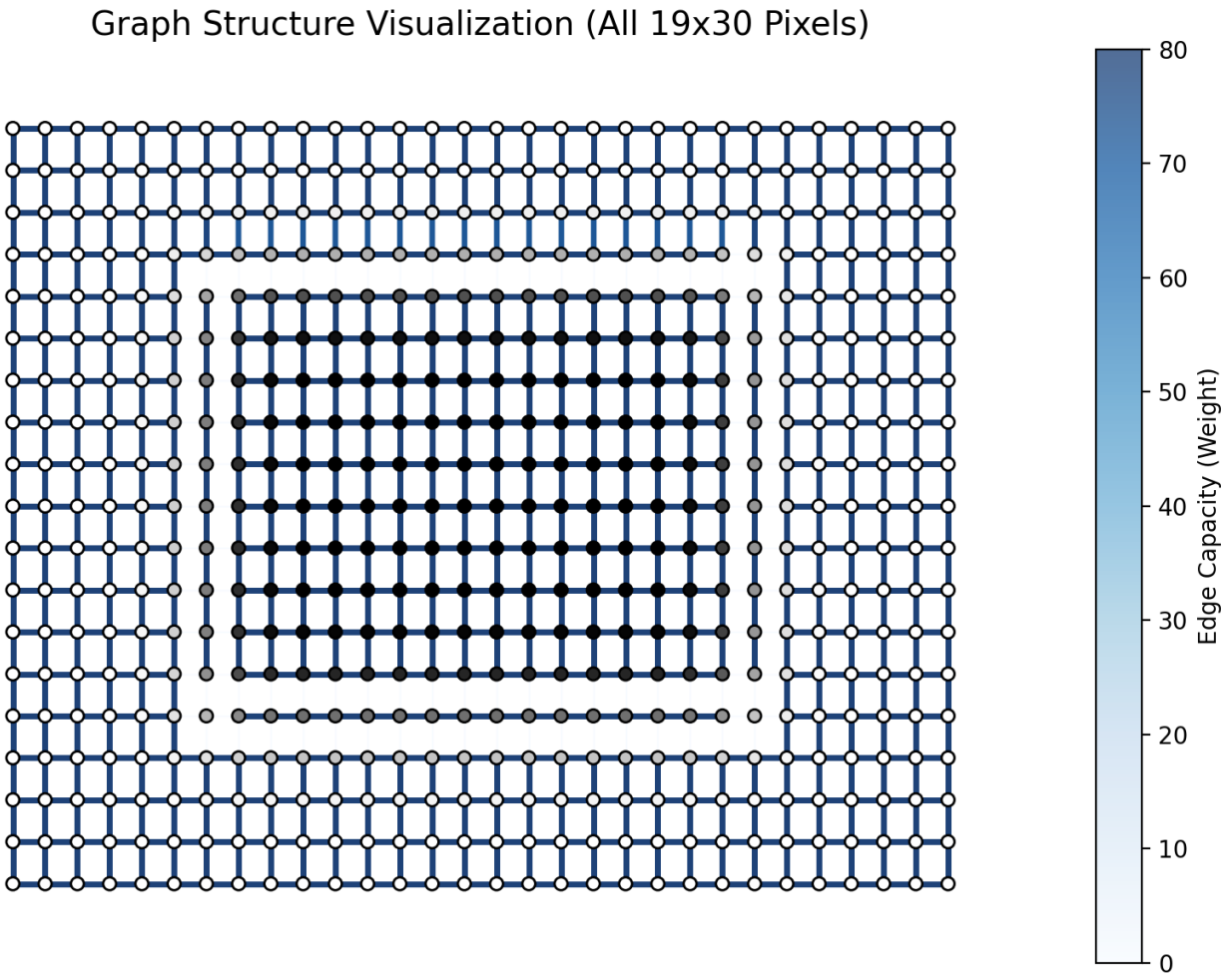
如图所示，图像被转化为一个图结构，每个像素点对应一个节点，此外还有源点S（前景）和汇点T（背景）两个特殊节点。由于图片是30*19，共570个像素点，构建的图结构包含570个节点（包括源点S和汇点T没有在其中展示），以及若干条n-links和t-links边。代码中的 `build_graph` 函数正是实现了这一结构的自动构建。

1. **节点设计**：每个像素点 (y, x) 都是一个节点，源点S和汇点T用于表达前景/背景归属。
2. **n-links（邻域边）**：连接相邻像素点，权重由像素灰度差异通过高斯函数计算，反映像素间的相似性。权重大表示像素更相似，算法更不愿意割断这条边，体现了分割时对区域连续性的偏好。图中像素之间的连线即为n-links。
3. **t-links（终端边）**：每个像素都与S和T相连，权重由像素亮度决定，反映其属于前景或背景的置信度。高亮度像素更倾向于前景，低亮度更倾向于背景。t-links的设计使得部分像素被“强制”归为前景或背景，其余像素则由整体能量最小化自动决定归属。

通过这种图结构，图像分割问题被转化为在图上寻找最小割。切断n-links意味着在像素之间划分边界，通常发生在像素差异大的地方；切断t-links则决定了像素最终归属前景还是背景。最大流/最小割算法会自动在图中找到一组割断的边，使得割的总代价最小，从而实现对图像的合理分割。

该结构的可视化图直观展示了像素之间的连接关系和边权分布，有助于理解分割算法的能量最小化原理。

构建图结构示意图：



3. 计算最大流与最小割

3.1 结合图像

如图所示，经过最大流最小割算法（如代码中的 `Dinic` 函数）处理后，图结构被自动分为两个集合：

- **前景集合（Source Set）**：在残余网络中，从源点S出发，沿着剩余容量大于0的边能够到达的所有像素节点。这些节点在分割结果中被归为前景（如图中绿色节点）。

- **背景集合（Sink Set）**：无法从S到达的像素节点，归为背景（如图中灰色节点）。
- **最小割边（Cut Edge）**：连接前景集合和背景集合的边（如图中红色虚线），即被算法割断的边。这些边往往位于像素灰度变化剧烈的区域，对应于图像的分割边界。

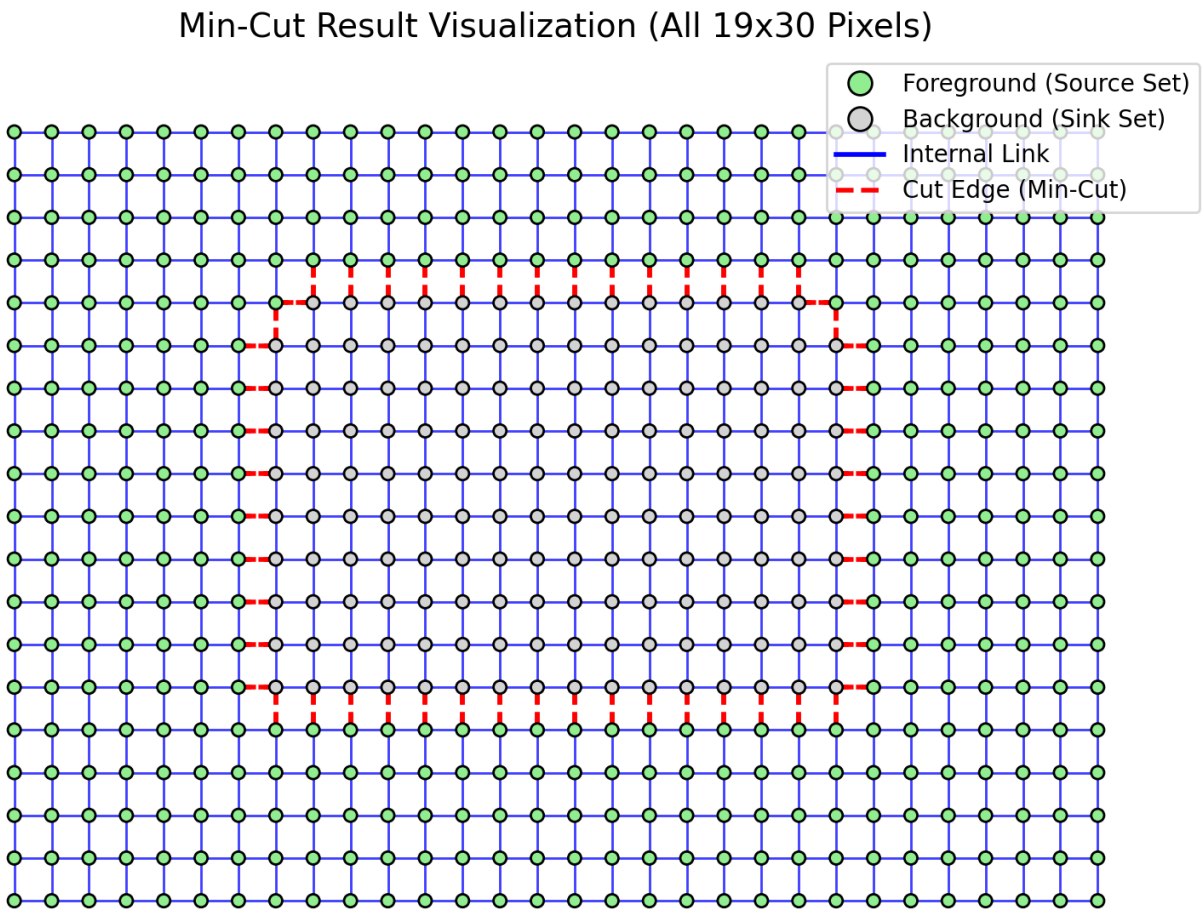
3.2 结合代码

代码实现中，`Dinic` 算法通过不断寻找增广路径，最终使得S和T之间不再连通。此时，前景和背景集合的划分即为最小割的物理意义：

- **分割边界自适应**：最小割自动选择割断权重最小的边，通常落在像素差异最大的地方，实现了自适应的图像分割。
- **能量最小化**：割断的总代价最小，保证了分割的合理性和边界的自然性。
- **像素归属明确**：每个像素根据其与S/T的连通性被唯一归类为前景或背景。

3.3 可视化结果

可视化图直观展示了最小割的分割效果，红色割边即为分割轮廓，绿色/灰色节点分别代表前景/背景区域。该过程体现了最大流最小割理论在图像分割中的实际应用。



4. 根据最小割结果生成分割图像的轮廓

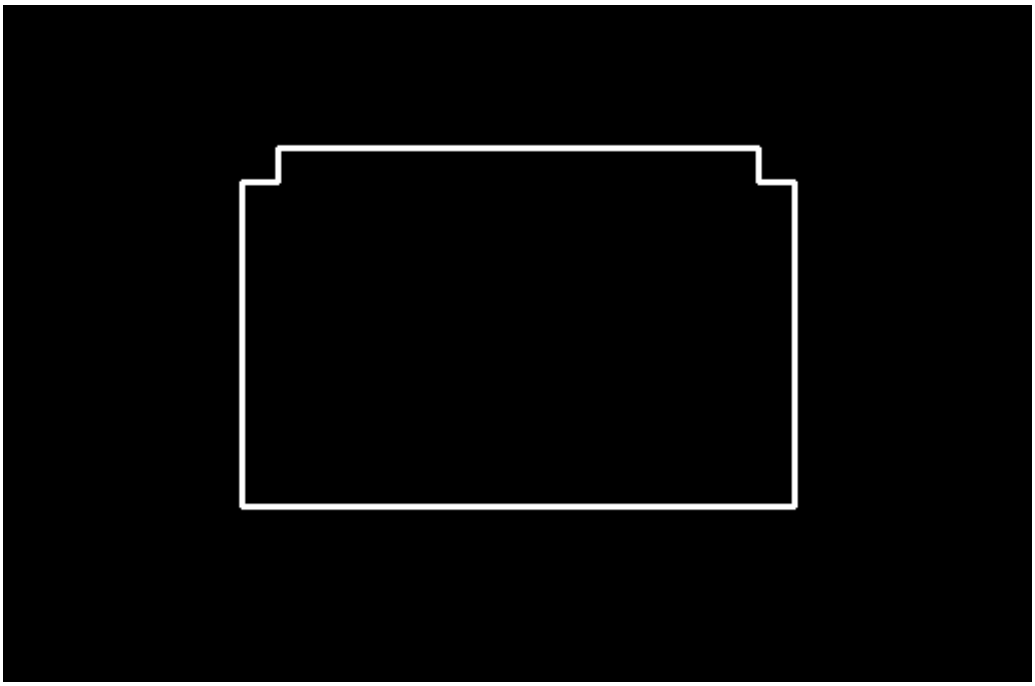
该流程实现了从最小割结果到可视化分割轮廓的完整转换，既保证了分割的准确性，也提升了结果的可读性。

说明

根据最大流最小割算法得到的最小割结果（即前景集合reachable和背景集合），我们可以将每个像素归类为前景或背景，进而生成分割掩膜和轮廓：

- **生成分割掩膜**：
    - 遍历所有像素，若像素节点属于前景集合（reachable），则掩膜值设为1，否则为0。
    - 得到的掩膜是低分辨率的（与缩放后的小图一致），需用`cv2.resize`放大回原图尺寸，保证分割结果与原图对齐。
- **提取分割轮廓**：
    - 将掩膜二值化（0/255），用`cv2.findContours`检测所有轮廓。
    - 过滤掉面积接近全图的轮廓（通常是误判的边框），只保留有效目标区域的轮廓。
    - 在纯黑背景上用白色线条单独显示轮廓，便于观察分割边界。

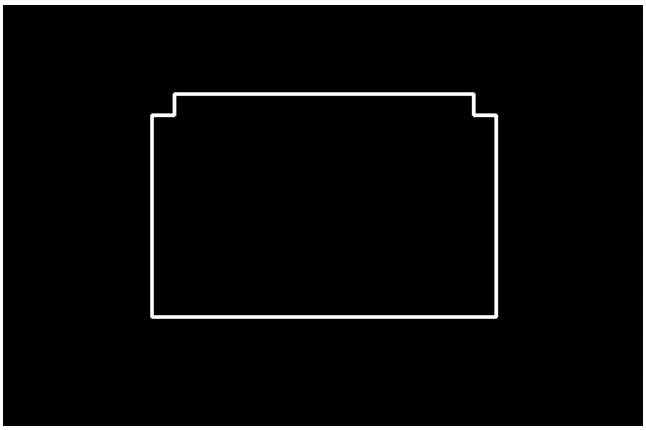
分割结果图像



5. 可视化分割结果

在原图上用绿色线条绘制分割轮廓，直观展示分割效果。

- 由于**像素较低分辨率**，轮廓可能不够平滑，但整体分割效果明显。
- 部分区域分割准确，边界清晰。
- 提高像素分辨率和优化图结构可进一步提升效果。

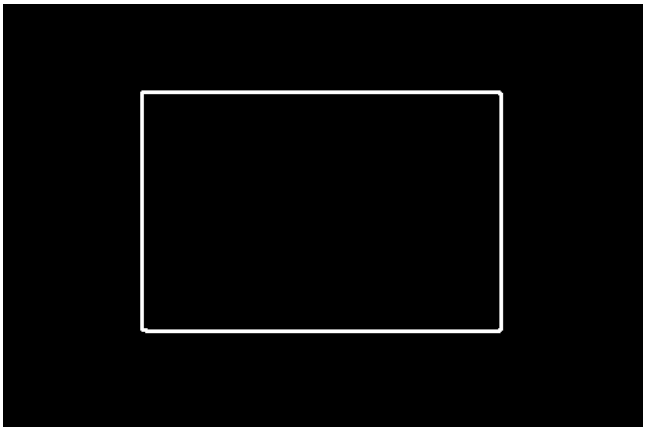


分割轮廓可视化图

分割结果与原图对比

6. 提升像素分辨率后的分割效果

为了提升分割效果，可以将输入图像的宽度从30提升到1000，增加像素分辨率。这样可以构建更精细的图结构，捕捉更多细节信息，从而改善分割质量。

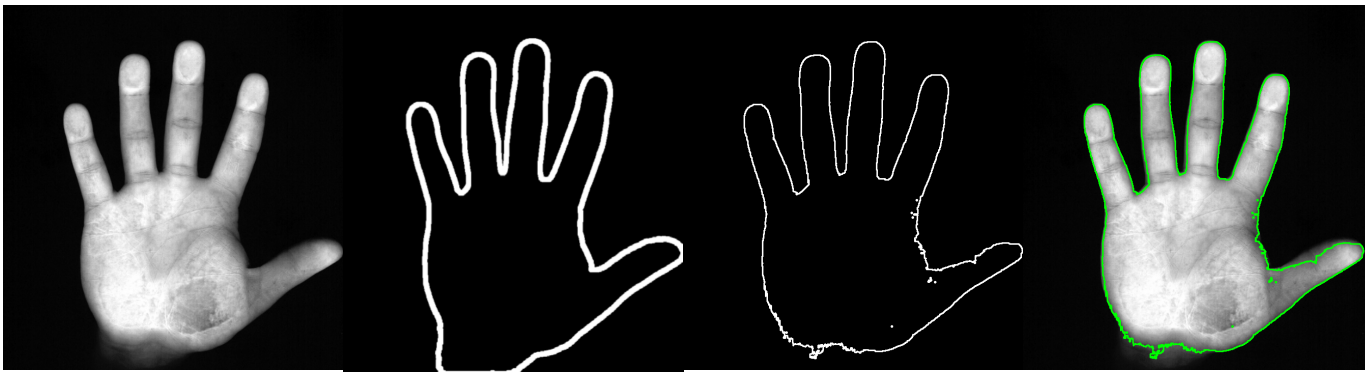


分割轮廓可视化图

提升像素后的分割效果

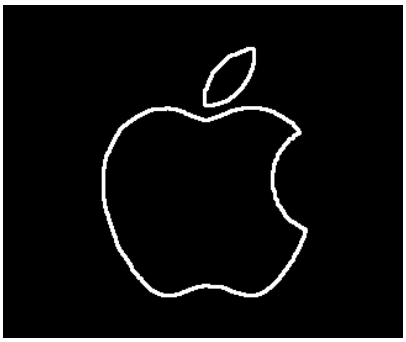
实验结果

题目测试图像



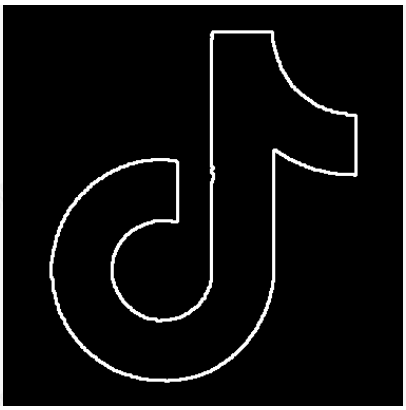
输入图像　　预计输出　　轮廓图　　对比图

测试图像1

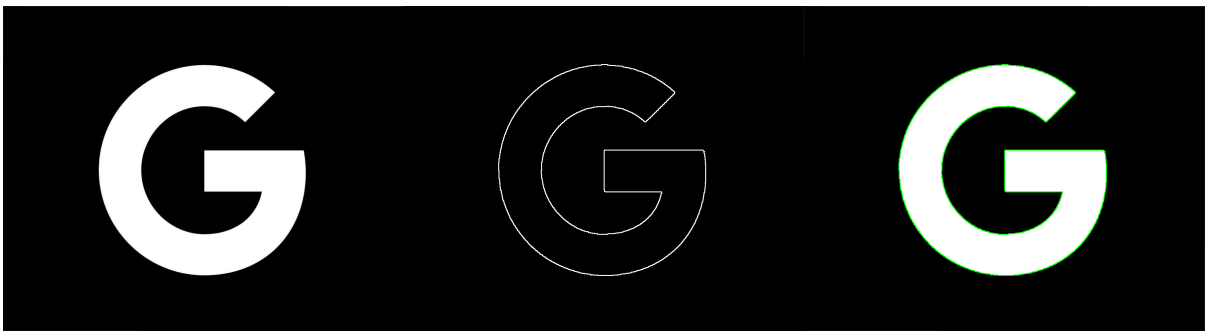

输入图像　　轮廓图　　对比图

测试图像2


输入图像　　轮廓图　　对比图

测试图像3



输入图像　　轮廓图　　对比图

更多测试图像结果

对比图

对比图

💡 实验总结

从理论到实践

本次实验成功将抽象的图论定理转化为具体的图像处理应用，打通了理论与实践的桥梁：

1. **数学模型的物理映射**：实验的核心在于将“图像分割”问题转化为“图的最小割”问题。我们将像素间的**颜色相似度**建模为边的**容量**（n-links），将像素属于前景/背景的**先验概率**建模为终端边的权重（t-links）。最小割算法寻找“割断代价最小”的边集，本质上就是在寻找图像中颜色差异最大、最不连续的边界，这完美契合了物体轮廓的物理特性。
2. **算法效率的工程权衡**：理论上最大流算法可以处理任意图，但在面对图像这种密集网格图（Grid Graph）时，节点数（$\mathrm{V} = \mathrm{W} \times \mathrm{H}$）和边数巨大。实验中发现直接处理高分辨率图像会导致计算耗时过长。为此，我采用了**降采样 Downsampling**策略，在低分辨率图上计算最小割，再将结果映射回原图。这种“空间换时间”或“精度换速度”的策略是工程实践中处理大规模数据的常见手段。
3. **Dinic算法的优势验证**：相比于朴素的 Ford-Fulkerson 算法，Dinic 算法通过构建**分层图（Level Graph）和多路增广**，有效避免了在环路中徘徊，显著提升了在稠密图上的收敛速度。

心得体会

1. **对算法复杂度的深刻认识**：在实验初期，尝试直接对原图构建图，程序运行极其缓慢甚至栈溢出。这让我直观体会到了 $\mathrm{O}(\mathrm{V}^2\mathrm{E})$ 复杂度在数据规模膨胀时的威力。通过引入 `sys.setrecursionlimit` 解决递归深度问题，以及通过图像缩放优化性能，我学会了如何在受限计算资源下优化算法实现。
2. **参数调优的重要性**：实验结果对参数非常敏感。例如高斯函数中的 σ 参数决定了对颜色差异的敏感度，亮度阈值决定了种子的选取。参数过大或过小都会导致分割欠佳（如背景误包含或前景破碎）。这让我认识到，一个好的算法不仅要有正确的逻辑，还需要根据具体数据分布进行精细的参数调整。
3. **图论应用的广阔前景**：通过这次实验，我看到了图论在计算机视觉中的优雅应用。最大流最小割不仅能用于图像分割，还能用于立体匹配、图像修复等领域。这种将图论只是应用于图像的思维方式，极大地拓宽了我的视野。

📚 参考资料

- 上课课件最大流最小割原理
- cv2 文档

附件

- 代码：`graph_cut.py`

```python
import cv2
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt
from collections import deque, defaultdict
import sys
def get_resized_image(img, width=60):
    """
    为了演示算法，我们将图像缩小，否则NetworkX构建图和计算最大流会非常慢。
    在实际工程中，通常使用C++实现的专门库（如PyMaxflow）。
    """
    h, w = img.shape[:2]
    r = width / float(w)
    dim = (width, int(h * r))
    resized = cv2.resize(img, dim, interpolation=cv2.INTER_AREA)
    return resized

def build_graph(img):
    """
    构建图:
    1. 节点：每个像素是一个节点，加上源点 'S' 和汇点 'T'。
    2. n-links (邻域边)：连接相邻像素，权重基于颜色相似度。
    3. t-links (终端边)：连接像素与S/T，权重基于亮度阈值（针对手掌图优化）。
    """
    # 图像像素与图结构的映射
    h, w = img.shape[:2] # 图像高度和宽度
    G = nx.DiGraph() # 有向图初始化

    # 转换为灰度图用于简单的亮度阈值判断
    if len(img.shape) == 3:
        gray = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
    else:
        gray = img

    # 高斯模糊减少噪声
    gray = cv2.GaussianBlur(gray, (5, 5), 0)

    # 定义源点和汇点
    S = 'source'
    T = 'sink'

    # 阈值与参数设定
    FG_THRESH = 100  # 前景亮度阈值
    BG_THRESH = 40   # 背景亮度阈值
    sigma = 8.0      # n-link 权重高斯参数

    # 边的容量越大，算法越不愿意割断这条边；容量越小，越容易被割断。
    FG_Capacity = 10000         # t-link 硬约束大权重
    Edge_Capacity = 10      # 模糊区域 t-link 权重


    # 遍历所有像素
```

```
    for y in range(h):
        for x in range(w):
            node_id = (y, x)
            pixel_color = img[y, x]
            gray_val = gray[y, x]

            # 添加 t-links (Terminal links) (像素与 S/T 的边)

            # 优化：不再仅依赖几何中心，而是依赖亮度
            if gray_val > FG_THRESH:
                # 亮区域 -> 前景种子
                G.add_edge(S, node_id, capacity=FG_Capacity)
                G.add_edge(node_id, T, capacity=0) # 容量0 “绝不可能”被分到背
景（T），即使割断这条边也没有任何代价
            elif gray_val < BG_THRESH:
                # 暗区域 -> 背景种子
                G.add_edge(S, node_id, capacity=0)
                G.add_edge(node_id, T, capacity=FG_Capacity)
            else:
                # 模糊区域（边缘） -> 由最小割决定
                # 给予较小的均匀权重，让 n-links 起主导作用
                G.add_edge(S, node_id, capacity=Edge_Capacity)
                G.add_edge(node_id, T, capacity=Edge_Capacity)

            # 添加 n-links (Neighbor links) (像素与邻居的边)

            neighbors = []
            if x < w - 1: neighbors.append(((y, x+1), img[y, x+1])) # 右
            if y < h - 1: neighbors.append(((y+1, x), img[y+1, x])) # 下

            for n_id, n_color in neighbors:
                # 计算颜色差异
                dist = np.linalg.norm(pixel_color - n_color)
                # 高斯函数权重公式 颜色越接近，weight越大，算法越不愿意割断这条边
                weight = 80 * np.exp(- (dist**2) / (2 * sigma**2))

                G.add_edge(node_id, n_id, capacity=weight)
                G.add_edge(n_id, node_id, capacity=weight)

    return G, S, T, gray

def Dinic(G, source, sink):
    """
    使用 Dinic 算法计算最大流和最小割
    """
    print("正在使用自定义 Dinic 算法计算最大流...")
    sys.setrecursionlimit(200000) # 增加递归深度以防止深层图遍历溢出

    # 1. 初始化残余图
    residual = defaultdict(dict)
    for u, v, data in G.edges(data=True):
        residual[u][v] = data['capacity']
        if u not in residual[v]:
            residual[v][u] = 0
```

```
    # 预处理邻接表（因为 residual 的结构不会变，只有权值变）
    adj = {u: list(residual[u].keys()) for u in residual}

    max_flow = 0
    level = {}

    def bfs():
        """构建分层图"""
        level.clear()
        level[source] = 0
        queue = deque([source])
        while queue:
            u = queue.popleft()
            if u == sink:
                return True
            for v in adj[u]:
                cap = residual[u][v]
                if cap > 0 and v not in level:
                    level[v] = level[u] + 1
                    queue.append(v)
        return False

    def dfs(u, pushed, ptr):
        """在分层图中寻找增广路径（多路增广）"""
        if pushed == 0 or u == sink:
            return pushed

        neighbors = adj[u]
        for i in range(ptr[u], len(neighbors)):
            ptr[u] = i # 当前弧优化
            v = neighbors[i]
            cap = residual[u][v]

            if v in level and level[v] == level[u] + 1 and cap > 0:
                tr = dfs(v, min(pushed, cap), ptr)
                if tr == 0:
                    continue

                residual[u][v] -= tr
                residual[v][u] += tr
                return tr
        return 0

    while bfs():
        ptr = {u: 0 for u in residual} # 每次构建分层图后重置当前弧
        while True:
            pushed = dfs(source, float('inf'), ptr)
            if pushed == 0:
                break
            max_flow += pushed

    # 5. 寻找最小割划分（S集合：从源点在残余图中可达的节点）
    reachable = set()
```

```
    queue = deque([source])
    visited = {source}
    while queue:
        u = queue.popleft()
        reachable.add(u)
        for v, cap in residual[u].items():
            if v not in visited and cap > 0:
                visited.add(v)
                queue.append(v)

    non_reachable = set(G.nodes()) - reachable

    return max_flow, (reachable, non_reachable)

def visualize_graph_structure(G, gray_img, h, w,
output_path='source/graph_structure.png'):
    """
    可视化图结构：展示压缩后全图的节点和边（大图建议只显示部分边/节点标签以防爆炸）
    """
    print("正在生成全图结构可视化...")
    # 全部像素节点
    all_nodes = [(y, x) for y in range(h) for x in range(w)]
    pos = {(y, x): (x, -y) for y in range(h) for x in range(w)}
    labels = {(y, x): f"{gray_img[y, x]}" for y in range(h) for x in
range(w)}
    colors = [gray_img[y, x] for y in range(h) for x in range(w)]

    # 只保留像素节点（不画source/sink）
    pixel_nodes = [n for n in G.nodes if isinstance(n, tuple)]
    subG = G.subgraph(pixel_nodes).copy()

    # 动态调整画布大小
    fig_w = max(8, w // 8)
    fig_h = max(6, h // 10)
    plt.figure(figsize=(fig_w, fig_h))
    ax = plt.gca()

    # 节点
    nx.draw_networkx_nodes(subG, pos, node_size=30, node_color=colors,
cmap='gray', vmin=0, vmax=255, edgecolors='black')
    # 只在小图时显示标签
    if h * w <= 400:
        nx.draw_networkx_labels(subG, pos, labels=labels,
font_color='red', font_weight='bold', font_size=8)

    # 边
    edges = []
    weights = []
    for u, v, data in subG.edges(data=True):
        edges.append((u, v))
        weights.append(data['capacity'])
    if weights:
        max_w = max(weights) + 1e-5
        widths = [0.5 + 2 * (w / max_w) for w in weights]
```

```
            edge_colors = [w for w in weights]
        else:
            widths = 1
            edge_colors = 'gray'
        edges_drawn = nx.draw_networkx_edges(subG, pos, edgelist=edges,
width=widths, edge_color=edge_colors, edge_cmap=plt.cm.Blues, edge_vmin=0,
edge_vmax=max(weights) if weights else 1, alpha=0.7, arrows=False)
        # 颜色条
        if edges_drawn and not isinstance(edges_drawn, list):
            plt.colorbar(edges_drawn, ax=ax, label='Edge Capacity (Weight)',
orientation='vertical', fraction=0.046, pad=0.04)
        elif edges_drawn and isinstance(edges_drawn, list) and
len(edges_drawn) > 0:
            sm = plt.cm.ScalarMappable(cmap=plt.cm.Blues,
norm=plt.Normalize(vmin=0, vmax=max(weights) if weights else 1))
            sm.set_array([])
            plt.colorbar(sm, ax=ax, label='Edge Capacity (Weight)',
orientation='vertical', fraction=0.046, pad=0.04)
        plt.title(f"Graph Structure Visualization (All {h}x{w} Pixels)",
fontsize=14)
        plt.axis('off')
        plt.tight_layout()
        plt.savefig(output_path, dpi=200)
        plt.close()
        print(f"全图结构可视化已保存至: {output_path}")

    def visualize_cut_result(G, gray_img, partition, h, w,
output_path='source/cut_result_structure.png'):
        """
        可视化最小割结果：展示全图的分割归属和割边
        """
        print("正在生成全图最小割结果可视化...")
        reachable, non_reachable = partition
        # 全部像素节点
        all_nodes = [(y, x) for y in range(h) for x in range(w)]
        pos = {(y, x): (x, -y) for y in range(h) for x in range(w)}
        labels = {(y, x): f"{gray_img[y, x]}" for y in range(h) for x in
range(w)}
        # 节点着色
        node_colors = ['lightgreen' if (y, x) in reachable else 'lightgray'
for y in range(h) for x in range(w)]
        # 只保留像素节点
        pixel_nodes = [n for n in G.nodes if isinstance(n, tuple)]
        subG = G.subgraph(pixel_nodes).copy()
        fig_w = max(8, w // 8)
        fig_h = max(6, h // 10)
        plt.figure(figsize=(fig_w, fig_h))
        ax = plt.gca()
        nx.draw_networkx_nodes(subG, pos, node_size=30,
node_color=node_colors, edgecolors='black')
        # 小图显示标签
        if h * w <= 400:
            nx.draw_networkx_labels(subG, pos, labels=labels,
font_color='black', font_weight='bold', font_size=8)
```

```
    # 边分类
    cut_edges = []
    internal_edges = []
    for u, v, data in subG.edges(data=True):
        u_in_S = u in reachable
        v_in_S = v in reachable
        if u_in_S != v_in_S:
            cut_edges.append((u, v))
        else:
            internal_edges.append((u, v))
    # 内部边
    nx.draw_networkx_edges(subG, pos, edgelist=internal_edges, width=1,
edge_color='blue', alpha=0.5, arrows=False)
    # 割边
    if cut_edges:
        nx.draw_networkx_edges(subG, pos, edgelist=cut_edges, width=2,
edge_color='red', style='dashed', arrows=False)
    from matplotlib.lines import Line2D
    legend_elements = [
        Line2D([0], [0], marker='o', color='w', label='Foreground (Source
Set)', markerfacecolor='lightgreen', markersize=10,
markeredgecolor='black'),
        Line2D([0], [0], marker='o', color='w', label='Background (Sink
Set)', markerfacecolor='lightgray', markersize=10,
markeredgecolor='black'),
        Line2D([0], [0], color='blue', lw=2, label='Internal Link'),
        Line2D([0], [0], color='red', lw=2, linestyle='--', label='Cut
Edge (Min-Cut)')
    ]
    ax.legend(handles=legend_elements, loc='upper right')
    plt.title(f"Min-Cut Result Visualization (All {h}x{w} Pixels)",
fontsize=14)
    plt.axis('off')
    plt.tight_layout()
    plt.savefig(output_path, dpi=200)
    plt.close()
    print(f"全图最小割结果可视化已保存至: {output_path}")

def main():
    # 0. 读取图像
    print("第零步：读取图像")
    img_name = 'source/test'
    img_path = img_name + '.png'
    img = cv2.imread(img_path)
    img_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

    # 1. 缩小图像（为了演示速度）
    print("第一步：缩小图像以加快处理速度")
    process_width = 1000   # 像素调整
    small_img = get_resized_image(img_rgb, width=process_width)
    h, w = small_img.shape[:2]
    print(f"处理图像大小: {w}x{h}")
    # plt.imsave(img_name + '_step1_resized.png', small_img)
```

```
    # 2. 构建图
    print("第二步：构建图结构")
    G, S, T, gray = build_graph(small_img)
    # plt.imsave(img_name + '_step2_gray.png', gray, cmap='gray')
    # visualize_graph_structure(G, gray, h, w, output_path=img_name +
'_step2_structure.png')


    # 3. 计算最大流 / 最小割
    print("第三步：计算最大流 / 最小割")
    cut_value, partition = Dinic(G, S, T)
    reachable, non_reachable = partition
    # visualize_cut_result(G, gray, partition, h, w, output_path=img_name
+ '_step3_minimumcut.png')

    # 4. 生成分割
    print("第四步：生成分割")
    mask = np.zeros((h, w), dtype=np.uint8)

    for y in range(h):
        for x in range(w):
            node_id = (y, x)
            if node_id in reachable:
                # 属于源点集合（前景）
                mask[y, x] = 1
            else:
                # 属于汇点集合（背景）
                mask[y, x] = 0
    # 将掩膜放大回原图尺寸
    original_h, original_w = img.shape[:2]
    full_mask = cv2.resize(mask, (original_w, original_h),
interpolation=cv2.INTER_NEAREST)
    # 提取轮廓
    mask_uint8 = (full_mask * 255).astype(np.uint8)
    # 改用 RETR_TREE 以便检测所有轮廓，防止因背景被识别为前景（掩膜反转）导致只检测到
最外层边框
    contours, _ = cv2.findContours(mask_uint8, cv2.RETR_TREE,
cv2.CHAIN_APPROX_SIMPLE)

    # 在原图上绘制轮廓，而不是全黑背景
    contour_img = img_rgb.copy()
    # 另外创建一个纯黑背景用于单独展示轮廓
    contour_only = np.zeros_like(img_rgb)

    valid_contours = []
    img_area = original_w * original_h
    for cnt in contours:
        x, y, w_box, h_box = cv2.boundingRect(cnt)
        # 如果轮廓面积接近全图面积，说明是图像边框（通常发生在背景被误判为前景时），将其
忽略
        if w_box * h_box > 0.90 * img_area:
            continue
        valid_contours.append(cnt)
```

```
    # 使用绿色（0, 255, 0）绘制轮廓，线条加粗
    cv2.drawContours(contour_img, valid_contours, -1, (0, 255, 0), 2)
    # 在纯黑背景上绘制白色轮廓
    cv2.drawContours(contour_only, valid_contours, -1, (255, 255, 255), 2)

    # 保存轮廓
    plt.imsave(img_name + '_contour.png', contour_only)

    # 5. 生成对比图
    print("第五步：生成对比图像")
    # plt.imsave(img_name + '_step5_compare.png', contour_img)
    plt.imsave(img_name + '_compare.png', contour_img)


if __name__ == "__main__":
    main()
```