


计算机视觉 课程实验报告

学 号： 201900161140	姓名： 张文浩	2021.10.22
实验题目：图像结构①		
<p>实验过程中遇到和解决的问题： (记录实验过程中遇到的问题，以及解决过程和实验结果。可以适当配以关键代码辅助说明，但不要大段贴代码。)</p> <h3>实验 5.1</h3> <p>实验要求：</p> <ul style="list-style-type: none">实现一个 8 连通的快速连通域算法（注意 ppt 里是 4 连通），并基于该算法对测试图像 horse_mask.png 进行以下处理：<ul style="list-style-type: none">计算白色连通区域的个数；删除较小的白色连通域，只保留最大的一个；  <p>实验步骤：</p> <h4>第一步：阈值分割</h4> <p>对输入的是灰度图像进行阈值分割，调用 threshold 函数，将读入的灰度图像像素值大于 127（前景部分）变为 1，像素值小于 127 的置为 0。</p> <pre>threshold(img, img_binary, 127, 1, THRESH_BINARY);</pre> <h4>第二步：快速连通域算法</h4> <pre>void two_pass(const Mat& src, Mat& dst)</pre> <p>此函数读入一个 src 图像是上一步得到的前景为 1，背景为 0 的图像。得到的图像 dst</p>		

为 image_label。即每个位置的 label 值。实现步骤如下：

把要得到的 image_label 图像中因为要存储 label 的值，所以要把格式置为 32 字节单通道的格式。

```
dst.release();  
src.convertTo(dst, CV_32SC1); //转换成有符号整形，单通道模式
```

构造一个 label_equal 数组，记录 label 的等价情况，例如 label_equal[a]=b 意思是 label=a 和 label=b 是等价的。

```
int label = 1;  
vector<int> label_equal; //记录标签的集合。 例如 label_equal[a]=b 意思是 label=a 和 label=b  
是等价的  
label_equal.push_back(0);  
label_equal.push_back(1);
```

为了后面方便判断遍历到的位置是否已经被打上标签，label 从 2 开始加。

用两个 for loop 遍历输出的 dst 图像矩阵，对于每个点，因为采用八邻域的方法，所以找到每个点对应的左上角、上边、右上角、左边这么 4 个像素。因为遍历是先按行顺序再按列顺序遍历的，所以这四个邻域像素一定已经被遍历过了。如果像素大于 1 说明之前被标记过，说明是前景，否则是背景，忽略即可。

如果邻域的点的 label>1，就放入当前点的邻域数组 neighborlabel 中。

```
int pixel1 = preline[j - 1]; //当前像素左上角的像素  
int pixel2 = preline[j]; //当前像素上边的像素  
int pixel3 = preline[j + 1]; //当前像素右上角的像素  
int pixel4 = curline[j - 1]; //当前像素左边的像素
```

```
neighborlabel.push_back(pixel);
```

下一步通过观察 neighborlabel 数组中是否为空来判断当前像素的邻域是否打过标签，如果邻域还没有打过标签就说明当前像素一定是一个新连通域的第一个点。就给当前像素打上一个新的标签。

```
label_equal.push_back(++label); //给当前像素打上一个新的标签  
curline[j] = label;
```

否则说明当前像素的邻域已经打过标签了，要找到当前邻域中最小的 label，并给当前像素打上这个 label

```
int label_min = *min_element(neighborlabel.begin(), neighborlabel.end());  
curline[j] = label_min; //给当前像素位置打上邻域中最小的 label
```

为了保证能找到与当前这个 label_min 等价的更小的 label，要再次遍历 neighborlabel 数组，利用维护好的 label_equal 数组，找到其中最小的 label，并修改 label_equal 数组，Wie 这两个 label 建立等价关系。

```
label_equal[smaller] = label_min; //表示 smaller 和 label_min 是等价的  
label_equal[label_neighbor] = label_min;
```

现在要利用维护好的数组 label_equal 更新 label_equal 数组，保证每个 label 都是等价中最小的那个 label

```
while (preLabel != curLabel) //说明有与 curLabel 等价且比 curLabel 更小的 label  
{  
    curLabel = preLabel;
```

```
preLabel = label_equal[preLabel];  
}
```

同时，根据题目要求，要找到最大的连通域，所以要用一个 label_sum 记录每个 label 出现的次数，出现次数最多的那个 label 就是最大的连通域。

```
label_sum[curLabel]++;
```

为了找到连通域的个数，用一个 set 集合，每次找到一个最小的等价类 label 后就加入集合 set，利用 set 的性质自动去重，最后 set 集合中 label 的个数就是连通域的数量。

```
label_set.insert(curLabel);
```

利用 label_set 集合和 label_sum 数组得到答案

```
//获得一共有多少个连通域  
maxlabel = label_set.size();  
//获得哪个 label 数量最多  
label_max = max_element(label_sum.begin(), label_sum.end()) -  
label_sum.begin();
```

最后一步第二遍遍历 dst 图像矩阵，利用 label_equal 数组，将每个像素更新成相应等价 label 中最小的 label。

```
data[j] = label_equal[data[j]];
```

第三步：上色

现在快速连通域算法的函数构造完毕，现在已经利用得到了一个 label 矩阵，表示每个像素属于哪个连通域，现在需要构造一个 addcolor 函数根据此进行上色，将相同连通域的像素打上相同的颜色。

```
void addcolor(const Mat & src, Mat & dst)
```

这个函数得到的矩阵就是上好颜色后的矩阵，所以格式为 RGB 三通道。

```
dst.create(rows, cols, CV_8UC3);
```

我们还需要一个 map，将每个 label 对应一个 RGB 颜色，这个颜色值利用函数 randomcolor 随机产生。

```
map<int, Scalar> colors;
```

有了和这个 map，在遍历结果 image 的时候只需要根据 label 进行颜色设置即可。

```
Scalar color = colors[pixelValue];  
*data_dst++ = color[0]; //因为当前 dst 设置为三通道，所以每个像素占 3 个位置  
*data_dst++ = color[1];  
*data_dst++ = color[2];
```

第四步：只给最大连通域上色

为了显示出最大连通域，构造一个 addcolor1 函数，只将最大连通域设置成白色。因为之前已经找到了最大连通域对应的 label，所以在遍历结果 image 的时候，只需要将 label 等于 label_max 的像素置为白色即可。

```
*data_dst++ = 255;  
*data_dst++ = 255;  
*data_dst++ = 255;
```

第五步：种子填充

为比较种子填充法和快速连通域算法的速度，我又构造了一个种子填充法的函数，来得到 label_image。

```
void seedfill(const Mat& src, Mat& dst)
```

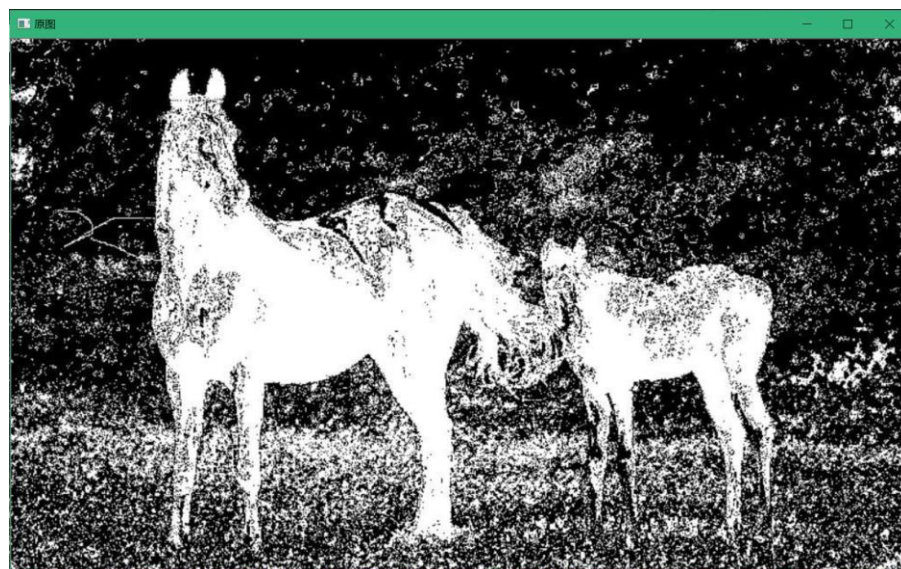
构造方法如下：

也是按照行优先再列优先的顺序遍历图像，每次将像素值为 1（白色的前景）的像素推入栈，每次从栈中取出一个像素，在将该像素八个邻域中像素值也是 1（白色前景）的像素推入栈，直到栈为空。

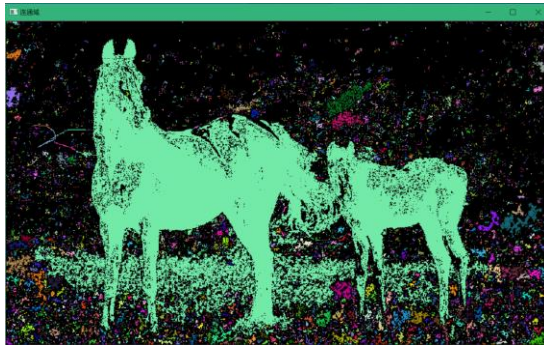
```
stack<pair<int, int>> neighborlabel;
neighborlabel.push(make_pair(i, j));
label++;
//cout << label << endl;
while (!neighborlabel.empty()) {
    pair<int, int> current = neighborlabel.top();
    int x = current.first, y = current.second;
    dst.at<int>(x, y) = label;
    neighborlabel.pop();
    //将邻域入栈
    if (x == 0 || y == 0 || x >= rows || y >= cols) continue;
    if (dst.at<int>(x - 1, y - 1) == 1)
    {
        neighborlabel.push(pair<int, int>(x - 1, y - 1));
    }
    .....
    .....
}
```

实验结果：

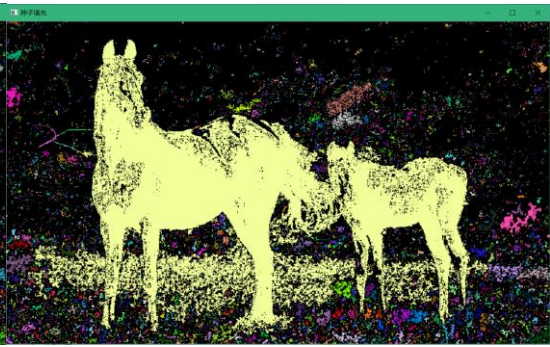
原图：



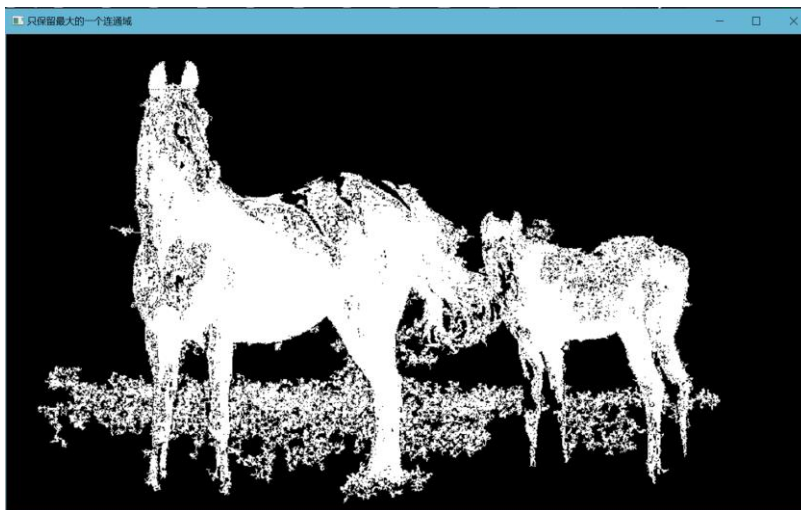
快速连通域算法



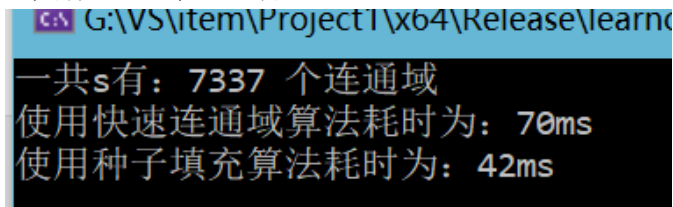
种子填充算法



最大的连通区域



一共有 7337 个连通域



通过速度对比发现种子填充比快速连通域算法更快。

实验 5.2

实验要求

- 了解 OpenCV 的距离变换函数 `distanceTransform`，使用合适的测试图像进行测试，并将距离场可视化输出。

实验步骤：

第一步：

将输入的图像转化为灰度图像

```
cvtColor(srcImage, grayImage, COLOR_BGR2GRAY);
```

第二步:

将灰度图像转化为二值图像, 将灰度值大于 127 的转化为 255, 小于 127 的转化为 0。

```
threshold(grayImage, binaryImage, 127, 255, THRESH_BINARY);
```

第三步:

计算距离矩阵, 采用欧氏距离

```
distanceTransform(binaryImage, image_dis, DIST_L2, DIST_MASK_PRECISE);
```

第四步:

因为距离矩阵的 Mat 类型为 32FC1, 我们最后要输出的图像的类型应该为 8UC1, 所以要进行类型转化。

```
Mat dstImage = Mat::zeros(binaryImage.size(), CV_8UC1);
for (int i = 0; i < image_dis.rows; i++)
{
    for (int j = 0; j < image_dis.cols; j++)
    {
        dstImage.at<uchar>(i, j) = image_dis.at<float>(i, j);
    }
}
```

第五步:

现在的 dstImage 图像偏暗, 为了使图像更加清晰, 进行归一化处理, 使数据分布在 0~255 之间。

```
normalize(dstImage, dstImage, 0, 255, NORM_MINMAX);
```

第六步: 分析 opencv 的距离变换函数 distanceTransform

DistanceTransform 函数的大致实现思路如下:

第一遍从左上角开始扫描, 按行遍历图像, 计算下式:

$$f(p) = \min[f(p), D(p, q) + f(q)]$$

第二遍从右下角开始, 从右向左扫描

根据上述结果得到最终图像

根据这个思路, 我自己实现了一个简单地距离变换函数, 两次扫描代码如下

```
//第一遍遍历图像, 使用左模板
for (int i = 1; i < rows - 1; i++)
{
    pDataOne = temp.ptr<uchar>(i);
    for (int j = 1; j < cols; j++)
    {
        pDataTwo = temp.ptr<uchar>(i - 1);
```

```

        dis_temp = eu(i, j, i - 1, j - 1);
        dismin = min((float)pDataOne[j], pDataTwo[j - 1] + dis_temp);

        dis_temp = eu(i, j, i - 1, j);
        dismin = min(dismin, pDataTwo[j] + dis_temp);

        pDataTwo = temp.ptr<uchar>(i);
        dis_temp = eu(i, j, i, j - 1);
        dismin = min(dismin, pDataTwo[j - 1] + dis_temp);

        pDataTwo = temp.ptr<uchar>(i + 1);
        dis_temp = eu(i, j, i + 1, j - 1);
        dismin = min(dismin, dis_temp + pDataTwo[j - 1]);
        pDataOne[j] = (uchar)cvRound(dismin);
    }
}
//第二遍使用右模板,从右下角开始
for (int i = rows - 2; i > 0; i--)
{
    pDataOne = temp.ptr<uchar>(i);
    for (int j = cols - 2; j >= 0; j--)
    {
        pDataTwo = temp.ptr<uchar>(i + 1);
        dis_temp = eu(i, j, i + 1, j);
        dismin = min((float)pDataOne[j], dis_temp + pDataTwo[j]);
        dis_temp = eu(i, j, i + 1, j + 1);
        dismin = min(dismin, pDataTwo[j + 1] + dis_temp);

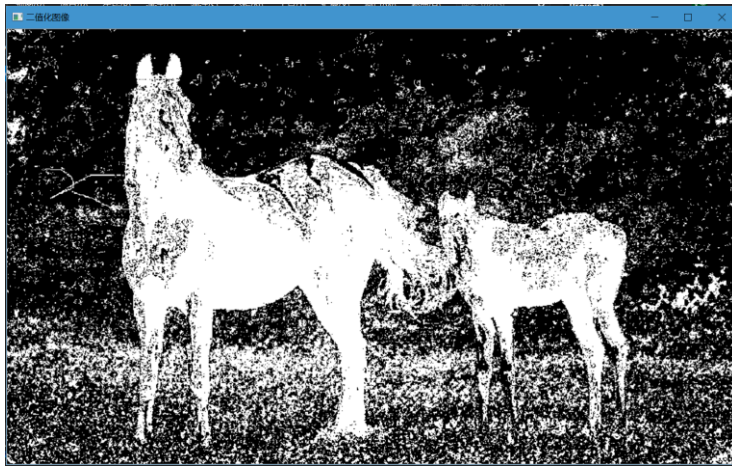
        pDataTwo = temp.ptr<uchar>(i);
        dis_temp = eu(i, j, i, j + 1);
        dismin = min(dismin, pDataTwo[j + 1] + dis_temp);

        pDataTwo = temp.ptr<uchar>(i - 1);
        dis_temp = eu(i, j, i - 1, j + 1);
        dismin = min(dismin, pDataTwo[j + 1] + dis_temp);
        pDataOne[j] = (uchar)cvRound(dismin);
    }
}

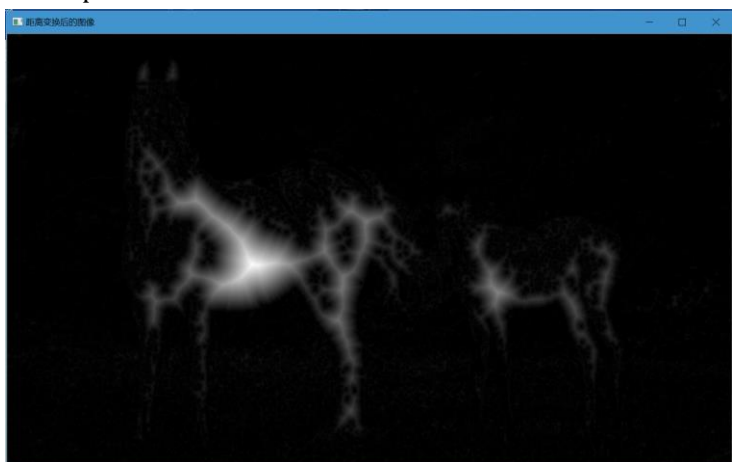
```

实验结果

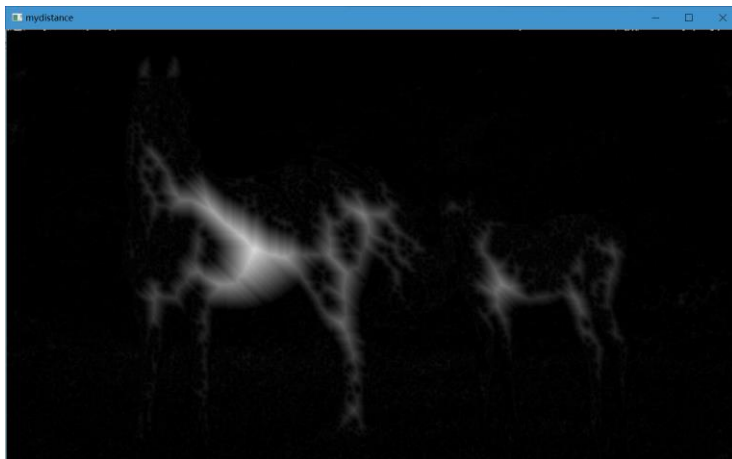
二值化的图像



使用 opencv 的 distanceTransform 函数计算出来的距离场



使用自己的编写的 distanceTransform 函数计算出来的距离场



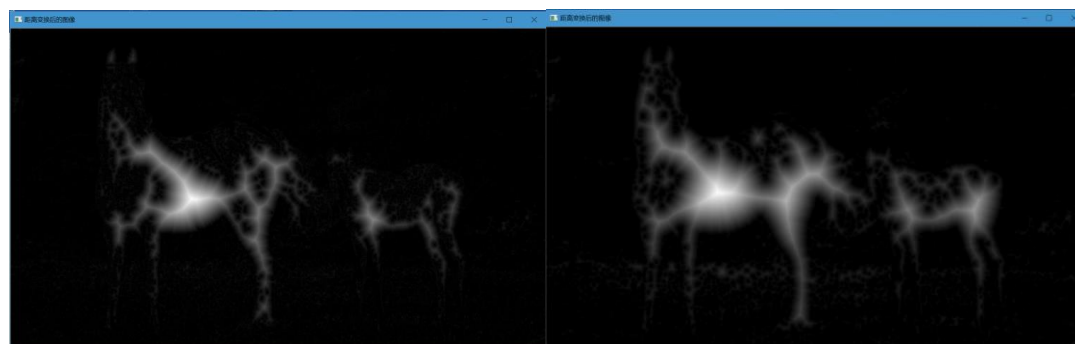
可以发现，自己编写的函数计算出来的距离场与 opencv 自带的函数效果虽然有区别，但整体差不多，说明思路是正确的。

另外，这个距离场常用在物体进行提取骨架处理方面，我看网上在进行骨架提取的时候经常对灰度图像先进行高斯模糊操作，说是可以提升骨架提取的效果。于是我也尝试先把回复图像进行高斯模糊处理。

对比如下：

未使用高斯滤波

使用高斯滤波



通过对比发现，使用高斯滤波后计算出来的距离场，骨架范围更大，骨架更粗，细节上不如未使用高斯滤波的版本。

结果分析与体会：

5.1

快速连通域算法：

扫描两遍图像，就可以将图像中存在的所有连通区域找出并标记。思路：第一遍扫描时赋予每个像素位置一个 label，扫描过程中同一个连通区域内的像素集合中可能会被赋予一个或多个不同 label，因此需要将这些属于同一个连通区域但具有不同值的 label 合并，也就是记录它们之间的相等关系；第二遍扫描就是将具有相等关系的 equal_labels 所标记的像素归为一个连通区域并赋予一个相同的 label（通常这个 label 是 label_equal 中的最小值）

种子填充：

种子填充方法来源于计算机图形学，常用于对某个图形进行填充。思路：选取一个前景像素点作为种子，然后根据连通区域的两个基本条件（像素值相同、位置相邻）将与种子相邻的前景像素合并到同一个像素集合中，最后得到的该像素集合则为一个连通区域。

5.2

DistanceTransform 函数的大致实现思路如下：

第一遍从左上角开始扫描，按行遍历图像，计算下式：

$$f(p) = \min[f(p), D(p, q) + f(q)]$$

第二遍从右下角开始，从右向左扫描

根据上述结果得到最终图像

根据这个思路，我自己编写了计算距离场的函数，最终效果与 opencv 自带的函数效果差不多。

并且对比了高斯模糊版本和非高斯模糊版本效果的不同。通过对比发现，使用高斯滤波后计算出来的距离场，骨架范围更大，骨架更粗，细节上不如未使用高斯滤波的版本。