目录

[一、问题描述 1](#_Toc22302)

[二、 实现工具及平台 1](#_Toc16248)

[三、 具体功能 1](#_Toc32359)

[四、 算法原理 1](#_Toc4821)

[4.1 Live-wire算法 1](#_Toc30285)

[4.1.1 cost计算 1](#_Toc27729)

[4.1.2 Dijkstra（堆优化）原理 3](#_Toc24813)

[4.2 撤销实现 4](#_Toc5968)

[4.3 图像扣取实现 5](#_Toc12018)

[五、 时间复杂度及优化 5](#_Toc4445)

[六、 遇到问题及解决 6](#_Toc8476)

[七、 讨论 7](#_Toc29709)

[八、 结论 8](#_Toc16155)

一、问题描述

学习图像处理基本知识，学习使用OpenCV工具，并实现利用live-wire算法实现Photoshop中的磁性套索工具。

1. 实现工具及平台

平台：visual studio 2017

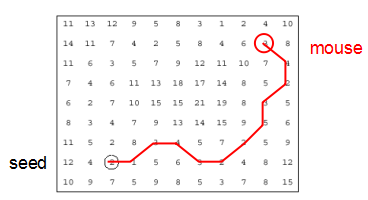
工具：OpenCV3.4.2

1. 具体功能

可执行文件启动后加载图片，左键点击图片中一点，滑动鼠标，红色的边缘检测线自动跟随鼠标检测边缘，再次左键点击一个点，前面检测的边缘固定，并用新的Seed开始检测边缘。点击右键，边缘线引伸到第一个点的点上，形成一个闭环，这时候点击回车键，就会自动把环里面的物体自动扣取出来。还可以按delete键进行撤销操作。

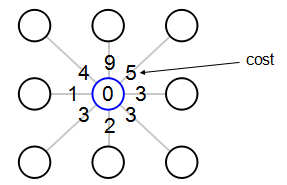
1. 算法原理

4.1 Live-wire算法



利用最短路算法dijkstra的原理，找出一个像素点到另一个像素点的最短路径，而计算出来的这个最短路径就是图像中物体的边缘。

4.1.1 cost计算



**l(p,q) = ωe · fe(q) + ωd · fd (p,q) + ωg · fg(q)** (fe代替论文中fz)

其中l(p,q)表示的就是p到q的局部距离cost（p、q像素点相邻）

we、wd、wg都为自定义的参数。

fe计算：

将图像经过canny算子检测到的边缘设为1，其余设为0得到。



fg梯度强度计算：

通过sobel算子计算出图像x、y两个方向的梯度，相加再开方得到梯度大小。



因为求cost，反向归一化。



fd梯度方向计算：

这一项其实是个平滑项，会给变化剧烈的边缘赋一个比较高的cost，让提取的边缘避免噪声的影响。具体公式如下：



其中：

D(p)表示与p点梯度方向垂直的单位向量



L(p, q) = { q – p; if D(p) · (q – p) ≥ 0,

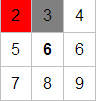
p – q; if D(p) · (q – p) < 0

dp(p, q) = D(p) · L(p, q)

dq(p, q) = L(p, q) · D(q)

斜对角方向cost修正：

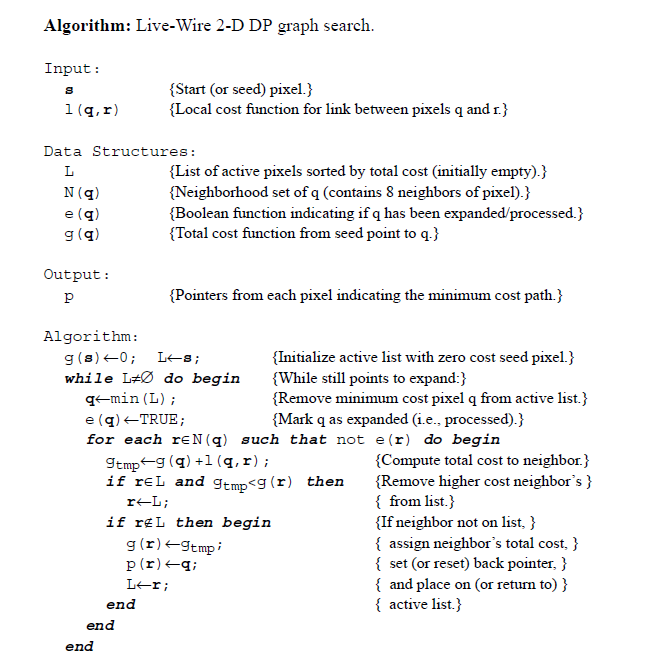
对于处于对角线上的点，由于欧氏距离相比于四个方向相邻的点为√2倍，所以相应计算的局部梯度值需要乘1/√2，达到和相邻四个方向一致。

(图中中心到左上角的梯度值应为6-(6-2)\*1/√2 = 3.17)

4.1.2 Dijkstra（堆优化）原理

1. 设立一个起始点s；
2. 将除了s的所有点到起始点的距离d都设为INF；
3. 建立一个优先队列que，队列中元素按照点到s的距离升序排序；
4. s压入优先队列que；
5. 如果que为空，结束程序，否则取出队头元素a；
6. 如果存储的a的d小于当先取出的a的d，那么goto 5；
7. 通过a计算a的邻接元素的d，如果小于原本存储的d，那么入优先队列，并更新d；
8. goto 5.

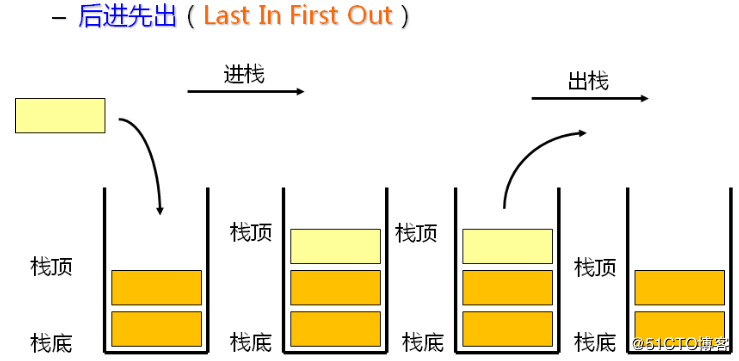
伪代码：



4.2 撤销实现

结构体数组模拟一个栈，有新的图像覆盖的时候，把图像入栈，当撤销的时候，将栈顶图像出栈，将现有图像覆盖。

|  |
| --- |
| struct img\_Stack //模拟栈  {  Mat img[15];  Point seed[15];  int sz;  }Sta; |



4.3 图像扣取实现

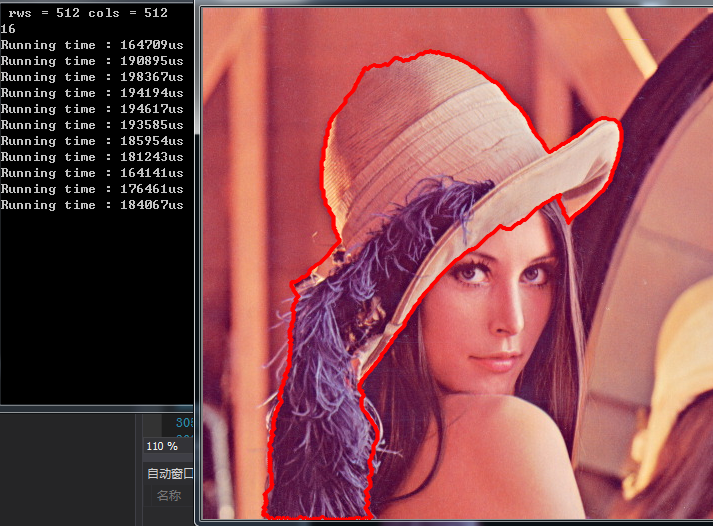
建立一个新Mat，将live-wire选取的边缘上的点Mat的值设为0，其余为1，在圈外选取一点，进行搜索（本代码采用宽搜bfs），搜到一点将它的Mat设为0，如果它的Mat值本来就是0，则回溯。直到圈外所有点的Mat值都变为0，结束搜索。 将Mat值为1的像素值显示出来，0的不显示。

1. 时间复杂度及优化

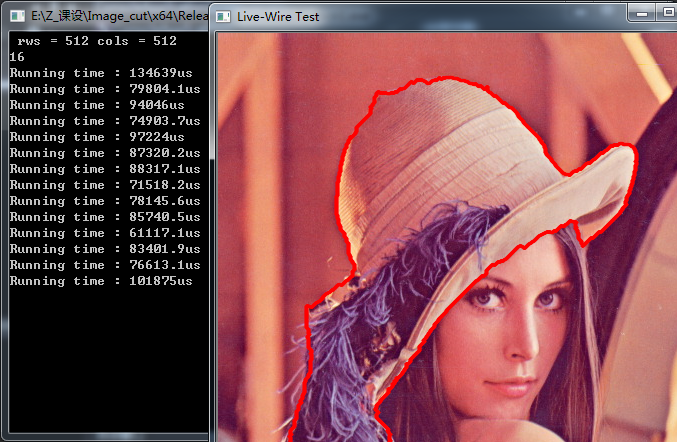
堆优化dijkstra的时间复杂度为O((n+m)logn)，其中n为点数，m为边数。在本算法中w为图片宽度，h为图片高度，则时间复杂度为O((h\*w + 8\*h\*w)log(h\*w)) =O(9h\*wlog(h\*w))。

对于一张512\*512的图像，在不加优化的情况下，每做一次dijkstra的时间在190ms左右，这一运算时间会在加大图片尺寸的时候成倍地增加。注意到如果每次dijkstra都需要计算一遍所有边的cost的话，那会增大dijkstra的常数。所以，我们把所有边的cost都预处理出来，预处理的时间复杂度为O(8h\*w)。这么做虽然不能减小算法的时间复杂度，但是却极大地减小了算法中的常数，从而实际上减小算法运行时间。**运行结果显示平均时间为80ms，效率提高一倍以上！**

此外，在dijkstra判断方向的时候巧妙使用位运算也可以减小运算时间；像素点遍历时采用ptr来代替at提高访问效率。

改进前：

改进后：



（其中第一次的Running time为预处理的时间）

1. 遇到问题及解决
2. 中间在使用Mat的ptr和at操作的时候遇到过越界的问题，发现是Mat没有初始化导致的。
3. Mat数据类型不匹配导致赋值错误。
4. 在写完后发现dijkstra运行很慢，寻找原因后发现是在debug版本下STL库中的priority\_queue运行慢导致，切换为release版本即可；或者重写优先队列，可以再debug模式下达到同样速度。
5. 第一次将fd的Mat数据类型写成了double，结果出现了两点距离大的时候检测到的线混乱的状况。



改为int后问题解决了。

1. 实际上像高斯模糊这些操作对Live-wire算法实质影响不大，起到优化作用，真正影响大的是在dijkstra时局部距离的计算，即l(p,q)的计算，而且并不是需要非常精确，在一段范围内边缘线的绘制效果都差不多。
2. 讨论

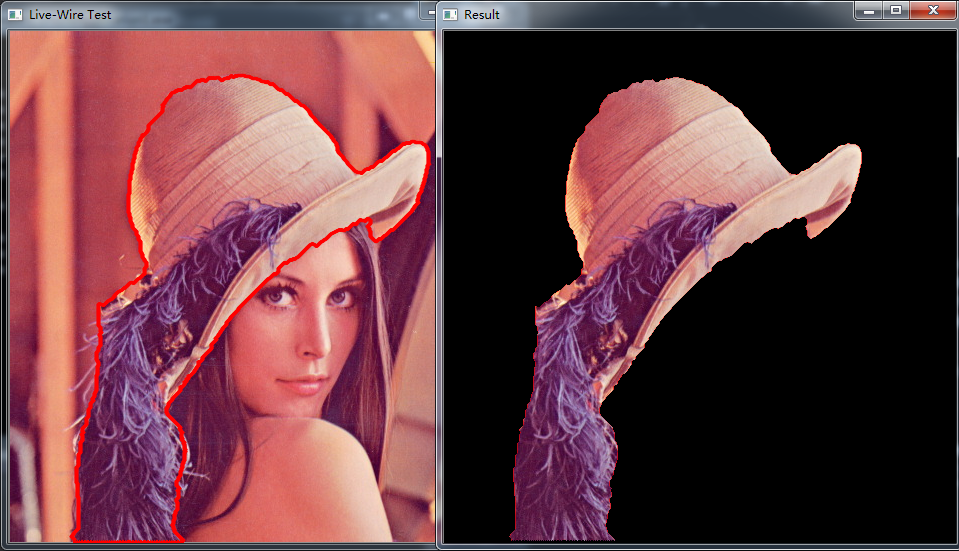
live-wire算法是在灰度图像层面上进行操作的，并没有充分利用彩色图像多通道的特性，或许改写算法使其能够利用**彩色信息**能够获得更好的效果。

其次，因其原理是dijkstra，所以在一些**曲率较大**的图形区域，算法并不能做的非常好，需要人工多点一些点，这也是算法不足的地方。

但总体来说，本算法能够解决一些实际需求，在学习过程中也学习到了很多新东西、新知识，非常不错。

1. 结论

live-wire算法是基于dijkstra的一种边缘检测算法，主要利用图像梯度值等，计算每个点和其相邻点的cost，寻找从原点到终点的最短路来找到图像中的边缘，这样做可以较精确地得到图像边缘，并且在一般尺寸图像上运行效率可观，是一个不错的边缘检测算法。加上一定的优化可以提高运行效率。在生产生活中的用处广泛。



（效果图）