专业：计算机科学与技术
姓名：秦嘉俊
学号：3210106182
日期：2022 年 10 月 23 日

浙江大学 实验报告

课程名称：图像信息处理 指导老师：宋明黎 成绩_____
实验名称：bmp 灰度图像二值化并对其进行形态学操作

一、 实验目的和要求

1. 熟悉二值图像，了解二值图像的用途和结构。
2. 掌握构建二值图像的算法：大津算法，并用代码实现算法以实际操作图像。
3. 基于二值化图像，实现形态学操作。包括图形腐蚀操作，膨胀操作，开操作和闭操作。

二、 实验内容和原理

内容

1. 图像二值化操作
2. 图像腐蚀操作
3. 图像膨胀操作
4. 图像开操作
5. 图像闭操作

原理

2.1 二值图像

二值图像（Binary Image）中像素（Pixel）的值只有 [0,1] 或者 [0,255]，编程中一般用 [0,255] 来构造二值图像。

二值图像的**优点**有：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。而**缺点**有：应用范围毕竟有限；更无法推广到三维空间中；表现力欠缺，不能表现物体内部细节；无法控制对比度。

2.2 大津算法

把一个图像转为二值化图像，我们的基本想法是：设置一个阈值 Threshold，比阈值小的置为 0, 比阈值大的就置为 255.

$$\begin{cases} I(x,y) = 0 \ if \ I(x,y) \leq \textbf{Threshold} \\ I(x,y) = 255 \ if \ I(x,y) \geq \textbf{Threshold} \end{cases}$$

但如何选取合适的 threshold, 这就是我们大津算法的核心部分，其具体分析如下：

1. 如何选取合适的 threshold 基本思想：将二值化得到的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 threshold 使得到的前景和背景的内部方差最小，而它们之间的方差则最大。

2. 推导过程

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

我们可以发现，最大外部方差和最小内部方差是等价的，所以我们找到其中一个即可。

同时我们可以将上述推导简化：

$$
\begin{aligned}
W_f = \frac{N_{Fgrd}}{N}, W_b = \frac{N_{Bgrd}}{N}, W_f + W_b = 1 \\
\mu = W_f\ times \mu_{Fgrd} + W_b\ times \mu_{Bgrd} \\
\sigma_{between} = W_f(\mu_{Fgrd} - \mu)^2 + W_b(\mu_{Bgrd} - \mu)^2 \\
= W_f(\mu_{Fgrd} - W_f \times \mu_{Fgrd} - W_b \times \mu_{Bgrd})^2 + W_b(\mu_{Bgrd} - W_f \times \mu_{Fgrd} - W_b \times \mu_{Bgrd})^2 \\
\rightarrow W_b W_f(\mu_f - \mu_b)^2
\end{aligned}
$$

3. 具体实现过程：

(a) 确定原始图像中像素的最大值和最小值；

(b) 最小值加 1 作为 threshold 对原始图像进行二值化操作；

(c) 根据对应关系确定前景和背景，分别计算当前 threshold 下的内部协方差和外部协方差；

(d) 回到 Step 2 直到达到像素最大值；

(e) 找到最大外部和最小内部协方差对应的 threshold.

值得注意的是，单纯的大津算法是有局限的。如果我们对于一张图，直接全局进行二值化操作，那么对那些不同区域像素差别很大的图来说，可能不会有好的效果。如对有阴影的文字，就会产生很强的割裂。因此我们的解决方法是局部自适应操作，即设定一个局部窗口，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

用数学形态学（也称图像代数）表示以形态为基础对图像进行分析的数学工具。基本思想是用具有一定形态的结构元素去度量和提取图像中的对应形状以达到对图像分析和识别的目的。形态学图像处理的数学基础和所用语言是集合论。形态学图像处理的应用可以简化图像数据，保持它们基本的形状特性，并除去不相干的结构。形态学图像处理的基本运算有 4 个：膨胀、腐蚀、开操作和闭操作。

在下面的介绍中，我们的 A 指二值图像; B 指二值模板，称为**结构元（structure element）**

2.3.1 膨胀操作

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）

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

上式表示 B 进行平移与 A 的交集不为空

2.3.2 腐蚀操作

腐蚀是一种消除边界点，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。

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

2.3.3 开操作

开运算：先腐蚀，后膨胀。用来消除小物体、在纤细点处分离物体、平滑较大物体的边界的同时并不明显改变其面积。

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

2.3.4 闭操作

闭运算：先膨胀，后腐蚀。用来填充物体内细小空洞、连接邻近物体、平滑其边界的同时并不明显改变其面积。

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

三、 实验步骤与分析

3.1 图像二值化

首先对于一个图像，我们使用 lab1 中的方法将其读入，同时转为灰度图以方便后面操作。

随后我们运用大津算法来进行图像二值化，具体实现基本类似上面的过程。

```
1  /* 用于转灰度图后的 getposition */
2  int Get_Position2(int x, int y)
3  {
4     return x * new_row_byte + y;
5  }
6  void Otto(int row1, int col1, int row2,int col2)
7  {
8     int i, j, k;
9     double N, N_fgrd, N_bgrd; /* 分别对应总像素数，前景像素个数，后景像素个数。 这里我们使用 double 以提高后面计算的精度
          */
10    int MaxPix = 0, MinPix = 255;
11    double mu_fgrd, mu_bgrd;
12    int Threshold;
13    double sigma = 0.0;
14    /* 找最大和最小的像素值 */
15    for (i = row1; i < row2 && i < ImageHeight; i++)
16       for (j = col1; j < col2 && j < ImageWidth; j++)
17       {
18          int now = Get_Position2(i, j);
19          BYTE tmp = b.aBitmapBits[now];
20          if (tmp > MaxPix) MaxPix = tmp;
21          if (tmp < MinPix) MinPix = tmp;
22       }
23    /* 找 Threshold */
24    for (k = MinPix + 1; k <= MaxPix ; k ++)
25    {
26       N = N_bgrd = N_fgrd = 0.0;
27       mu_fgrd = mu_bgrd = 0.0;
28       for (i = row1; i < row2 && i < ImageHeight; i++)
29          for (j = col1; j < col2 && j < ImageWidth; j++)
30          {
31             int now = Get_Position2(i, j);
```

```
                N += 1.0;
                BYTE tmp = b.aBitmapBits[now];
                if (tmp >= k) N_fgrd += 1.0, mu_fgrd += tmp;
                else N_bgrd += 1.0, mu_bgrd += tmp;
            }
        mu_fgrd /= N_fgrd; /* 计算均值 */
        mu_bgrd /= N_bgrd;
        double w = (N_bgrd * N_fgrd) / N / N * (mu_bgrd - mu_fgrd) * (mu_bgrd - mu_fgrd);
        if (w > sigma)
        {
            sigma = w;
            Threshold = k;
        }
    }
    /* 对图像二值化 即大于等于阈值的赋为全 1(255), 否则赋为 0 */
    for (i = row1; i < row2 && i < ImageHeight; i++)
        for (j = col1; j < col2 && j < ImageWidth; j++)
        {
            int now = Get_Position2(i, j);
            BYTE tmp = b.aBitmapBits[now];

            if (tmp >= Threshold)
                b.aBitmapBits[now] = 255;
            else
                b.aBitmapBits[now] = 0;
        }
}
void Solve()
{
    int i, j;
    /* 大津 二值化图像 */
    int block_size_h = ImageHeight / 2;
    int block_size_w = ImageWidth / 2;
    for (i = 0; i < ImageHeight; i += block_size_h)
        for (j = 0; j < ImageWidth; j += block_size_w)
            Otto(i, j, i+block_size_h, j+block_size_w);
}
```

我们的大津算法 `Otto(row1, col1, row2, col2)` 对以 (row1, col1) 为左上角，(row2, col2) 为右下角的矩形进行二值化操作。实现过程如下：

1. 首先遍历这个矩阵的所有像素点，找到其中最大的像素值和最小的像素值 `MaxPix` 和 `MinPix`
2. 随后我们开始从 `MinPix+1` 开始依次枚举 Threshold. 循环中我们每次假定 k 当前枚举到的阈值，由此区分出前景和后景，并计算出均值 `mu_fgrd` `mu_bgrd` 和方差 `w`. 注意到这里我们在统计像素个数时，将 `N,N_fgrd,N_bgrd` 定义为浮点型变量，以便后面计算时有更高的精度。然后我们通过比对方差，找到方差最大时的阈值，并将其作为 Threshold.
3. 接着我们开始二值化，我们将大于 Threshold 的像素值赋为 255, 小于的赋为 0. 这样就实现了对这一部分的矩形的二值化。

注意的是我们这里使用了分块的做法，即将图像分为四部分，然后分别对其进行二值化。分块的好处是可以处理某些某些部分的像素值明显和其他部分的情况，但也会带来问题，详见后面的实验结果展示和心得

体会部分。

但我们发现简单粗暴的分块效果并不好，特别是当块分多之后，各个块之间有很强的割裂感，块边界看上去是不连续的。因此我们还实现了另一种方法，即滑动窗口下的局部大津法：

```
    void LocalOTSU(int row1, int col1, int row2, int col2, int x, int y)
{
    if(row1 < 0)row1 = 0;
    if(row2 < 0)row2 = 0;
    int i, j, k;
    double N, N_fgrd, N_bgrd; /* 分别对应总像素数，前景像素个数，后景像素个数。 这里我们使用 double 以提高后面计算的精度 */
    int MaxPix = 0, MinPix = 255;
    double mu_fgrd, mu_bgrd;
    int Threshold;
    double sigma = 0.0;
    /* 找最大和最小的像素值 */
    for (i = row1; i < row2 && i < ImageHeight; i++)
        for (j = col1; j < col2 && j < ImageWidth; j++)
        {
            int now = Get_Position2(i, j);
            BYTE tmp = b.aBitmapBits[now];
            if (tmp > MaxPix) MaxPix = tmp;
            if (tmp < MinPix) MinPix = tmp;
        }
    /* 找 Threshold */
    for (k = MinPix + 1; k <= MaxPix ; k ++)
    {
        N = N_bgrd = N_fgrd = 0.0;
        mu_fgrd = mu_bgrd = 0.0;
        for (i = row1; i < row2 && i < ImageHeight; i++)
            for (j = col1; j < col2 && j < ImageWidth; j++)
            {
                int now = Get_Position2(i, j);
                N += 1.0;
                BYTE tmp = b.aBitmapBits[now];
                if (tmp >= k) N_fgrd += 1.0, mu_fgrd += tmp;
                else N_bgrd += 1.0, mu_bgrd += tmp;
            }
        mu_fgrd /= N_fgrd; /* 计算均值 */
        mu_bgrd /= N_bgrd;
        double w = (N_bgrd * N_fgrd) / N / N * (mu_bgrd - mu_fgrd) * (mu_bgrd - mu_fgrd);
        if (w > sigma)
        {
            sigma = w;
            Threshold = k;
        }
    }
    /* 对图像二值化 即大于等于阈值的赋为全 1(255), 否则赋为 0 */
    int now = Get_Position2(x, y);
    BYTE tmp = b.aBitmapBits[now];
    if (tmp >= Threshold)
        saved.aBitmapBits[now] = 255;
    else
        saved.aBitmapBits[now] = 0;
}
void Solve()
```

```
{
    memcpy(&saved, &b, sizeof(saved));
    saved.aBitmapBits = (BYTE *)calloc(saved.bmih.biSizeImage, sizeof(BYTE));
    /* 滑动窗口版本的局部大津 */
    int window_size = 50;
    for (i = 0; i < ImageHeight; i ++)
        for (j = 0; j < ImageWidth; j ++)
              LocalOTSU(i-window_size/2, j-window_size/2, i+window_size/2, j+window_size/2, i, j);
    Print(&saved, "Local_OTSU.bmp");
}
```

我们依次枚举灰度图中的每个像素，对于像素点 (i, j) 我们确定一个以这个像素点为中心，边长为 window 的方形局部区域，在这个区域里进行局部大津算法，得到阈值后记录 (i, j) 的二值化情况，随后继续处理后面的像素。这样可以使各个区域之间的连接更加平滑。

3.2 膨胀

```
/* 检查(x,y)是否超出边界，而且这个这个点的像素之前是否为 1 */
int Check(int x,int y,BMPFILE *a)
{
    return x>=0 && y >= 0 && x <= ImageHeight && y <= ImageWidth && a->aBitmapBits[Get_Position2(x, y)] == 255;
}
/* a 表示原图， ans 用来存答案*/
void Dilation(BMPFILE *a, BMPFILE *ans)
{
    int i, j, k, flag;
    int dx[10] = {0, -1, 0, 0, 1, -1, -1, 1, 1};
    int dy[10] = {0, 0, 1, -1, 0, -1, 1, -1, 1};
    /* 枚举中心点 这里我们采用 3 * 3 的方形作为 structure element */
    for (i = 0; i < ImageHeight; i++)
        for (j = 0; j < ImageWidth; j++)
        {
            flag = 0; /* 表示会不会膨胀到(i,j) */
            for (k = 0; k < 9; k++)
            {
                int nx = i + dx[k];
                int ny = j + dy[k];
                if (Check(nx, ny, a)) {
                    flag = 1;
                    break;
                }
            }
            if (flag) ans->aBitmapBits[Get_Position2(i, j)] = 255;
            else ans->aBitmapBits[Get_Position2(i,j)] = 0;
        }
}
```

进行膨胀操作时，我们选用 3 * 3 的方形作为结构元去膨胀 A. 具体过程如下：

1. 首先我们枚举这个方形的中心点，中心点确定了结构元的位置就确定了。
2. 结构元的每个像素的位置我们预先成对地存入了 `dx[] dy[]` 中，对于结构元中第 i 个点，他的位置就是 `(x+dx[i], y+dy[i])`(`(x, y)`是结构元中心点的位置)。随后我们判断每个点是否在 A 范围

内，如果在那么这个点是否值为 1(Check 函数部分). 如果都满足说明我们结构元和图像的交集不为空，将 flag 赋值为 1 即可直接跳出判断的循环。

3. 最后对于每个中心点的位置，如果 flag=1 就说明这个点需要被膨胀为 1, 将其赋值为 255, 否则赋值为 0.

3.3 腐蚀

```
/* Check() 函数同上 */
void Erosion(BMPFILE *a, BMPFILE *ans)
{
   int i, j, k, flag;
   int dx[10] = {0, -1, 0, 0, 1, -1, -1, 1, 1};
   int dy[10] = {0, 0, 1, -1, 0, -1, 1, -1, 1};
   /* 枚举中心点 这里我们采用 3 * 3 的方形作为 structure element */
   for (i = 0; i < ImageHeight; i++)
      for (j = 0; j < ImageWidth; j++)
      {
         flag = 1; /* 表示要不要保留(i,j) */
         for (k = 0; k < 9; k++)
         {
            int nx = i + dx[k];
            int ny = j + dy[k];
            if (!Check(nx, ny, a)) {
               flag = 0;
               break;
            }
         }
         if (flag) ans->aBitmapBits[Get_Position2(i, j)] = 255;
         else ans->aBitmapBits[Get_Position2(i, j)] = 0;
      }
}
```

进行腐蚀操作时，我们依然选用 3 * 3 的方形作为结构元去腐蚀 A. 具体过程如下：

1. 首先我们枚举这个方形的中心点，中心点确定了结构元的位置就确定了。

2. 判断阶段与膨胀基本相同。不同点在于，如果存在结构元的点，A 在这个位置上的像素为 0, 那么说明这个目前结构元的中心点不能保留，因此我们直接将 flag 赋值为 0, 并退出循环。

   值得注意的是，对于边界的点，有两种处理方法，一种是将边界如第一排第一列的像素再复制一遍，以便判断，还有一种判断就是直接将边界外的点视为 0, 这样相当于将原图像中第一排、最后一排、第一列和最后一列的点全部变为 0 了。这里我们采取的是后者的解决方法，如果结构元的点超过了边界，我们直接返回 False.

3. 最后对于每个中心点的位置，如果 flag=1 就说明这个点需要被膨胀为 1, 将其赋值为 255, 否则赋值为 0.

3.4 开操作与闭操作

```
/* 注意是在第一个操作后的图上继续修改 */
void Open()
{
   Erosion(&b, &saved);
   Dilation(&saved, &c);
}
void Close()
{
   Dilation(&b, &saved);
   Erosion(&saved, &c);
}
```

根据前面的定义，这里我们的开闭操作直接调用前面的 `Dilation()` 和 `Erosion()` 函数即可，注意二者的顺序，一个是先腐蚀再膨胀，另一个是先膨胀再腐蚀。

四、 实验环境及运行方法

4.1 实验环境

Windows 10 系统

gcc 10.3.0 (tdm64-1) x86_64-w64-mingw32

4.2 运行方法

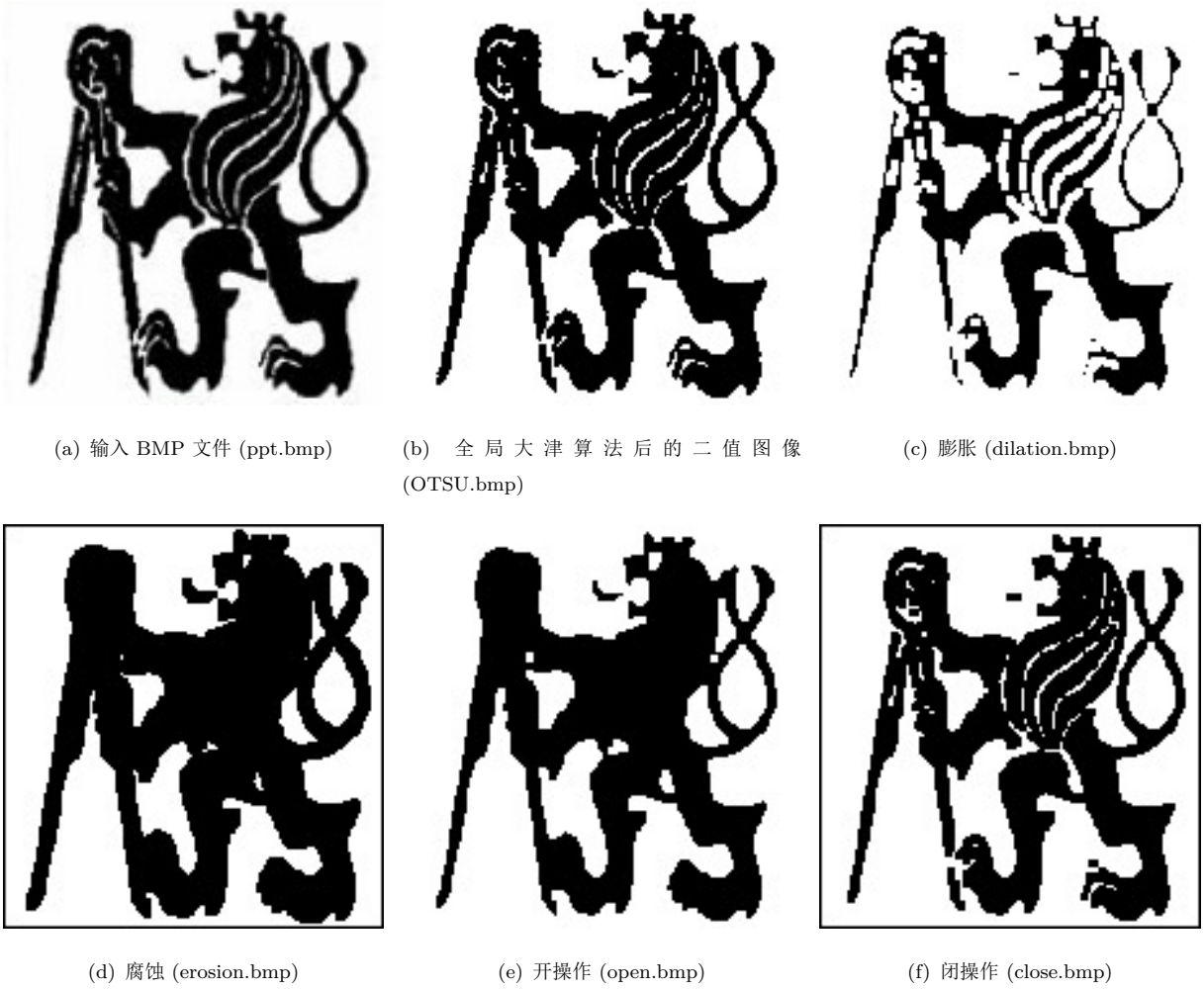
源文件为 lab2.c, 在源文件里有我们的样例输入, 一张 24 色彩色 BMP 图像文件 (LN.bmp), 使用 VSCode 打开这个文件夹，并选中 lab2.c 点击 Run Code 即可开始运行。当终端出现“Successfully open the image”时说明我们成功打开了图像，否则会输出“BMP Image Not Found!”.

随后我们的图像会进行转化,分别输出 Local_OTSU.bmp(滑动窗口 + 局部大津的成果), OTSU.bmp(简单分块 + 大津算法的成果), dilation.bmp(膨胀后的图像) erosion.bmp(腐蚀后的图像) open.bmp(开操作后的图像) close.bmp(闭操作后的图像). 我们可以改变输入时的 LN.bmp 以改变输入图片，观看算法在不同图片下的效果，也可以改变简单分块中的块规模，还可以改变滑动窗口的大小，通过对比了解差异，感受不同算法的优缺点。

需要注意的是，滑动窗口会有相当的时间成本，因此最开始我们注释掉了这一段代码，如有需要可以自行取消注释。

五、 实验结果展示

5.1 检验我们的形态学操作



(a) 输入 BMP 文件 (ppt.bmp)　(b) 全局大津算法后的二值图像 (OTSU.bmp)　(c) 膨胀 (dilation.bmp)

(d) 腐蚀 (erosion.bmp)　(e) 开操作 (open.bmp)　(f) 闭操作 (close.bmp)

图 1: 用 PPT 的例子检验形态学操作

值得注意的是，在图像里 255 表示白色, 0 表示黑色，而我们是将白色作为前景 (1)，后景为黑色 (0)，因此形态学操作与老师 ppt 展示也有所不同。

此外可以发现我们的腐蚀和闭操作，外围有一圈黑框。这是因为我们的算法实现中，我们直接把边缘的点赋为 0, 即黑色，因此会有这样的情况。

5.2 优化大津算法

5.2.1 基本大津算法

(a) 输入 BMP 文件 (LN.bmp)

(b) 不分块的大津算法 (LN1.bmp)

(c) 分为 3 * 3 块的大津算法 (LN3.bmp)

图 2: 分块 + 大津算法

5.2.2 滑动窗口 + 局部大津

(a) 窗口大小 40 * 40(40.bmp)

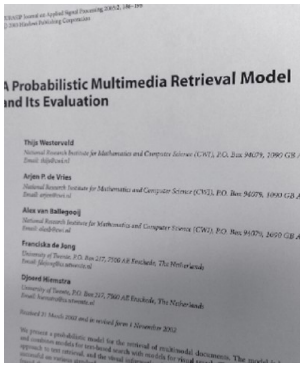
(b) 窗口大小 50 * 50(50.bmp)

(c) 窗口大小 60 * 60(60.bmp)

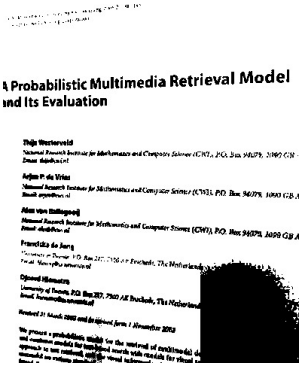
(d) 窗口大小 70 * 70(70.bmp)

图 3: 滑动窗口 + 局部大津算法

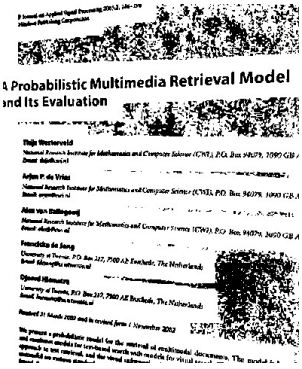
这里还有一个不利于全局大津的例子：



(a) 原图 (excep.bmp)



(b) 全局大津 (excep_otsu.bmp)



(c) 滑动窗口 40 * 40(excep_local.bmp)

可以看出，两种方法各有优劣，全局大津很难直接处理阴影的情况，但滑动窗口可以。此外滑动窗口因为每次局部大津的范围比较小（否则时间消耗太大），存在很多黑斑，全局大津则不会。

六、 心得体会

大津算法本身就是一个全局阈值的算法，因此它不易处理局部方差大的情况。如果只用一次大津算法，如莱纳图中的高频的羽毛、镜子部分的纹路消失了很多，丢失了很多重要的图像信息。于是我开始尝试改进。

最开始使用的简单分块，但后来发现这样对于不同块差距较大的图像会带来边界极强的割裂感。后来想起老师上课说的滑动窗口，经过网上的各种搜索，总算是了解了这种方法，并加以实现。从莱纳图的小窗遍历可以看出，窗的尺寸越小，二值化的颗粒越强，对低频（如皮肤）的处理效果差，但对高频率（如羽毛）的处理效果好，而且还能保留帽子和背景的纹路。需要注意的是，局部大津带来时间的额外成本是显著的，因此我们没有尝试太大的滑动窗口。

经过了这么多折腾，也算是完成了这个 lab，我也感受到了图像信息处理的有趣之处，期待之后更加精彩的实验！