

图像信息处理-实验二

图像二值化

3130000696 黄昭阳 - 11/15/15

实验信息

24位1680x1121bmp实验用图, 原图如下所示



stanleychen

全局二值化

使用**within variance**计算得到的全局二值化



stanleychen

使用between_variance计算得到的全局二值化



局部二值化

sliding window

gap = 3,
windowsize = 100
min within variance



gap = 1

windowsize = 50

between variance 局部二值化

页面显得有点脏，背景偏模糊，细节不清楚，人物细节清楚。



gap = 1

windowsize = 100

between variance 局部二值化

可以看到很多很多的小细节，电线杆，衣服，楼房等等，但是人物轮廓逐渐模糊。



gap = 1

windowsize = 100

between variance 局部二值化

可以看到更多的小细节，电线杆，楼房的轮廓更加清晰，人物轮廓更加模糊，但是山在水中的投影得到显现。



gap = 1

windowsize = 200

between variance 局部二值化

山在水中的效果投影十分明显，可以观察到背景的电线杆、烟囱轮廓十分清楚，不再像之前被浓重的阴影遮住或者中间断掉。



算法优化

局部二值化采用sliding window的算法，对于每一个window，需要计算这个window每一个threshold的between variance，然后取最优解。最暴力的做法是对于每个window都直接扫描window中的所有数据进行计算，但这样的复杂度是 windowsize*bmpsize*255，当 windowsize稍大时，时间将特别长，例如本次实验用图的数据量将达到200*200*1680*1121*255 这个恐怖的数字，显得十分之大，这里使用优化后的算法

优化算法：

1.每次只写window左下角这一个像素点，因为其他的window内的像素点会被后面的值所覆盖，写入没有任何意义。只有当sliding window到达边界的时候才将其他必要部分写下，保证整个数据文件，这样可以保证每个像素点只写一次，所以写的复杂度将从 windowsize*bmpsize优化到bmpsize的级别。

实验用图的优化跳跃是：200*200*1680*1121 -> 1680*1121

```
int finishRow = startRow+deltaHeight; //deltaHeight为纵向的gap
int finishCol = startCol+deltaWidth; //deltaWidth为横向的gap

if(flagRow) finishRow = startRow+height;//已经移到最后一行，写满所有行
if(flagCol) finishCol = startCol+width; //已经移到最后一列，写满所有列

for(int row = startRow; row<finishRow; row++)
    for(int col = startCol; col<finishCol; col++)
        if(bmpFileData[row][col]<threshold)
            data[row][col]= 0;
        else
            data[row][col]= 255;
```

2.计算一个window的值

我们可以观察到sliding window相邻两个window的信息有大量重复，利用这一点进行哟花。假设我们的计算顺序为：

```
for(row :0 -> biHeight)
    for(col:0 -> biWidth)
        operation;
```

每一个(row,col)都对应一个window。

a.对于每一行都维护一个amount数组，其作用是：

amount[i] - 当前行的第一个window (即(row,0)坐标对应的window) 中灰度值为i的像素有amount[i]个。

当前行的amount数组已经得出，则计算下一行的amount数组时，有一行像素将被移出window，有一行像素被移入window，amount只需要扫描这两行像素，减去被移除的，加上被移入的。更新数据即可得到新的一行的amount数据。

```
for(int j=0;j<width;j++)
{
    amount[bmpFileData[row][j]]--;
    //bmpFileData[i][j]表示第i行，第j列像素的灰度值，这里减去被移除的
    if((row+height)<bmpInfoHeader.biHeight)
        amount[bmpFileData[row+height][j]]++;
    //加上被移入的
}
memcpy(row_amount,amount,sizeof(amount));
//将这一行第一个window的数据拷贝给row_amount，这样就可以按照b的做法操作新的一行了
```

b.对于每一个window都维护一个row_amount数组，其作用是：

row_amount[i] - 当前window中灰度值为i的像素有row_amount[i]个

对于每一行，从第一列开始处理，row_amount使用memcpy从amount处拷贝一份来获得该行第一个window的数据，在window沿着这一行平移的时候，window每向右移一个像素点，则有一列像素被移出window，有一列像素被移入window，row_amount需要扫描这两列像素来更新数据，这样每移动一次window只需要消耗2*window height的时间。

```
for(int i=row;i<row+height;i++)
{
    row_amount[bmpFileData[i][col]]--;
    if((col+width)<bmpInfoHeader.biWidth)
        row_amount[bmpFileData[i][col+width]]++;
}
```

这样子算法的复杂度为 $bmpSize * window height * 255$ ，在本次试验中的优化数据是：

$200 * 200 * 1680 * 1121 * 255 \rightarrow 200 * 1680 * 1121 * 255$

变得可以接受。

使用row_amount计算一个window threshold的做法如下：

```
tot_amount = height*width;
sum_back = 0;
minimal = 255;
maximal = 0;
for(int i=0; i<256; i++)
{
    sum_back+= row_amount[i]*i;
    if(row_amount[i]>0)
    {
        if(minimal>i) minimal = i;
        if(maximal<i) maximal = i;
    }
}
amountf = row_amount[minimal];
wf = amountf/tot_amount;
average_fore = minimal;
chosen_threshold = minimal;
sum_fore = minimal*row_amount[minimal];

amountb = tot_amount - amountf;
wb = amountb/tot_amount;
sum_back -= minimal*row_amount[minimal];
average_back = sum_back/amountb;
max_between_variance = sqr(average_fore-average_back)*wb*wf;
int chosen_threshold = minimal;
for(int threshold = minimal+1; threshold < maximal ;
threshold++)
{
    amountf+= row_amount[threshold];
    amountb-= row_amount[threshold];
    wf = amountf/tot_amount;
    wb = amountb/tot_amount;
    sum_fore += row_amount[threshold]*threshold;
    sum_back -= row_amount[threshold]*threshold;
    average_fore = sum_fore/amountf;
    average_back = sum_back/amountb;
    double between_variance = sqr(average_fore-
average_back)*wb*wf;
    if(between_variance > max_between_variance)
    {
        max_between_variance = between_variance;
        chosen_threshold = threshold;
    }
}
```

本次实验示例：

windowsize = 200x200

bmpsize = 1680x1121

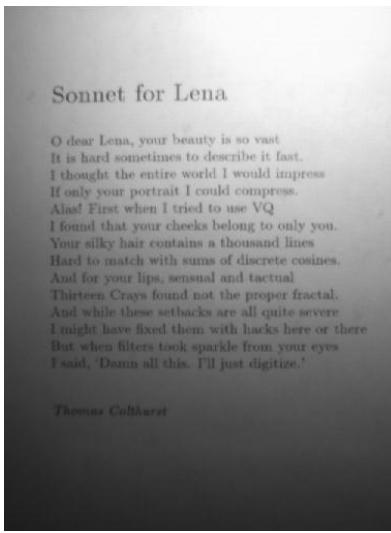
gap = 1

约十秒内能够出解。

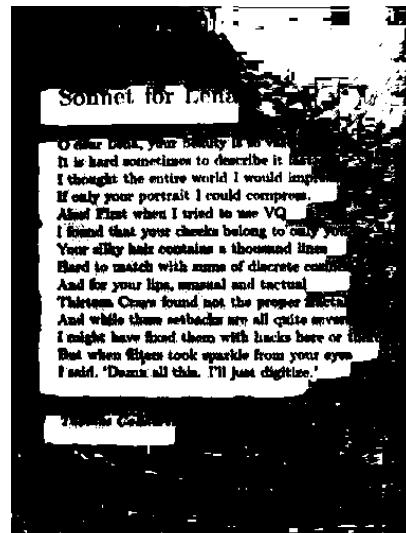
文本扫描图二值化

在扫描文本的时候，由于阴影的存在，直接使用局部二值化将出现大团阴影，考虑到当一个window的max between variance偏小时，说明这一个window内的灰度十分接近，区分度很低，可以直接视为背景（当它是文本时必然不会出现区分度低的情况），因此传入一个constriction，当 $\text{max between variance} < \text{constriction}$ 时直接将它视为背景而不采用threshold策略，

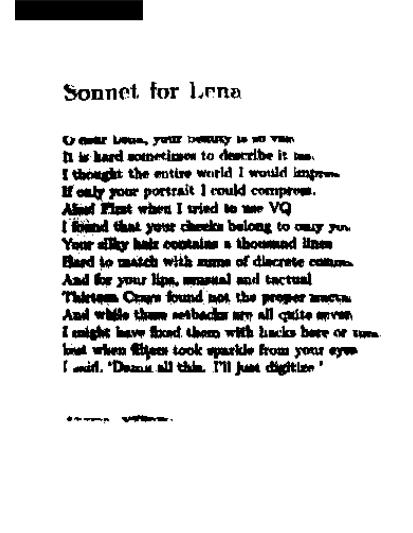
原图：



直接局部二值化：



采用该策略的二值化 $\text{constriction} = 50$



A Probabilistic Multimedia Retrieval Model and Its Evaluation

constriction = 45
windowsize = 50x50

A Probabilistic Multimedia Retrieval Model and Its Evaluation

之前的所有图像都是基于window sliding的算法（一个像素一个像素的滑过去），考虑到text扫描的特殊性，这里使用constriction来清除黑暗块，并且不再是一个像素一个像素的滑，相邻的两个window之间不取重叠部分。

即由原来的row+1, col+1的滑动改为row+window height, col+window width的移动，因为字与周围的界限十分明显，当window内有字时，between variance会比window内无字的要大很多，此时如果使用sliding的算法，滑到边界情况时势必会引起模糊，而如果用window将整张图网格化，一个网格内有字的可以直接使用threshold进行二值化，而网格内无字的则会被constriction限制直接变成背景色。下面为使用20x20, constriction为30时的图。

Sonnet for Lena

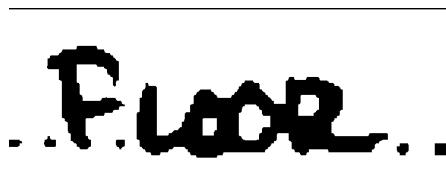
O dear Lena, your beauty is so vast
It is hard sometimes to describe it just.
I thought the entire world I would improve
If only your portrait I could comprise
Alas! First when I tried to use VQ
I found that your cheeks belong to only you
Your silky hair contains a thousand lines
Hard to match with sums of discrete cosines
And for your lips, sensual and tactile
Thirteen Crags found not the proper fractal
And while these setbacks are all quite severe
I might have fixed them with lucks here or there
But when filters took sparkle from your eyes
I said "Never mind, I'll use bigtime!"

Never

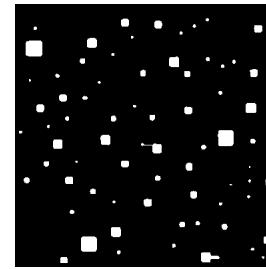
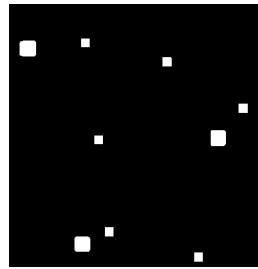
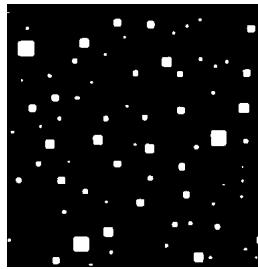
膨胀与腐蚀

这里不上单纯的膨胀腐蚀效果图，没什么意义，直接上它的功能图。

1.floor的开和关，分别是原图、开、关。由于是从PPT上屏幕截图下来的，像素有点多，这里使用18*18的方阵来进行操作的。



2.rectangular的开和关，可以看到关的图有些方块连起来了，这里用的13*13的方阵操作



3.monster的extract。用的十字架（一个像素级别）来抽取的

