技术文档--车标定位

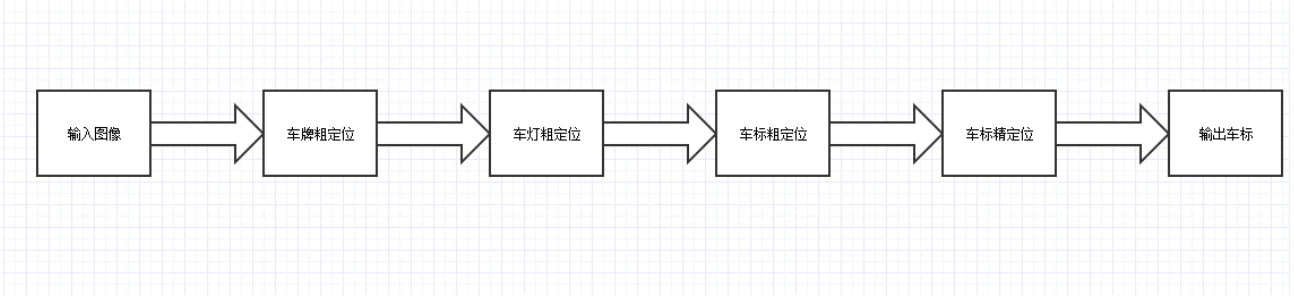
一、简述

随着社会经济的不断发展，道路通行能力日益不能满足交通量的增长。从20世纪90年代起，我国也逐步展开了智能交通系统的研究和开发，探讨在现有的交通运输网的基础上应用现状互联网技术，使交通管理智能化。特别是在我国的国情和我国经济的快速发展，社会信息化程度日益提高的同时，交通管理智能化成为发展的趋势。

图像处理技术是处理交通车辆的图像和视频序列的有效方法之一。车标是车辆的一个重要特征，车标的定位和识别可以为车辆的跟踪提供重要的信息，进而更好的识别跟踪相应的车辆。车标定位可以用于安全控制、交通法的执行、监控系统等方面。虽然人能够很容易地识别车标，但是在数量过多或者长时间的持续监控下很容易导致疲劳而读取到错误信息或者漏掉信息，因而自动化的车标定位和识别是必要的

使用图像处理开源库 openCV 对图像中车辆的车标进行定位并截取用于车标的匹配。本版本中的使用的方法是先验知识定位：在图像中的车牌的轮廓特征在形态学变换很明显，易于定位；再定位到车牌的情况下，车灯区域与车牌区域有明显的位置关系（车灯区域一般位于车牌上方），这样再定位到车灯区域；而车标一般位于车灯区域中，这样就定位到了车标的大概位置；将车标区域的大概位置截取后，对其进行形态学变换定位车标的具体位置

本程序中的车标定位的处理的算法由六部分组成，如下图所示，其中车牌的粗定位、车灯的粗定位、车标的粗定位、车标的精定位为关键技术：



二、开发环境

- Windows7
- VS2010
- openCV2.4.4

三、核心代码思路说明

1 应用技术

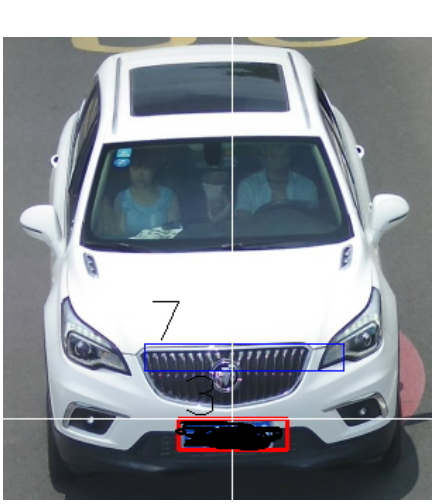
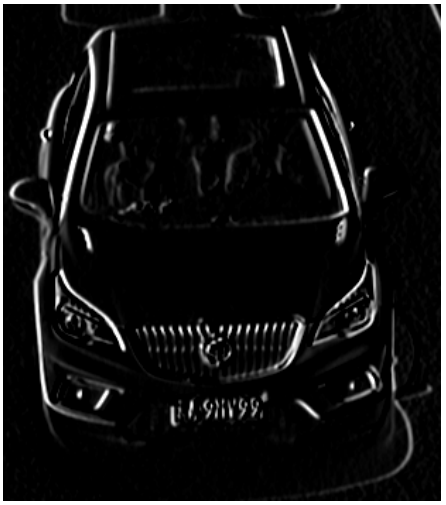
首先介绍在整个代码中使用的两个图像处理技术：基于灰度图像、边缘特征的算法和形态学变换

1.1 基于灰度图像、边缘特征的算法

边缘是图像的最重要的特征之一，往往携带着一幅图像的大部分信息，它反映了图像中一个物体区别于其他物体最基本的特征。所谓边缘是指其周围像素灰度值有明显变化的那些像素的集合。边缘广泛存在与目标与背景直接、目标与目标之间、区域与区域之间、基元与基元之间。

边缘检测是基于边界的图像分割方法，是采用某种算法来提取出图像中对象与背景间的交界线。传统的图像边缘检测是利用图像一阶导数的极值或二阶导数的过零点信息来提供边缘点的依据。利用一阶微分的算子有 Roberts、Sobel、Prewitt、Kirsch，利用二阶微分的算子有 Laplace、LOG等。在本程序中使用的是 Sobel 算子

车辆的边缘特征图像如下：

1.2 形态学变换

数学形态学的基本运算有4个：腐蚀、膨胀、开启和闭合。数学形态学方法利用一个称作结构元素的”探针”收集图像的信息，当探针在图像中不断移动时，便可考察图像各个部分之间的相互关系，从而了解图像的结构特征。在程序中主要使用膨胀和腐蚀进行形态学变换

膨胀是在二值图像中“加长”或“变粗”的操作。这种特殊的方式和变粗的程度由一个称为结构元素的集合控制，使图像中的高亮区域逐渐增长。效果如下图所示：

腐蚀就是求局部最小值的操作，腐蚀“收缩”或“细化”二值图像中的对象。收缩的方式和程度由一个结构元素控制。效果如下图所示：

2 代码思路流程

下面着重获取的思路和流程，只展示一些必要的代码，相应的代码在程序中都有较详细的注释。获取车标的大概流程如下：

- 车牌的粗定位
- 车灯的粗定位
- 车标的粗定位
- 车标的精定位：将图像进行 X 和 Y 方向的投影

- X 方向的车标检测：定位到则返回结果，没有则进行 Y 方向上的定位
- Y 方向的车标定位：定位到则返回结果

2.1 车牌的粗定位

车牌具有丰富的形态学特征，在进行形态学变换再根据其轮廓的矩形特征(其长高比例等），与其他干扰的轮廓特征进行区别，筛选后得到车牌的大概位置

2.1.1 图像预处理

进行定位之前一般要调整图像到规范大小，然后转换图像为灰度图并进行平滑处理，然后进行 Sobel 和二值化等，然后使用腐蚀和膨胀对图像进行形态学处理

2.1.2 定位车牌位置

找到图像所有轮廓的最下方界限：通过前面的腐蚀和膨胀，图像中的路面和路旁边的干扰将被基本排除，则再获取的图像轮廓中，车牌的轮廓将会处于图像的最下方部分，所有先找到所有图像轮廓的最下方界限

```
//找出图中轮廓的下边界
    int bottom = -1;
    for(int i = 0; i < contours.size(); i++) {
        CvRect aRect =  boundingRect(contours[i]);
        if(bottom == -1) {
            bottom = aRect.y;
        }
        else if(bottom < aRect.y) {
            bottom = aRect.y;
        }
    }
```

获取最下方轮廓：在确定最下方界限以后，在界限往上一定范围（程序中取值100）内的所有轮廓将被获取，被获取的轮廓都有可能是车牌轮廓

确定车牌轮廓：如果获取的轮廓只有一个，那么这个轮廓就是车牌；但如果获取的车牌轮廓有多个，将采取下面的策略进行筛选：

- 车牌的高宽比筛选：轮廓的高宽比要在一定范围内，程序中的是在2~6倍之间并且面积小于2000

```
(double)aRect.width/(double)aRect.height>2)&& ((double)aRect.width/(double)aRect.height<6)&& tmparea>=100&&tmpar
```

- 2 居中筛选：如果符合高宽比的轮廓有多个时，首先获取筛选过后所以轮廓的左边界和右边界，如果左边界和右边界之间的范围在合理范围内（在这个图像大小中是100px，大概是整个车辆的宽），则说明可以获取车辆的中心线，而车牌往往在车辆的中心线位置

```
    //水平左边界、右边界
    int left = -1;
    int right = -1;
    vector<Rect> centerRects;
    for(int i = 0; i < rects.size(); i++) {
        if(rects[i].y >= bottom - 200 && rects[i].y <= bottom) {
            centerRects.push_back(rects[i]);

            if(left == -1) {
                left = rects[i].x;
            }
            else if(left > rects[i].x) {
                left = rects[i].x;
            }

            if(right == -1) {
                right = rects[i].x;
            }
            else if(right < rects[i].x) {
                right = rects[i].x;
            }
```

```
        }
    }
```

- 3 取最下部分：如果居中筛选仍没有结果，则选择最靠近最下界限的轮廓

2.2 车灯粗定位

在经过车牌粗定位以后，获得了车牌的位置，把车牌上边界值传入；在车灯定位中使用图像 Y 方向的投影图像来获取车灯的位置，因为在车辆中车灯与车窗之间的车盖部分几乎是平滑的，而使用 Y 方向的投影将会去除图像中的所以垂直线，导致车盖部分在处理后是一段真空区域（这样导致车盖有贴海报之类的对结果有较大的影响），利用车灯位于车牌和车盖之间这个位置信息来粗定位车灯位置

2.2.1 图像预处理

对图像进行简单的大小缩放、颜色空间转换、Y方向投影、二值化等处理，与车牌预处理类似

2.2.2 车灯粗定位

在程序中的图像大小下，轿车类的车牌距离车盖较近，而打卡则较远，但其两者的距离不会超过100px，所以在程序中车牌上边界到往上100px之间对图像进行扫描。在扫描过程中来确定车灯与车盖之间的真空区域，由于图像处理后车盖部分没有像素点（在图像中为黑色既0），通过自上而下的扫描，如果发现这一行图像像素点为0的像素点过多，那就认为这行图像像素点在车盖区域内，并且取车盖区域的下边界。这样车盖的下边界和车牌的上边界直接便是车灯区域

```
bool isLamp = false;
int bottom = -1, top = -1, left = -1, right = -1;
int count;
for (int i = plateTop-90; i < plateTop; i++) {
    count = 0;
    uchar* data = dilateImage.ptr<uchar>(i);
    //车牌中心线左右加上100几乎车辆的宽
    for(int j = plateCenter-100; j < plateCenter+100; j++) {
        if(data[j] == 255) {
            ++count;

            if(left == -1) {
                left = j;
            }
            else if(left > j) {
                left = j;
            }

            if(right == -1) {
                right = j;
            }
            else if(right < j) {
                right = j;
            }
        }
    }

    if(count == 0) {
        if(top != -1 && bottom == -1) {
            bottom = i;

            if(bottom-top < 20) {
                top = -1;
                bottom = -1;
            }
            else {
                break;
            }
        }
    }
    else if(count != 0) {
        if(top == -1 && bottom == -1) {
            top = i;
        }
    }
}
```

2.3 车标粗定位

车标粗定位相对比较简单，在图像大小都一样的情况下，车灯的宽度基本都是没有太大差异的，用获取的车灯区域去除掉车灯部分，只截取剩下部分即可

```
//提取出大致的车标区域
if(plateCenter-40 < 0 || plateCenter+40 > dst.cols-1 || lampBottom-lampTop < 0 || lampTop < 0
    || lampTop > dst.rows-1 || lampBottom < 0 || lampBottom > dst.rows-1) {
    return;
}
```

2.4 车标精定位

在获取大概的车标区域以后，对这个图像区域进行处理而得到车标。在大部分的车辆中车标区域一般都会有很多的横条纹或者竖条纹，这些条纹对图像的检测会造成很大的干扰。在程序中分别对对车标区域进行了 X 和 Y 方向的投影，X 方向的投影可以很大程度上消除横条纹的影响，Y 方向的投影可以很大程度消除竖条纹的影响。但不可能同时消除横竖条纹的影响，所以程序中首页对 X 方向投影后的图像进行车标的定位，如果定位失败则再进行 Y 方向投影图像的定位

2.4.1 图像预处理

缩放图像到固定大小、进行 X 和 Y 方向的投影等

2.4.2 X 方向投影定位

对图像进行简单膨胀腐蚀形态学变换以后，对得到的轮廓进行一系列的筛选，程序中的筛选是有严格的顺序的，而且每次筛选以后如果轮廓就剩下一个，那么这个轮廓便是车标轮廓，其详细情况如下：

- 判断轮廓是否为1，唯一则这个轮廓便是车标，为0便进行 Y 方向的投影定位
- 在获取的轮廓有很多的时候，而且存在一个较大的轮廓，其他轮廓都偏小，则这个轮廓就是车标
- 判断轮廓中的竖条纹的情况，如果过多，说明此图像适合做 Y 方向的投影检测

```
int vertical_20 = 0;
int vertical_40 = 0;
for(int i = 0; i < contours.size(); i++) {
    Rect rect = boundingRect(contours[i]);
    if(rect.height > 20) { //矩形轮廓的高度是否超过一定范围
        ++vertical_20;
    }
    if(rect.height > 40) { //矩形轮廓的高度是否超过一定范围
        ++vertical_40;
    }
}
if(vertical_20 > 5 || vertical_40 > 2) {
    cout << "判断为竖条纹" << endl;
    return false;
}
```

- 边界筛选：把位于上、左、右边界的轮廓去除掉，而下边界的轮廓上方存在比其面积大或差不多的轮廓，则排除

```
/*****************************************************************************/
/* 上、左、右边界限制条件筛选：
    不得很接近边界*/
/*****************************************************************************/
bool LogoDetect::TLR_delete(Rect r) {
    //上、左、右边界不得很接近
    if(r.x < 3 || r.y < 3 || r.x+r.width > 150-3) {
        return true;
    }

    return false;
}

/*****************************************************************************/
/* 判断是矩形轮廓是否可以用下边界规则排除：
```

```
    1.此矩形轮廓处于下边界上
    2.在轮廓上方存在与面积差不多或者大于的矩形轮廓*/
/*************************************************************************/
bool LogoDetect::isBottomDelete(int index, vector<vector<Point>> contours) {
    Rect r = boundingRect(contours[index]);
    for(int i=0; i < contours.size(); i++) {
        Rect rect = boundingRect(contours[i]);
        if(i != index && !TLR_delete(rect) && r.y-1 > rect.y+rect.height-10 ) {
            if(r.area() <= rect.area()) {
                return true;
            }

            if(r.area() > 500 && rect.area() > 200) {
                return true;
            }
        }
    }
    return false;
}
```

- 比较法最小面积排除：在经过上面的筛选以后如果存在小面积的轮廓与最大轮廓之间的面积差距过大，说明这个小面积轮廓是干扰轮廓，把它去除掉

```
    //首先求出最大的矩形轮廓
    Rect maxRect = boundRects[0];
    for(int i=1; i < boundRects.size(); i++) {
        Rect r = boundRects[i];
        if(maxRect.area() < r.area()) {
            maxRect = r;
        }
    }

    //进行比较筛选掉小面积的轮廓
    vector<Rect> minDeleteRects;
    for(int i=0; i < boundRects.size(); i++) {
        Rect r = boundRects[i];
        if((double)(maxRect.area())/(double)(r.area()) > 10) {
            continue;
        }

        minDeleteRects.push_back(r);
    }
```

- 矩形轮廓联通：由于车标的形状各异，经过膨胀腐蚀以后的有的轮廓会被切割成各自独立的部分，但如果这部分车标膨胀腐蚀调整过轻使它的整体成一个部分，其他的车标便会无法独立的获取，往往和干扰的轮廓连在一起，导致无法定位出来。所以针对这些小面积的具有一定关联性的轮廓使用轮廓联通来使它们成为一个整体。联通的标准如下：
  - 首先进行交叉联通：小面积矩形轮廓直接联通，大面积的在同一水平上才进行联通
  - 优先联通：在一级水平线上且距离较近,5距离或者在二级水平线上且距离更近，3距离

```
/*************************************************************************/
/*  首先进行交叉联通：小面积矩形轮廓直接联通，大面积的在同一水平上才进行联通*/
/*************************************************************************/
bool LogoDetect::cross_Link(vector<Rect> &rects, bool all_small) {
    for(int i=0; i < rects.size(); i++) {
        for(int j=0; j < rects.size(); j++) {
            if(i!=j && all_small && isCross(rects[i], rects[j])) {
                Rect rect = link_x(rects[i], rects[j]);
                rects[i] = rect;
                rects[j] = rect;

                vector<Rect> linkRects;
                linkRects.push_back(rect);
                for(int k = 0; k < rects.size(); k++) {
                    if(k != i && k != j) {
                        linkRects.push_back(rects[k]);
                    }
                }
                rects = linkRects;
                return false;
            }
```

```
            else if(i!=j && !all_small && isHorizontalLevel(rects[i], rects[j], 10, true) && isCross(rects[i], r
                Rect rect = link_x(rects[i], rects[j]);
                rects[i] = rect;
                rects[j] = rect;

                vector<Rect> linkRects;
                linkRects.push_back(rect);
                for(int k = 0; k < rects.size(); k++) {
                    if(k != i && k != j) {
                        linkRects.push_back(rects[k]);
                    }
                }
                rects = linkRects;
                return false;
            }
        }
    }

    return true;
}

/*************************************************************************/
/* 优先联通：*/
/*************************************************************************/
bool LogoDetect::priority_Link(vector<Rect> &rects) {
    //在一级水平线上且距离较近,5距离
    //在二级水平线上且距离更近，3距离
    int distance = -1, r1NO, r2NO;
    Rect nearRect1, nearRect2;
    for(int i=0; i < rects.size(); i++) {
        for(int j=0; j < rects.size(); j++) {
            int dis = distanceRects(rects[i], rects[j], 0);
            if(i!=j && isHorizontalLevel(rects[i], rects[j], 3, true) && dis <= 3) {
                    if(distance == -1) {
                        distance = dis;
                        r1NO = i;
                        r2NO = j;
                        nearRect1 = rects[i];
                        nearRect2 = rects[j];
                    }
                    else if(distance > dis) {
                        distance = dis;
                        r1NO = i;
                        r2NO = j;
                        nearRect1 = rects[i];
                        nearRect2 = rects[j];
                    }
            }
            else if(i!=j && isHorizontalLevel(rects[i], rects[j], 5, true) && dis <= 5) {
                    if(distance == -1) {
                        distance = dis;
                        r1NO = i;
                        r2NO = j;
                        nearRect1 = rects[i];
                        nearRect2 = rects[j];
                    }
                    else if(distance > dis) {
                        distance = dis;
                        r1NO = i;
                        r2NO = j;
                        nearRect1 = rects[i];
                        nearRect2 = rects[j];
                    }
            }
        }
    }

    if(distance != -1) {
        Rect rect = link_x(nearRect1, nearRect2);
        rects[r1NO] = rect;
        rects[r2NO] = rect;

        vector<Rect> linkRects;
        linkRects.push_back(rect);
```

```
        for(int k = 0; k < rects.size(); k++) {
            if(k != r1NO && k != r2NO) {
                linkRects.push_back(rects[k]);
            }
        }
        rects = linkRects;
        return false;
    }

    return true;
}
```

- 高宽比筛选：车标中高度不会大于宽度太大，利用这一特性进行一次筛选
- 最小面积排除：去除轮廓中面积小于100的轮廓，小于100的轮廓可以确定是干扰轮廓。而最小面积轮廓排除不放到前面而放到这的原因是前面的轮廓联通需要，有些小面积轮廓有可能与其他轮廓有联系，可以联通成一个整体的轮廓。而经过联通以后便不存在了，可以直接排除
- 优先选择居中且唯一上部分的轮廓：在经过上面的筛选以后轮廓还是不唯一的情况下，则再所以轮廓中优先选择在图像中部的轮廓（因为车标位于中心的概率较大），而有两个轮廓同时位于中心时，处于上方的轮廓较大概率是车标
- 如果在上面的筛选过还是没确定车标，那就选择其中面积最大的一个轮廓

2.4.3 Y 方向投影定位

对图像进行简单膨胀腐蚀形态学变换以后，对得到的轮廓进行一系列的筛选，程序中的筛选是有严格的顺序的，而且每次筛选以后如果轮廓就剩下一个，那么这个轮廓便是车标轮廓，其详细情况如下：

- 初步筛选：去除左右边界上的轮廓
- 矩形轮廓联通：将在同一垂直线上的具有关联性的轮廓进行联通
- 居中轮廓选择：优先选择位于图像中心位置处的轮廓，位于中心轮廓不止一个便优先选择位于图像上部分的
- 选择所有轮廓中面积最大的一个

三、代码结构说明

```
void detect();
//对车标进行定位
void logoDetect(cv::Mat image, int index, int dirIndex);
//在X方向上对车标进行检测
bool X_Detect(Mat image, Mat &resultImage, int index, int dirIndex);
//针对X方向上的相邻轮廓的联通处理
bool X_Link(vector<Rect> &rects, int x_dif, int y_dif, bool all_small);
//在Y方向上对车标进行检测
bool Y_Detect(Mat imMat, Mat &resultImage, int index, int dirIndex);
//Y方向上的联通
bool Y_Link(vector<Rect> &rects, int x_dif, int y_dif);
//当矩形轮廓面积都偏小且符合宽高规范和在水平线上时，直接相连
bool small_Link(vector<Rect> &rects);
//定位车牌
bool plateDetect(Mat image, int& plateTop, int &plateCenter, int index);
//定位车灯
bool lampDetect(Mat image, int& lampTop, int& lampBottom, int index);

//图像预处理
void preprocess(string path, int dirIndex);
void test(vector<string> files);
void rangeTest(Mat image);
//判断两矩形轮廓是否相邻联通
bool isAdjacent_x(Rect rect1, Rect rect2, int x_dif, int y_dif, bool all_small);
bool isAdjacent_y(Rect rect1, Rect rect2, int x_dif, int y_dif);
//判断所有的轮廓中是否存在一个相对其他轮廓面积巨大的轮廓
int maxDif(vector<Rect> rects);
int maxDif(vector<Rect> rects, int multiple);
int maxDif(vector<vector<Point>> contours, int multiple);
//判断矩形是否符合标准
bool isStandard(Rect rect);
//判断两个矩形轮廓是否在在水平线上
bool isHorizontalLevel(Rect rect1, Rect rect2, int y_dif, bool isDouble);
```

```
//将两个矩形轮廓进行融合联通
Rect link_x(Rect rect1, Rect rect2);
Rect link_y(Rect rect1, Rect rect2);
//将交叉的矩形轮廓进行联通
bool cross_Link(vector<Rect> &rects, bool all_small);
//优先联通
bool priority_Link(vector<Rect> &rects);
//将相互包含接触的矩形融合
bool mergeRects(vector<Rect> &rects);
//判断两个矩形是否相互包含
bool withContain(Rect r1, Rect r2);
//判断两个矩形是否交叉
bool isCross(Rect r1, Rect r2);
//测量两个矩形轮廓之间的水平或垂直距离
int distanceRects(Rect r1, Rect r2, int direction);
//保存图片
void saveImage(Mat writeImage, int index, int dirIndex);
//判断是否可以用下边界规则排除矩形轮廓
bool isBottomDelete(int index, vector<vector<Point>> contours);
bool isBottomDelete(int index, vector<Rect> rects);
//上、左、右限制筛选
bool TLR_delete(Rect r);
```

四、总结

经验

形态学变换

形态学变换是从边缘特征图像中获取目标轮廓的有效处理手段，而适当的形态学变换至关重要，可以影响后面目标轮廓获取的难道。而适当形态学变换只有不到的使用不同的参数，且车标的形态多样，边缘特征丰富，找到通用的形态学变换使其得到最好的效果的比较难的。所以只能照顾绝大部分，而且不能过度膨胀使目标与干扰因素联通，也不能过度腐蚀使目标轮廓消失。

车标前横竖杠的消除

车标往往在车辆前部的横竖杠区域，而在进行边缘特征算子计算后，横竖杠对目标检测会造成巨大的影响。而在 openCV 中通过 X 方向和 Y 方向的投影可以消除大部分的横竖杠影响

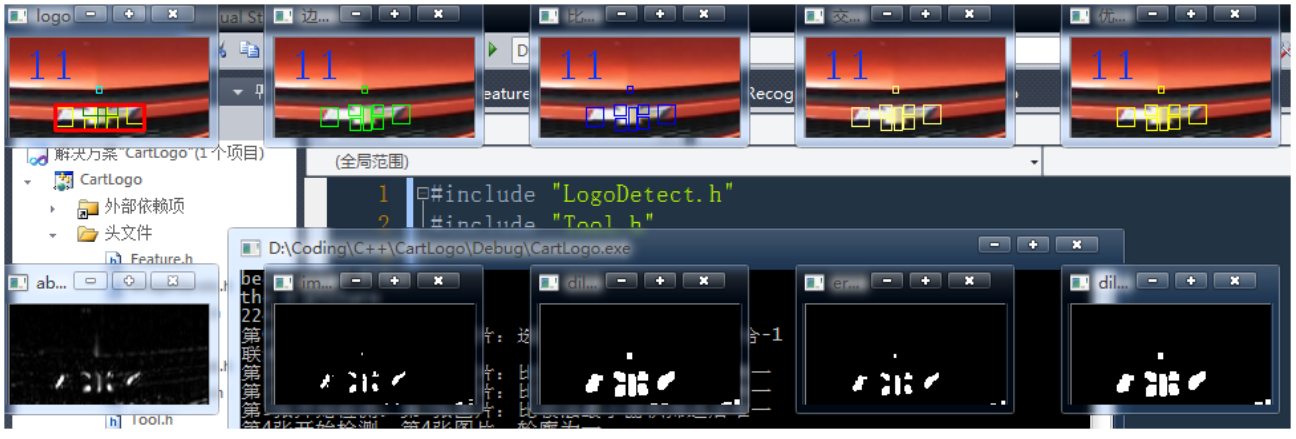
同一水平线和同一垂直线矩形轮廓的联通

因为大部分的车标的边缘信息不够明显，导致经过边缘检测并进行形态学变换以后得到的图像中轮廓基本偏小而且基本在同一水平线上，通过联通操作将它们进行联通形成一个整体的轮廓

依然存在的问题

膨胀腐蚀后车标整体被切割问题

这类车标表面光滑、或者较小、轮廓特征不明显等，使得膨胀腐蚀以后的不到一个整体的车标轮廓，导致不进行处理直接筛选不能得到车标轮廓。但这类车标一般有一些关联性，如面积都比较小且在同一水平线或者垂直线上，将它们进行联通以后便会得到一个车标整体轮廓。不进行联通的轮廓情况如下图：



车盖污染问题

这类问题基本无解，因为在膨胀腐蚀以后基本无法筛选，还好这类情况极少。污染情况如下图：