山东大学 计算机科学与技术 学院

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 | | |
| 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。）  **实验5.1**  **实验要求：**   * 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：   + 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；     **实验步骤：**  **第一步：阈值分割**  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。  threshold(img, img\_binary, 127, 1, THRESH\_BINARY);  **第二步：快速连通域算法**  void two\_pass(const Mat& src, Mat& dst)  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curLable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)] q为模板像素  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，说是可是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)] q为模板像素  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。 | | |