大作业：图片网络爬虫设计与图像处理（10 分）

1. 代码检查日期：2018 年 1 月 4 日, 18:00-22:00pm
2. 书面材料提交
   截止日期：2018 年 1 月 7 日, 23:59pm，截止日期之后不再接收提交。
   提交内容：书面报告文档（Word 版以及 PDF 版本）+程序源码+程序使用说明，打包成 rar 压缩文件，提交文件命名方式：学号_姓名_数据结构 2017 大作业.rar
   提交方式：以邮件形式发送至 datastructure_nju2@hotmail.com

1. 概述

设计、实现图片网络爬虫，从网站上批量下载图片；并对图片进行图像处理。

2. 功能

2.1 网址获取（25%）

1） 自选图片网站，将其主站网址设为爬虫的初始网址。

2） 使用正则表达式，从每个网页中提取更多的网址，进行存储，以便后续访问。

PS：不同网页中的网址其展现格式有所区别。分析选定网站中的网址格式，正确使用正则表达式完成提取。（用浏览器打开一个网页，点击右键->查看源代码即可查看网页源码）

2.2 网页去重（20%）

每当获取一个网址，验证其是否在此前访问过，从而避免重复访问、以及访问环路。（利用 Bloom filter 实现）

2.3 图片下载（5%）

分析网页中的图片链接，对其图片进行下载、存储（正则表达式）。

2.4 图像处理（25%）

1）实现图像平滑降噪（见附录一）

2）实现 Sobel 边缘检测功能（见附录二）

3. 设计要求

3.1 上述功能必须自己独立完成，不得采用开源软件。其中网络爬虫的实现语言不限，图像处理实现语言指定为 C++。

3.2 爬取不重复网页的数量不少于 1000 个，图片数量不少于 10000 张。正则表达式自学。

3.3 对于待访问的网址，使用深度优先遍历的方式访问。

（自行选择合适的存储结构）

3.4 网页去重要求 Bloom filter（BF）的假阳性概率(false positive ratio)控制在 0.1%以内。编写 Bloom filter 类，实现 BF 的初始化操作（给定存储元素个数 n，计算最优 Hash 函数的个数 k 以及 BF 的长度 m），以及 BF 构建（插入操作）。

注：Hash 函数可以参见附录三。可测试不同 Hash 的随机性，选择你认为最佳的 k 个 Hash 函数。也可以选择某一个 Hash 函数，然后利用不同 Hash 种子，来模拟多 Hash。

3.5 实现图像处理类 PixImage，包含模糊方法 Bluring()，以及 Sobel 边缘检测方法 Sobel()。将处理后的图片按照一定的格式(tif, png, jpg, bmp 等皆可)进行保存。

3.6 代码注释详尽。

3.7 给出书面报告（25%）

1） 展现项目的总体设计框架

2） 描述各模块实现的思路、方法、细节

3） 展现最终项目达到的效果

附录一、图像模糊（平滑）

图像是一个由彩色像素构成的矩阵，如下图所示（4x3 图像）：

```
      ------> x

  |   -----------------------------
  |   | 0, 0 | 1, 0 | 2, 0 | 3, 0 |
y |   -----------------------------
  |   | 0, 1 | 1, 1 | 2, 1 | 3, 1 |
  v   -----------------------------
      | 0, 2 | 1, 2 | 2, 2 | 3, 2 |
      -----------------------------
```

图像原点位于左上方，向右为 x 轴正方向，向下为 y 轴正方向。我们使用二维坐标（i，j）来表示 x 坐标为 i，y 坐标为 j 的像素。每个像素有三个分量，范围都在 0~255 之间，代表像素的红、绿、蓝的强度。 这三个分量被称为图像的 RGB 值。三个值均为零的像素是黑色，三个值均为 255 的像素是白色。

实现一个名为 PixImage 的类，用于存储一幅彩色图像。PixImage 类包含了读取图像像素、修改图像像素、模糊图像、检测图像中的边缘等方法。PixImage 的信息包括图片大小、图片名称、和每幅图片中所有像素的 RGB 值，其具体存储方式由你决定。PixImage 的大小在构造时确定，且之后不会改变。

原图

模糊处理

如上图所示，原图中会有图像噪声，导致图片质量下降。图像模糊是图像降噪常见的处理方法。本作业要求使用均值滤波器来消除噪声。原理：对于任意一个像素，使用其周围 n x n 像素范围内的平均值来置换该像素值。如：n=3,则权值矩阵模板为：

$$1/9 \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

$$b(i，j) = (f(i-1,j-1) + f(i,j-1)+ f(i+1,j-1) + f(i-1,j) + f(i,j) +$$

$$f(i+1,j) + f(i-1,j+1) + f(i,j+1) + f(i+1,j+1))/9$$

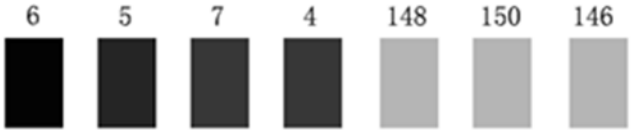
其中，b(i,j)为像素（i,j）模糊化后的像素值，f(i,j)为像素（i,j）的原始像素值。通过均值滤波，可以使图像模糊，达到消除细小噪声的目的。其副作用是在噪声被消除的同时，目标图像也变模糊了，见灰太狼右图。

本作业要求实现 bluring()方法，对输入的图片进行均值过滤模糊化处理（注：对 RGB 三个分量分别处理）。变量 n 可作为形参传入。同时，模糊处理可以多次迭代执行，以达到满意的降噪效果。

附录二、图像边缘检测

边缘检测是图像处理与计算机视觉中极为重要的一种分析图像的方法。边缘检测的目的就是找到图像中亮度变化剧烈的像素点构成的集合，表现出来往往是轮廓。如果图像中边缘能够精确的测量和定位，那么，就意味着实际的物体能够被定位和测量，包括物体的面积、物体的直径、物体的形状等就能被测量。在对现实世界的图像采集中，有下面 4 种情况会表现在图像中时形成一个边缘。

1. 深度的不连续（物体处在不同的物平面上）；
2. 表面方向的不连续（如正方体的不同的两个面）；
3. 物体材料不同（这样会导致光的反射系数不同）；
4. 场景中光照不同（如被树萌投向的地面）；



上面的图像是图像中水平方向 7 个像素点的灰度值显示效果，我们很容易地判断在第 4 和第 5 个像素之间有一个边缘，因为它俩之间发生了强烈的灰度跳变。在实际的边缘检测中，边缘远没有上图这样简单明显，我们需要取对应的阈值来区分出它们。边缘检测的算法有很多，本项目要求实现索贝尔 Sobel 边缘检测。如下图所示：

原图

Sobel 边缘

索贝尔算子（Sobel operator）是一离散性差分算子，用来运算图像亮度函数的灰度之近似值。

-1	0	+1
-2	0	+2
-1	0	+1

Gx

+1	+2	+1
0	0	0
-1	-2	-1

Gy

该算子包含两组 3x3 的矩阵，分别为横向及纵向，将之与图像作平面卷积，即可分别得出横向及纵向的亮度差分近似值。如果以 A 代表原始图像，Gx 及 Gy 分别代表经横向及纵向边缘检测的图像灰度值，其公式如下：

$$\mathbf{G_x} = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} * \mathbf{A} \quad \text{and} \quad \mathbf{G_y} = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * \mathbf{A}$$

Gx = (-1)*f(x-1, y-1) + 0*f(x,y-1) + 1*f(x+1,y-1)

+ (-2)*f(x-1,y) + 0*f(x,y)+2*f(x+1,y)

+ (-1)*f(x-1,y+1) + 0*f(x,y+1) + 1*f(x+1,y+1)

= [f(x+1,y-1)+2*f(x+1,y)+f(x+1,y+1)]-[f(x-1,y-1)+2*f(x-1,y)+f(x-1,y+1)]

Gy = 1* f(x-1, y-1) + 2*f(x,y-1)+ 1*f(x+1,y-1)

+ 0*f(x-1,y) 0*f(x,y) + 0*f(x+1,y)

+ (-1)*f(x-1,y+1) + (-2)*f(x,y+1) + (-1)*f(x+1, y+1)

= [f(x-1,y-1) + 2f(x,y-1) + f(x+1,y-1)]-[f(x-1, y+1) + 2*f(x,y+1)+f(x+1,y+1)]

图像的每一个像素的横向及纵向梯度近似值可用以下的公式结合，来计算梯度的大小。

$$\mathbf{G} = \sqrt{\mathbf{G_x}^2 + \mathbf{G_y}^2}$$

得到梯度图像后，我们需要的是计算阈值，这是 Sobel 算法很核心的一部分，也是对效果影响较大的地方。为统一操作，本项目使用的 Sobel 阈值计算如下：

```
scale = 4;
cutoff = scale*mean(b);
thresh = sqrt(cutoff);
```

其中 mean 函数是求图像所有点灰度的平均值。scale 是一个系数（大家可以尝试不同系数下的输出效果）。最终将图片转换为二值（黑白）图像保存。

附录三、哈希函数

1.RS 从 Robert Sedgwicks 的 *Algorithms in C*一书中得到了。原文作者已经添加了一些简单的优化的算法，以加快其散列过程。

```
1.  public long RSHash(String str)
2.    {
3.      int b     = 378551;
4.      int a     = 63689;
5.      long hash = 0;
6.      for(int i = 0; i < str.length(); i++)
7.      {
8.        hash = hash * a + str.charAt(i);
9.        a    = a * b;
10.     }
11.     return hash;
12.   }
```

2.JS Justin Sobel 写的一个位操作的哈希函数。

```
1.  public long JSHash(String str)
2.    {
3.      long hash = 1315423911;
4.      for(int i = 0; i < str.length(); i++)
5.      {
6.        hash ^= ((hash << 5) + str.charAt(i) + (hash >> 2));
7.      }
8.      return hash;
9.    }
```

3.PJW 该散列算法是基于贝尔实验室的彼得 J 温伯格的的研究。

```
1.  public long PJWHash(String str)
2.    {
3.      long BitsInUnsignedInt = (long)(4 * 8);
4.      long ThreeQuarters     = (long)((BitsInUnsignedInt  * 3) / 4);
5.      long OneEighth         = (long)(BitsInUnsignedInt / 8);
6.      long HighBits          = (long)(0xFFFFFFFF) << (BitsInUnsignedIn
    t - OneEighth);
7.      long hash              = 0;
```

```
        long test   = 0;
        for(int i = 0; i < str.length(); i++)
        {
            hash = (hash << OneEighth) + str.charAt(i);
            if((test = hash & HighBits)  != 0)
            {
                hash = (( hash ^ (test >> ThreeQuarters)) & (~HighBits));
            }
        }
        return hash;
    }
```

4.ELF 和 PJW 很相似，在 Unix 系统中使用的较多。

```
    public long ELFHash(String str)
    {
        long hash = 0;
        long x    = 0;
        for(int i = 0; i < str.length(); i++)
        {
            hash = (hash << 4) + str.charAt(i);
            if((x = hash & 0xF0000000L) != 0)
            {
                hash ^= (x >> 24);
            }
            hash &= ~x;
        }
        return hash;
    }
```

5.BKDR 这个算法来自 Brian Kernighan 和 Dennis Ritchie 的 *The C Programming Language*。

```
    public long BKDRHash(String str)
    {
        long seed = 131; // 31 131 1313 13131 131313 etc..
        long hash = 0;
        for(int i = 0; i < str.length(); i++)
        {
```

```
        hash = (hash * seed) + str.charAt(i);
    }
    return hash;
}
```

6.SDBM 这个算法在开源的 SDBM 中使用，似乎对很多不同类型的数据都能得到不错的分布。

```
public long SDBMHash(String str)
{
    long hash = 0;
    for(int i = 0; i < str.length(); i++)
    {
        hash = str.charAt(i) + (hash << 6) + (hash << 16) - hash;
    }
    return hash;
}
```

7.DJB 这个算法是 Daniel J.Bernstein 教授发明的，是目前公布的最有效的哈希函数。

```
public long DJBHash(String str)
{
    long hash = 5381;
    for(int i = 0; i < str.length(); i++)
    {
        hash = ((hash << 5) + hash) + str.charAt(i);
    }
    return hash;
}
```

8.DEK 由伟大的 Knuth 在《编程的艺术 第三卷》的第六章排序和搜索中给出。

```
public long DEKHash(String str)   {
    long hash = str.length();
    for(int i = 0; i < str.length(); i++)
    {
        hash = ((hash << 5) ^ (hash >> 27)) ^ str.charAt(i);
    }
    return hash;    }
```