浙江大学实验报告

专业：计算机科学与技术
姓名：NoughtQ
学号：1145141919810
日期：2024 年 10 月 22 日

课程名称：图像信息处理　指导老师：宋明黎　成绩：_____

实验名称：二值图像与形态学运算

一、 实验目的和要求

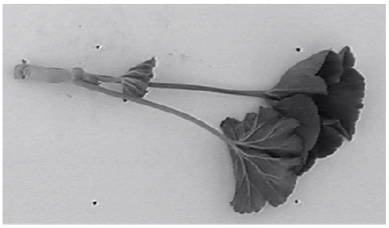
1. 掌握并实现图像二值化的算法——大津算法，以及它的一些变体。
2. 学习并实现各类形态学操作，包括膨胀、腐蚀、开操作和闭操作。

二、 实验内容和原理

2.1 二值图像

二值图像(binary image)：像素值只有 0 和 1 两种值，因此每个像素仅需 1bit 空间。但是在实际编程中，我们通常用 0 和 255 两个值来分别表示 0 和 1。

Binary image



Grayscale image

- 优点：
  - 更小的内存需求
  - 运行速度更快
  - 为二值图像开发的算法往往可以用于灰度级图像
  - 更便宜
- 缺点：
  - 应用范围有限；
  - 无法推广到三维空间中
  - 表现力欠缺，不能表现图像内部细节
  - 无法控制对比度(contrast)

2.2 图像二值化

通过对灰度图像阈值化(thresholding) 操作，将图像的像素值进行重置，从而实现图像二值化，基本思想为：

- 将二值图像视为两部分，一部分对应**前景**（foreground），另一部分对应**背景**（background）
- 尝试找到一个合适的阈值，使得前景和背景的内部方差(variance)最小化，而让它们之间的方差最大化

用到的算法是大津算法(Otsu’s method)，它的公式为：

$$
\begin{aligned}
\sigma^2_{\text{within}}(T) &= \frac{N_{\text{Fgrd}}(T)}{N}\sigma^2_{\text{Fgrd}}(T) + \frac{N_{\text{Bgrd}}(T)}{N}\sigma^2_{\text{Bgrd}}(T) \\
\sigma^2_{\text{between}}(T) &= \sigma^2 - \sigma^2_{\text{within}}(T) \\
&= \left(\frac{1}{N}\sum_{x,y}(f^2[x,y]-\mu^2)\right) - \frac{N_{\text{Fgrd}}}{N}\left(\frac{1}{N_{\text{Fgrd}}}\sum_{x,y\in\text{ Fgrd}}\left(f^2[x,y]-\mu^2_{\text{Fgrd}}\right)\right) \\
&\quad -\frac{N_{\text{Bgrd}}}{N}\left(\frac{1}{N_{\text{Bgrd}}}\sum_{x,y\in\text{ Bgrd}}\left(f^2[x,y]-\mu^2_{\text{Bgrd}}\right)\right) \\
&= -\mu^2 + \frac{N_{\text{Fgrd}}}{N},\mu^2_{\text{Fgrd}} + \frac{N_{\text{Bgrd}}}{N}\mu^2_{\text{Bgrd}} \\
&= \frac{N_{\text{Fgrd}}}{N}\left(\mu_{\text{Fgrd}}-\mu\right)^2 + \frac{N_{\text{Bgrd}}}{N}\left(\mu_{\text{Bgrd}}-\mu\right)^2 \\
&= \frac{N_{\text{Fgrd}}(T)\cdot N_{\text{Bgrd}}(T)}{N^2}\left(\mu_{\text{Fgrd}}(T)-\mu_{\text{Bgrd}}(T)\right)^2
\end{aligned}
$$

计算内部方差的公式过于复杂，因此有下面的化简方法：

$$
\begin{aligned}
W_f = \frac{N_{\text{Fgrd}}}{N}, W_b &= \frac{N_{\text{Bgrd}}}{N} \\
\mu &= W_f\cdot\mu_{\text{Fgrd}} + W_b\cdot\mu_{\text{Bgrd}} \\
\sigma_{\text{between}} &= W_f\left(\mu_{\text{Fgrd}}-\mu\right)^2 + W_b\left(\mu_{\text{Bgrd}}-\mu\right)^2 \\
&= W_f\left(\mu_{\text{Fgrd}} - W_f\cdot\mu_{\text{Fgrd}} - W_b\cdot\mu_{\text{Bgrd}}\right)^2 + W_b\left(\mu_{\text{Bgrd}} - W_f\cdot\mu_{\text{Fgrd}} - W_b\cdot\mu_{\text{Bgrd}}\right)^2 \\
&= W_bW_f\left(\mu_f-\mu_b\right)^2
\end{aligned}
$$

具体步骤为：

1. 确定原始图像中像素的最大值和最小值
2. 令阈值 = 像素最小值 + 1，对原始图像进行二值化操作
3. 确定前景和背景，分别计算当前阈值下的内部协方差和外部协方差，然后阈值++
4. 重复 2、3 两步，直到阈值达到像素最大值
5. 找到最大外部协方差和最小内部协方差对应的阈值

阈值策略的缺陷：全局二值化操作不给力

解决方法——**局部自适应操作** (local adaptive operation)：

- 设定一个局部窗口，在整个图像上滑动该窗口
- 对于每一窗口位置，确定针对该窗口的 threshold

2.3 形态学操作

形态学图像处理的数学基础和所用语言是**集合论** (set theory)，它的功能是简化图像数据，保持它们基本的形状特性，并除去不相干的结构。它包含以下基本运算：膨胀(dilation)、腐蚀(erosion)、开操作(opening)和闭操作(closing)。

2.3.1 膨胀

膨胀 (dilation)的作用：通过扩大前景来填充二值图像中的空洞。

物理意义：膨胀是将与物体“接触”的所有背景点合并到该物体中，使边界向外部扩张的过程，可以用来填补物体中的空洞（其中“接触”的含义由结构元描述）。

令集合A为二值图像，B为二值模板（称为**结构元** (structure element)，可以类比为电路中的滤波器），计算公式为：

$$A \oplus B = \{z \mid (B)_z \cap A \neq \emptyset\}$$

上式表示平移后的B与A的交集不为空。

2.3.2 腐蚀

腐蚀 (erosion)的作用：它是一种消除边界点，使边界向内部收缩的过程。可以用来消除小且无意义的物体。

$$A \ominus B = \{(x, y) \mid (B)_{\text{xy}} \subseteq A\}$$

- 只有当扫描区域的像素情况与结构元一致时，输出图像上的对应像素值为1，否则为0
- 与膨胀一样，某些像素点无法被结构元扫到，那么这些像素点在输出图像上的值就设为 0

2.3.3 开操作

开操作 (opening)：先腐蚀，再膨胀，公式为：

$$A \circ B = (A \ominus B) \oplus B$$

作用：从局部上看，它能够消除小而无意义的点；从全局上看，它使边界更加平滑，并保留了原图的特征。

2.3.4 闭操作

闭操作 (closing)：先膨胀，再腐蚀，公式为：

$$A \cdot B = (A \oplus B) \ominus B$$

作用：填充小的空洞，连接相邻物体，在保留原图特征的同时使边界更平滑。

三、 实验步骤与分析

3.1 图像二值化

3.1.1 全局大津算法

我们先对整张图片使用大津算法确定阈值。虽然前面已经简要介绍过计算步骤，但这里根据 C++代码再次说明一下：

1. 先将图片转化成灰度图（lab1 已实现），遍历灰度图中的每个像素点，找到最小和最大的像素值（记为 `minPixel` 和 `maxPixel`）
2. 令阈值 `t = minPixel + 1`，根据阈值确定前景和背景（这里规定比阈值小的为前景，设为黑色；比阈值大的为背景，设为白色）
3. 根据大津算法计算当前阈值下的方差 `var`，如果比最大协方差 `maxVar` 大则更新 `maxVar` 和对应阈值 `threshold`
4. 若 `t ⩽ maxPixel`，重复 2、3 两步
5. 遍历所有像素，若比 `threshold` 小，设为 `0`（黑色），否则设为 `255`（白色）

代码如下所示：

```
void OtsuAlgo(BMPFILE bf, int x1, int y1, int x2, int y2) {
    int x, y, t, pixel;
    int minPixel = 255, maxPixel = 0;        // 最小、最大像素值
    int threshold;                           // 利用大津算法确定的阈值
    double nF, nB, n;                        // 属于前景、背景的像素数,
以及总的像素数
    double expF, expB, var, maxVar;          // 前景、背景像素值的均值、
方差、最大方差

    // 遍历所有像素点，确定最小、最大像素值
    for (y = y1; y < y2; y++)
        for (x = x1; x < x2; x++) {
            pixel = bf→aBitmapBits[y * GrayBitWidth + x];
            minPixel = pixel < minPixel ? pixel : minPixel;
            maxPixel = pixel > maxPixel ? pixel : maxPixel;
        }

    // 根据简化后的大津算法，计算最大外部协方差
    maxVar = 0;
    for (t = minPixel + 1; t ⩽ maxPixel; t++) {
        nF = nB = n = 0;
        expF = expB = 0;
        for (y = y1; y < y2; y++)
```

```
            for (x = x1; x < x2; x++) {
                pixel = bf->aBitmapBits[y * GrayBitWidth + x];
                if (pixel < t) {
                    nF++;
                    expF += pixel;
                } else {
                    nB++;
                    expB += pixel;
                }
                n++;
            }
        expF /= nF;
        expB /= nB;
        var = (nF / n) * (nB / n) * pow((expF - expB), 2);
        if (var > maxVar) {
            maxVar = var;
            threshold = t;
        }
    }

    // 根据阈值改变给定区域下所有的像素值
    for (y = y1; y < y2; y++)
        for (x = x1; x < x2; x++) {
            pixel = bf->aBitmapBits[y * GrayBitWidth + x];
          bf->aBitmapBits[y * GrayBitWidth + x] = pixel < threshold ?
0 : 255;
        }

}

// 调用
OtsuAlgo(bf, 0, 0, bf->bmih.biWidth, bf->bmih.biHeight);
```

3.1.2 分块大津算法

全局大津算法的缺点是仅关注整体，而没有关注局部的像素点，导致它在处理细节上效果不是很好。一种可能的解决方案是将整张图片划分为多个小块，对每个小块进行一次大津算法的计算，获得该小块的阈值并修改该小块下的像素值。代码如下所示：

```
blockWsize = bf->bmih.biWidth / BLOCKWIDTHNUM;
blockHsize = bf->bmih.biHeight / BLOCKWIDTHNUM;

for (y = 0; y ≤ BLOCKWIDTHNUM; y++)
```

```
    for (x = 0; x ⩽ BLOCKWIDTHNUM; x++) {
        if (x * blockWsize == bf→bmih.biWidth || y * blockHsize ==
bf→bmih.biHeight)
            break;
        OtsuAlgo(bf, newImg, x * blockWsize, y * blockHsize,
                 min((x + 1) * blockWsize, bf→bmih.biWidth),
                  min((y + 1) * blockHsize, bf→bmih.biHeight),
-1, -1);
    }
```

3.1.3 滑动窗口大津算法

分块大津算法虽然克服了全局大津算法的一些局限性，但同时也产生了新的问题：图片的割裂感比较明显，仿佛被切成了多个小块。因此我们采取另一种改进措施“滑动窗口大津算法”：对于图像中的每一个像素点，先根据它周围的像素（即滑动窗口）使用大津算法来确定阈值，然后根据这个阈值来改变这个点的像素值。代码如下所示：

```
newImg = (BMPFILE)malloc(sizeof(struct tagBMPFILESTRUCT));
memcpy(&(newImg→bmfh), &(bf→bmfh), sizeof(BITMAPFILEHEADER));
memcpy(&(newImg→bmih), &(bf→bmih), sizeof(BITMAPINFOHEADER));
memcpy(&(newImg→aColors),  &(bf→aColors),  sizeof(RGBQUAD)  *
COLORNUM);
newImg→aBitmapBits = (BYTE *)calloc(newImg→bmih.biSizeImage,
sizeof(BYTE));

// 遍历每个像素点
for (y = 0; y < bf→bmih.biHeight; y++)
    for (x = 0; x < bf→bmih.biWidth; x++) {
        // 确定滑动窗口的区域
        lx = max(x - WINDOWSIZE, 0);
        rx = min(x + WINDOWSIZE, bf→bmih.biWidth);
        ty = max(y - WINDOWSIZE, 0);
        by = min(y + WINDOWSIZE, bf→bmih.biHeight);
        // printf("%d %d %d %d\n", lx, ty, rx, by);
        OtsuAlgo(bf, newImg, lx, ty, rx, by, x, y);
    }

return newImg;
```

最后，我将大津算法三种不同的实现方法放在同一个函数 `GenerateBinary()`内。在运行主程序的时候，该函数会先调用询问函数 `AskforBinaryChoice()`，用户可根据需求选择不同的方法。代码如下所示：

```
// 询问要使用何种图像二值化方法
int AskforBinaryChoice(void) {
    int choice;

    printf("Here are the choices of image binarization.\n");
    printf("1) Global Binarization\n");
    printf("2) Local Adaptive Binarization(multiple parts)\n");
    printf("3) Local Adaptive Binarization(sliding window)\n");
    printf("Please input your choice(1, 2 or 3): ");
    scanf("%d", &choice);
    if (choice < 1 || choice > 3) {
        printf("Invalid choice. Try Again!\n");
        exit(1);
    } else {
       printf("Valid choice!\nPlease wait a minute for the generation
of the binary image...\n");
    }

    return choice;
}


// 生成二值图像
BMPFILE GenerateBinary(BMPFILE bf, int * choice) {
    BMPFILE newImg;                         // 新图像（用于滑动窗口）

    int x, y;
    int blockWsize, blockHsize;             // 每个块在宽度上、高度上的个数
    int lx, rx, ty, by;                     // 确定滑动窗口左上和右下角的坐标

    *choice = AskforBinaryChoice();         // 先询问选择哪种大津算法

    switch (*choice) {
        case 1:       // 全局
                  OtsuAlgo(bf, newImg, 0, 0, bf->bmih.biWidth, bf-
>bmih.biHeight, -1, -1);
            break;
        case 2:       // 分块
            blockWsize = bf->bmih.biWidth / BLOCKWIDTHNUM;
            blockHsize = bf->bmih.biHeight / BLOCKWIDTHNUM;
            // 遍历每个图像的每个块，对每个块使用大津算法
            for (y = 0; y <= BLOCKWIDTHNUM; y++)
```

```
                for (x = 0; x ⩽ BLOCKWIDTHNUM; x++) {
                    if (x * blockWsize == bf->bmih.biWidth || y *
blockHsize == bf->bmih.biHeight)
                        break;
                    OtsuAlgo(bf, newImg, x * blockWsize, y * blockHsize,
                                         min((x + 1) * blockWsize,
bf->bmih.biWidth),
                                     min((y + 1) * blockHsize, bf-
>bmih.biHeight), -1, -1);
                }
            break;
        case 3:       // 滑动窗口
            // 将灰度图的除图像数据外的内容全部拷贝给 newImg（这里很容易弄错,
我自己在这里 debug 了半天...）
            newImg = (BMPFILE)malloc(sizeof(struct tagBMPFILESTRUCT));
                            memcpy(&(newImg->bmfh),  &(bf->bmfh),
sizeof(BITMAPFILEHEADER));
                            memcpy(&(newImg->bmih),  &(bf->bmih),
sizeof(BITMAPINFOHEADER));
           memcpy(&(newImg->aColors), &(bf->aColors), sizeof(RGBQUAD)
* COLORNUM);
                    newImg->aBitmapBits  =  (BYTE  *)calloc(newImg-
>bmih.biSizeImage, sizeof(BYTE));

            // 遍历每个像素点
            for (y = 0; y < bf->bmih.biHeight; y++)
                for (x = 0; x < bf->bmih.biWidth; x++) {
                    // 确定滑动窗口的区域
                    lx = max(x - WINDOWSIZE, 0);
                    rx = min(x + WINDOWSIZE, bf->bmih.biWidth);
                    ty = max(y - WINDOWSIZE, 0);
                    by = min(y + WINDOWSIZE, bf->bmih.biHeight);
                    // printf("%d %d %d %d\n", lx, ty, rx, by);
                    OtsuAlgo(bf, newImg, lx, ty, rx, by, x, y);
                }
            return newImg;
        default:     // 不存在的选择
            printf("Invalid Option! Try Again!\n");
            exit(1);
    }
```

```
    return bf;
}
```

3.2 形态学操作

在这里，我借鉴了挑选图像二值化的方法：用一个总起的函数 MorphologyOp()处理各种形态学操作，它会先向用户询问选择何种形态学操作，然后根据选择调用对应的函数，最后返回处理好的图像。代码如下所示：

```
// 询问要进行何种形态学操作（膨胀/腐蚀/开操作/闭操作）
int AskforChoicev2(void) {
    int choice;

    printf("Here are the choices of morphology operations.\n");
    printf("4) Dilation\n");
    printf("5) Erosion\n");
    printf("6) Opening\n");
    printf("7) Closing\n");
    printf("Other numbers can cancel the morphology operations.\n");
    printf("Please input your choice(4, 5, 6 or 7): ");
    scanf("%d", &choice);
    if (choice < 4 || choice > 7) {
        printf("Morphology operations canceled!\n");
        exit(1);
    } else {
       printf("Valid choice!\nPlease wait a minute for the generation
of the binary image...\n");
    }

    return choice;
}

// 形态学操作
BMPFILE MorphologyOp(BMPFILE bf, int choice) {
    BMPFILE newImg;
    newImg = (BMPFILE)malloc(sizeof(struct tagBMPFILESTRUCT));

    generateStructElem(SESIZE);       // 生成结构元（方块形状）

    switch(choice) {                  // 选择不同的形态学操作
        case 4:
            printf("Your choice is: Dilation.\n");
            newImg = Dilation(bf);
```

```
            break;
        case 5:
            printf("Your choice is: Erosion.\n");
            newImg = Erosion(bf);
            break;
        case 6:
            printf("Your choice is: Opening.\n");
            newImg = Opening(bf);
            break;
        case 7:
            printf("Your choice is: Closing.\n");
            newImg = Closing(bf);
            break;
        default:
            printf("Invalid choice!\n");
            exit(1);
    }

    return newImg;
}

// 调用
choice = AskforChoicev2();
MphImg = MorphologyOp(binaryImg, choice);
```

3.2.1 膨胀

```
// 膨胀
BMPFILE Dilation(BMPFILE bf) {
    int x, y, i;
    int scan_x, scan_y;

    BYTE * oldImgData;                    // 用于存储原图图像数据
    LONG width = bf->bmih.biWidth;        // 图像宽度
    LONG height = bf->bmih.biHeight;      // 图像高度
    oldImgData = (BYTE *)malloc(sizeof(BYTE) * width * height);

    // 保存原图图像数据
    for (y = 0; y < height; y++)
        for (x = 0; x < width; x++) {
            oldImgData[y * GrayBitWidth + x] = bf->aBitmapBits[y *
GrayBitWidth + x];
        }
```

```
    // 遍历每个像素点
    for (y = 0; y < height; y++)
        for (x = 0; x < width; x++) {
            for (i = 0; i < STURCTELEMNUM; i++) {
                scan_x = x + StructElemX[i];
                scan_y = y + StructElemY[i];
                // 如果结构元与该像素周围的重叠区域中存在黑点，则该像素也设为黑色，从而实现膨胀
                if (scan_x >= 0 && scan_x < width && scan_y >= 0 && scan_y < height
                  && !oldImgData[scan_y * GrayBitWidth + scan_x]) {
                    bf->aBitmapBits[y * GrayBitWidth + x] = 0;
                    break;
                }
            }
        }

    return bf;
}
```

3.2.2 腐蚀

```
// 腐蚀
BMPFILE Erosion(BMPFILE bf) {
    // 实现上与膨胀类似
    int x, y, i;
    int scan_x, scan_y;

    BYTE * oldImgData;
    LONG width = bf->bmih.biWidth;
    LONG height = bf->bmih.biHeight;
    oldImgData = (BYTE *)malloc(sizeof(BYTE) * width * height);

    for (y = 0; y < height; y++)
        for (x = 0; x < width; x++) {
            oldImgData[y * GrayBitWidth + x] = bf->aBitmapBits[y * GrayBitWidth + x];
        }

    for (y = 0; y < height; y++)
        for (x = 0; x < width; x++) {
            for (i = 0; i < STURCTELEMNUM; i++) {
```

```
                scan_x = x + StructElemX[i];
                scan_y = y + StructElemY[i];
                // 如果结构元与该像素周围的重叠区域中存在白点，则该像素也设为白色，从而实现腐蚀
                if (scan_x ⩾ 0 && scan_x < width && scan_y ⩾ 0 && scan_y < height
                   && oldImgData[scan_y * GrayBitWidth + scan_x] == 255) {
                    bf→aBitmapBits[y * GrayBitWidth + x] = 255;
                    break;
                }
            }
        }

    return bf;
}
```

3.2.3 开操作

```
// 开操作
BMPFILE Opening(BMPFILE bf) {
    // 先腐蚀再膨胀
    Erosion(bf);
    Dilation(bf);

    return bf;
}
```

3.2.4 闭操作

```
// 闭操作
BMPFILE Closing(BMPFILE bf) {
    // 先膨胀再腐蚀
    Dilation(bf);
    Erosion(bf);

    return bf;
}
```

3.3 主程序

```
#include "bmp.h"
#include <stdio.h>
#include <stdlib.h>
```

```c
int main() {
    int bi_choice, choice;
    BMPFILE originImg, grayImg, binaryImg, MphImg;

    // 分配存储空间
    originImg = (BMPFILE)malloc(sizeof(struct tagBMPFILESTRUCT));
    grayImg = (BMPFILE)malloc(sizeof(struct tagBMPFILESTRUCT));
    binaryImg = (BMPFILE)malloc(sizeof(struct tagBMPFILESTRUCT));
    MphImg = (BMPFILE)malloc(sizeof(struct tagBMPFILESTRUCT));

    // 读取原图
    originImg = ReadBMPFile();

    // 转化为灰度图 & 输出
    grayImg = GenerateGrayScale(originImg);
    WriteBMPFile(grayImg, 1);

    // 转化为二值图像 & 输出
    binaryImg = GenerateBinary(grayImg, &bi_choice);
    WriteBMPFile(binaryImg, 3 * 10 + bi_choice);

    // 形态学操作 & 输出
    choice = AskforChoicev2();
    MphImg = MorphologyOp(binaryImg, choice);
    WriteBMPFile(MphImg, choice);

    return 0;
}
```

四、 实验环境及运行方法

4.1 实验环境

我的开发环境如下:

- WSL2 + Ubuntu 24.04 LTS
- gcc version 13.2.0 (Ubuntu 13.2.0-23ubuntu4)
- GNU Make 4.3

4.2 运行方法

若要运行主程序，需要遵循以下步骤:

> **注意**
>
> 在执行以下命令前，请检查一下路径下 `./代码/test` 是否存在`.bmp` 图片，并且检查一下`./代码/scripts/bmp.h` 文件的宏定义 `BMPFILEPATH` 是否指的是该图片。若有问题请自行修改，否则程序无法正常运行。

1. 将目录切换至 `./代码/build`
2. 执行以下命令:

```
# 编译代码
$ make

# 运行可执行文件
$ ./lab2
Successfully open the file!
Size: 1548022(bit)
ColorBitWidth: 2144
Width: 714
Height: 722
Image Size: 1547968
Finish the conversion successfully!
Here are the choices of image binarization.
1) Global Binarization
2) Local Adaptive Binarization(multiple parts)
3) Local Adaptive Binarization(sliding window)
Please input your choice(1, 2 or 3):
```

现在程序询问使用何种方式生成二值图像，可以选择的方法有：全局大津算法（输入 1），分块大津算法（输入 2），以及滑动窗口大津算法（输入 3）。下面以输入 1 为例:

```
Please input your choice(1 or 2): 1
Valid choice!
Please wait a minute for the generation of the binary image...
Finish the conversion successfully!
Here are the choices of morphology operations.
4) Dilation
5) Erosion
6) Opening
7) Closing
Other numbers can cancel the morphology operations.
Please input your choice(4, 5, 6 or 7):
```

现在程序询问使用何种形态学操作，可以选择的操作有：膨胀（输入 4），腐蚀（输入 5），开操作（输入 6），闭操作（输入 7），以及啥也不做（输入其他整数，或者直接按 Ctrl+C 结束）。下面以输入 4 为例：

```
Please input your choice(4, 5, 6 or 7): 4

Valid choice!
Please wait a minute for the generation of the binary image...
Your choice is: Dilation.
Finish the conversion successfully!
```

3. 来到 ./代码/tests 目录，此时可以看到得到的新图（灰度图，以及膨胀后的图像）

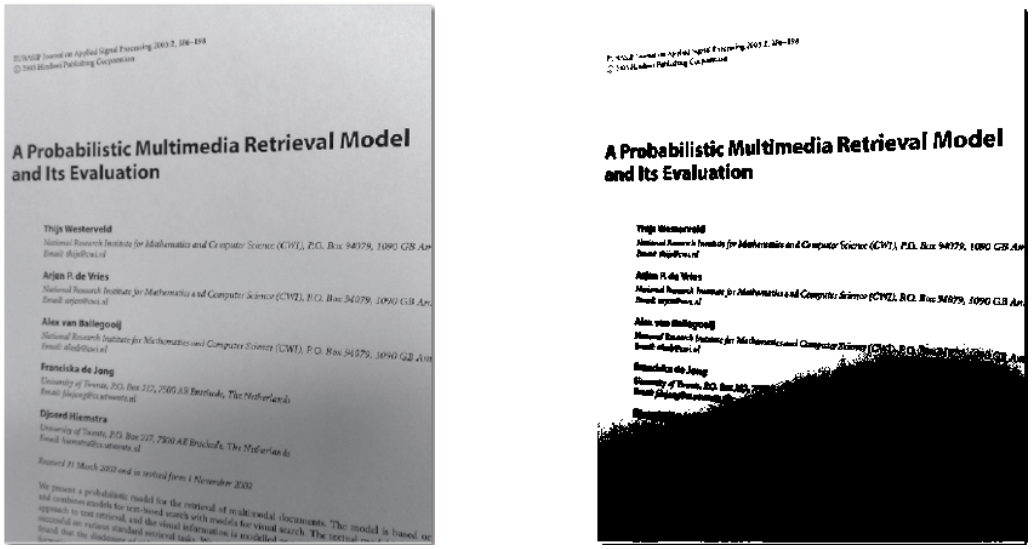
五、 实验结果展示

5.1 图像二值化

先来看一下全局大津算法下的二值图像：

（从左到右依次为原图、灰度图和二值图像）

可以看到，在这种情况下，全局大津算法的效果还是比较令人满意的。但是在下面的情况下，全局大津算法可能就不是那么理想了：



（从左到右依次为原图和二值图像）

由于原图的底部区域有一块阴影，虽然人眼并不会因此受到很大的干扰，但是程序会认为这些阴影部分的像素值偏低，因而会被转化为黑色，导致阴影下的文字被遮住。于是，我们先用分块大津算法尝试解决这一问题：

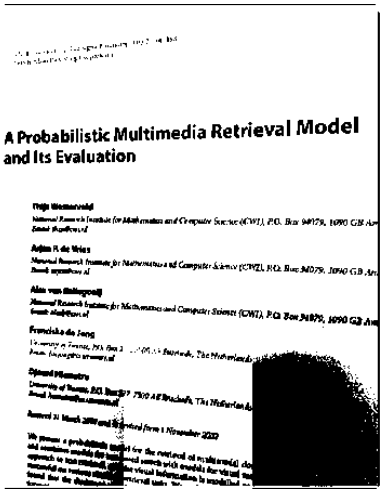


Figure 2: 3×3 分块

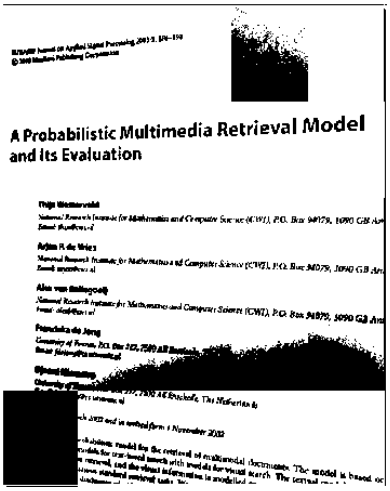


Figure 3: 5×5 分块

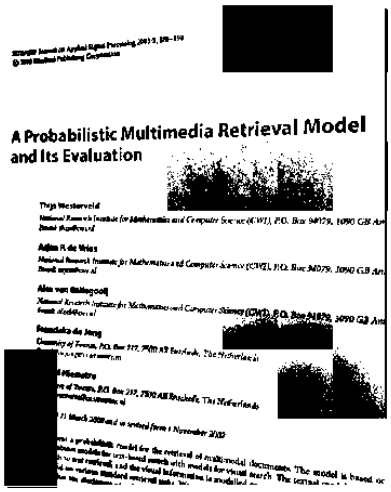


Figure 4: 7×7 分块

可以看到，分块局部算法能够适当消除一部分阴影的影响，且分块越多，阴影区域的面积更少。但原本空白的区域多了好几块黑色小块，个人推测是因为在块内，深色区域少而浅色区域多，导致最大协方差对应的阈值大，因而很多浅色的区域也会被转化为黑色。

接下来看另一种方案——滑动窗口大津算法的实现效果：

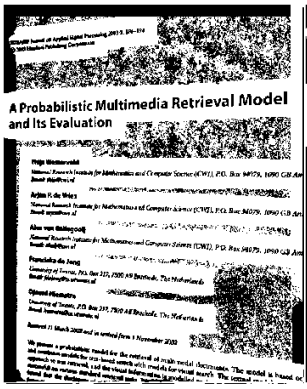


Figure 5: 窗口大小 30×30

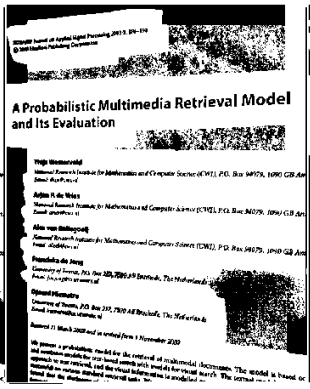


Figure 6: 窗口大小 50×50

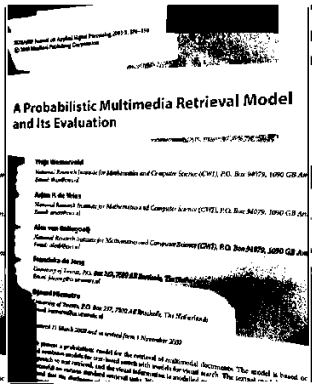


Figure 7: 窗口大小 70×70

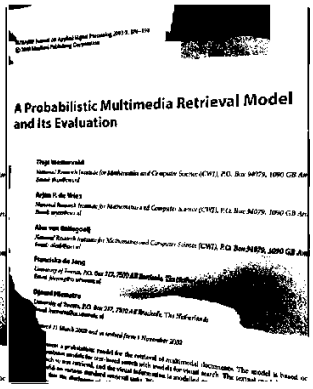


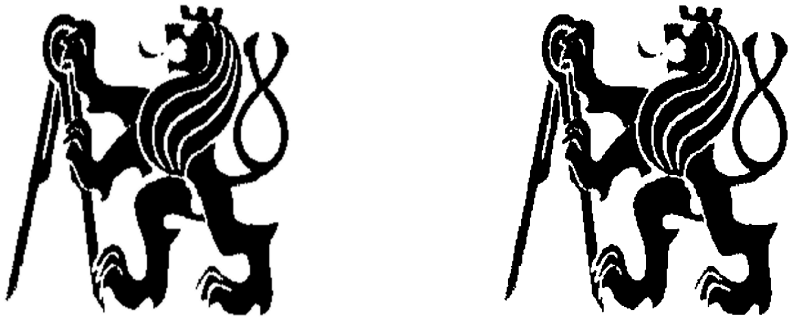
Figure 8: 窗口大小 90×90

可以看到，滑动窗口大津算法也能在一定程度上消除阴影的影响，且随着窗口的增大，多生成的黑色区域会减少，但是那些黑色区域的会更加密集（看起来深度更深），但从整体效果上而言稍微优于分块大津算法。

我对开头部分的莱娜图也进行了分块和滑动窗口下的大津算法，但是从效果上而言不如全局大津算法，故不将图片放入报告内（可以到`./代码/tests/lena`目录下看）。

5.2 形态学操作

直接来看效果：



（从左到右依次为原图和二值图像，~~几乎~~没有什么区别）

（结构元为 3×3 的方块，从左到右依次为膨胀、腐蚀、开操作、闭操作后的图像）

（结构元为 5×5 的方块，从左到右依次为膨胀、腐蚀、开操作、闭操作后的图像）

（结构元为 7×7 的方块，从左到右依次为膨胀、腐蚀、开操作、闭操作后的图像）

从这些结果中，可以确定形态学操作函数的设计是成功的。

六、 心得体会

本实验的算法难度适中，稍微涉及一些统计学和集合论的数学知识，代码实现上还算比较容易的。形态学操作的实现相对比较顺利，但是大津算法的设计上遇到了不少的磕磕绊绊，尤其是在滑动窗口大津算法中，我遇到了以下问题：

- 最开始的时候，我并不是遍历每一个像素，而是根据窗口的大小遍历一小块一小块的区域，这实质上就是划分更细的分块大津算法。在前面我也提到过，分块大津算法在莱娜图上的效果更糟，现在划分太细效果还要差——感觉图像是由大颗粒的像素点构成的：

Figure 23: 失败作品 1

- 而且，我原先的滑动窗口是根据当前正在处理的灰度图上（一部分已经二值化了）确定阈值，而不是根据原图确定阈值，这样生成的二值图像割裂感更强（失败作品 2 和失败作品 3 具体怎么生成的有些遗忘，但可以确定的是它们的出现都和这个原因有关）：

Figure 24: 失败作品 2

Figure 25: 失败作品 3

- 最后，在滑动窗口大津算法中，我打算将新生成的图像放在 `newImg` 变量中，所以首先要将灰度图中除了图像数据外的所有字段拷贝到 `newImg` 上。但由于我的粗心大意，

在复制调色盘的时候没有处理好，导致图像漆黑一片（调色盘的数据全是 0）（但是它的图像数据是正确的，可以用 010 Editor 之类的编辑器查看）：

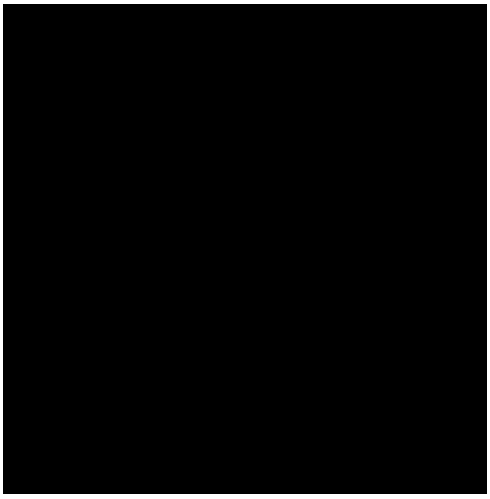


Figure 26: 失败作品 4

虽然我听说 Lab1 之后的实验相对比较轻松，但这次实验中我遇到的麻烦可不比首次实验中遇到的少。希望在之后的实验中，我能够吸取前面几次实验的教训，争取少犯一些不该犯的错误。除此之外，这次实验让我更深入地学习了通过大津算法实现的图像二值化，以及各种形态学操作，使我受益匪浅。