重 庆 交 通 大 学

学生实验报告

课 程 名 称：数 字 图 像 处 理 .

开 课实验室：软 件 实 验 中 心 .

学 院：信息学院 2021 年级物联网工程专业 2 班

学 生 姓 名：李骏飞学 号 632109160602

指 导 教 师：蓝 章 礼 .

开 课 时 间：2023 至 2024 学年第 二 学期

成 绩	
教师签名	

实验项目名称		实验三：图像形态学变换			
姓名	李骏飞	学号	**632109160602**	实验日期	**2024.4.17**
教师评阅： 1:实验目的明确□A□B□C□D 2:内容与原理□A□B□C□D 3:实验报告规范□A□B□C□D 4:实验主要代码与效果展示□A□B□C□D 5:实验分析总结全面□A□B□C□D					
实验记录					

一、实验目的

对二值图像进行形态学变换。选做要求：可选择对彩色图像、灰度图像进行形态学变换。

二、实验主要内容及原理

（0）结构元（SE）

结构元(SE,Structure Element)就是卷积操作中的卷积核,或者是空间域滤波中提到的滤波器。虽然形态学操作中结构元的形状可以是任意的，但是由于在图像操作中，为了方便计算，通常要求结构元是矩形的阵列，对于任意形状的结构元，如果不满足矩形的要求，则用 0 将其填充为矩形即可。

另外，结构元内部的有效元素不像滤波器那样有权值，通常结构元中只分为两种元素，就是 0 和 1，不会出现其他数值的系数。(当然对有些算法来说也有例外)。结构元对图像进行的操作也和卷积非常类似，就是由结构元的中心依次滑过图像，然后进行设计好的操作即可。

（1）图像的腐蚀：

图像的腐蚀（Erosion），用于改变图像的形状和结构。腐蚀操作可以使图像中**边缘细化**、物体缩小，并且可以**去除小的连通区域**。它在图像处理中常用于去除噪声、断开物体之间的连接以及图像分割等应用。

腐蚀操作基于**结构元素（SE）**的概念,腐蚀操作将结构元素在图像上滑动，并将结构元素的中心与图像中的像素进行比较。如果结构元素的所有元素与图像中的对应像素都匹配，那么该像素保持不变；否则，该像素被置为背景色。

（2）图像的膨胀：

图像的膨胀（Dilation）是数字图像处理中的一种形态学操作，用于改变图像的形状和结构。膨胀操作可以使图像中的物体变大、**边缘加粗**，并且可以**填充物体间的空隙**。它在图像处理中常用于填充小的空洞、连接断开的物体以及图像分割等应用。

（3）图像的开运算

图像开运算是图像依次经过腐蚀、膨胀处理后的过程。具体来说，图

像被腐蚀后，可以去除噪声，但也会压缩图像；接着对腐蚀过的图像进行膨胀处理，可以进一步去除噪声，并恢复原有图像的大小。开运算能够去除图像中的小物体，在纤细点分离物体，平滑较大物体的边界同时并不明显改变其面积。

（4）图像的闭运算

对于每个像素的 RGB 分量值，可以通过增加或减小其值来调整像素的亮度。可以使用以下公式来进行亮度调节

（5）图像的边缘提取

形态学梯度（Morphological Gradient）为**膨胀图与腐蚀图之差**，对二值图像进行这一操作可以将团块的边缘突出出来。我们可以用形态学梯度来**保留物体的边缘轮廓**。

（6）图像的顶帽运算

顶帽运算（Top Hat）又常常被译为“礼帽”运算。为**原始图像**与开**运算**之后得到的图像的

因为开运算带来的结果是放大了裂缝或者局部低亮度的区域，因此，从原图中减去开运算后的图，得到的效果图突出了比原图轮廓周围的区域更明亮的区域，且这一操作和选择的核的大小相关。

顶帽运算的作用是可以提取噪声，突出原图像中比周围亮的部分。（因为开运算本身可以去除一些孤立点，细微连接，毛刺等细节，所以这些细节就可以通过顶帽操作来提取出来）

（7）图像的黑帽运算

黑帽（Black Hat）运算为**"闭运算"的结果图与原图像之差**。黑帽运算后的效果图突出了比原图轮廓周围的区域更暗的区域，且这一操作和选择的核的大小相关。

黑帽操作的作用是**突出原图像中比周围暗的区域。(**比如闭运算本身可以填补物体内部的一些黑洞，这些黑洞就可以通过黑帽运算来凸显）。

三、实验环境

Windows11

Visual Studio2021

C#语言

四、实验主要代码与效果展示

二值图像的腐蚀

➢ 算法描述：

从文本框中获取用户输入的腐蚀结构元 SE 的大小。如果用户输入的内容可以成功转换为整数类型，则将其保存在 target 变量中。根据 target 大小创建一个为全 1 模板 SE，并将所有元素设置为 1。

遍历图像除了边界的每个像素点，对于每个像素点(i, j)，检查以当前像素点为中心的区域内的像素是否与腐蚀结构元匹配。如果结构元区域内的某个像素为 1，并且对应的原始图像中的像素不是白色（255），则说明结构元与周围像素不匹配，将腐蚀标志 erode 设置为 false，否则置为 true。根据 erode 的情况进行赋值，如果存在不匹配的情况，将当前像素点设置为黑色，即执行腐蚀操作；如果结构元区域内的所有像素都匹配，将当前像素点设置为白色，即不执行腐蚀操作。

最后处理边界像素，这里选择全部置为黑色（0）。

该算法的核心思想是根据用户指定的腐蚀结构元的大小，在图像中遍历每个像素点，并将其周围的区域与腐蚀结构元进行比较，根据匹配结果确定是否进行腐蚀操作。

```
1.  //二值图像腐蚀
2.  private void BinaryErosion()//二值图像的腐蚀函数(腐蚀白色)
3.  {
4.      Bitmap bt1 = new Bitmap(pictureBox_show.Image);
5.
6.      int target;//SE 大小
7.      int target_binary;
8.      if (int.TryParse(textBox_structureElementValue.Text, out target))
9.      {
10.         if (target % 2 != 0)//奇数
11.         {
12.             // 转换成功，可以使用 target 变量和 target_binary 变量进行后续操作
13.             target = int.Parse(textBox_structureElementValue.Text); // 均值
14.             target_binary = target / 2;
15.         }
16.         else
17.         {
18.             // 转换失败，弹窗提示报错
19.             MessageBox.Show("请输入奇数值", "错误
    ", MessageBoxButtons.OK, MessageBoxIcon.Error);
20.             return;
```

```
        }
    }
    else
    {
        // 转换失败，弹窗提示报错
        MessageBox.Show("输入的文本不是有效的数字", "错误
", MessageBoxButtons.OK, MessageBoxIcon.Error);
        return;
    }

    // 创建 11x11 的全 1 模板
    int[,] se = new int[target, target];
    for (int i = 0; i < target; i++)
    {
        for (int j = 0; j < target; j++)
        {
            se[i, j] = 1;
        }
    }

    // 获取图像的宽度和高度
    int width = bt1.Width;
    int height = bt1.Height;

    // 创建新的 Bitmap 对象用于保存处理后的图像
    Bitmap resultImage = new Bitmap(width, height);

    // 遍历图像的每个像素点
    for (int i = target_binary; i < width - target_binary; i++)
    {
        for (int j = target_binary; j < height - target_binary; j++)
        {
            bool erode = true;

            // 检查结构元素是否与周围像素匹配
            for (int k = -target_binary; k <= target_binary; k++)
            {
                for (int l = -target_binary; l <= target_binary; l++)
                {
                    // 若结构元素与周围像素不匹配，则不进行腐蚀操作
                    if (se[k + target_binary, l + target_binary] == 1 && bt1.GetPi
xel(i + k, j + l).R != 255)//有黑色部分
                    {
                        erode = false;
                        break;
                    }
                }
                if (!erode)
                {
                    break;
                }
            }

            // 如果结构元素与周围像素匹配，则将该像素设置为白色
            if (erode)
            {
```

```
                resultImage.SetPixel(i, j, Color.White);
            }
            else//否则设置为黑色
            {
                resultImage.SetPixel(i, j, Color.Black);
            }

            //pictureBox_show.Refresh();
        }
    }

    // 处理图像边界
    for (int i = 0; i < width; i++)
    {
        for (int j = 0; j < target_binary; j++)
        {
            resultImage.SetPixel(i, j, Color.Black);
            resultImage.SetPixel(i, height - j - 1, Color.Black);
        }
    }
    for (int j = 0; j < height; j++)
    {
        for (int i = 0; i < target_binary; i++)
        {
            resultImage.SetPixel(i, j, Color.Black);
              resultImage.SetPixel(width - i - 1, j, Color.Black);
          }
      }

      // 显示处理后的图像
      pictureBox_show.Image = resultImage;
}
```

- **演示效果：**

可视化界面展示：

下面是腐蚀操作对实际图像的处理效果。

第一张为原图。

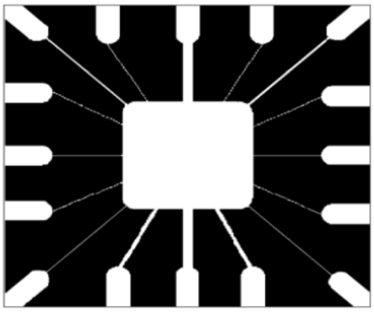
第二张为用 3*3 的全 1 模板作结构元进行的腐蚀，可以看到，由于细线的宽度小于 3 像素，所以细线不可能完全包含 SE，因此细线部分被“腐蚀”。

第三张图结构元尺寸为 5x5，可以看到，只留下的最粗的线条。

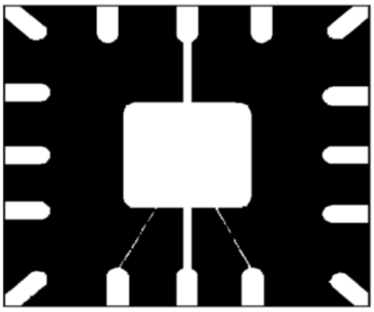
第五张 SE 尺寸为 45x45，连最粗的线都被腐蚀了。

第五张 SE 尺寸为 31x31，就可以将除了中心方块以外的所有元素腐蚀掉。

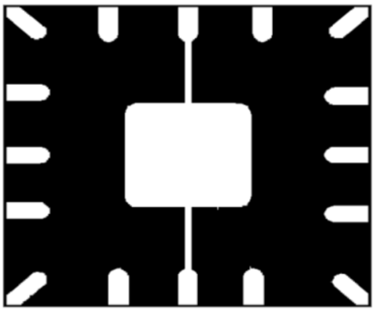
另外，因为我自己写的腐蚀操作是带 padding 的，也就是会根据 SE 的尺寸对图像周边填 0，所以不会因为一遍遍的腐蚀操作导致图像变小。



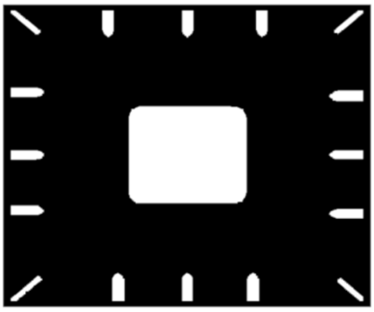
原图



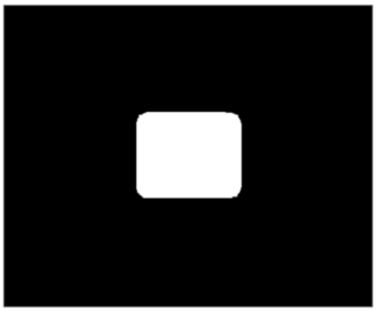
3*3的腐蚀结构元



5*5的腐蚀结构元



15*15的腐蚀结构元



31*31的腐蚀结构元

二值图像的膨胀

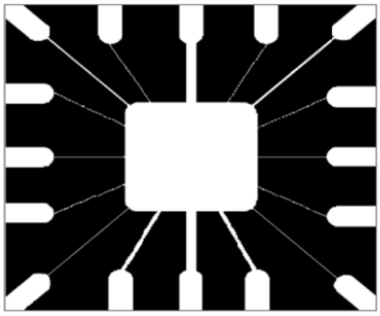
➢ 算法描述：

与二值图像的腐蚀操作类似，只是修改了下判断条件即可，在结构元素中如果出现元素不为白色（255），直接退出循环，将该像素点赋值为 0，开启下一轮循环即可。

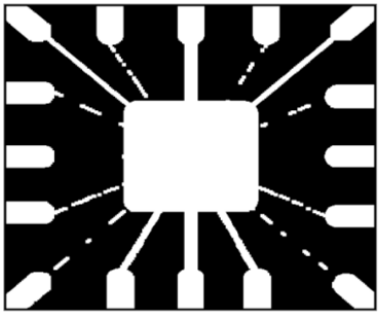
```
//二值图像的膨胀函数? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ?
private void BinaryDilation()//二值图像的膨胀函数(膨胀白色)
{
    int target;
    int target_binary;
    if (int.TryParse(textBox_structureElementValue.Text, out target))
    {
        if (target % 2 != 0)//奇数
        {
            // 转换成功，可以使用 target 变量和 target_binary 变量进行后续操作
            target = int.Parse(textBox_structureElementValue.Text); // 均值
            target_binary = target / 2;
        }
        else
        {
            // 转换失败，弹窗提示报错
            MessageBox.Show("请输入奇数", "错误
", MessageBoxButtons.OK, MessageBoxIcon.Error);
            return;
```

```
        }

    }
    else
    {
        // 转换失败，弹窗提示报错
        MessageBox.Show("输入的文本不是有效的数字", "错误
", MessageBoxButtons.OK, MessageBoxIcon.Error);
        return;
    }


    Bitmap bt1 = new Bitmap(pictureBox_show.Image);
    Bitmap bt2 = new Bitmap(pictureBox_show.Image);
    int R;

    for (int i = 0; i < bt1.Width; i++)
    {
        for (int j = 0; j < bt1.Height; j++)
        {
            R = bt1.GetPixel(i, j).R;
            if (R == 255)//是白色
            {
                for (int k = -target_binary; k <= target_binary; k++)//以 i,j 为中心
点
                {
                    for (int l = -target_binary; l <= target_binary; l++)
                    {
                        if (i + k >= 0 && i + k < bt1.Width &&
                            j + l >= 0 && j + l < bt1.Height)
                        {
                            bt2.SetPixel(i + k, j + l, Color.FromArgb(255, 255, 25
5));
                        }

                    }
                }
            }
        }
        pictureBox_show.Refresh();//刷新图片框
        pictureBox_show.Image = bt2;
    }
}
```

- **演示效果：**



原图



5*5的膨胀结构元

11*11的膨胀结构元

二值图像的开启

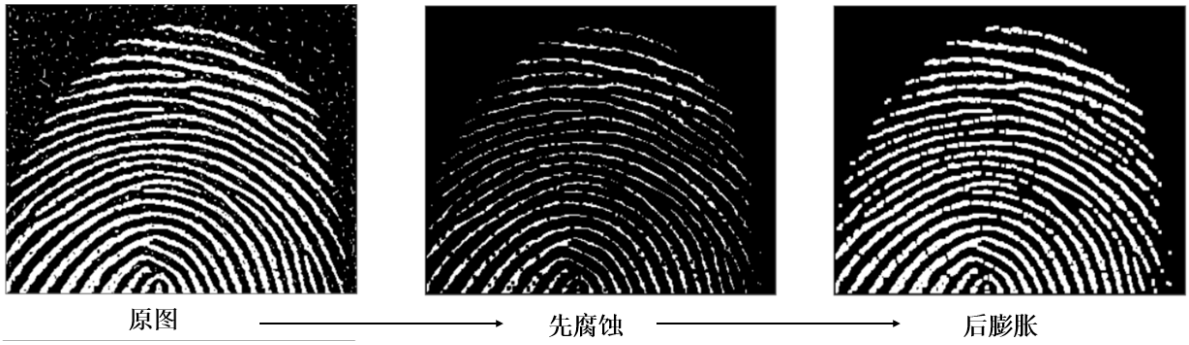
➢ 算法描述：

图像开运算是图像依次经过**腐蚀、膨胀**处理后的过程。具体来说，图像被腐蚀后，可以去除噪声，但也会压缩图像；接着对腐蚀过的图像进行膨胀处理，可以进一步去除噪声，并恢复原有图像的大小。开运算能够去除图像中的小物体，在纤细点分离物体，平滑较大物体的边界同时并不明显改变其面积。

```
1.  BinaryErosion(); //先腐蚀
2.  BinaryDilation(); //再膨胀
```

➢ 演示效果：

图 1 是原图像，图 3 是开启之后的图像，可以看到，在采集指纹的图像中，存在很多不需要的噪声干扰，根据编写的腐蚀、膨胀操作代码实现开操作，使得细小的噪声完全消失了，而指纹主体也通过膨胀很好的恢复了。



二值图像的闭合

➢ 算法描述：

图像闭运算是图像依次经过**膨胀、腐蚀**处理后的过程。具体来说，图像先膨胀，后腐蚀，它能够帮助关闭前景物体内部的小孔，或去除物体上的小黑点。闭运算能够填平小孔，弥合小裂缝，而总的位置和形状不变。

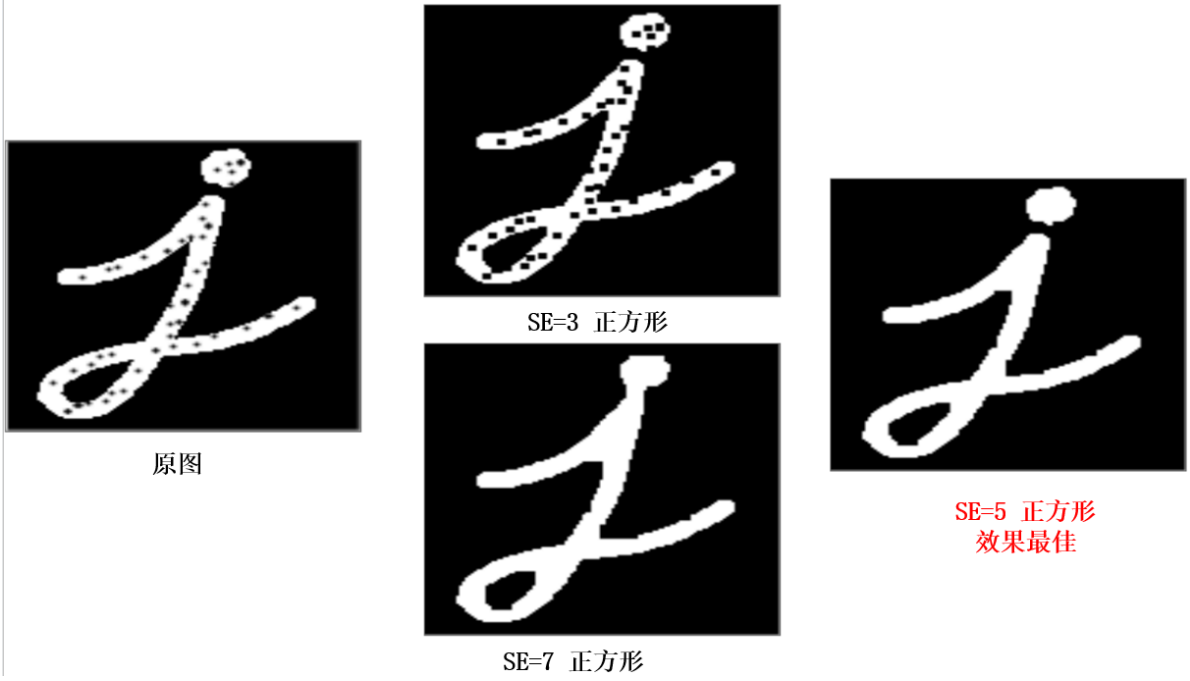
```
1.  BinaryDilation(); //先膨胀
2.  BinaryErosion(); //再腐蚀
3.
```

➢ 演示效果：

可视化界面展示：

下图是在**开操作**得到的图 3 基础上进一步进行**闭操作**的结果图，可以看到，相比于之前的图像，闭操作将很多指纹中断裂的区域连接起来，使得指纹的图像更加完整。但是对于较大的裂缝，闭操作是无能为力的，可以增大 SE 的尺寸，但是这样很可能使得指纹不同纹路也连接起来，反而得效果不好。

以下是另一幅图像进行比操作之后的结果，只要调整好 SE 的尺寸，就能得到较好的修复效果。



二值图像的形态学梯度

➢ 算法描述：

从图像框中获取原始图像和显示图像的副本，确保在处理过程中不影响原始图像，然后调用 BinaryErosion() 方法进行腐蚀操作，将腐蚀操作后的图像存储在 bt1 中。

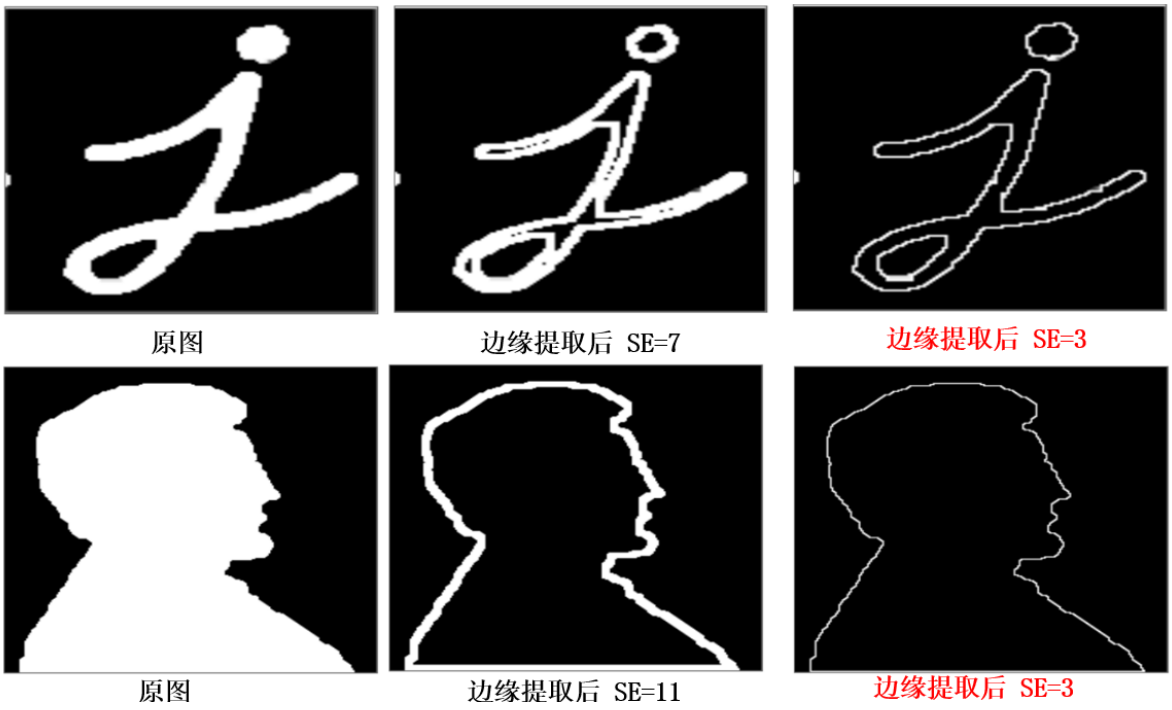
接下来，通过遍历图像的每个像素来计算边缘值 R，它是当前像素的红色通道值与原始图像对应位置像素的红色通道值之差的绝对值。这个值表示了当前像素与原始图像对应位置像素的差异，也可以是绿色或蓝色。使用计算得到的边缘值 R3，创建一个新的颜色，并将其设置为 bt2 中相同位置的像素的颜色在进行赋值即可。

```
1.  //二值图像的边缘提取
2.  private void binary_handle_change_边缘提取(object sender, EventArgs e)
3.  {
4.      Bitmap bt1 = new Bitmap(pictureBox_show.Image);
5.      Bitmap bt2 = new Bitmap(pictureBox_show.Image);
6.      Bitmap originalBt = new Bitmap(original_image.Image);
7.      BinaryErosion(); //先腐蚀
8.      bt1 = new Bitmap(pictureBox_show.Image);
9.      int R1, R2, R3;
10.     for (int i = 0; i < bt1.Width; i++)
11.     {
12.         for (int j = 0; j < bt1.Height; j++)
13.         {
14.             R1 = bt1.GetPixel(i, j).R;
```

```
            R2 = originalBt.GetPixel(i, j).R;
            R3 = (R2 - R1) > 0 ? (R2 - R1) : (R1 - R2);
            bt2.SetPixel(i, j, Color.FromArgb(R3, R3, R3));
        }
        pictureBox_show.Refresh();//刷新图片框
        pictureBox_show.Image = bt2;
    }
}
```

- **实现效果：**

测试用例：



二值图像的顶帽运算

➢ 算法描述：

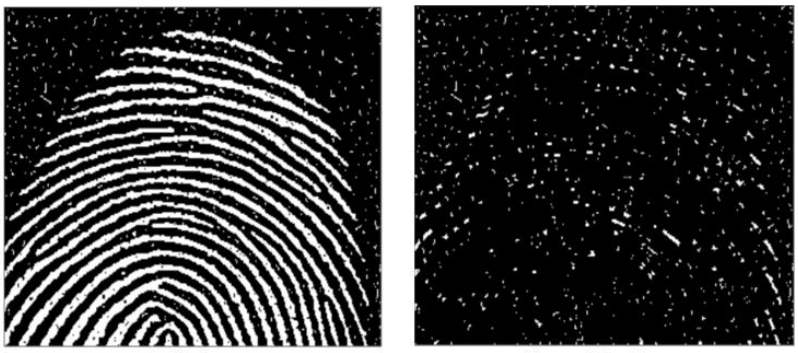
顶帽运算是通过开运算结果图像与原图像之差来获取的。先通过执行开运算（BinaryErosion 和 BinaryDilation）获取开运算之后的图像（bt_open）。然后遍历图像的每个像素，获取开运算图像的像素值（R1）和原图像的像素值（R2）。根据 R1 和 R2 的差值计算顶帽图像的像素值（R3），并将其设置到 bt2 图像的相应位置。

```
1.  //二值图像的顶帽运算
2.  private void binary_top_hat(object sender, EventArgs e)
3.  {
4.      Bitmap bt2 = new Bitmap(pictureBox_show.Image);
5.      Bitmap originalBt = new Bitmap(pictureBox_show.Image);//原始图像
6.      //执行开运算，先腐蚀，再开启
7.      BinaryErosion();
8.      BinaryDilation();
9.      Bitmap bt_open = new Bitmap(pictureBox_show.Image);//获取开运算之后的图像
10.     // 循环遍历图像的每个像素
11.     int R1, R2, R3;
12.     for (int i = 0; i < bt_open.Width; i++)
13.     {
14.         for (int j = 0; j < bt_open.Height; j++)
15.         {
16.             R1 = bt_open.GetPixel(i, j).R; // 获取顶帽图像的像素值
17.             R2 = originalBt.GetPixel(i, j).R;// 获取原图像的像素值
18.             R3 = (R2 - R1) > 0 ? (R2 - R1) : (R1 - R2);
```

```
            bt2.SetPixel(i, j, Color.FromArgb(R3, R3, R3));
        }
        pictureBox_show.Refresh();//刷新图片框
        pictureBox_show.Image = bt2;
    }

}
```

- **实现效果：**

进行顶帽运算后，可以一定程度上提取到噪声，但是由于在实验中图片的分辨率较小，对于像素集中起来的图片处理效果一般，下面是两幅图的对比，明显第一幅提取噪声的效果要好一些。



原图　　顶帽运算后 SE=3

原图　　顶帽运算后 SE=1　　顶帽运算后 SE=3

二值图像的黑帽运算

- **算法描述：**

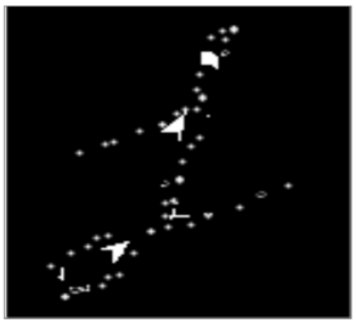
创建一个新的 Bitmap 对象（bt2）作为黑帽图像的容器，并将原始图像赋值给 bt2 和 originalBt。通过执行闭运算（先膨胀再腐蚀）获取闭运算之后的图像。遍历 bt_close 图像的每个像素，在循环中，获取 bt_close 图像的像素值（R1）和 originalBt 图像的像素值（R2）。根据 R1 和 R2 的差值计算黑帽图像的像素值（R3），并将其设置到 bt2 图像的相应位

置。计算闭运算图像与原图像之差，即可获取到二值图像的黑帽图像。

```
//二值图像的黑帽运算
private void binary_black_hat(object sender, EventArgs e)
{
    Bitmap bt2 = new Bitmap(pictureBox_show.Image);
    Bitmap originalBt = new Bitmap(pictureBox_show.Image);//原始图像

    BinaryDilation();//执行闭运算，先开启，再腐蚀
    BinaryErosion();


    Bitmap bt_close = new Bitmap(pictureBox_show.Image);//获取开运算之后的图像
    // 循环遍历图像的每个像素
    int R1, R2, R3;
    for (int i = 0; i < bt_close.Width; i++)
    {
        for (int j = 0; j < bt_close.Height; j++)
        {
            R1 = bt_close.GetPixel(i, j).R; // 获取黑帽图像的像素值
            R2 = originalBt.GetPixel(i, j).R;// 获取原图像的像素值
            R3 = (R2 - R1) > 0 ? (R2 - R1) : (R1 - R2);
            bt2.SetPixel(i, j, Color.FromArgb(R3, R3, R3));
        }
        pictureBox_show.Refresh();//刷新图片框
        pictureBox_show.Image = bt2;
    }
}
```

- **实现效果：**

原图

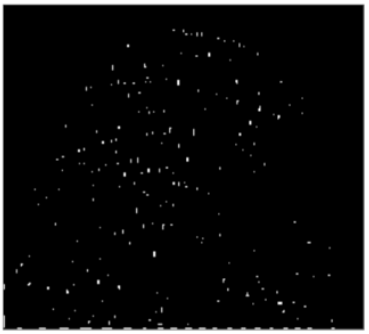


黑帽运算后 SE=7

黑帽运算后 SE=1 效果最好

原图

黑帽运算后 SE=7



黑帽运算后 SE=3 效果最好

彩色图像(灰度图像)的膨胀

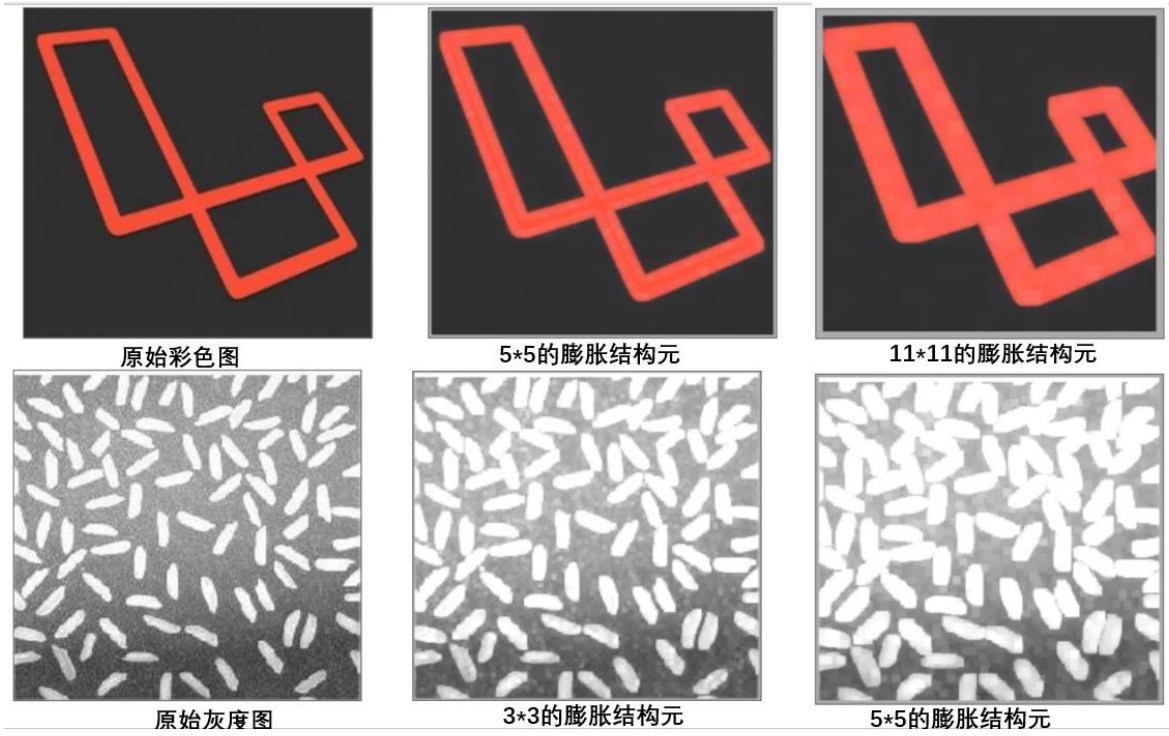
➢ 算法描述：

创建原始图像的副本和用于存储膨胀后图像的对象。然后循环遍历原始图像的每个像素，在内层循环中，创建一个颜色数组来存储结构元素内每个像素的颜色通道值。同时，使用一个索引变量来追踪颜色数组的位置，获取结构元素内每个像素的颜色值，并将其存储到相应的颜色通道数组中。内层循环结束后，对每个颜色通道的数组进行排序，以获取膨胀后的颜色值。膨胀后的颜色值是颜色通道数组中最大的值。根据膨胀后的颜色值，使用 SetPixel 方法将其设置到膨胀图像的相应位置即可。

```
1.  //彩色图像的膨胀函数
2.  private void ColorDilation()
3.  {
4.      int target;//SE 大小
5.      int target_binary;
6.      if (int.TryParse(textBox_structureElementValue.Text, out target))
7.      {
8.          if (target % 2 != 0)//奇数
9.          {
10.             // 转换成功，可以使用 target 变量和 target_binary 变量进行后续操作
11.             target = int.Parse(textBox_structureElementValue.Text); // 均值
12.             target_binary = target / 2;
13.         }
14.         else
15.         {
16.             // 转换失败，弹窗提示报错
17.             MessageBox.Show("请输入奇数值", "错误
    ", MessageBoxButtons.OK, MessageBoxIcon.Error);
18.             return;
19.         }
20.     }
21.     else
22.     {
23.         // 转换失败，弹窗提示报错
24.         MessageBox.Show("输入的文本不是有效的数字", "错误
    ", MessageBoxButtons.OK, MessageBoxIcon.Error);
25.         return;
26.     }
27.
28.     Bitmap bt1 = new Bitmap(pictureBox_show.Image);
29.     // 创建膨胀后的图像对象
30.     Bitmap dilatedImage = new Bitmap(bt1.Width, bt1.Height);
31.
32.     // 定义结构元素大小和形状
33.     int structElementSize = target;  // 结构元素大小
34.     int structElementHalfSize = target_binary;
35.
36.     // 循环遍历图像的每个像素
```

```
    for (int i = structElementHalfSize; i < bt1.Width - structElementHalfSize; i++)
    {
        for (int j = structElementHalfSize; j < bt1.Height - structElementHalfSize; j++)
        {
            // 创建一个颜色数组来存储每个颜色通道的值
            int[] redChannel = new int[structElementSize * structElementSize];
            int[] greenChannel = new int[structElementSize * structElementSize];
            int[] blueChannel = new int[structElementSize * structElementSize];

            int index = 0;

            // 循环遍历结构元素的每个像素
            for (int k = -structElementHalfSize; k <= structElementHalfSize; k++)

            {
                for (int l = -structElementHalfSize; l <= structElementHalfSize; l++)
                {
                    // 获取结构元素的颜色
                    Color structElementColor = bt1.GetPixel(i + k, j + l);

                    // 存储颜色通道的值
                    redChannel[index] = structElementColor.R;
                    greenChannel[index] = structElementColor.G;
                    blueChannel[index] = structElementColor.B;

                    index++;
                }
            }

            // 对每个颜色通道进行排序并获取膨胀后的颜色值
            Array.Sort(redChannel);
            Array.Sort(greenChannel);
            Array.Sort(blueChannel);

            // 获取膨胀后的颜色值
            int dilatedRed = redChannel[structElementSize * structElementSize - 1];
            int dilatedGreen = greenChannel[structElementSize * structElementSize - 1];
            int dilatedBlue = blueChannel[structElementSize * structElementSize - 1];

            // 设置膨胀后的颜色值到膨胀图像中
            dilatedImage.SetPixel(i, j, Color.FromArgb(dilatedRed, dilatedGreen, dilatedBlue));
        }
    }

    pictureBox_show.Image = dilatedImage;

}
```

➢ **实现效果：**



彩色图像(灰度图像)的腐蚀

➢ **算法描述：**

创建一个与原始图像大小相同的新图像对象 erodedImage，用于存储

腐蚀后的图像。遍历原始图像的每个像素，对于每个像素，初始化最小的红色值 minRed、绿色值 minGreen 和蓝色值 minBlue 为 255（最大值），然后在结构元素的范围内，遍历与当前像素相邻的像素，对于每个相邻像素，获取其红色、绿色和蓝色分量的值。并检查相邻像素的分量值是否小于当前最小分量值，如果是，则更新对应的最小分量值。

在经过结构元素范围内的所有相邻像素中，找到最小的红色、绿色和蓝色分量值。使用最小分量值创建一个新的颜色 Color 对象，并将其赋值给腐蚀后图像 erodedImage 中对应位置的像素。

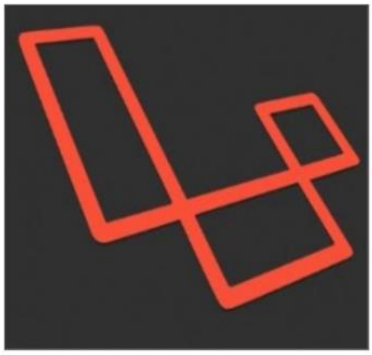
循环遍历所有像素，完成腐蚀操作后，将腐蚀后的图像显示在 PictureBox 控件中。

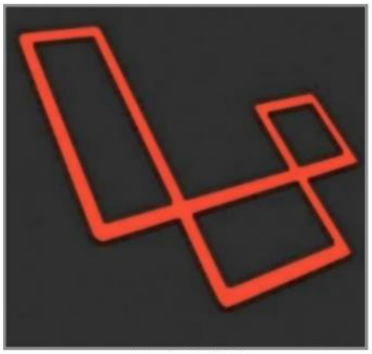
由于是彩色或者灰度图像，腐蚀操作是针对每个颜色通道分别进行的，就保持了当前图像的颜色信息。

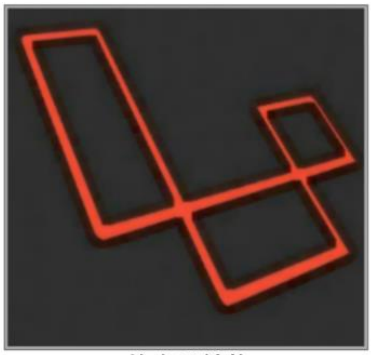
```
1.  //彩色图像腐蚀
2.  private void ColorErosion()
3.  {
4.      int target;//SE 大小
5.      int target_binary;
6.      if (int.TryParse(textBox_structureElementValue.Text, out target))
7.      {
8.          if (target % 2 != 0)//奇数
9.          {
10.             // 转换成功，可以使用 target 变量和 target_binary 变量进行后续操作
11.             target = int.Parse(textBox_structureElementValue.Text); // 均值
12.             target_binary = target / 2;
13.         }
14.         else
15.         {
16.             // 转换失败，弹窗提示报错
17.             MessageBox.Show("请输入奇数值", "错误
    ", MessageBoxButtons.OK, MessageBoxIcon.Error);
18.             return;
19.         }
20.     }
21.     else
22.     {
23.         // 转换失败，弹窗提示报错
24.         MessageBox.Show("输入的文本不是有效的数字", "错误
    ", MessageBoxButtons.OK, MessageBoxIcon.Error);
25.         return;
26.     }
27.
28.     Bitmap bt1 = new Bitmap(pictureBox_show.Image);
29.
30.     // 创建腐蚀后的图像对象
31.     Bitmap erodedImage = new Bitmap(bt1.Width, bt1.Height);
32.
33.     // 定义结构元素大小和形状
```

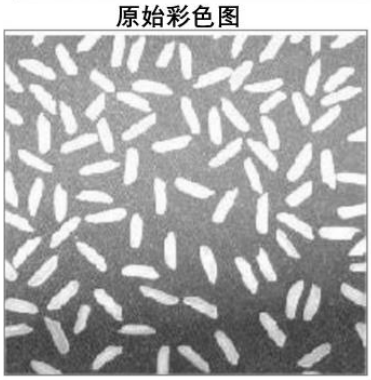
```
34.     int structElementSize = target;  // 结构元素大小
35.     int structElementHalfSize = target_binary;
36.
37.
38.     for (int i = structElementHalfSize; i < bt1.Width - structElementHalfSize; i++
    )
39.     {
40.         for (int j = structElementHalfSize; j < bt1.Height - structElementHalfSize
    ; j++)
41.         {
42.             int minRed = 255;
43.             int minGreen = 255;
44.             int minBlue = 255;
45.
46.             for (int k = -structElementHalfSize; k <= structElementHalfSize; k++)

47.             {
48.                 for (int l = -structElementHalfSize; l <= structElementHalfSize; l
    ++)
49.                 {
50.                     Color pixelColor = bt1.GetPixel(i + k, j + l);
51.                     if (pixelColor.R < minRed)
52.                     {
53.                         minRed = pixelColor.R;
54.                     }
55.                     if (pixelColor.G < minGreen)
56.                     {
57.                         minGreen = pixelColor.G;
58.                     }
59.                     if (pixelColor.B < minBlue)
60.                     {
61.                         minBlue = pixelColor.B;
62.                     }
63.                 }
64.             }
65.
66.             erodedImage.SetPixel(i, j, Color.FromArgb(minRed, minGreen, minBlue));

67.         }
68.     }
69.     pictureBox_show.Image = erodedImage;
70.
71. }
```

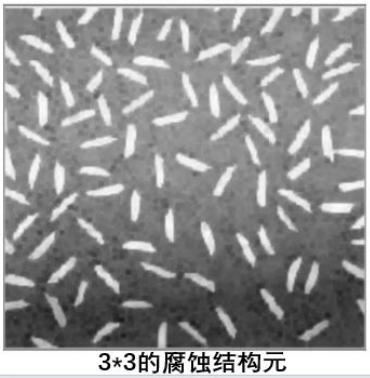
- **实现效果：**

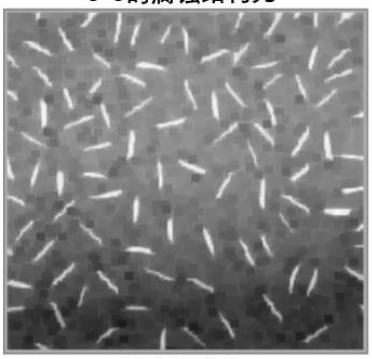

原始彩色图


5*5的腐蚀结构元


5*5的腐蚀结构元


原始灰度图


3*3的腐蚀结构元


5*5的腐蚀结构元

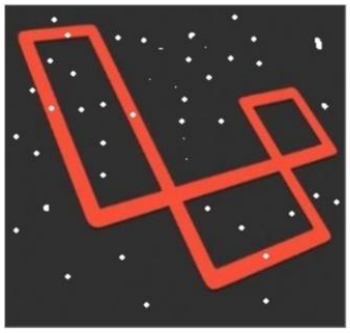
彩色图像(灰度图像)的开操作

➢ 算法描述：

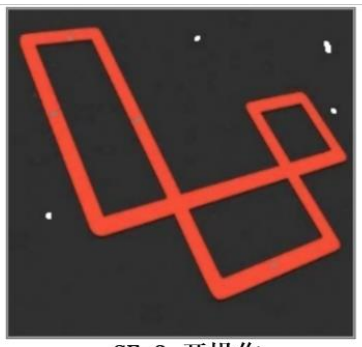
和二值图像的开操作一样，先执行**腐蚀操作**，再执行**膨胀操作**即可。

```
1.  ColorErosion(); //先腐蚀
2.  ColorDilation(); //再膨胀
3.
```

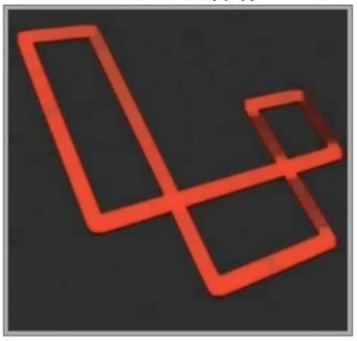
➢ 实现效果：



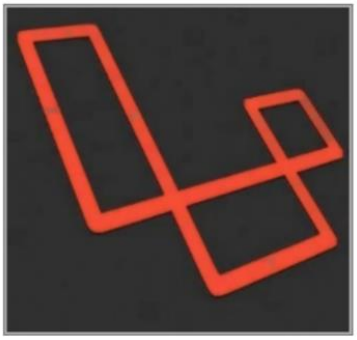
原图



SE=3 开操作



SE=7 开操作



SE=5 开操作 效果最好

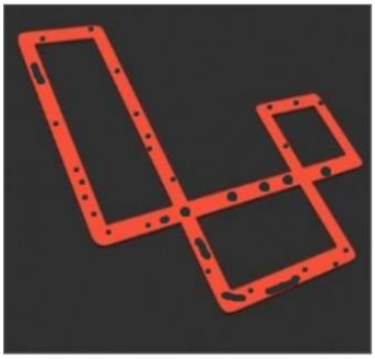
彩色图像(灰度图像)的闭操作

➢ 算法描述：

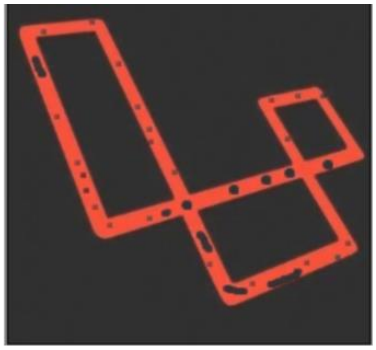
和二值图像的闭操作一样，先执行**膨胀操作**，再执行**腐蚀操作**即可。

```
1.  ColorDilation(); //先膨胀
2.  ColorErosion(); //再腐蚀
```

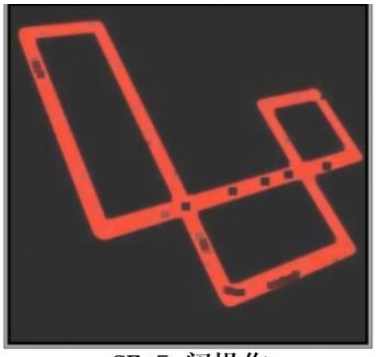
➢ **实现效果：**



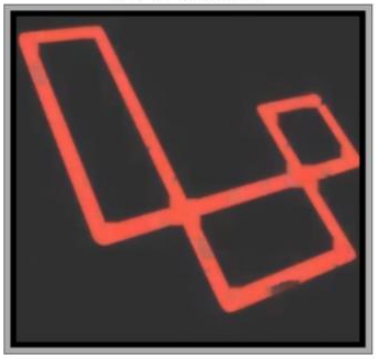
原图



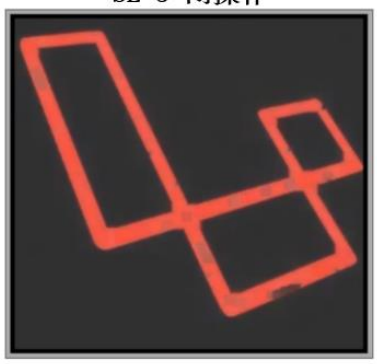
SE=3 闭操作



SE=5 闭操作



SE=9 闭操作



SE=7 闭操作 效果最好

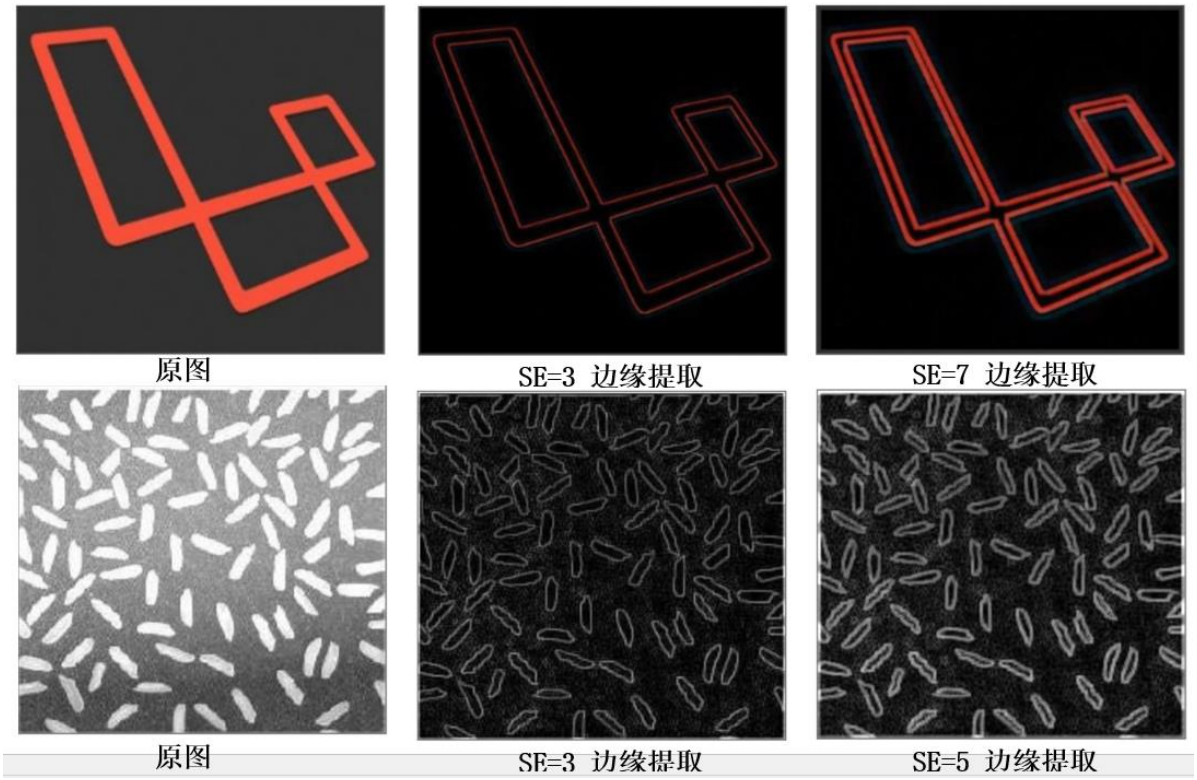
彩色图像(灰度图像)的边缘提取操作

➢ **算法描述：**

与二值图像的边缘提取操作类似，通过对原始图像和腐蚀后的图像进行像素值的比较，得到边缘处的差异值。不同的是，彩色图像（灰度图像）需要分别对红色、绿色和蓝色三个通道进行比较，计算得到各个通道的差异值，然后将差异值作为新图像对应位置的像素值。

```
1.  //彩色图像的边缘提取
2.  private void color_handle_change_边缘提取(object sender, EventArgs e)
3.  {
4.      Bitmap bt1;
5.      Bitmap bt2 = new Bitmap(pictureBox_show.Image);
6.      Bitmap originalBt = new Bitmap(original_image.Image);
7.      ColorErosion();//先腐蚀
8.      bt1 = new Bitmap(pictureBox_show.Image);
9.      for (int i = 0; i < bt1.Width; i++)
10.     {
11.         for (int j = 0; j < bt1.Height; j++)
12.         {
13.             Color color1 = bt1.GetPixel(i, j);
14.             Color color2 = originalBt.GetPixel(i, j);
15.             int diffR = Math.Abs(color2.R - color1.R);
16.             int diffG = Math.Abs(color2.G - color1.G);
17.             int diffB = Math.Abs(color2.B - color1.B);
18.             bt2.SetPixel(i, j, Color.FromArgb(diffR, diffG, diffB));
19.         }
20.         pictureBox_show.Refresh();//刷新图片框
21.         pictureBox_show.Image = bt2;
22.     }
23.
24.
25.
26. }
```

➢ **实现效果：**



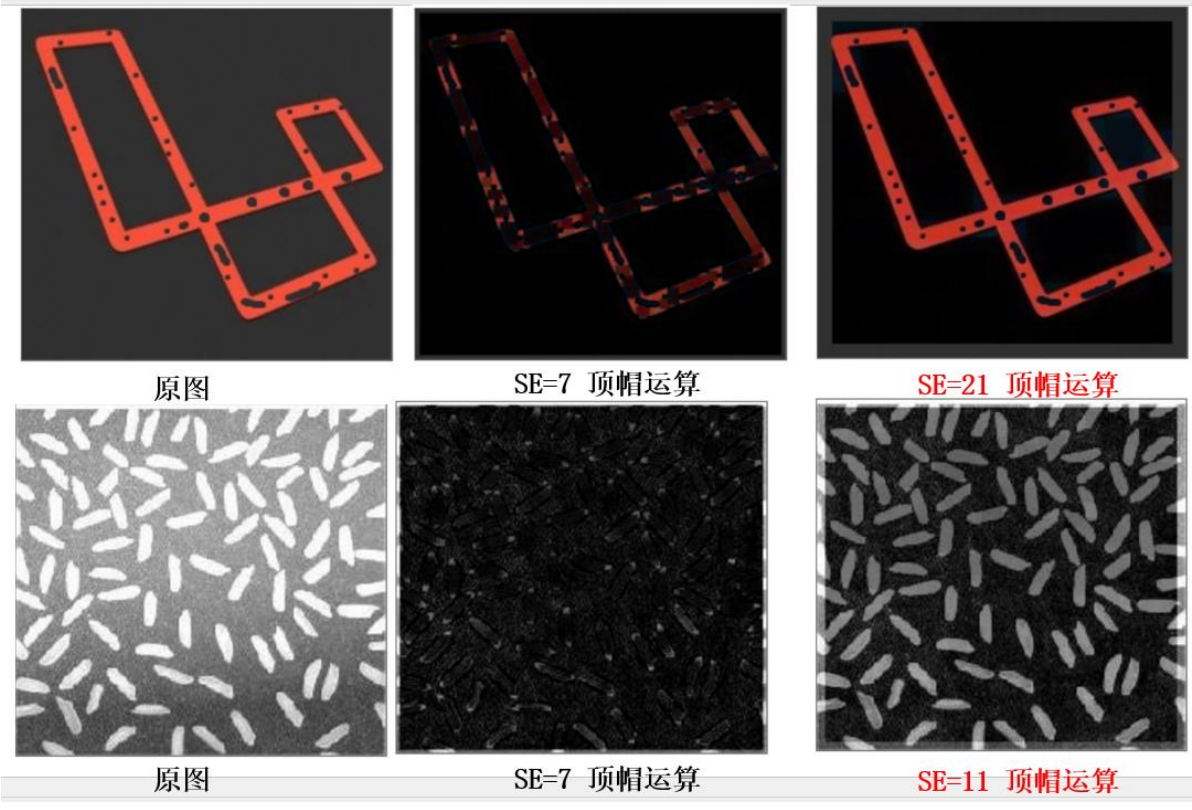
彩色图像(灰度图像)的顶帽运算操作

➢ 算法描述：

核心算法思想同彩色图像的边缘提取操作。

```
1.  //彩色图像的顶帽运算
2.  private void color_top_hat(object sender, EventArgs e)
3.  {
4.      Bitmap bt1 = new Bitmap(pictureBox_show.Image);
5.      Bitmap originalBt = new Bitmap(pictureBox_show.Image);//原始图像
6.      //执行开运算，先腐蚀，再开启
7.      ColorErosion();
8.      ColorDilation();
9.      Bitmap bt_open = new Bitmap(pictureBox_show.Image);//获取开运算之后的图像
10.     // 循环遍历图像的每个像素
11.     int R1, R2, R3;
12.     for (int i = 0; i < bt_open.Width; i++)
13.     {
14.         for (int j = 0; j < bt_open.Height; j++)
15.         {
16.             Color color1 = originalBt.GetPixel(i, j);
17.             Color color2 = bt_open.GetPixel(i, j);
18.             int diffR = Math.Abs(color2.R - color1.R);
19.             int diffG = Math.Abs(color2.G - color1.G);
20.             int diffB = Math.Abs(color2.B - color1.B);
21.             bt1.SetPixel(i, j, Color.FromArgb(diffR, diffG, diffB));
22.         }
23.         pictureBox_show.Refresh();//刷新图片框
24.         pictureBox_show.Image = bt1;
25.     }
26. }
```

- **实现效果：**



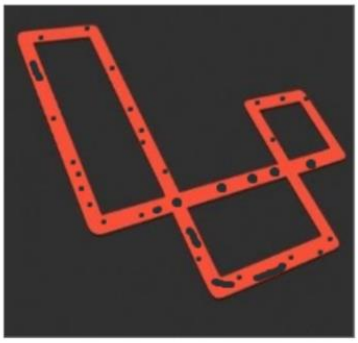
彩色图像(灰度图像)的黑帽运算操作

➢ 算法描述：

核心算法思想同彩色图像的边缘提取操作。

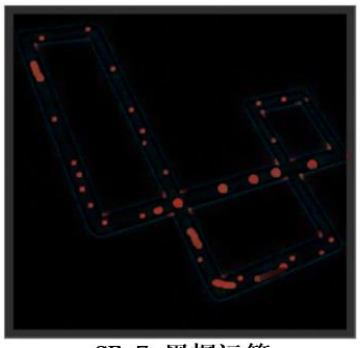
```
1.  //彩色图像的黑帽运算
2.  private void color_black_hat(object sender, EventArgs e)
3.  {
4.      Bitmap bt1 = new Bitmap(pictureBox_show.Image);
5.      Bitmap originalBt = new Bitmap(pictureBox_show.Image);//原始图像
6.
7.      ColorDilation();//执行闭运算，先开启，再腐蚀
8.      ColorErosion();
9.
10.
11.     Bitmap bt_close = new Bitmap(pictureBox_show.Image);//获取开运算之后的图像
12.     // 循环遍历图像的每个像素
13.     int R1, R2, R3;
14.     for (int i = 0; i < bt_close.Width; i++)
15.     {
16.         for (int j = 0; j < bt_close.Height; j++)
17.         {
18.             Color color1 = originalBt.GetPixel(i, j);
19.             Color color2 = bt_close.GetPixel(i, j);
20.             int diffR = Math.Abs(color2.R - color1.R);
21.             int diffG = Math.Abs(color2.G - color1.G);
22.             int diffB = Math.Abs(color2.B - color1.B);
23.             bt1.SetPixel(i, j, Color.FromArgb(diffR, diffG, diffB));
24.         }
25.         pictureBox_show.Refresh();//刷新图片框
26.         pictureBox_show.Image = bt1;
27.     }
28. }
```

➢ 实现效果：



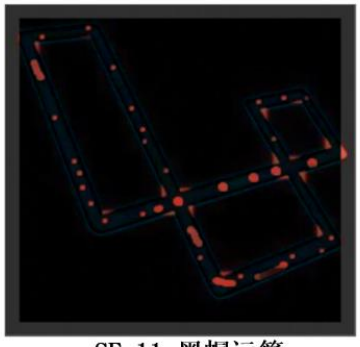
原图

SE=5 黑帽运算



SE=7 黑帽运算

SE=9 黑帽运算 效果最好



SE=11 黑帽运算

五、实验结果及分析(包括心得体会，本部分为重点，不能抄袭复制)

➢ 完成情况:

完成了实验**全部的基本要求**和**全部的扩展要求**，最终的结果基本达到了我的预期

➢ 实验心得

在这次实验中，我对图像进行了形态学变换，包括**腐蚀、膨胀、开运算、闭运算、边缘提取、顶帽运算和黑帽运算**。通过完成这些操作，我对图像处理中的形态学变换有了更深入的理解，并且学会了如何将这些操作应用于不同类型的图像，包括**二值图像、灰度图像和彩色图像**。

在过程中，我先从简单的二值图像开始，实现了基本的形态学变换操作。让我熟悉了形态学变换的算法原理和实现过程。通过腐蚀和膨胀操作，我了解了如何通过改变**结构元素的大小和形状**来实现图像的收缩和膨胀效果。开运算和闭运算则展示了如何组合腐蚀和膨胀操作来平滑图像、填充空洞或者去除噪声。

随后，我将注意力转移到**彩色图像和灰度图像**上，探索如何将形态学变换应用于不同通道的颜色值或灰度值。通过对彩色图像的腐蚀、膨胀、开运算和闭运算操作，我了解了如何在保留颜色信息的同时，对图像进行形态学变换。边缘提取操作则让我能够突出显示图像中的边缘部分，对于边缘检测和特征提取非常有用。顶帽运算和黑帽运算则展示了如何通过形态学操作突出图像中的亮区域或暗区域。

通过这次实验，我深入理解了形态学变换在图像处理中的作用和应用。我学会了如何将形态学变换操作从简单的二值图像扩展到彩色图像和灰度图像上，以应对更复杂的图像处理任务。这为我进一步探索和应用形态学变换提供了很好的基础，并让我更好地理解了图像处理领域的相关概念和技术。

总的来说，**这次实验难度比较大，关于图像的运算我查阅了很多资料也踩了很多坑，处理图像时的先后顺序不对也会导致图像效果千差万别，我必须在熟悉原理的情况下，恰当地调整参数大小，才能得到符合期望的图像。**但是这次实验也让我在形态学变换方面获得了实际的经验，并且通过逐步扩展到不同类型的图像，我对图像处理的应用和方法有了更深入的认识。这对于我的学习和进一步研究图像处理技术将非常有帮助。