南京理工大学计算机科学与工程学院

算法分析课程设计 报告

题　　目 智能剪刀

起止时间 2020.09.02-2020.10.12

指导教师 孔慧

学生姓名 黄海浪

学生学号 9181040G0818

目 录

1.引言

随着时代的不断进步,科学技术水平的不断提升,计算机技术也得到了长足的发展。在众多领域诸如医学分析、摄影工作者、工业分析中图像的使用智能套索工具对图片重要信息进行提取显得尤为重要。全自动的一般图像分割是一个悬而未决的问题，原因是图像来源，内容和复杂性的理论，因此已经让位于各种半自动匹配的方法，初始化方案等。在许多情况下，只能手动分段（即跟踪），将图像分量从复杂的背景提取出来，利用高水平的智能细分工具视觉专长，只需要最少的用户交互即能满足用户的独特需求。

本文详细介绍了一种交互式的数字图像分割工具，这种工具被成为图像的“套索”，而英文名字叫“livewire”，这里将其称为“智能剪刀”。此工具可从任意复杂的背景中快速提取对象边界，其中边界为表示为图中的最佳路径。该工具的优势是能够在人的控制下智能地完成图像边界的处理，通过人为操作选取的种子点计算到此点的距离最终获得每个像素点到种子点的最短路径，当人为操作时能够直接在图片中勾勒出边界部分，智能地交互式地完成图像的分割与截取功能。

本文处理方式是将整个图像看作一个无向图，通过canny算子检测连通的像素，再利用梯度方向与梯度强度对图像进行处理，通过最后的斜对角的cost修正后最终得到相邻像素之间的一个局部cost，通过此cost在图像中运用Dijkstra算法计算出种子点到该点的总cost的最短路径即作为划分图像的标准。用户选取种子点时为了最大程度地减少手动选择所需的用户交互，种子点通常是通过边界“冷却”沿当前活动边界线自动设置。边界当边界的当前交互部分的一部分未更改时，发生冷却最近并因此“冻结”，沉积新的种子点，同时重新启动最佳路径扩展。

本文介绍了整个“智能剪刀”算法的原理、处理、以及python实现。并在最终给出了比较完美的结果。

2.计算理论概述

2.1 图像的基础知识

数字图像处理是指将图像信号转换成数字信号并利用计算机对其进行处理的过程。图像处理最早出现于 20 世纪 50 年代，当时的电子计算机已经发展到一定水平，人们开始利用计算机来处理图形和图像信息。数字图像处理作为一门学科大约形成于 20 世纪 60 年代初期。早期的图像处理的目的是改善图像的质量，它以人为对象，以改善人的视觉效果为目的。图像处理中，输入的是质量低的图像，输出的是改善质量后的图像，常用的图像处理方法有图像增强、复原、编码、压缩等。

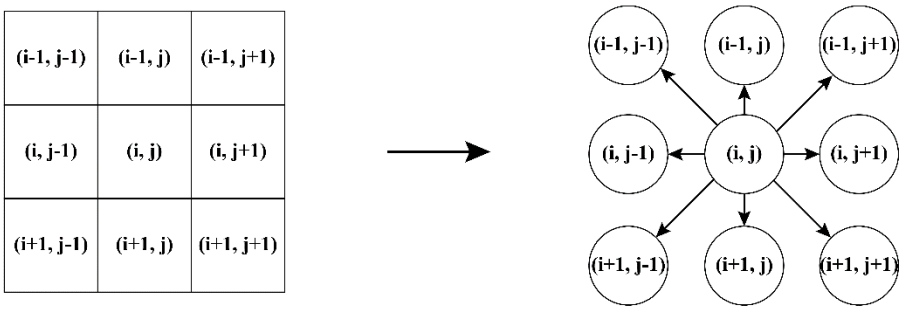
图像的属性有亮度、对比度、饱和度、灰度值、RGB、位深度/每像素位数、像素、分辨率、DPI、位图、RGB 与位图、CMKY 与位图、Alpha 通道、图像数据、图像矩阵等。其中彩色图像的每个元素（像素）都是一个 RGB 值，RGB 每个分量占一个字节，共三个字节。而灰度图像中，值的范围 0-255，灰度值为 0 代表黑色，255 代表白色，0-255 即黑-灰-白的程度过渡。.bmp 格式文件即位图没有灰度图这个概念，但是可以在.bmp 文件中表示灰度图。方法是用 256 色的调色板，没一项的 RGB 值都是相同的，也就是说 RGB 值从(0，0，0)，(1，1，1)一直到(255，255，255)。(0，0，0)是全黑色，(255，255，255)是全白色，中间的是灰色。这样，灰度图就可以用 256 色图来表示了。可以理解为灰度图像为特殊的 RGB 图。像素是图像中不可再分割的单元，每幅图像都是由许许多多的像素点构成的，它是以一个单一颜色的小格存在，灰度图用一个字节的容量来存储一个像素，24 真彩色 RGB 则用三个字节来存储一个像素。在本文的算法处理中，将会涉及到上述部分图像的属性，故之后不再过多赘述。

数字图像处理常用方法有图像变换、图像编码压缩、图像增强和复原、图像分割、图像描述和图像分类（识别）。而图像分割是数字图像处理中的关键技术之一。图像分割是将图像中有意义的特征部分提取出来，其有意义的特征有图像中的边缘、区域等，这是进一步进行图像识别、分析和理解的基础。虽然目前已研究出不少边缘提取、区域分割的方法，但还没有一种普遍适用于各种图像的有效方法。因此，本文基于交互式的“智能剪刀”设计将在增强效果的同时减少人为的工作量。

2.2 基本思想

基本思想是把二维图像看作无向图,像素点(pixel)作为节点(node),相邻的像素点有边(edge)相连,边上根据一定的规则赋以不同的权重(weight)/损失(cost)以表示从一点到边的另一点所需要的能量(energy)。相邻像素之间就可以计算出一个局部 cost，于是就转化成了最短路径问题了，接下来就是基于 Dijkstra 算法产生路径，就是需要提取的边缘。主要涉及的算法有两部分：1)

相邻像素的 cost 计算；2)最短路径算法。



智能剪刀的操作流程如下:

1.根据图像构造图,包括节点和边,使用八联通.

2.为边计算损失,包括静态权重和动态权重.

3.用户在待分割目标的边界给定初始种子点

4.即时计算上一种子点到鼠标点间的最短路径,移动鼠标直到用户认为最短路径恰好就是图像的边界,点击鼠标生成下一个种子点。

5.重复第 4 步直到目标物体的轮廓勾画完成。

2.3 算法计算

2.3.1 边缘检测

边缘检测的算法主要是基于图像强度的一阶和二阶导数，但导数通常对噪声很敏感，因此必须采用滤波器来改善与噪声有关的边缘检测器的性能。常见的滤波方法主要有高斯滤波，即采用离散化的高斯函数产生一组归一化的高斯核，然后基于高斯核函数对图像灰度矩阵的每一点进行加权求和。计算相邻像素 cost 的最终目的还是为了寻找边缘，所以本质还是边缘检测。从最短路径的角度来说，就是边缘越明显的地方，cost 的值越小。本文采用三种指标求加权：（1）边缘检测算子（2）梯度强度(Gradient Magnitude)（3）梯度方向(Gradient Direction)。其中边缘检测算子本文使用了 OpenCV-python 中的 Canny 算子检测边缘。表达式如下：

$$l(p,q) = \omega_Z \cdot f_Z(q) + \omega_D \cdot f_D(p,q) + \omega_G \cdot f_G(q)$$

名	说明	名	说明
$f_Z(q)$	Canny 算子检测边缘	W_Z	加权值，0.425
$f_D(p, q)$	梯度方向	W_D	加权值，0.15
$f_G(q)$	梯度值	W_G	加权值，0.425

2.3.2 Canny 算子

从表面效果上来讲，Canny 算法是对 Sobel、Prewitt 等算子效果的进一步细化和更加准确的定位。Canny 算法基于三个基本目标：低错误率。所有边缘都应被找到，且没有伪响应。边缘点应该被很好地定位。已定位的边缘必须尽可能接近真实边缘。单一的边缘点响应。这意味在仅存一个单一边缘点的位置，检测器不应指出多个像素边缘。

进而，Canny 的工作本质是，从数学上表达前面的三个准则。因此 Canny 的步骤如下：对输入图像进行高斯平滑，降低错误率。计算梯度幅度和方向来估计每一点处的边缘强度与方向。根据梯度方向，对梯度幅值进行非极大值抑制。本质上是对 Sobel、Prewitt 等算子结果的进一步细化。用双阈值处理和连接边缘。

基本思想是根据梯度信息，先检测出许多连通的像素，然后对于每一块连通的像素只取其中最大值且连通的部分，将周围置零，得到初始的边缘(Edges)，这个过程叫做 Non-Maximum Suppression。然后用二阈值的办法将这些检测到的初始边缘分为 Strong, Weak, and None 三个等级，顾名思义，Strong 就是很确定一定是边缘了，None 就被舍弃，然后从 Weak 中挑选和 Strong 连通的作为保留的边缘，得到最后的结果，这个过程叫做 Hysteresis Thresholding。

它是一个经典的算法，公式如下：

$$f_Z(\mathbf{q}) = \begin{cases} 0; & \text{if } I_L(\mathbf{q}) = 0, \\ 1; & \text{otherwise.} \end{cases}$$

2.3.3 梯度强度

梯度强度，顾名思义，就是梯度求模而已，x 和 y 两个方向的梯度值平方相加在开方，公式如下：

$$G = \sqrt{I_x^2 + I_y^2}.$$

因为要求 cost，所以反向并归一化：

$$f_G = 1 - \frac{G - \min(G)}{\max(G) - \min(G)}$$

2.3.4 梯度方向

梯度方向对边界增加了一个光滑性约束，对边界方向发生突变的地方给以较大的损失. 用 D(p) 表示 p 点梯度的单位向量， D′ (p)表示与 D(p)正交的方向(顺时针旋转 90 度)，即

$$\mathbf{D}(\mathbf{p}) = [I_x(\mathbf{p}), I_y(\mathbf{p})], \quad \mathbf{D}'(\mathbf{p}) = [I_y(\mathbf{p}), -I_x(\mathbf{p})]$$

那么，梯度方向对应的损失定义为

$$f_D(\mathbf{p}, \mathbf{q}) = \frac{2}{3\pi}[\arccos(d_{\mathbf{p}}(\mathbf{p}, \mathbf{q})) + \arccos(d_{\mathbf{q}}(\mathbf{p}, \mathbf{q}))]$$

其中,

$$d_{\mathbf{p}}(\mathbf{p}, \mathbf{q})) = \mathbf{D}'(\mathbf{p}) \cdot \mathbf{L}(\mathbf{p}, \mathbf{q})$$
$$d_{\mathbf{q}}(\mathbf{p}, \mathbf{q})) = \mathbf{L}(\mathbf{p}, \mathbf{q}) \cdot \mathbf{D}'(\mathbf{q})$$

其中,

$$L(p, q) = \frac{1}{\|\mathbf{p} - \mathbf{q}\|} \begin{cases} \mathbf{q} - \mathbf{p}; & \text{if } \mathbf{D}'(\mathbf{p}) \cdot (\mathbf{q} - \mathbf{p}) \geq 0 \\ \mathbf{p} - \mathbf{q}; & \text{otherwise} \end{cases}$$

是个单位向量. 注意到反余弦 arccos 是 [-1, 1] 上单调递减的函数，该损失函数定义的方式使得:

- 边的两个端点的梯度方向一致并且与边的方向也一致时损失较低
- 边的两个端点的梯度方向一致但与边的方向正交时损失较高
- 边的两个端点的梯度方向不一致时损失较高

2.3.5 欧式距离(Euclidean Distance)

定义：欧几里得距离或欧几里得度量是欧几里得空间中两点间“普通”（即直线）距离。欧式距离是一个通常采用的距离定义，指在 m 维空间中两个点之间的真实距离，或者向量的自然长度（即该点到原点的距离），欧式距离公式如下：

$$d(x, y) = \sqrt{\sum_{k=1}^{n} (x_k - y_k)^2}$$

2.3.6 Cost 修正

在二维图像中，相邻的像素通常按照间隔欧式距离分为两种：1)上下左右相邻，间隔为像素边长；2)斜对角相邻，间隔为像素边长的$\sqrt{2}$在计算局部 cost 的时候通常要把这种距离差异的影响考虑进去，比如下面这幅图：

2	3	4
5	6	6
7	8	9

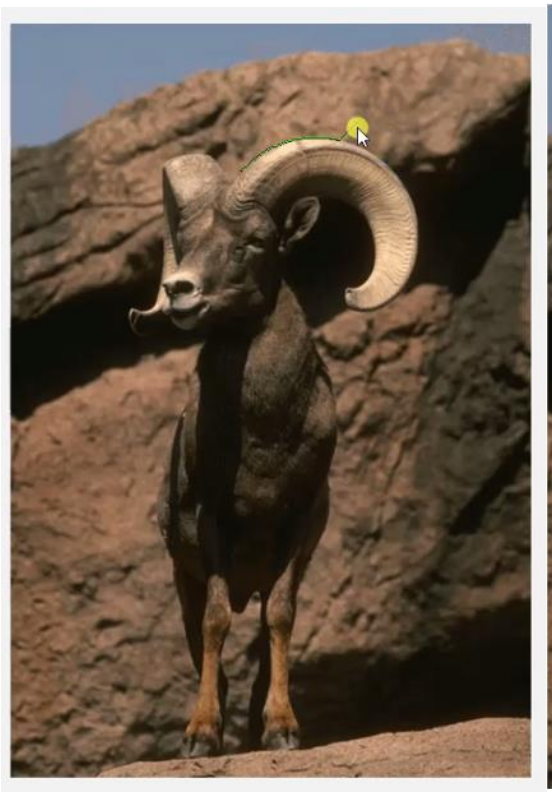
如果不考虑像素位置的影响，那么查找最小 cost 的时候会认为左上角的 cost=2 最小。然而如果考虑到像素间距的影响，我们来看左上角方向，和中心的差异是 6-2，做个线性插值的话，则左上角距中心单位距离上的值应该为：

$$6-(6-2)\times 1/\sqrt{2}\ =3.17>3$$

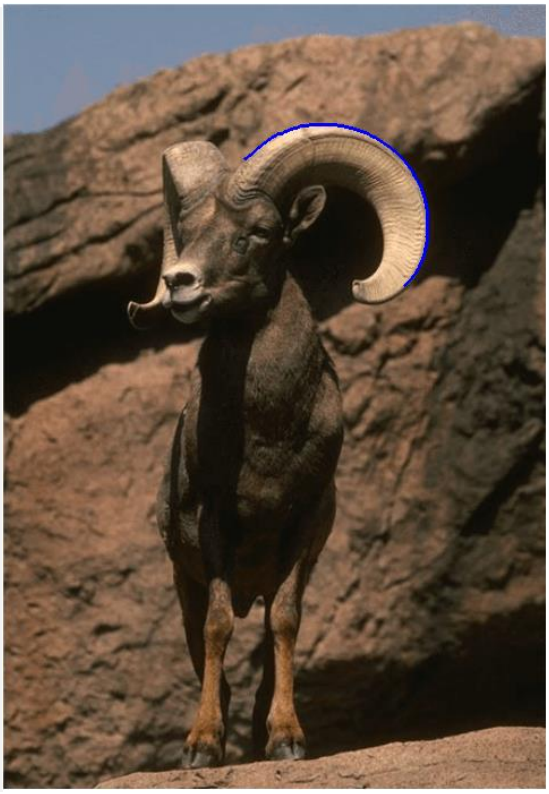
所以正上方的才是最小 cost 的正确方向。

2.3.7 Dijkstra 算法求最短路

在磁力套索中，一般的用法是先单击一个点，然后移动鼠标，在鼠标和一开始单击的点之间就会出现自动贴近边缘的线，这里我们定义一开始单击的像素点为种子点(seed)，而磁力套索其实在考虑上部分提到的边缘相关cost的情况下查找种子点到当前鼠标的最短路径。移动鼠标时，最贴近边缘的种子点和鼠标坐标的连线就会实时显示。

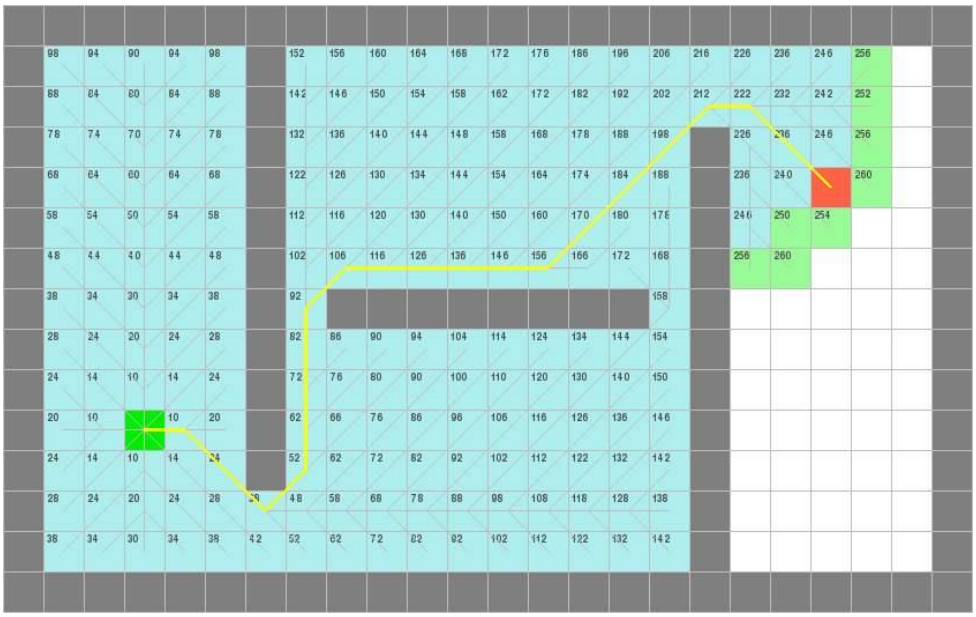


图表 1 用户正在操作

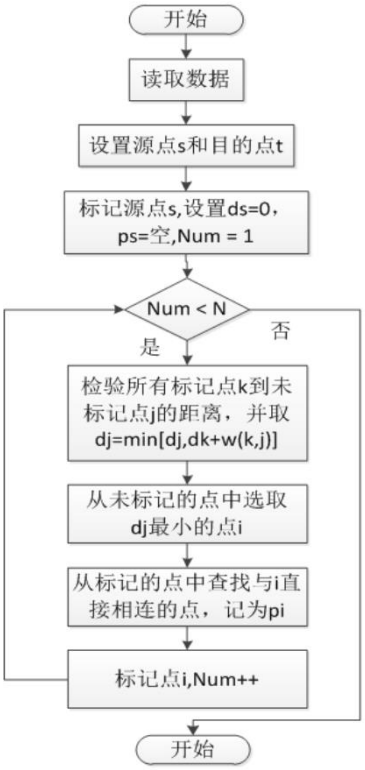


图表 2 用户操作完成

给定种子点后，执行Dijkstra算法将图像的所有像素遍历，得到每个像素到种子点的最短路径。以下面这幅图为例，在一个cost矩阵中，利用Dijkstra算法遍历每一个元素后，每个元素都会指向一个相邻的元素，这样任意一个像素都能找到一条到seed的路径，比如图所示两个像素，红色沿着花费cost最少的最短路径到达绿点。



图表 3 最短路示意图



图表 4Dijkstra 算法流程图

```
输入:
  s                // 种子点
  l(q,r)           // 计算局部cost

数据结构:
  L                // 当前待处理的像素
  N(q)             // 当前像素相邻的像素
  e(q)             // 标记一个像素是否已经做过相邻像素展开的Bool函数
  g(q)             // 从s到q的总cost

输出:
  p                // 记录所有路径的map

算法:
  g(s)←0; L←s;                    // 将种子点作为第一点初始化
  while L≠∅:                      // 遍历尚未结束
    q←min(L);                     // 取出最小cost的像素并从待处理像素中移除
    e(q)←TRUE;                    // 将当前像素记录为已经做过相邻像素展开
    for each r∈N(q) and not e(r):
      gtemp←g(q)+l(q,r);          // 计算相邻像素的总cost
      if r∈L and gtemp
        r←L; { from list.}        // 舍弃较大cost的路径
      else if r∉L:
        g(r)←gtemp;               // 记录当前找到的最小路径
        p(r)←q;
        L←r;                      // 加入待处理以试图寻找更短的路径
```

图表 5Dijkstra 算法伪代码

遍历的过程会优先经过cost最低的区域，所有像素对应的到种子像素的最短路径都找到后，移动鼠标时就直接画出到seed的最短路径就可以了。

3.编程语言预备与类库

3.1 Tkinter 窗口

Python 提供了多个图形开发界面的库，几个常用 PythonGUI 库有 Tkinter、wxPython、Jython。Tkinter 模块(Tk 接口)是 Python 的标准 TkGUI 工具包的接口。Tk8.0 的后续版本可以实现本地窗口风格,并良好地运行在绝大多数平台中。wxPython 是一款开源软件，是 Python 语言的一套优秀的 GUI 图形库，允许 Python 程序员很方便的创建完整的、功能健全的 GUI 用户界面。Jython 程序可以和 Java 无缝集成。除了一些标准模块，Jython 使用 Java 的模块。Jython 几乎拥有标准的 Python 中不依赖于 C 语言的全部模块。比如，Jython 的用户界面将使用 Swing，AWT 或者 SWT。Jython 可以被动态或静态地编译成 Java 字节码。

Tkinter 是 Python 的标准 GUI 库。Python 使用 Tkinter 可以快速的创建 GUI 应用程序。由于 Tkinter 是内置到 python 的安装包中、只要安装好 Python 之后就能 importTkinter 库、而且 IDLE 也是用 Tkinter 编写而成、对于简单的图形界面 Tkinter 还是能应付自如。Tkinter(也叫 Tk 接口)是 Tk 图形用户界面工具包标准的 Python 接口。Tk 是一个轻量级的跨平台图形用户界面(GUI)开发工具。Tk 和 Tkinter 可以运行在大多数的 Unix 平台、Windows、和 Macintosh 系统。

Tkinter 由一定数量的模块组成。Tkinter 位于一个名为_tkinter(较早的版本名为 tkinter)的二进制模块中。Tkinter 包含了对 Tk 的低级接口模块，低级接口并不会被应用级程序员直接使用，通常是一个共享库(或 DLL)，但是在一些情况下它也被 Python 解释器静态链接。

本文用到了关于 tkinter 窗口创建轻量级的操作程序。

3.2 Opencv-python

opencv-python 是一个用于解决计算机视觉问题的 python 绑定库。

Python 是由 Guido van Rossum 开发的一款通用编程语言，由于其编程的简洁性和可读性，很快地风靡全球。它可以让码农用少量的几行代码表达自己的想法并且不失可读性。与其他语言如 C/C++相比，Python 运行较慢。也就是说，python 可以轻易的使用 C/C++进行扩展，这将允许我们对高强度算力的代码使用 C/C++编程，并且创建 python 封装使其可以以 Python 模块的形式被调用。这种方式有两个优势：第一，代码与原生 C/C++代码运行一样快（因为它实际上在后台运行的是 C++代码）；第二，使用 Python 编程与 C/C++更容易。opencv-Python 是一个原生 OpenCV C++实现的 Python 封装器。OpenCV-Python 使用了 Numpy，numpy 是一个高度优化的数值运算库使用 matlab 语法风格。所有的 OpenCV 数组结构都将转化成 Numpy 数组或者由 Numpy 数组转化而来。这同样使得其可以轻松地与其他使用 Numpy 的库如 scipy 和 matplotlib 进行集成。

本文用到了 opencv 对图像进行基本操作以及 opencv 里面的 canny 算子。

3.3 Numpy 数学工具

numpy 提供了一个在 Python 中做科学计算的基础库，重在数值计算，主要用于多维数组（矩阵）处理的库。用来存储和处理大型矩阵，比 Python 自身的嵌套列表结构要高效的多。本身是由 C 语言开发，是个很基础的扩展，Python 其余的科学计算扩展大部分都是以此为基础。以下是其特点：

1.高性能科学计算和数据分析的基础包。

2.ndarray，多维数组（矩阵），具有矢量运算能力，快速、节省空间。

3.矩阵运算，无需循环，可完成类似 Matlab 中的矢量运算。

4.线性代数、随机数生成。

本文用到了 numpy 对矩阵进行运算，以及图像梯度的计算。

4.具体开发

4.1 模块式设计

本文使用 python 对其进行处理，算法部分直接调用了 OpenCV 的 Canny 函数和 Sobel 函数(求梯度)。代码分为三个部分，一个 main 函数作为启动入口，一个 livewire 类对图像进行数据计算与处理，以及路径的寻找，一个 GUI 类对整个窗口的进行显示以及与用户进行交互。

4.2 计算类

计算类为 livewire 类，实现的功能有 1.Canny 边缘检测 + 梯度幅度 + 梯度方向；2. Dijkstra 算法。模块有：1.使用 Sobel 运算符返回图像的梯度幅度；2. 返回像素 p 周围的 8 个邻居；3. 计算 梯度变化 refer to the link direction；4. 欧几里德距离 计算 Canny 边和梯度幅度成本。5. 从成本矩阵中获取整个图像的路径矩阵。

其中用到的变量再代码的 init 里面：

```
class Livewire():
    """
        套索工具类 使用
        1. Canny 边缘检测 + 梯度幅度 + 梯度方向
        2. Dijkstra 算法
    """

    def __init__(self, image):
        self.image = image  #传入的image
        self.x_lim = image.shape[0]  # 高
        self.y_lim = image.shape[1]  # 宽
        # cost矩阵 0-1的范围
        # https://www.cnblogs.com/my-love-is-python/p/10397482.html
        self.cost_edges = 1 - cv2.Canny(image, 85,
                                        170) / 255.0  # Canny opencv 自带 边缘检测
        self.grad_x, self.grad_y, self.grad_mag = self._get_grad(image)  # 使用Sobel运算符返回图像的梯度幅度

        self.cost_grad_mag = 1 - self.grad_mag / np.max(self.grad_mag)  # gard矩阵
        # 权重（Canny边缘，渐变幅度，渐变方向）
        self.weight = (0.425, 0.425, 0.15)

        self.n_pixs = self.x_lim * self.y_lim   # pixs
        self.n_processed = 0    # 进度
```

4.3 GUI 类

Gui 类使用 python 自带得 tkinter 库以创建轻量简单的窗口程序。其中模块主要有 1.打开图片按钮；2.保存图片按钮；3.canvas 作图窗口（显示图片和绘制点）4.绑定左键右键单击操作模块；5.cv2img 保存图片时的特殊处理（不同于 canvas 作图）。用到的变量具体如下：

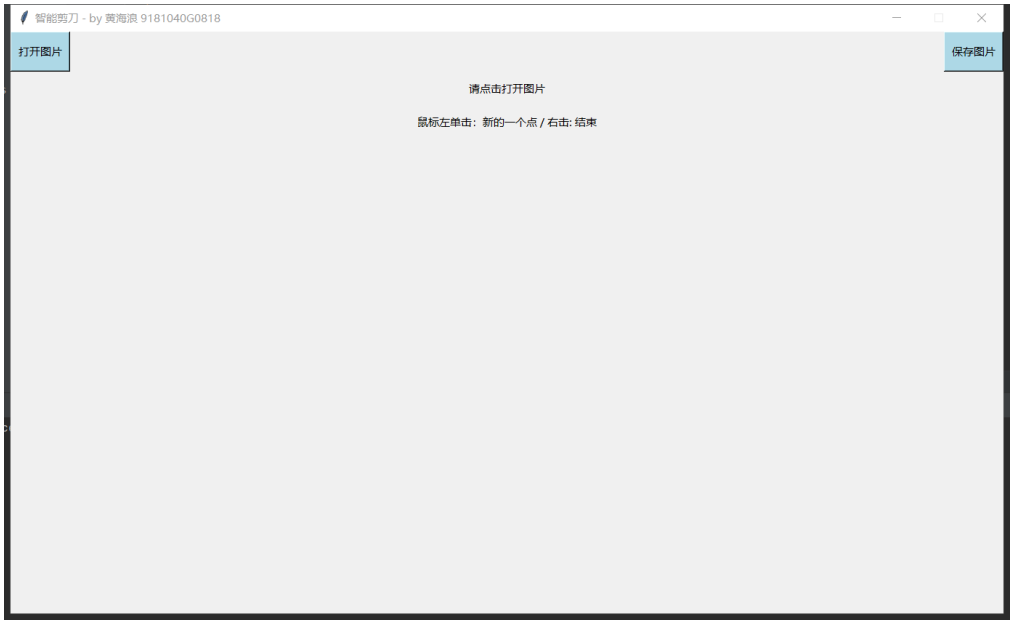
```python
class MY_GUI():
    # 创建一个窗口
    def __init__(self, init_window):
        self.tk = tk  # 调用tk方法使用
        self.mainWindow = init_window  # 初始化父窗口
        self.mainWindow.title("智能剪刀 - by 黄海浪 9181040G0818")  # 窗口名
        self.mainWindow.geometry('1367x800+280+90')  # 1367x800为窗口大小，+280+90 定义窗口弹出时的默认展示位置
        self.mainWindow.resizable(0, 0)  # 窗口不可调节大小

        self.imageFilePath = None  # 初始化图片地址
        self.cv2_image = None  # 原始读入图片      进行处理的图片
        self.pil_image = None  # 初始化调整后的显示图片 tk必须使用pil类型的
        self.lw = None  # 算法处理  # liveWire的算法类
        self.seed = None  # 点（之前的点） 种子
        self.path_map = {}  # 存放所有路径     计算的路径
        self.path = []  # 存放点击之前的路径   保存图片需要
        self.active = False  # 是否允许实时计算 计算有结果后为 True
        self.seed_enabled = True  # 是否允许点一个点     刚开始允许计算点
        self.finish_enable = False
        self.allOvals = []  # 画的点 用来重新计算时进行清除    move时清除的临时点
```

5.实现结果分析

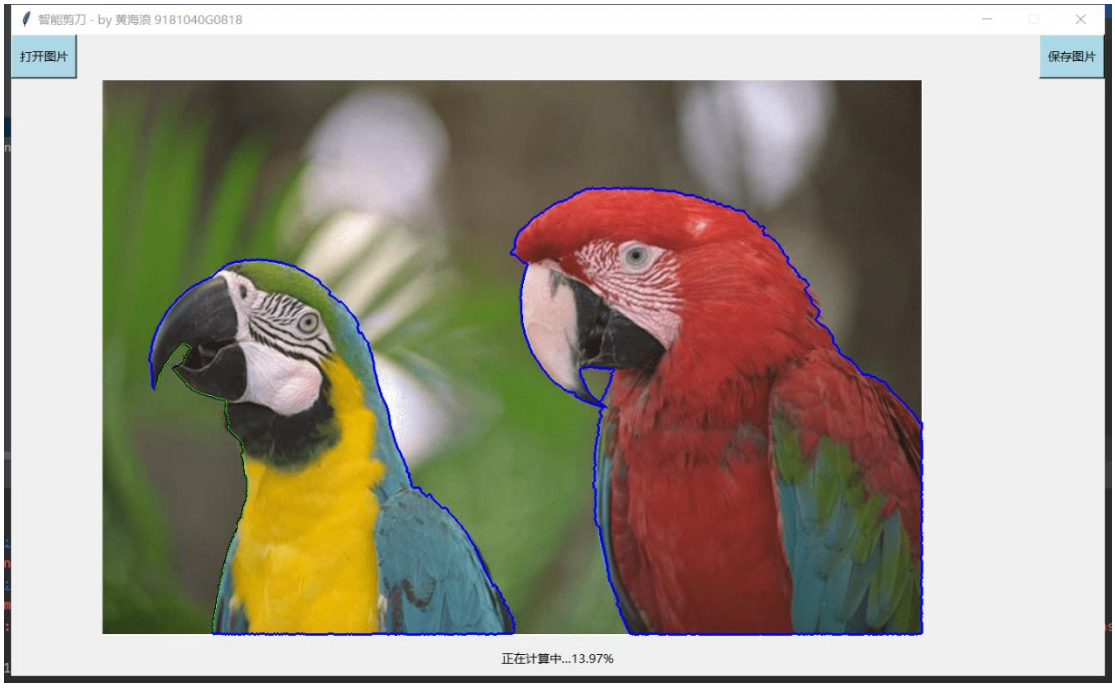
5.1 实验结果

运行 main 函数，将会出现如图所示的窗口，不同的机器可能因为电脑屏幕分辨率不同而导致按钮位置不同，可以修改代码也可以使用电脑屏幕分辨率为 1980*1080 的电脑。



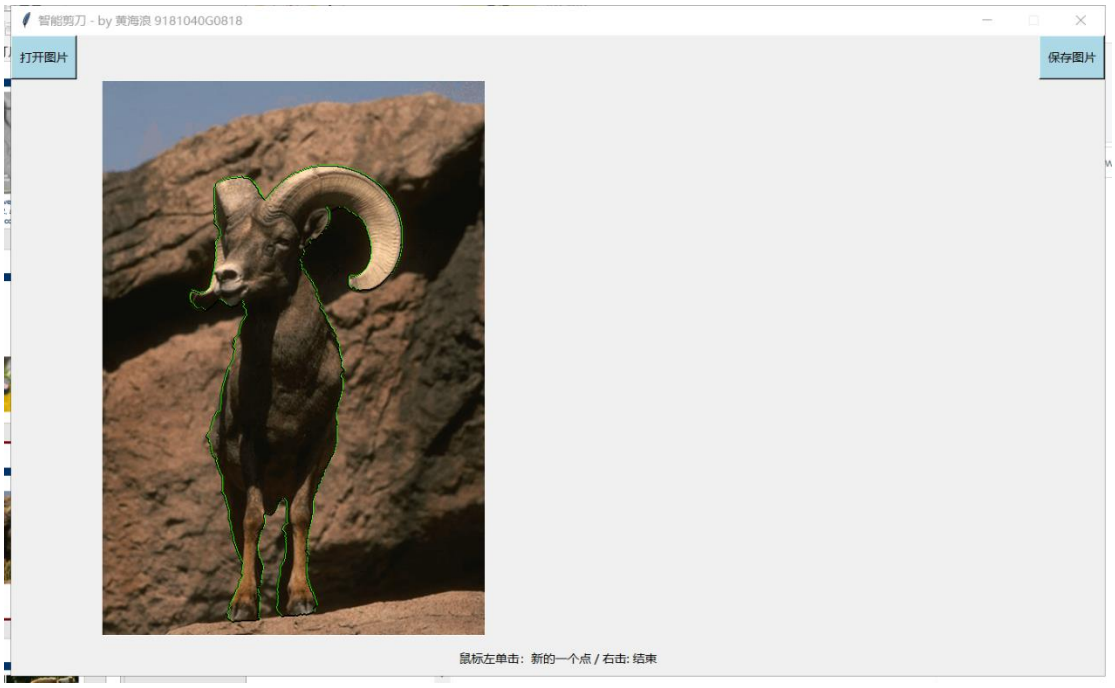
图表 6 刚启动时的窗口

如图为圈出一部分图像保存后继续进行运算，在底部有提示进度条。



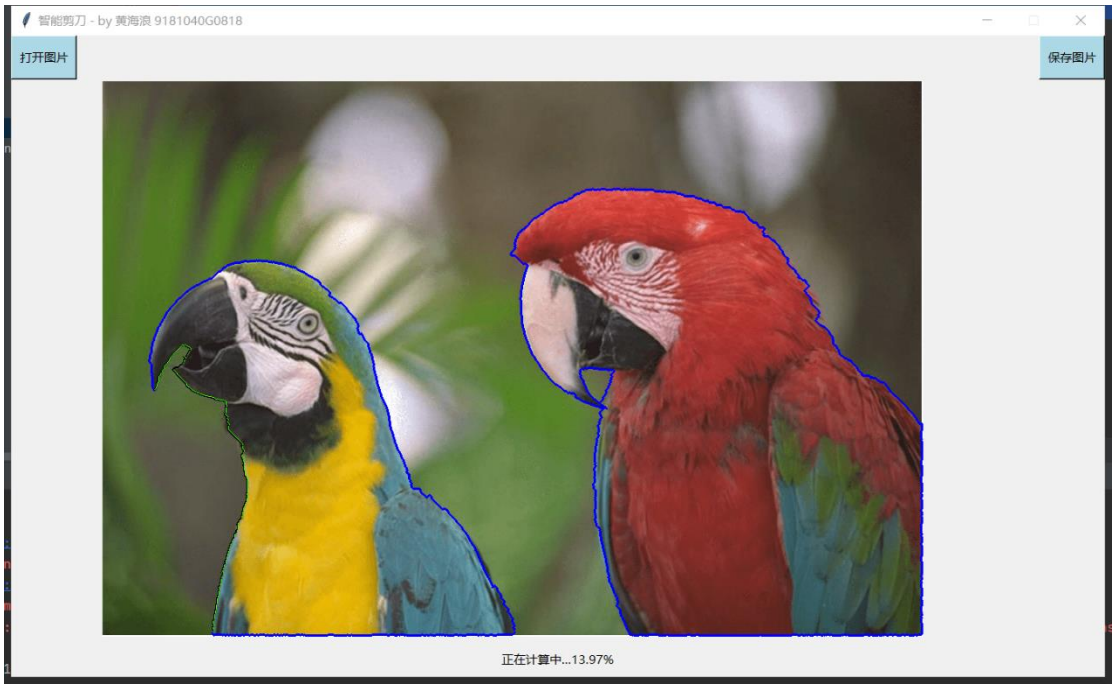
图表 7 处理一部分后正在计算的图片

在 canvas 上面作图完毕后右击鼠标得到的图片，之后依然可以使用鼠标左键对其他地方进行类似的处理。

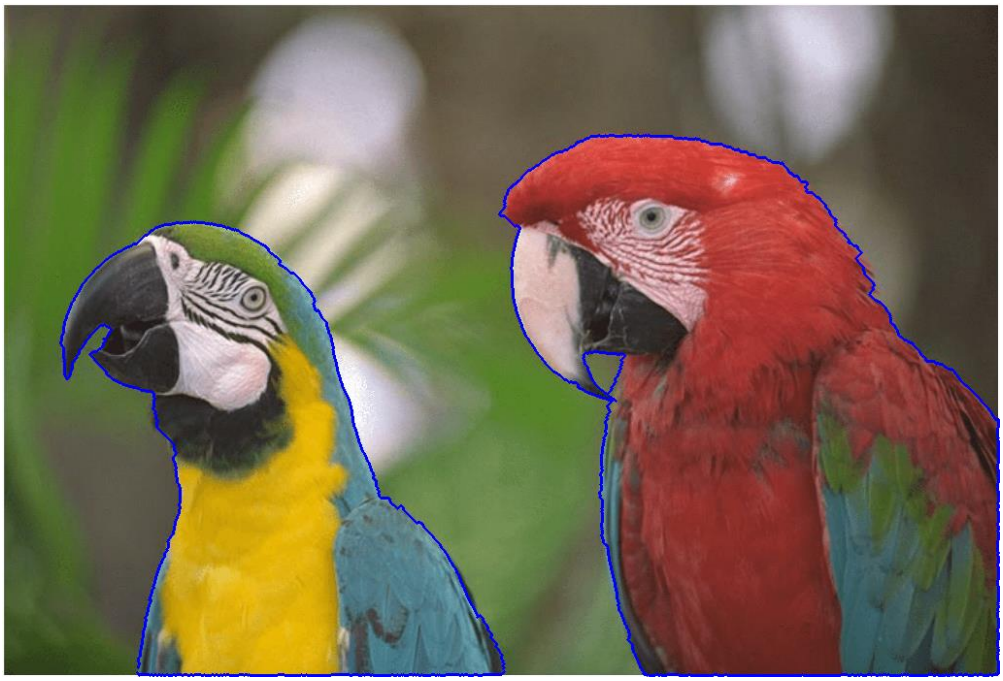


图表 8 鼠标右击后处理完成

两只鸟即将完成“剪切”。



图表 9 即将完成的“剪切”



图表 10 最终效果图 1

图表 11 最终效果图 2

5.2 不足与改进

获取轮廓是用的 Canny 算子直接检测边缘，调用的为 opencv 的库函数，并没有自己手动实现。边缘效果在某些情况下会多点一些点才能确定边缘，增加了与用户的交互。可以利用机器学习配合拉普拉斯交叉零点、canny 等对轮廓进行提取，减少与用户的交互，提高“智能性”。

算法上面的改进是本文每次都是对整幅图像进行计算，感觉是实际应用中没必要一上来就计算整幅图像，可以根据 seed 位置做一些区块划分，鼠标本身也会留下轨迹，也或许可以考虑只在鼠标轨迹方向进行启发式搜索。另外计算路径的时候也许可以考虑借鉴有点类似于 Image Pyramid 的思想，没必要一上来就对全分辨率下的路径进行查找。

Gui 部分不是特别美观，因为用的只是 python 自带的 tkinter 库，可以使用 pyqt 优化 Gui，提升用户交互感。

5.3 遇到的问题与解决方案

由于之前基本没有用 python 做过窗口程序，所以面对 Tkinter 有个不熟悉到熟悉的过程。所以导致花了较多时间取学习 Tkinter 窗口程序。原本的绘制 canvas 和取点和图片是对应不上的，之后通过经验换绑定组件（仅对 image 进行绑定）解决了这个问题。具体绑定图如下：

```
def updateCanvas(self):
    cc = self.mainWindow.canvas.create_image(0, 0, anchor=NW, image=self.mainWindow.tkImage)
    # 监听鼠标
    self.mainWindow.canvas.tag_bind(cc, "<Motion>", self.motionEvent)
    self.mainWindow.canvas.bind("<Button-1>", self.clickEvent)
    self.mainWindow.canvas.bind("<Button-3>", self.clickClearEvent)
    # 格式化窗口
    self.mainWindow.canvas.grid(row=20, column=1)
```

图表 12 绑定组件图

算法的实现，出现了边缘计算数组越界问题，每次遇到边缘都是我头疼的一个问题，最后不断在草稿纸上算并且不断实验解决了边缘处理的问题。

图片显示在窗口上失真，原本直接使用 pil 打开图片会出现 canvas（tk 只能显示 pil 类型的图片）显示模糊，使用 cv2 打开后格式化，再转化为 pil 的图片解决了这个问题。

多线程计算时候的限制，刚开始计算多线程的时候，如果不禁止用户随意点击会造成线程过多导致程序停止运行，于是每次计算时就禁止用户操作（除了保存图片）

算法类实现起来并不算太难，除了边缘问题，经过公式的铺垫实现代码起来还算顺畅。这告诉我以后写代码一定要先有理论支持！

6.个人小结

这次的课设，虽然在架构整个系统和写代码的能力方面没有提升特别多，但是却让我学会了很多算法以及图像处理的东西，尤其是感觉自己耐心和写报告的能力又变强了。之前做过图像处理，但是像这次这样深入去研究公式，研究其中的原理还是头一次，这让我发现了算法的乐趣，发现了它内在实现的闪光点，让我对原来的基础 c++，java 等内在实现又增加了兴趣，有一种想再回头深入研究一下的冲动。

从图像的特殊处理、图像去雾、人脸识别、视频处理，直到这次算法课让我了解了以前从来没做过的套索工具，体验感很不错，做起来很有意思，尤其是最短路径那块又让我回去看了看八竿子打不着的网络技术哈哈哈😄。这次代码就写了不到 500 行，但是研究却花了较多的时间，有了理论支持，算法部分写的真的比以往快多了。这是以前没有学会一定要有理论支持这件事！

最后感谢老师百忙之中抽空垂阅我的报告~