连通域计数问题求解

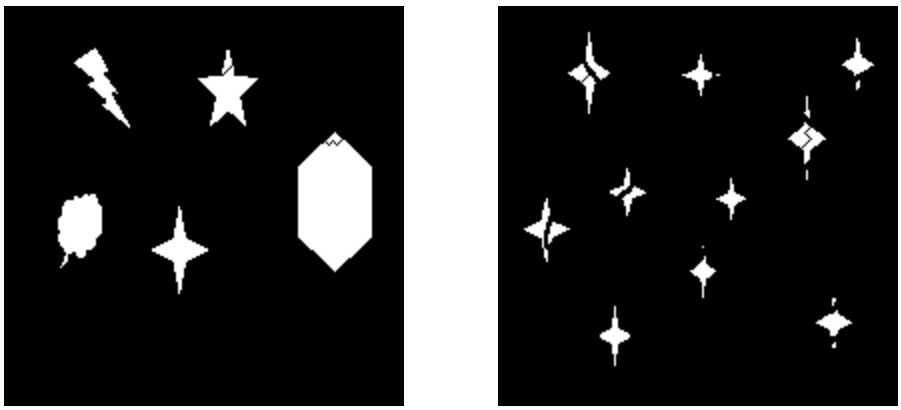
自 64　　赵文亮
2016011452

2018 年 10 月 13 日

目录

1　问题重述

请编程计算图 1 两个二值图中的 4 连通域数量、8 连通域数量，以及每个连通域内的总像素个数，并打印出结果。不能直接调用库函数。（图片文件：img1.bmp，img2.bmp）



(a) img1.bmp　　(b) 1mg2.bmp

图 1: 示例图片

2 原理介绍

基于该问题的要求，我们需要设计一个函数，该函数的输入为图片文件和连通性类型（4 或 8），输出为连通域数量和每个连通域内的总像素个数。由于 4 连通与 8 连通在算法实现上区别不大，在本节统称为连通。

我的算法基于 DFS 完成。不难看出，只要找到连通域中的某一个点，再通过 DFS 即可将这整个连通域的像素点全部搜索到。为了避免重复搜索，只需要设置一个访问标记矩阵，用来记录访问过的像素点。这样，每搜索到一个连通域，都可以将这个连通域搜索完全后再进行下一个连通域的搜索。

我使用一个变量 n 来完成连通区域的计数，程序开始时将其初始化为 0，并在每次发现一个新连通域时递增 n；使用变量 n_{pixel} 记录每个连通域内像素个数，在发现一个新连通域时将其初始化为 1，并在 DFS 内部每找到区域中一个点就递增 n_{pixel}，DFS 结束（即该连通区域搜索完毕）后，将 n_{pixel} 的数值添加到用于记录所有连通区域的像素数的列表中。主算法如算法 1 所示。

算法 1: 主算法

input : A bitmap, type of connectivity

output: Number of connected domains and number of pixels of each

initialization;

foreach *pixel p in the picture* **do**

  if *p is white and unvisitied* **then**

    $n \longleftarrow n + 1$;

    set visit flag;

    $n_{\text{pixel}} \longleftarrow 1$;

    call `DFS` from p;

    add $n_p ixel$ to pixel list;

  end

end

DFS 的算法实现比较简单，即从某一点出发，遍历它的邻居像素点。如果邻居像素点未被访问过且为白色，则可知它是连通区域的一部分，应该增加该连通区域像素点计数，并从这个点递归 DFS。DFS 的算法如算法 2 所示。

算法 2: DFS

input : Coordinate of current pixel p

foreach *neighbor pixel* p_{nbr} *of current pixel* **do**

  if p_{nbr} *is out of range* **then**

    continue;

  end

  if p_{nbr} *is white and unvisited* **then**

    set visit flag;

    $n_{\text{pixel}} \longleftarrow n_{\text{pixel}} + 1$;

    call DFS from p_{nbr};

  end

end

3 实现代码

基于上一节中的算法思路，使用 matlab 编写程序。主算法的函数为 get_connected_domain(filename, connect_type)：

```
% author: Zhao Wenliang
% function get_connected_domain
% input:
%   filename: a string of the name of the picture, e.g. 'img1.bmp'
%   type: connectivity type, must be 4 or 8
% output:
%   n: the number of connected domain
%   pix: the number pixels in each connected domain

function [n, pix] = get_connected_domain(filename, connect_type)
    % visit flag
    global flag;
    % neighbor position difference
    global nbr_dp;
    % graph
    global G;
    % size
    global s;
    % pixel count
    global n_pixel;
    % connectivity type
    global c_type;
    % initial
    c_type = connect_type;
    n = 0;
    G = imread(filename) / 255;
    s = size(G);
    flag = zeros(s);
    nbr_dp = [ -1,  0;
                1,  0;
                0,  1;
                0, -1;
               -1, -1;
                1, -1;
                1,  1;
               -1,  1;];
    pix = zeros(1, ceil(numel(G) / 2));
    % main loop
    for i = 1 : s(1)
        for j = 1 : s(2)
            if flag(i, j) == 0 && G(i, j) == 1 % new domain found
                n = n + 1;
                flag(i, j) = 1;
                n_pixel = 1;
                DFS(i, j); % DFS to find the whole domain
                pix(n) = n_pixel;
            end
        end
    end
    pix = sort(pix(1:n));
end
```

其中第 29 行定义了寻找邻居像素的坐标偏差。前四行为四连通时的情形，整体为八连通时的情形。例如当前像素为 (i,j)，通过第一行即可寻找到 $(i-1,j)$。最后将各区域的像素数排序，便于检验结果的正确性。

DFS 的代码如下：

```
1  % author: Zhao Wenliang
2  % function DFS
3  % input:
4  %   i, j: the row and column index from which we start the DFS
5  %   type: connectivity type, must be 4 or 8
6  function DFS(i, j)
7      global flag;
8      global nbr_dp;
9      global s;
10     global G;
11     global n_pixel;
12     global c_type;
13
14     for nbr_n = 1 : c_type
15         nbr = [i, j] + nbr_dp(nbr_n, :);
16         x = nbr(1);
17         y = nbr(2);
18         % out of range
19         if x < 1 || x > s(1) || y < 1 || y > s(2)
20             continue;
21         end
22         % unvisited and white, do DFS recursively
23         if G(x, y) == 1 && flag(x, y) == 0
24             flag(x, y) = 1;
25             n_pixel = n_pixel + 1;
26             DFS(x, y);
27         end
28     end
29 end
```

其中 15 行巧妙地利用 nbr_np 来计算出邻居的坐标，19 行为判断邻居坐标点是否出界。

最后则通过调用主算法的函数来求解原题：

```
% author: Zhao Wenliang
% main function, calculate the number of connected domain

% img1
% connectivity 4
[img1_4, pix1_4] = get_connected_domain('img1.bmp', 4);
% connectivity 8
[img1_8, pix1_8] = get_connected_domain('img1.bmp', 8);

% img2
% connectivity 4
[img2_4, pix2_4] = get_connected_domain('img2.bmp', 4);
% connectivity 8
[img2_8, pix2_8] = get_connected_domain('img2.bmp', 8);
```

4 计算结果

调用 main.m 后，查看各个变量可得计算结果。首先得出连通域的数目，如图 2 所示。从左至右分别对应了 img1.bmp（4 连通）、img1.bmp（8 连通）、img2.bmp（4 连通）、img2.bmp（8 连通）。

```
>> main
>> [img1_4, img1_8, img2_4, img2_8]

ans =

     9     5    23    20
```

图 2: 连通域个数求解结果

进而得出各个连通域内的像素个数的结果，如图 3 所示。从上到下分别对应了 img1.bmp（4 连通）、img1.bmp（8 连通）、img2.bmp（4 连通）、img2.bmp（8 连通）的结果。

```
>> pix1_4, pix1_8, pix2_4, pix2_8
pix1_4 =
          1          1         24         32        305        328        395        554       1918
pix1_8 =
        306        328        427        555       1942
pix2_4 =
     1     2     5     5     6     8    11    15    43    48    52    56    57    66    80    82    84    85    93    94   104   106   126
pix2_8 =
     1     2     5     5     6     8    15    43    52    57    82    84    85    93   104   104   105   106   126   146
```

图 3: 连通域内像素个数求解结果

5 分析

5.1 准确性

为了验证算法的准确性，我使用 Matlb 中的函数 bwlabel 编写了测试代码，同样可以输出图片中连通域的个数和每个连通域中的像素点个数。具体的实现代码见附录。连通域计数输出结果如图 4 所示。连通域内像素个数如图 5 所示。可见我的算法的输出的结果与使用库函数得到的结果完全相同。

```
>> using_lib
>> [N1_4, N1_8, N2_4, N2_8]

ans =

     9     5    23    20
```

图 4: 连通域个数求解结果（库函数）

```
>> pix1_4_lib, pix1_8_lib, pix2_4_lib, pix2_8_lib

pix1_4_lib =

          1           1          24          32         305         328         395         554        1918


pix1_8_lib =

        306         328         427         555        1942


pix2_4_lib =

    1    2    5    5    6    8   11   15   43   48   52   56   57   66   80   82   84   85   93   94  104  106  126


pix2_8_lib =

    1    2    5    5    6    8   15   43   52   57   82   84   85   93  104  104  105  106  126  146
```

图 5: 连通域内像素个数求解结果（库函数）

5.2 复杂度

该算法的核心是 DFS，这就保证了每个点能够被不重不漏地访问。初次之外，为了提升代码效率，我还做了如下的考虑：

- 适当使用全局变量。将需要两个函数共同访问或修改的变量设置为全局变量，免去了来回传递参数带来的时间和空间的复杂度。例如访问标志 flag 矩阵、像素点计数变量 n_{pixel} 等等。
- 预分配内存提升效率。各个连通域的像素点数需要使用一个列表 pix 来保存，但是这个列表的长度事先并不知道。如果在每次得到一个连通域时，都执行pix = [pix, n_pixel]会不断改变 pix 占用的内存，效率较低。为此，我估计出 pix 的最大取值，事先分配内存，最后返回时再截取需要的部分，而多余的内存会在函数执行完毕释放，减小了时间复杂度。

A 库函数解法代码

主函数 using_lib.m：

```
% author: Zhao Wenliang
% using matlab library functions for comparation

[L1_4, N1_4] = bwlabel(imread('img1.bmp'), 4);
pix1_4_lib = using_lib_count(L1_4, N1_4);
[L1_8, N1_8] = bwlabel(imread('img1.bmp'), 8);
pix1_8_lib = using_lib_count(L1_8, N1_8);
[L2_4, N2_4] = bwlabel(imread('img2.bmp'), 4);
pix2_4_lib = using_lib_count(L2_4, N2_4);
[L2_8, N2_8] = bwlabel(imread('img2.bmp'), 8);
pix2_8_lib = using_lib_count(L2_8, N2_8);
```

对各区域像素点计数函数 using_lib_count.m

```
function pix = using_lib_count(L, N)
    pix = zeros(1, N);
    for i = 1 : N
        pix(i) = sum(sum(L == i * ones(size(L))));
    end
    pix = sort(pix);
end
```