第 3 章 答题卡评分

3.1 概述

从小到大我们都经历了无数次考试，而其中用 2B 铅笔涂黑的答题卡对大家来讲应该都不会陌生，这个案例就是用 OpenCV 对答题卡进行识别并评分。

3.1.1 案例描述

在本案例中用到的答题卡如图 3-1 所示。该答题卡共有 30 道选择题，每题 1 分，共 30 分。选择题区域的 4 个角上设有用于定位的黑色色块。

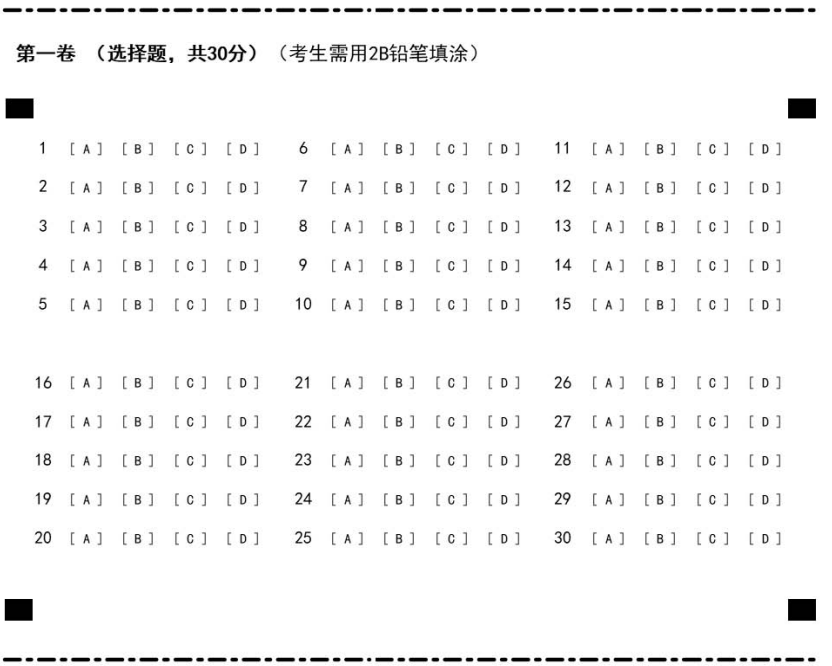


图 3-1 答题卡样例

答题卡中涉及的尺寸（单位：像素）如下：

（1）定位块之间宽度：900。

（2）定位块之间高度：580。

（3）答案横向距离：60（A-B-C-D）或 120（D 到下一个 A）。

（4）答案纵向距离：45（如 1-2 题间）或 90（如 5-16 题间）。

（5）定位块大小：32×24。

（6）答案涂黑区域：35×15。

用于评分的答题卡如图 3-2 所示。该图大致水平，略有倾斜，其中的 30 道选择题的答案已经用铅笔涂黑。为了简单起见，所有选择题的标准答案都为 B。

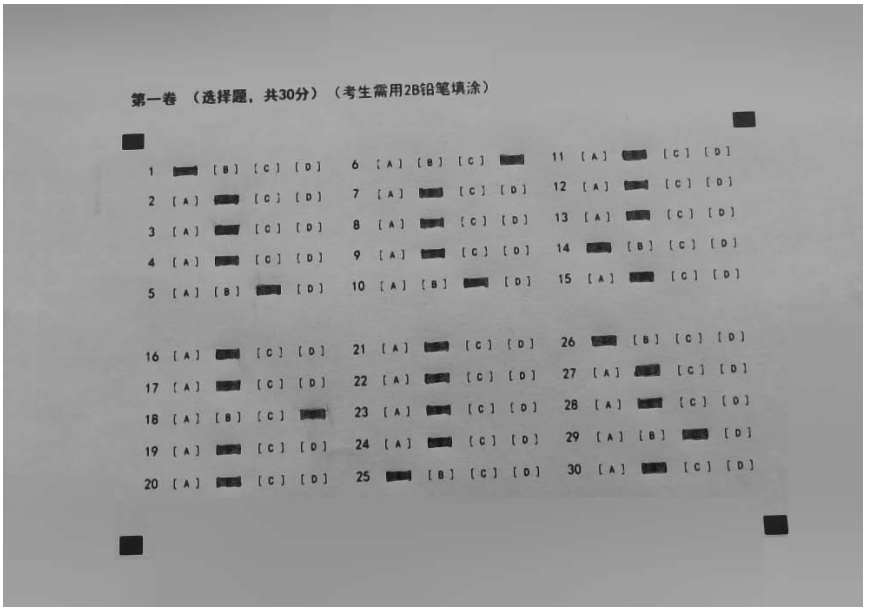


图 3-2　用于评分的答题卡

3.1.2　案例分析

显而易见，本案例的关键是如何获取定位块的位置。定位块为全黑的矩形，面积比答案涂黑区域大一些，要识别出定位块可以根据轮廓求出最小外接矩形，但是识别过程中会受到一些干扰。一种干扰是涂黑的答案（以下称“答案块”），同样是全黑的矩形。虽然理论上定位块比答案块要大一些，但是由于答题卡有倾斜，加上考生涂黑时可能比标准区域涂得大一些，因此涂黑的面积未必比定位块小。另一种干扰是定位块上方的文字，它们也会形成黑色的矩形。不过文字并不是整体全黑，可以用黑色像素所占比例进行区分。剩下的问题就是如何区分定位块和答案块。

观察答题卡后不难发现，定位块位于边缘区域，如果把定位块连接成一个矩形，则所有的答案块都应该在这个矩形内部，如图 3-3 所示。这样，问题就简化成如何识别最外围的 4

个黑块了。

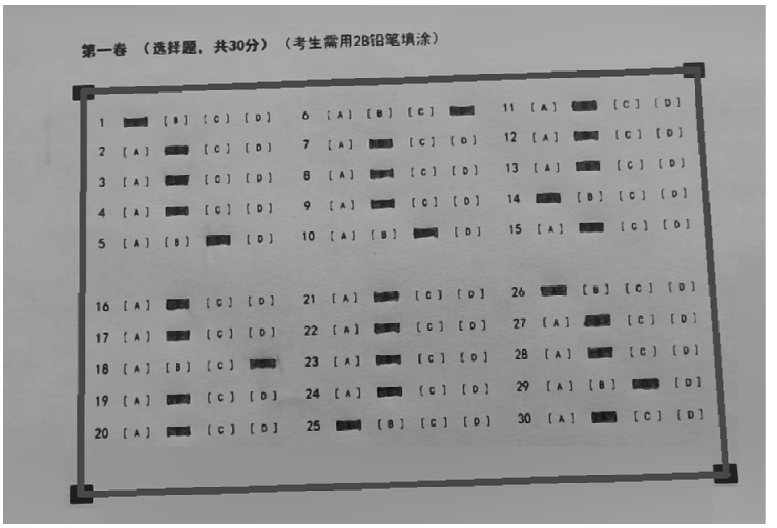


图 3-3　定位块连接成矩形后

在 OpenCV 中解决此类问题有一个简单而又有效的方法。首先，每个外接矩形可以用其中心点来表示。假设选定 A、B、C、D 共 4 个点来测试它们是否是定位块。可以在黑色背景上以此 4 个点为顶点绘制一个实心（假定为白色）的矩形，然后依次测定其余中心点所在位置是否为白色，如为白色则表示该点在此矩形内部，否则在矩形外部。通过不断地试错，最终可以测试出最外围的 4 个点。此方法简单实用，不过测试前需要先调整 4 个点的相对位置，否则可能会使矩形的两条边形成交叉，如图 3-4 所示，这样测试结果就不准确了。

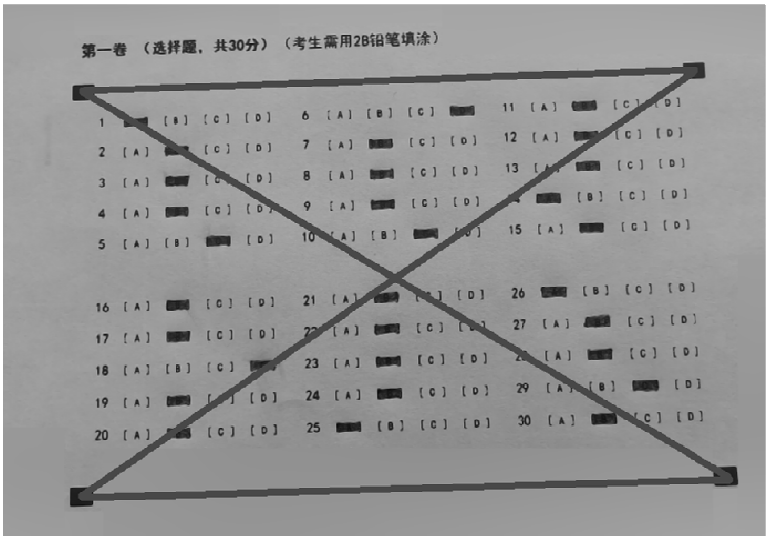


图 3-4　4 个顶点未调整所引发的问题

定位块确定以后，只需通过透视变换将答题卡转换成水平状态就可以根据答案块的坐标位置识别答案并评分了。

3.2　总体设计

3.2.1　系统需求

本案例只需 OpenCV，不需要任何第三方库。

3.2.2　总体思路及流程

根据上述分析，本案例的总体流程如下：

（1）将输入图案转换成二值图。

（2）提取轮廓并获取轮廓的最小外接矩形。

（3）根据黑色像素占比筛选掉汉字部分。

（4）测试定位块的位置。

（5）对答题卡进行透视变换。

（6）判定考生涂黑的答案。

（7）给答题卡评分。

3.3　答题卡自动评分的实现

3.3.1　二值化

由于程序中需要判断黑色像素占比，因此二值化时不能用 Canny()算法，而应该用 threshold()函数。

OpenCV 中 threshold()函数的原型如下：

```
    double Imgproc.threshold(Mat src, Mat dst, double thresh, double maxval, int
type)
    函数用途：对图像二值化。

    【参数说明】
    (1) src：输入图像，要求是 CV_8U 或 CV_32F 类型。
    (2) dst：输出图像，和 src 具有相同的尺寸、数据类型和通道数。
    (3) thresh：阈值。
    (4) maxval：二值化的最大值，只用于 Imgproc.THRESH_BINARY 和 Imgproc.THRESH_
BINARY_INV 两种类型。
    (5) type：二值化类型，可选参数如下：
    ◆ Imgproc.THRESH_BINARY：当大于阈值时取 maxval，否则取 0。
```

- Imgproc.THRESH_BINARY_INV：当大于阈值时取 0，否则取 maxval。
- Imgproc.THRESH_TRUNC：当大于阈值时为阈值，否则不变。
- Imgproc.THRESH_TOZERO：当大于阈值时不变，否则取 0。
- Imgproc.THRESH_TOZERO_INV：当大于阈值时取 0，否则不变。
- Imgproc.THRESH_OTSU：大津法自动寻找全局阈值。
- Imgproc.THRESH_TRIANGLE：三角形法自动寻找全局阈值。

其中 Imgproc.THRESH_OTSU 和 Imgproc.THRESH_TRIANGLE 是获取阈值的方法，可以和另外 5 种联用，如"Imgproc.THRESH_BINARY | Imgproc.THRESH_OTSU"。

程序中用 makeBinary()函数对输入图像进行二值化，相关代码如下：

```
Mat gray = new Mat();
Imgproc.cvtColor(src, gray, Imgproc.COLOR_BGR2GRAY);
Mat binary = new Mat();
Imgproc.threshold(gray, binary, 120, 255, Imgproc.THRESH_BINARY);
```

代码中先用 cvtColor()函数将彩色图像转换成灰度图，然后调用 threshold()函数进行二值化。在最后一行代码中，threshold()函数将最后一个参数 type 设为 Imgproc.THRESH_BINARY，将阈值 thresh 设为 120，该行代码可理解为如果像素值大于 120，则二值化为 255，否则为 0。调用上述代码后生成的图像如图 3-5 所示。该图只有黑白两色，像素值分别为 0 和 255。

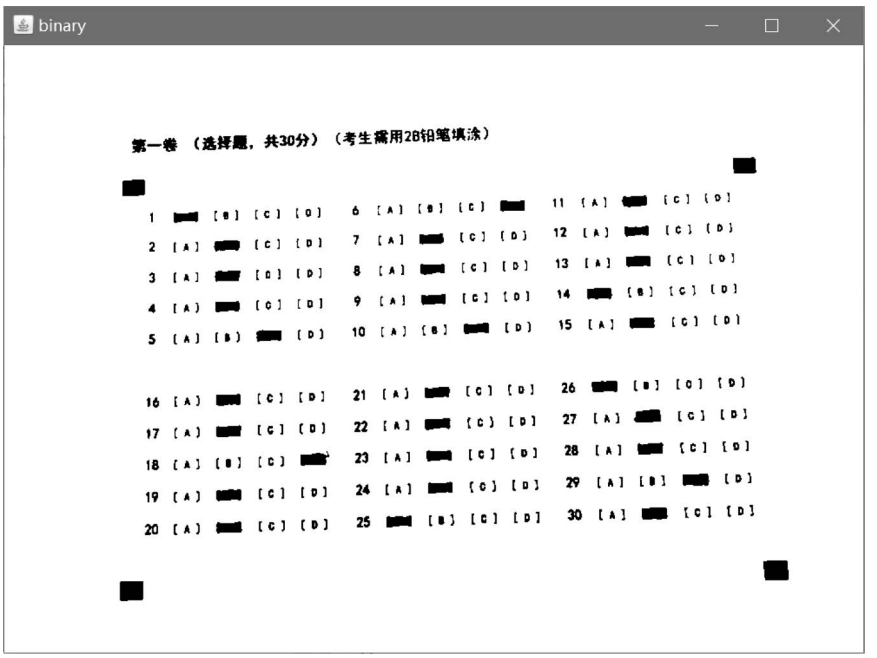


图 3-5　二值化后的答题卡

3.3.2　提取轮廓

在二值图的基础上就可以用 findContours()函数来提取轮廓了。提取轮廓的代码如下：

```
List<MatOfPoint> contour = new ArrayList<MatOfPoint>();
Imgproc.findContours(binary, contour, new Mat(), Imgproc.RETR_TREE,
        Imgproc.CHAIN_APPROX_SIMPLE);
```

调用 findContours()函数很简单，但是其参数较为复杂，有必要说明一下相关用法，该函数的原型如下：

```
void Imgproc.findContours(Mat image, List<MatOfPoint> contours, Mat hierarchy,
int mode, int method)
```

函数用途：在二值图像中寻找轮廓。

【参数说明】

(1) image：输入图像，必须是 8 位单通道二值图或灰度图。如果是灰度图，则像素值为 0 的仍视作 0，像素值不为 0 的视作 1，如此灰度图也可作为二值图处理。

(2) contours：检测到的轮廓。

(3) hierarchy：轮廓的层级，包含对轮廓之间的拓扑关系的描述。hierarchy 中的元素数量和轮廓中的元素数量是一样的。第 i 个轮廓 contours[i]有着相对应的 4 个 hierarchy 索引，分别是 hierarchy[i][0]、hierarchy[i][1]、hierarchy[i][2]和 hierarchy[i][3]，它们分别是轮廓的同层下一个轮廓索引、同层上一个轮廓索引、第 1 个子轮廓索引和父轮廓索引。如果第 i 个轮廓没有下一个同层轮廓、子轮廓或父轮廓，则对应的索引用负数表示。

(4) mode：轮廓提取模式，具体如下。

◆ Imgproc.RETR_EXTERNAL：只检测最外层轮廓，所有轮廓的 hierarchy[i][2]和 hierarchy[i][3]均设为-1。

◆ Imgproc.RETR_LIST：检测所有的轮廓，但轮廓之间不建立层级关系。

◆ Imgproc.RETR_CCOMP：检测所有的轮廓并将它们组织成双层层级关系。

◆ Imgproc.RETR_TREE：检测所有轮廓，所有轮廓建立一个树形层级结构。

(5) method：轮廓逼近方法，可选参数如下。

◆ Imgproc.CHAIN_APPROX_NONE：存储所有轮廓点，两个相邻的轮廓点(x1,y1)和(x2,y2)必须是 8 连通，即 max(abs(x1-x2),abs(y2-y1))=1。

◆ Imgproc.CHAIN_APPROX_SIMPLE：压缩水平方向、垂直方向和对角线方向的线段，只保存线段的端点。

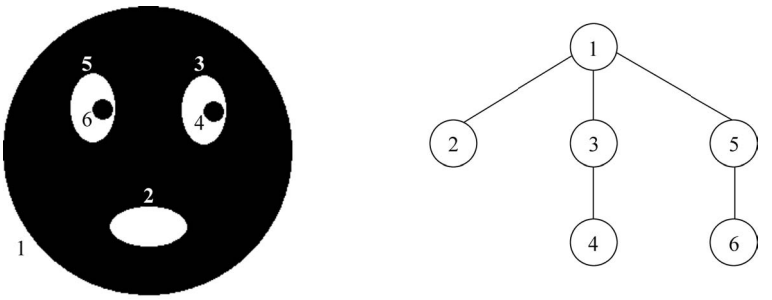
◆ Imgproc.CHAIN_APPROX_TC89_L1：使用 teh-Chinl chain 近似算法中的一个。

◆ Imgproc.CHAIN_APPROX_TC89_KCOS：使用 teh-Chinl chain 近似算法中的一个。

该函数提取的轮廓保存在参数 contours 中，其数据类型是 MatOfPoint 的列表。顾名思义，MatOfPoint 就是用矩阵存储了一个点集，列表中每个 MatOfPoint 对象都是一个轮廓，所以 contours.size()就是提取的轮廓数量。

需要注意的是，轮廓是有层级的，如图 3-6 所示，图中是一个有着 6 个轮廓的图形及其层级关系，其中最大的 1 号轮廓层级最高，2 号、3 号和 5 号轮廓是其子轮廓，1 号轮廓则

是它们的父轮廓。轮廓是可以嵌套的，例如 3 号和 5 号轮廓又有其各自的子轮廓。



（a）标有轮廓编号的图像　　（b）轮廓的树形结构图

图 3-6　轮廓的层级结构

为了描述轮廓之间的拓扑关系，findContours()函数用参数 hierarchy 来描述轮廓的层级关系。首先，hierarchy 中的元素数量和 contours 的元素数量是一致的。其次，每个轮廓都对应 4 个索引值。例如，第 i 个轮廓 contours[i]的 4 个 hierarchy 索引分别是 hierarchy[i][0]、hierarchy[i][1]、hierarchy[i][2]和 hierarchy[i][3]，它们分别是轮廓的同层下一个轮廓索引、同层上一个轮廓索引、第 1 个子轮廓索引和父轮廓索引。例如，5 号轮廓可以表示为[−1, 3, 6, 1]，它没有同层下一个轮廓，所以第 1 个索引值用−1 表示，它的同层上一个轮廓为 3 号轮廓，第 1 个子轮廓为 6 号轮廓，父轮廓为 1 号轮廓。

为了在提取轮廓时包括它们的拓扑关系，需要设置 mode 参数，详见函数原型中的参数说明。由于在本案例中并未涉及轮廓之间的拓扑关系，因此调用 findContours()函数时参数 hierarchy 用 new Mat()代表了。

总而言之，提取轮廓后的数据保存在 contours 参数中，在此基础上可以获取相应的最小外接矩形。最小外接矩形的概念如图 3-7 所示，图中有两个矩形，其中倾斜的矩形就是最小外接矩形，另一个未经旋转的矩形称为直边界矩形。很明显，最小外接矩形的面积远远小于直边界矩形，其形状也更贴近图形的轮廓。

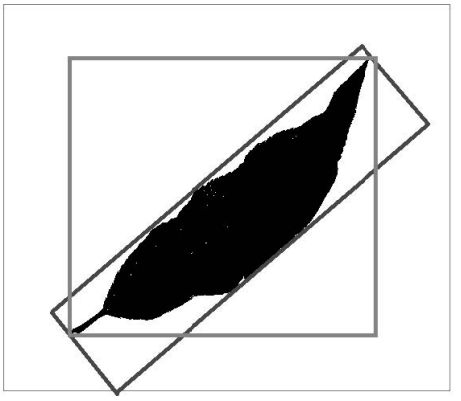


图 3-7　最小外接矩形的概念

在 OpenCV 中，最小外接矩形用 RotatedRect 类表示，该类的成员变量有 center、width、height、angle 等，如图 3-8 所示。

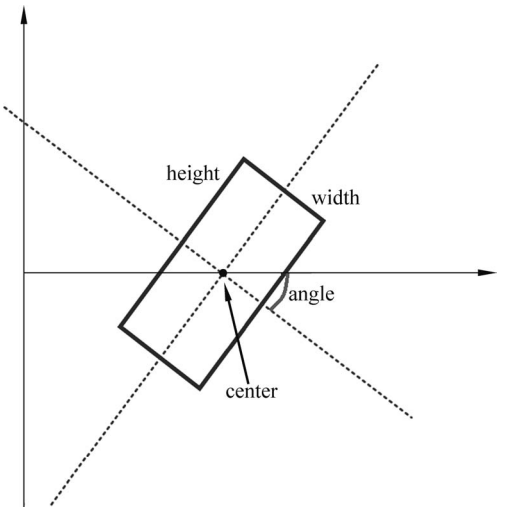


图 3-8　RotatedRect 类的成员变量

有时，需要用到这个旋转矩形的 4 个顶点，而根据上述成员变量计算起来较为烦琐，为此 OpenCV 中提供了 boxPoints()函数，可以一次性获取旋转矩形的 4 个顶点，该函数的原型如下：

```
void Imgproc.boxPoints(RotatedRect box, Mat points)
函数用途：获取旋转矩形的 4 个顶点。

【参数说明】
(1) box：输入的旋转矩形。
(2) points：输出的 4 个顶点。
```

在了解了相关的基础知识后，就不难理解获取最小外接矩形的 minRect()函数了。该函数的声明行如下：

```
public static int[][] minRect(List<MatOfPoint> contour)
```

该函数的参数 contour 就是 findContours()函数输出的结果，类型同样是 MatOfPoint 的列表。由于轮廓很多，为了提高运行速度，代码中先用矩形面积进行初步筛选，符合条件的用 block 数组输出。由于 block 数组的长度事先无法知晓，因此 block[0][0]被用来表示该数组的有效长度，后续章节中有不少案例也用到了这种方法。

对筛选后的外接矩形进行标注，标注后的结果如图 3-9 所示。可以看到，答题卡上部的几个汉字也包括在内。为了防止它们对定位块的判断构成干扰，需要将这些汉字过滤掉。

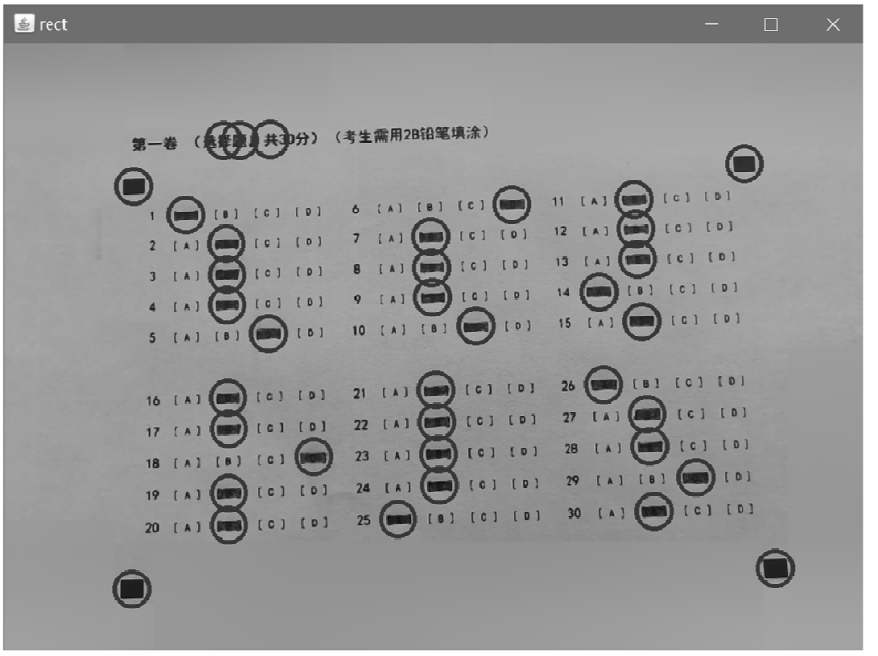


图 3-9　初步筛选后的最小外接矩形

3.3.3　汉字过滤

如前所述，汉字可以通过黑色像素占比的方法进行过滤，程序中用 isBlack()函数来判断某个外接矩形是否为全黑。当然，由于获取外接矩形时中心的坐标或多或少会存在误差，二值化时也会丢失一部分黑色像素，因此判断全黑不能用 100%这么严格的标准。

该函数先用前面介绍的 boxPoints()函数获取旋转矩形的 4 个顶点，然后经过透视变换将旋转矩形转换成水平放置，接着调用 OpenCV 中的 countNonZero()函数统计非零像素的个数。在二值图中，非零像素即不是黑色的像素，用像素总数减去非零像素就是黑色像素数量了。代码中将阈值设定为 70%，如果黑色像素占比没有达到这个比率，则认为该矩形不是全黑而被过滤掉。

经过过滤以后答题卡上剩余的外接矩形如图 3-10 所示。可以看出，上方的汉字已经被过滤掉。虽然有的答案块也被过滤掉了，但这并不会影响对定位块的判断。

3.3.4　定位块位置

过滤后还剩 13 个外接矩形，接下来就可以测试外围的定位块了，测试的原理在 3.1.2 节已经介绍过。每次测试需要选取 4 个点，为了保证测试的准确，这 4 个点要按照一定的顺序排列，程序中用 arrangeFour()函数实现这一步。

排列完成后，在黑色背景上用这 4 个点画一个实心的矩形，然后逐个测试其余点是否

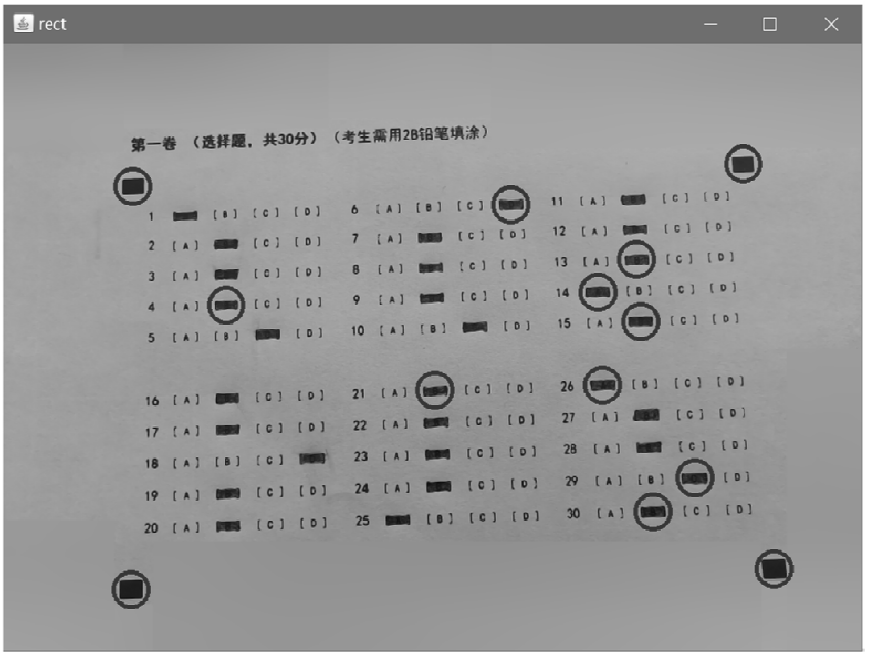


图 3-10　排除汉字干扰后的最小外接矩形

在矩形内部。绘制实心矩形的是 drawPoly()函数。全黑的背景可以直接用 Mat 类的 zeros()方法完成，接着调用 OpenCV 的 fillPoly()函数绘制实心的多边形，该函数并不复杂，只要定义好多边形的顶点，然后转换成函数需要的数据类型即可。

绘制完成后就可以测试有多少中心点在矩形中了，程序中用 countInside()函数完成此任务。函数对所有中心点都进行了测试，包括矩形的 4 个顶点，因此只有包括所有点的矩形才是最外围的定位块。通过这项测试也就找到了 4 个定位块的坐标。程序中用 findFour()函数对所有组合进行测试，这个函数包含一个多重循环，测试完成后以数组形式返回 4 个定位块中心的坐标。

3.3.5　透视变换

接下来需要根据 4 个定位块的位置进行透视变换，在本案例中先用 perspMatrix()函数求出相应的转换矩阵。该函数参数中的 pt 数组就是含有 4 个定位块坐标的数组，该数组共 8 个元素，每两个元素表示一个点的 x 坐标和 y 坐标。函数返回的是转换矩阵，根据此矩阵就能将定位块构成的矩形转换成水平放置的矩形，如图 3-11 所示。

3.3.6　答案的判断

下一步是根据透视变换后的图像判断考生涂写的答案。由于答案块都有固定的位置，因此相关的定位非常容易。

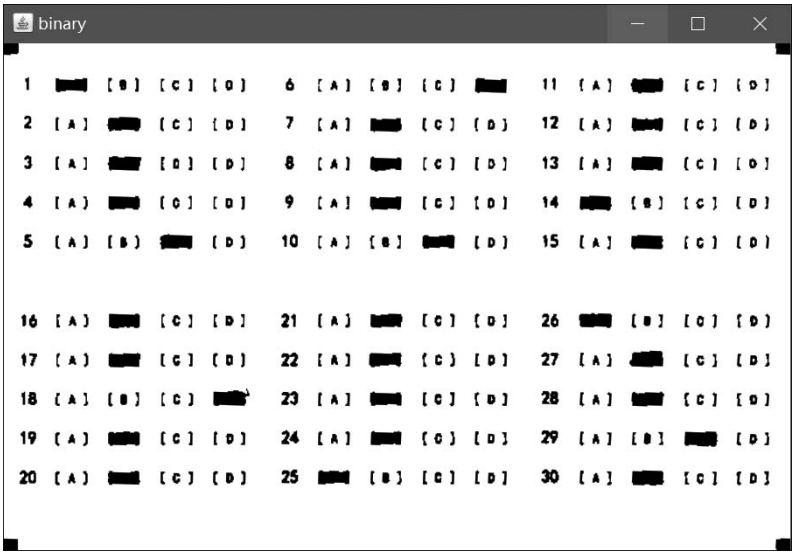


图 3-11　透视变换后的答题卡

判断是否涂黑的标准仍然是黑色像素占比，程序中通过 countArea()函数实现。该函数先用子矩阵截取 35×15 大小的区域并复制到 sub 中，然后调用 countNonZero()函数进行计数。如果返回值为 1，则表示该区域被涂黑，如果返回值为 0，则表示没有。只有当黑色像素占比超过 70%时才返回 1，因此如果涂的范围较小就可能被判断为 0。

上述函数只是判断一个答案块是否被涂黑，而一道题的答案由 A、B、C、D 共 4 个答案块构成，因此判断考生究竟涂黑了哪个答案需要综合 4 个答案块。如果 A 被涂黑而 B、C、D 没有，则可以认为考生的答案是 A，但是如果除了 A 以外还有其他答案也被涂黑，则考生实际上选择了多项答案，应该判错。程序中用 oneAnswer()函数进行此项判断，该函数取 4 个答案块的判断结果（1 表示涂黑，0 表示没有），其中 A 的结果乘以 1000，B 的结果乘以 100，C 的结果乘以 10，D 的结果仍取该数，然后将 4 个数相加的和用 1 个 4 位整数来表示答案。如果和等于 1000，则表示考生选了 A，如为 100 表示考生选了 B，如为 10 表示考生选了 C，如为 1 表示考生选了 D，其余情况说明考生选错（多选或未选）。在此基础上，程序用 allAnswer()函数对所有题的答案进行判断。

得出所有答案后，就可以将考生的答案与标准答案进行比对从而给出得分，程序中用 finalScore()函数实现这个功能。

3.4　完整代码

最后，给出本案例的完整代码：

```
//第 3 章/AnswerSheet.java
```

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Comparator;
import java.util.List;
import org.opencv.core.Core;
import org.opencv.core.CvType;
import org.opencv.core.Mat;
import org.opencv.core.MatOfPoint;
import org.opencv.core.MatOfPoint2f;
import org.opencv.core.Point;
import org.opencv.core.RotatedRect;
import org.opencv.core.Scalar;
import org.opencv.core.Size;
import org.opencv.highgui.HighGui;
import org.opencv.imgcodecs.Imgcodecs;
import org.opencv.imgproc.Imgproc;

public class AnswerSheet {

    public static int width;        //图像长
    public static int height;       //图像宽
    public static int lenX = 900;   //定位块横向距离
    public static int lenY = 580;   //定位块纵向距离
    public static int topX = 78;    //第1题答案A离定位块中心的横向距离
    public static int topY = 48;    //第1题答案A离定位块中心的纵向距离

    public static void main(String[] args) {
        System.loadLibrary(Core.NATIVE_LIBRARY_NAME);

        //读取图像并转换成二值图
        Mat src = Imgcodecs.imread("Answersheet1.png");
        width = src.width();
        height = src.height();
        Mat binary = makeBinary(src);
        HighGui.imshow("binary", binary);
        HighGui.waitKey(0);

        //提取轮廓
        List<MatOfPoint> contour = new ArrayList<MatOfPoint>();
        Imgproc.findContours(binary, contour, new Mat(), Imgproc.RETR_TREE,
                Imgproc.CHAIN_APPROX_SIMPLE);

        //筛选符合条件的块
```

```
        int[][] block = minRect(contour);
        drawRect(src, block);
        int[][] blk = checkBlocks(binary, contour, block);

        //确定 4 个定位块的位置
        int[] out = findFour(blk);
        System.out.println("定位块中心坐标: " + Arrays.toString(out));

        //根据定位块对原图像进行透视变换
        Mat trans = new Mat();
        Mat matrix = perspMatrix(out, lenX, lenY);
        Imgproc.warpPerspective(binary, trans, matrix, new Size(lenX, lenY));
        HighGui.imshow("binary", trans);
        HighGui.waitKey(0);

        //获取答案所在位置的涂黑数据
        int[][] area = blockColor(trans);

        //根据涂黑数据判断答案正确与否并评分
        int[] answer = allAnswer(area);
        System.out.println("考生答案: " + Arrays.toString(answer));
        int score = finalScore(answer);
        System.out.println("考生得分: " + score + "分 / 共 30 分");

        System.exit(0);
    }

    public static Mat makeBinary(Mat src) {
        //将图像转换成二值图
        Mat gray = new Mat();
        Imgproc.cvtColor(src, gray, Imgproc.COLOR_BGR2GRAY);
        Mat binary = new Mat();
        Imgproc.threshold(gray, binary, 120, 255, Imgproc.THRESH_BINARY);
        return binary;
    }

    public static int[][] minRect(List<MatOfPoint> contour) {
        int total = contour.size();
        int[][] block = new int[total + 1][3];
        MatOfPoint2f dst = new MatOfPoint2f();

        //获取各轮廓的最小外接矩形并进行筛选
        int count = 0;
```

```
        for (int n = 0; n < total; n++) {
            //获取轮廓的最小外接矩形
            contour.get(n).convertTo(dst, CvType.CV_32F);
            RotatedRect rect = Imgproc.minAreaRect(dst);

            //排除太大和太小的外接矩形
            double w = rect.size.width;
            double h = rect.size.height;
            if ((w * h < 300) || (w * h > 800))
                continue;

            //返回轮廓编号及最小外接矩形中心坐标
            count++;
            block[count][0] = n;  //轮廓编号
            block[count][1] = (int) rect.center.x;
            block[count][2] = (int) rect.center.y;
        }

        block[0][0] = count;     //有效长度
        return block;
    }

    public static boolean isBlack(Mat binary, RotatedRect rect) {
        //旋转矩形的 4 个顶点
        Mat pts = new Mat();
        float[] f = new float[8];
        Imgproc.boxPoints(rect, pts);
        pts.get(0, 0, f);
        int[] data = new int[8];
        for (int i = 0; i < 8; i++) {
            data[i] = (int) f[i];
        }

        //将旋转矩形转换成直边界矩形
        Mat matrix = perspMatrix(data, 20, 20);
        Mat area = new Mat();
        Imgproc.warpPerspective(binary, area, matrix, new Size(20, 20));

        //清点黑色像素个数并判断是否全黑
        int count = Core.countNonZero(area);
        double rate = 1 - count / 400.0;
        if (rate > 0.7) {
            return true;
```

```
        } else {
            return false;
        }

    }

    public static void drawRect(Mat src, int[][] block) {
        for (int i = 1; i <= block[0][0]; i++) {
            Point center = new Point(block[i][1], block[i][2]);
            Imgproc.circle(src, center, 20, new Scalar(0, 0, 255), 3);
        }
        HighGui.imshow("rect", src);
        HighGui.waitKey(0);
    }

    public static int[][] checkBlocks(Mat binary, List<MatOfPoint> contour,
            int[][] block) {
        MatOfPoint2f dst = new MatOfPoint2f();
        int count = 0;
        int num = block[0][0];

        //逐个检查各旋转矩形是否全黑并标记
        for (int i = 1; i <= num; i++) {
            int id = block[i][0];
            contour.get(id).convertTo(dst, CvType.CV_32F);
            RotatedRect rect = Imgproc.minAreaRect(dst);
            boolean black = isBlack(binary, rect);
            if (black)
                count++;
            else {
                block[i][0] = 0; //标记不合格的
            }
        }

        //将全黑的重置为新的数组
        int[][] blk = new int[count][3];
        int next = 0;
        for (int i = 1; i <= num; i++) {
            if (block[i][0] != 0) {
                blk[next][0] = block[i][0];
                blk[next][1] = block[i][1];
                blk[next][2] = block[i][2];
                next++;
```

```
            }
        }
        return blk;

    }

    public static int[][] sort2D2(int[][] arr) {
        //用二维数组的第二维排序
        Arrays.sort(arr, new Comparator<int[]>() {
            public int compare(int[] o1, int[] o2) {
                return o1[1] - o2[1];
            }
        });
        return arr;
    }

    public static int[] arrangeFour(int[][] blk, int n1, int n2, int n3,
int n4){
        //获取 4 个点的坐标并按 x 坐标排序
        int[][] p = new int[4][3];
        for (int i = 0; i < 3; i++) {
            p[0][i] = blk[n1][i];
            p[1][i] = blk[n2][i];
            p[2][i] = blk[n3][i];
            p[3][i] = blk[n4][i];
        }
        int[][] sorted = sort2D2(p);

        //重新排列 4 个点
        int[] pt = new int[8];
        if (sorted[0][2] < sorted[1][2]) {
            pt[0] = sorted[0][1];
            pt[1] = sorted[0][2];
            pt[6] = sorted[1][1];
            pt[7] = sorted[1][2];
        } else {
            pt[0] = sorted[1][1];
            pt[1] = sorted[1][2];
            pt[6] = sorted[0][1];
            pt[7] = sorted[0][2];
        }

        if (sorted[2][2] < sorted[3][2]) {
```

```
            pt[2] = sorted[2][1];
            pt[3] = sorted[2][2];
            pt[4] = sorted[3][1];
            pt[5] = sorted[3][2];
        } else {
            pt[2] = sorted[3][1];
            pt[3] = sorted[3][2];
            pt[4] = sorted[2][1];
            pt[5] = sorted[2][2];
        }
        return pt;
    }

    public static Mat drawPoly(int[] data) {
        //多边形的顶点
        Point[] pt1 = new Point[4];
        pt1[0] = new Point(data[0], data[1]);
        pt1[1] = new Point(data[2], data[3]);
        pt1[2] = new Point(data[4], data[5]);
        pt1[3] = new Point(data[6], data[7]);
        MatOfPoint mop = new MatOfPoint(pt1);
        List<MatOfPoint> pts = new ArrayList<MatOfPoint>();
        pts.add(mop);

        //以黑色背景绘制实心的多边形
        Size size = new Size(width, height);
        Mat img = Mat.zeros(size, CvType.CV_8UC1);
        Imgproc.fillPoly(img, pts, new Scalar(127));
        return img;
    }

    public static int countInside(Mat gray, int[][] blk) {
        int count = 0;
        for (int i = 0; i < blk.length; i++) {
            int x = blk[i][1];
            int y = blk[i][2];
            byte[] data = new byte[1];
            gray.get(y, x, data);
            if (data[0] == 127)
                count++;
        }
        return count;
    }
```

```
public static int[] findFour(int[][] blk) {
    int[] out = new int[8];
    int n = blk.length;
    for (int i = 0; i < n - 3; i++) {
        for (int j = i + 1; j < n - 2; j++) {
            for (int k = j + 1; k < n - 1; k++) {
                for (int l = k + 1; l < n; l++) {
                    int[] pt = arrangeFour(blk, i, j, k, l);
                    Mat img = drawPoly(pt);
                    int count = countInside(img, blk);
                    if (count == n) {
                        return pt;
                    }
                }
            }
        }
    }
    return out;
}

public static Mat perspMatrix(int[] pt, int width, int height) {
    //定义原图像中 4 个点的坐标
    Point[] pt1 = new Point[4];
    pt1[0] = new Point(pt[0], pt[1]);
    pt1[1] = new Point(pt[2], pt[3]);
    pt1[2] = new Point(pt[4], pt[5]);
    pt1[3] = new Point(pt[6], pt[7]);

    //定义目标图像中 4 个点的坐标
    Point[] pt2 = new Point[4];
    pt2[0] = new Point(0, 0);
    pt2[1] = new Point(width, 0);
    pt2[2] = new Point(width, height);
    pt2[3] = new Point(0, height);

    //计算透视变换的转换矩阵
    MatOfPoint2f mop1 = new MatOfPoint2f(pt1);
    MatOfPoint2f mop2 = new MatOfPoint2f(pt2);
    Mat matrix = Imgproc.getPerspectiveTransform(mop1, mop2);

    return matrix;
}
```

```
public static int countArea(Mat binary, int x, int y) {
    //截取答案区域并判断是否被涂黑
    Mat roi = binary.submat(y - 8, y + 7, x - 18, x + 17);
    Mat sub = new Mat();
    roi.copyTo(sub);
    double total = 35 * 15.0;
    int count = Core.countNonZero(roi);
    if ((total - count)/ total > 0.7)
        return 1;
    else
        return 0;
}

public static int[][] blockColor(Mat m) {
    int[][] area = new int[14][11];
    for (int row = 0; row < 11; row++) {
        for (int col = 0; col < 14; col++) {
            int type = countArea(m, topX + col * 60, topY + row * 45);
            area[col][row] = type;
        }
    }
    return area;
}

public static int oneAnswer(int[][] area, int row, int col) {
    int n1 = area[col][row];
    int n2 = area[col + 1][row];
    int n3 = area[col + 2][row];
    int n4 = area[col + 3][row];
    int num = n1 * 1000 + n2 * 100 + n3 * 10 + n4;
    if (num == 1000)
        return 1;     //代表A
    if (num == 100)
        return 2;     //代表B
    if (num == 10)
        return 3;     //代表C
    if (num == 1)
        return 4;     //代表D
    return 0;
}

public static int[] allAnswer(int[][] area) {
```

```
        int[][] map = new int[30][2];
        for (int n = 0; n < 5; n++) {
            map[n][0] = n;
            map[n][1] = 0;
        }

        for (int n = 5; n < 10; n++) {
            map[n][0] = n - 5;
            map[n][1] = 5;
        }

        for (int n = 10; n < 15; n++) {
            map[n][0] = n - 10;
            map[n][1] = 10;
        }

        for (int n = 15; n < 20; n++) {
            map[n][0] = n - 9;
            map[n][1] = 0;
        }

        for (int n = 20; n < 25; n++) {
            map[n][0] = n - 14;
            map[n][1] = 5;
        }

        for (int n = 25; n < 30; n++) {
            map[n][0] = n - 19;
            map[n][1] = 10;
        }

        int[] answer = new int[30];
        for (int n = 0; n < 30; n++) {
            answer[n] = oneAnswer(area, map[n][0], map[n][1]);
        }

        return answer;

    }

    public static int finalScore(int[] answer) {
        //所有选择题的标准答案
        int[] correct = new int[30];
```