

数据结构大作业实验报告

图片网络爬虫设计与图像处理

欧阳鸿荣

计算机科学与技术系

161220096

一、问题分析

本次大实验可以分为两个部分：

- 1.设计、实现图片网络爬虫，从网站上批量下载图片；
- 2.对图片进行图像模糊和边缘化检测的处理。

（1）网络爬虫方面

在查阅资料比较各种语言后，考虑到 `python` 有比较丰富的库，且打算新学一门语言，决定使用 `python` 进行处理。实验要求爬取不重复网页的数量不少于 1000 个，图片数量不少于 10000 张，对于待访问的网址，使用队列存储，按深度优先访问。

因此，实验的基本思路是，以一个网页为起始网页，先获得其超链接指向的网页，存储于队列中，然后依次访问网页和其子网页，下载网页上的图片。

实验中可能出现环路问题，因此建立一个 `BF` 布隆过滤器类，通过选取恰当额哈希函数，对已经访问过的网址进行过滤。

由于所爬取的网址和图片数目较多，因此查阅资料后采用了多线程。

（2）图像处理方面

由于图片种类繁多，则选取基于 `C++` 的开源库 `Opencv` 进行图片的解析。而图像处理方面，也选择使用 `C++`，基于均值滤波器和索贝尔边缘检测的原理来处理。

二、总体设计框架

(1) 网络爬虫方面

主要将借助 **re**, **requests**, **urllib** 三个库实现网络网页的爬取。

(a) 库 **re** 主要用于实现正则表达式

利用浏览器查看网页源代码，可以发现几乎所有的跳转链接的格式都是 `href = "..."`，然后图片几乎都是 `src = "... + 后缀名"`：

```
" href="http://www.gamersky.com/z/hmm7/">英雄无敌7</a></div>
" href="http://www.gamersky.com/z/enemyfront/">敌军前线</a></div>
" href="http://www.gamersky.com/z/sniperelite3/">狙击精英3</a></div>
" href="http://www.gamersky.com/z/sc2/">星际争霸2</a></div>
" href="http://www.gamersky.com/z/valkyria/">战场女武神</a></div>
```

故将爬取跳转链接的正则表达式定义为：

```
href="(http.[^\\t\\r\\n\\n^]*?)"
```

```
">
/"/>
.5/">
```

将爬取图片的正则表达式定义为：

```
src="(http.[^\\t\\r\\n\\n^]*?\\.)(jpg|png|jpeg|gif|bmp)"
```

然后通过 **re.findall**(模式串,待匹配串)方法即可进行模式串的匹配。

(b) 库 **requests** 主要用来查看网页内容

这样一来便可利用正则表达式匹配到其中的跳转链接和图片下载地址。

在翻阅网上的资料是时，发现，我们需要设置一个浏览器的标识以伪装成浏览器，防止被服务器查水表。请求头部定义如下：

```
hd = {'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/51.0.2704.63 Safari/537.36 Qiyu/2.1.0.0'}
```

(c) 库 **urllib** 用来将图片下载到指定的位置

通过 **urllib.request.urlretrieve**(下载地址,存储路径)即可。

(d) 对于网页去重问题，实现布隆过滤器。

将采取选取一个哈希函数，利用不同 Hash 种子来模拟多 Hash 的方式。设需要爬取的不同网页数为 n ，布隆过滤器的 bit 数为 m ，误判率为 p ，哈希函数的个数为 k ，根据公式推导有如下结论：

$$m = -\frac{n \cdot \ln p}{(\ln 2)^2}$$

误判率最小时有：

$$k = \ln 2 \cdot \frac{m}{n}$$

因此在 Bloom filter 的初始化中，需要给定最大爬取的网页数以及最大误判率，通过以上公式计算最优的 m 和 k ，以实现一个布隆过滤器。对于该 k 个哈希函数，编号为 $0, 1, 2, \dots, k-1$ ，以各自的编号作为哈希函数的随机种子。

对于待访问网址的存储，选用队列作为存储结构，实现深度优先遍历。同时为了提高效率，需要使用多线程，故将先爬够足够的网址，并分别创建相应的待执行线程，最后再一起执行。

在多线程的设计中考虑到有可能出现某个网址获取很慢（有可能是爬取到一些不方便访问的网站，比如以 .exe, .pdf, .apk 结尾的下载链接的网站，或者是已经 404 not found（不存在）的网站），甚至是一直不返回数据而导致线程假死的情况，因此我们在代码中将为每次获取信息报设置一个 timeout 参数，当请求相应超过 timeout 秒时将放弃请求，同时每一个线程将有一个生命周期 lifetime 的参数，若访问失败，则 lifetime--，当 lifetime 减到 0 的时候放弃该线程的执行，也即放弃该线程负责的网址中图片的爬取。

(2) 图像处理方面

利用 opencv 的 cvLoadImage 读取图片，cvSaveImage 保存图片，同时使用 cvGet2D 方法可获取图片某个位置处的 RGB 三个像素值。

(a) 图像模糊方法的实现

可以在读取图像之后，用一个 RGB_Array 的全局数组记录图像的所有位置处的 RGB 像素，然后遍历图像上的每个点，分别求出其周围 $n \times n$ 个位置的 RGB 三个值的平均值，并将其赋给 RGB_Array 的对应位置的记录值，全部处理完成后通过 cvSet2D 设置修改后的像素，最后再用 cvSaveImage 进行保存即可。

(b) 图像边缘检测方法的实现

首先利用 cvLoadImage 方法读取图片，同时在传入参数的时候设置 iscolor=0 读取图片，同样可以通过 cvGet2D 读取到灰度值，并将之存储到一个 Gray_Array 的全局数组中，通过遍历所有的像素点，利用 sobel 算子分别估算每一个位置处的梯度值，同时算出整个图像的灰度平均值，根据公式 $\sqrt{\text{scale} \times \text{mean}}$ 计算阈值，然后将所有梯度大于等于该阈值的点的灰度值置为 255，小于该阈值的点的灰度值置为 0。同样通过 cvSet2D 设置修改后的像素，最后利用 cvSaveImage 保存图片即可。

三、各模块思路和代码详解

(1) 网络爬虫方面

爬虫部分的文件组织架构如下图：

image	2018/1/2 21:13	文件夹	
lib	2017/12/24 19:18	文件夹	
main.py	2018/1/2 21:07	Python File	1 KB
__pycache__	2018/1/2 21:13	文件夹	
bloomFilter.py	2017/12/27 22:14	Python File	2 KB
imaScript.py	2018/1/2 21:12	Python File	8 KB
mmh.c	2017/12/28 19:11	C Source	2 KB
mmhlib.so	2017/12/24 17:15	SO 文件	46 KB

main.py 是主函数，imaScript.py 用于实现图片的爬取，而 bloomFilter.py 用来实现布隆过滤器，而 mmh.c 是在网上查找的一个哈希随机性非常强大的哈希函数，（由于实验初始阶段布隆过滤器写错导致网页爬取速度非常慢，但没当即检查出来，因此曾怀疑过哈希函数的速度太慢以及随机性不好导致形成环路，便先引入了现成的哈希函数，也尝试了如多线程等方法，后来才发现其实是布隆过滤器写错了）

1.imaScript.py

(a) 准备工作与初始化

该文件引用了许多外部库以及自己所实现的 bloomFilter

其中，re 用于正则表达式的匹配，request 用于网页内容的获取，urllib 用于图片的下载，queue 用于队列的实现，time 用于获取系统时间，os 用于文件目录的操作，threading 用于多线程的实现，

```
1 import re,requests,urllib
2 import queue
3 import time
4 import os
5 import threading
6 from lib import bloomFilter
7
```

hd 是浏览器的标识

imaRe 和 htmlRe 是匹配图片链接和网页链接的正则表达式

```
hd = {'User-Agent':'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
#伪装成浏览器，防止被查水表
imaRe = re.compile( 'src="(http.[^ \t^\r^\n^]*?\.)(jpg|png|jpeg|gif|bmp)"',re.S
htmlRe = re.compile( 'href="(http.[^ \t^\r^\n^]*?)"',re.S )#正则匹配网址
```

把 imaScript 封装为一个类，便于代码的逻辑清晰与实现，其中参数的意义在注释中已经注明。

htmlStart:网络图片爬取的起始网页
rootPath:图片存储的根目录
htmlMaxCount:需要爬取的最大网址数
imaMaxCount:需要爬取的最大图片数
errors:布隆过滤器需要达到的误判率
lifetime:每个网址在爬取过程中的最大重复请求数
timeout:每次请求的最大时长，以秒为单位

`_init_`是 imaScript 类的构造函数，同时初始化布隆过滤器

```
class imaScript:
    def __init__(self,htmlStart,rootPath,htmlMaxCount=1000,imaMaxCount=10000,errors=0.001,lifetime=2,timeout=
        '''
            htmlStart:网络图片爬取的起始网页
            rootPath:图片存储的根目录
            htmlMaxCount:需要爬取的最大网址数
            imaMaxCount:需要爬取的最大图片数
            errors:布隆过滤器需要达到的误判率
            lifetime:每个网址在爬取过程中的最大重复请求数
            timeout:每次请求的最大时长，以秒为单位
        '''
        if not rootPath.endswith('\\'): #让存储目录以'\'结尾，方便存储目录的层次化实现
            rootPath += '\\'
        if not os.path.exists(rootPath):
            os.mkdir(rootPath)
        self.rootPath = rootPath
        self.htmlQueue = queue.Queue()
        self.htmlStart = htmlStart
        self.htmlQueue.put(htmlStart) #将起始网址与以之对应的图片存放文件夹入队列
        self.imaId = 0 #记录下载的图片数量
        self.imaFailCnt = 0 #记录下载失败的图片数量
        self.htmlId = 0 #记录爬取的网址数量
        self.htmlFailCnt = 0 #记录爬取失败的网址数量
        self.htmlMaxCount = htmlMaxCount
        self.imaMaxCount = imaMaxCount
        par = bloomFilter.optimalParameters(htmlMaxCount,errors) #计算布隆过滤器的最优m和k
        self.BFlist = bloomFilter.BF(par[0],par[1]) #初始化布隆过滤器
        self.BFlist.add(htmlStart) #将初始网址加入布隆存储器中
        self.threadLifetime = lifetime
        self.timeout = timeout
```

(b) run 函数

```
def run(self,thread_num):

    if thread_num<1: #判断线程数是否小于1
        print('线程数不能小于1;')
        return
    thread_num += 1 #将主线程加上
    self.thread_num = thread_num
    st = time.time() #记录开始爬虫的时间
    print('爬虫开始;')
    self.threadings = list() #初始化线程队列
    self.__mulScript() #执行多线程爬虫方法
    #输出爬取的结果信息
    print('爬虫结束,总耗时%d秒,信息如下;'%(time.time()-st))
    print('爬取网站%d个,其中%d个爬取失败;'%(self.htmlId,self.imaFailCnt))
    print('其中爬取图片%d张,其中%d张下载失败;'%(self.imaId,self.imaFailCnt))
    print('原始图片已全部保存至目录%s下;'%self.rootPath)
```

该函数主要用来执行爬虫的主程序,其中 thread_num 是允许开启的最大线程数,通过__mulScript 执行多线程爬虫,并且记录时间等信息,待爬虫结束后输出信息。

(c) __mulScript 函数

```
def __mulScript(self):#通过布隆过滤器的检验,将非重复的网址存入htmlQueue队列,同时在threading中添加相应的待执行线程。

    def htmlGet(lifetime,html):# 获取html中的所有子链接,并存入htmlQueue和threading两个队列中。
```

__mulScript 函数是一个私有的函数,通过布隆过滤器的检验,将非重复的网址存入 htmlQueue 队列,同时在 threading 中添加相应的待执行线程,最后再执行。内置有 htmlGet 函数,通过该函数,获取 html 中的所有子链接,在布隆过滤器检验不形成环路后,存入 htmlQueue 和 threading 两个队列中。具体代码中有注释。

```
def htmlGet(lifetime,html):# 获取html中的所有子链接,并存入htmlQueue和threading两个队列中。

    if lifetime<=0: #当次数达上限后,放弃该网址的请求
        print('多次请求失败,放弃',html)
        return
    try: #尝试在timeout在时间内请求数据
        rq = requests.get(html,headers=hd,timeout=self.timeout)
    except: #若请求失败则重试,并lifetime-1
        print(html+'重新连接;')
        return htmlGet(lifetime-1,html)
    if rq.status_code!=requests.codes.ok: #若响应码异常则返回
        print()
        print(html)
        print('页面错误;')
        print()
        rq.close()
        return
    code = rq.content.decode('utf8','ignore') #以utf8编码解析网页内容,对于无法解码的情况忽略错误
    rq.close() #断开连接
    htmlList = re.findall( htmlRe,code ) #用正则表达式匹配页面中的所有跳转链接
    for i in htmlList: #遍历所有得到的跳转链接
        if len(self.threadings)>=self.htmlMaxCount: #若已达到了最大爬取网址数,则返回
            return
        if self.BFlist.isContains(i): #若布隆过滤器检测到重复,则跳过该链接
            continue
        print("捕获到过滤后的网址"+i)
        t = threading.Thread(target=self.__script,args=(self.threadLifetime,i,))
        self.threadings.append(t) #爬取该跳转链接的线程入线程队列
        self.htmlQueue.put(i) #html入htmlQueue队列,以供之后的深度优先遍历搜索
```

__mulScript 函数的剩余部分用于多线程的开启，先将爬取根网址图片的线程入线程队列，再深度优先遍历搜寻跳转链接，并且设置守护线程，以及开启阻塞主线程的执行

```
t = threading.Thread(target=self.__script,args=(self.threadLifetime,self.htmlStart,))
self.threadings.append(t) #将爬取根网址图片的线程入线程队列
while not self.htmlQueue.empty(): #深度优先遍历搜寻跳转链接
    element = self.htmlQueue.get() #获取队头元素
    htmlGet(self.threadLifetime,element) #以该队头元素搜寻其子链接，并填充线程队列
    if len(self.threadings)>self.htmlMaxCount: #若达到了最大网页爬取数，则跳出搜索
        break
print('总共找到网站%d个'%len(self.threadings)) #输出信息
self.htmlId = len(self.threadings) #记录总爬取网页数
for t in self.threadings: #遍历线程队列
    t.setDaemon(True) #设置守护线程
    t.start() #开启线程
while True: #死循环，直到正在运行的线程数少于最大线程数，才会开启新的线程
    if threading.activeCount()<self.thread_num:
        break
for t in self.threadings: #阻塞主线程的执行
    t.join()
```

(d) __script 函数

该函数负责将 html 中的所有图片下载到 rootPath 中，并且在该函数中，也负责一些异常情况以及终止的响应和处理。若多次请求访问失败，则直接返回。若数量达到上限或者响应码异常，亦返回。若下载失败，也返回。

```
def __script(self,lifetime,html):# 将html中的所有图片下载到rootPath中。

    if self.imaId>=self.imaMaxCount: #当图片数量达到最大图片数量时，返回
        return
    if lifetime<=0: #当多次请求失败后，放弃该地址
        print('多次请求失败,放弃',html)
        self.htmlFailCnt += 1
        return
    try: #尝试在指定的timeout时间内请求网页数据
        rq = requests.get(html,headers=hd,timeout=self.timeout)
    except: #请求失败后再次尝试请求，并lifetime-1
        print(html+'重新连接;')
        return self.__script(lifetime-1,html)

    if rq.status_code!=requests.codes.ok: #若响应码异常则返回
        print()
        print(html)
        print('页面错误;')
        print()
        rq.close()
        return
    code = rq.content.decode('utf8','ignore') #以utf8编码解析网页内容，对于无法解码的情况忽略错误
    rq.close() #断开连接

    imaList = re.findall(imaRe,code) #利用正则表达式匹配网页中的所有图片url地址
    print('爬取网站:'+ html + ' ')
    for imaLink in imaList: #遍历该网页的所有的图片url地址
        if self.imaId>=self.imaMaxCount: #若图片下载达数量上限，则返回
            return
        self.imaId += 1
        name = str(self.imaId) + '.*'%( str(imaLink[1]) ) #图片存储的图片名
        try: #尝试下载图片
            urllib.request.urlretrieve(''.join(imaLink),self.rootPath+name)
        except: #若下载失败，则记录失败的次数，并输出信息
            self.imaFailCnt += 1
            print()
            print(''.join(imaLink))
            print("图片下载失败;")
```


2.bloomFilter.py

(a) 准备工作与初始化

```
import math
from bitarray import bitarray
import ctypes

lib = ctypes.cdll.LoadLibrary('lib/mmhlib.so')
```

bloomFilter.py 用来实现布隆过滤器，而 mmh.c 是在网上查找的一个哈希随机性非常强大的哈希函数

（由于实验初始阶段布隆过滤器写错导致网页爬取速度非常慢，但没当即检查出来，因此曾怀疑过哈希函数的速度太慢以及随机性不好导致形成环路，便先引入了现成的哈希函数，也尝试了如多线程等方法，后来才发现其实是布隆过滤器写错了）

在这里引用了基于 c 语言的哈希函数 mmh

```
/*gcc -o mmhlib.so -shared -fPIC mmh.c*/
#include <string.h>
unsigned int murmurHash(const char *key, const int seed)
{
    //由随机种子seed计算字符串key的哈希值

    int len = strlen(key); //获取字符串长度
    const unsigned int m = 0x5bd1e995;
    const int r = 24;
    unsigned int h = seed ^ len; //异或运算
    // Mix 4 bytes at a time into the hash
    const unsigned char *data = (const unsigned char *)key;
    while(len >= 4)
    {
        unsigned int k = *(unsigned int *)data; //四个字节解码为int
        k *= m;
        k ^= k >> r; //移位运算:k右移r位
        k *= m;
        h *= m;
        h ^= k;
        data += 4;
        len -= 4;
    }
    // Handle the last few bytes of the input array
    switch(len)
    {
        case 3:h ^= data[2] << 16;
        case 2:h ^= data[1] << 8;
        case 1:h ^= data[0];h *= m;
    };
    // Do a few final mixes of the hash to ensure the last few
    // bytes are well-incorporated.
    h ^= h >> 13;
    h *= m;
    h ^= h >> 15;
    return h;
}
```

这次也学习了一下用 python 调用 c 的一些方法和技巧，通过命令行窗口在运行的目录下键入 gcc -o 生成文件名.so -shared -fPIC mmh.c。然后会在目录生成一个 .so 文件 这个文件就可以直接被 python 调用了。

python 调用的方式就是通过 ctypes 库。然后就可以调用 C 的函数。这里有个要注意的地方是在 python 里调用这个传入字符串的时候，要传入字符串的二进制流，而不能直接传 unicode 编码的 str，在传入的时候，加个.encode()就可以了，encode 将字符串重新编码成二进制的形式。

(b) optimalParameters

通过给定的需要存储的元素个数和需要达到的假阳性率计算布隆过滤器中的最优二进制数组大小和哈希函数个数，其中计算公式在前文有。

```
def optimalParameters(eleNum,errors):# 通过给定的需要存储的元素个数和需要达到的假阳性率计算布隆过滤器中的最优二进制数组大小和哈希函数个数
    size = math.ceil(-eleNum*math.log(errors)/(math.log(2)*math.log(2)))
    hashCount = math.ceil(math.log(2)*size/eleNum)
    return size,hashCount
```

(c) BF 类

实现一个 BF 类，BF 类中包含三个函数，分别是 `_init_`，`add`，`isContains`

• `_init_`：构造函数

初始化布隆过滤器，其中 `size` 为布隆过滤器中二进制数组的大小，即 `m`。
`hash_count` 为哈希函数的个数，即 `k`。在这里用了 `bitarray` 类型来方便操作，其中 `m` 和 `k` 为上面的 `optimalParameter` 计算出来的使得假阳性概率低于 0.01% 的合适的 `m` 和 `k`。

```
class BF:
    def __init__(self,size,hash_count):#初始化布隆过滤器，size为二进制数组大小，hash_count为哈希函数个数
        self.bitArray = bitarray(size) #初始化二进制数组
        self.bitArray.setall(0) #二进制数组全部置0
        self.size = size
        self.hashCount = hash_count
```

• `add`：添加元素

对于传入的 `item`，通过调用哈希函数，向布隆过滤器中添加网址 `item`

```
def add(self,item):#向布隆过滤器中添加字符串元素item
    item = item.encode()#调用C就是坑，把字符串转化为二进制形式
    for i in range(self.hashCount):
        index = lib.murmurHash(item,i)%self.size #以i为随机种子得到哈希映射值
        self.bitArray[index] = True #将相应二进制数组位置置1
```

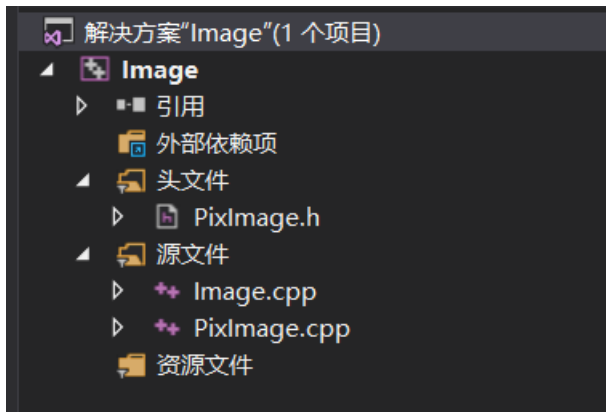
• `isContains`：添加元素

对于网址 `item`，通过计算其哈希值，在 `bitArray` 中检索，看是否访问过

```
def isContains(self,item):#判断元素item是否已经存储过
    item = item.encode()
    for i in range(self.hashCount):
        index = lib.murmurHash(item,i)%self.size #以i为随机种子
        if not self.bitArray[index]: #若有一个哈希函数没有映射到1
            return False
    return True #否则认为存储过该元素
```


(2) 图片处理方面

图片处理部分的文件组织架构如下图：

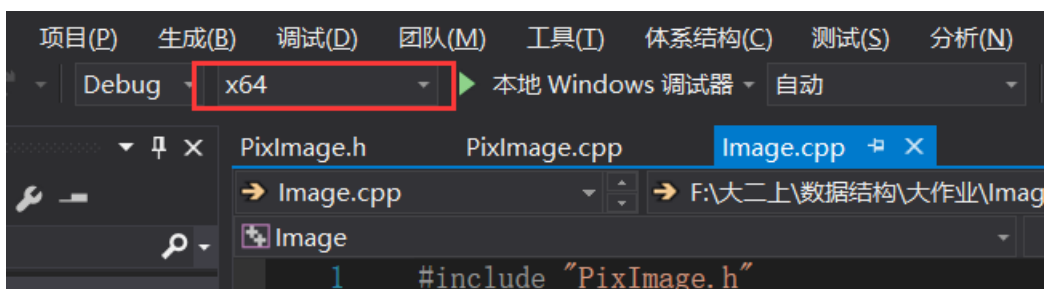


PixImage.h 是对于 PixImage 类以及一些全局变量的定义
PixImage.cpp 是对 PixImage 类内部函数的具体事项
Image.cpp 是 main 函数所在的文件，用于与用户交互

本次实验用了 opencv 库对图片进行解析

 opencv-3.3.1-vc14.exe

在配置好系统的环境变量等变量之后，在 vs2015 中设置



选择 x64 的环境以适配 opencv 的要求

1. PixImage.h

PixImage.h 是对于 PixImage 类以及一些全局变量的定义

先引用一些函数库 以及声明名空间

```
#pragma once
#include <opencv2/opencv.hpp>
#include <iostream>
#include <fstream>
#include <stdlib.h>
#include <math.h>
#include <io.h>

using namespace std;
using namespace cv;
```

定义 MAX = 5000，这是图片像素处理的最大宽度

同时定义两个数组 RGB_Array 和 Gray_Array 来存储待处理图片的信息

（其实这是在 Image.cpp 中定义的，但是都是属于定义，就放这一起说明）

```
const int MAX = 5000;
double RGB_Array[MAX][MAX][3];
double Gray_Array[MAX][MAX];
```

最后便是整个 PixImage 类的定义

```
class PixImage {
public:
    int ImageHeight, ImageWidth; //记录图像的宽高
    char *ImageName; //记录图像的名称
    bool grayDefault; //是否以opencv默认的灰度获取方法获取图像灰度矩阵
private:
    IplImage *img, *imgGray; //图像读取的指针变量
    int Sobel_OperatorX[3][3] =
    { { -1, 0, 1 },
      { -2, 0, 2 },
      { -1, 0, 1 } }; //横向sobel算子
    int Sobel_OperatorY[3][3] =
    { { 1, 2, 1 },
      { 0, 0, 0 },
      { -1, -2, -1 } }; //纵向sobel算子
public:
    PixImage(char *fileName, bool isGrayDefault = false) {...}
    CvScalar Get_RGB(int i, int j); //获取图像第i行第j列的rgb值
    //直接调用cvGet2D的方法获取该位置rgb值
    double Get_RGB(int i, int j, int k); //获取图像第i行第j列的rgb值
    void Set_RGB(int i, int j, double *value); //将图像第i行第j列的像素设置为value参数:
    void Set_RGB(int i, int j, CvScalar value); //将图像第i行第j列的像素设置为value参数:
    void Set_Gray(int i, int j, double value); //设置图像第i行第j列的灰度值为value
```

Image 类中定义了一些类的公共变量和私有变量，其中

- ImageHeight 和 ImageWidth 记录图像的宽高
- ImageName 记录图像的名称
- grayDefault 表示是否以 opencv 默认的灰度获取方法获取图像灰度矩阵
- *img, *imgGray 是 图像读取的指针变量

并且用预先定义的 3*3 的矩阵记录 sobel 算子，方便后来的边缘化检测

```
class Image {
public:
    int ImageHeight, ImageWidth; //记录图像的宽高
    char *ImageName; //记录图像的名称
    bool grayDefault; //是否以opencv默认的灰度获取方法获取图像灰度矩阵
private:
    IplImage *img, *imgGray; //图像读取的指针变量
    int Sobel_OperatorX[3][3] =
    { { -1, 0, 1 },
      { -2, 0, 2 },
      { -1, 0, 1 } }; //横向sobel算子
    int Sobel_OperatorY[3][3] =
    { { 1, 2, 1 },
      { 0, 0, 0 },
      { -1, -2, -1 } }; //纵向sobel算子
};
```

Image 类也同时定义了对应图像的一些处理函数：

```
public:
    Image(char *fileName, bool isGrayDefault = false) {...}
    CvScalar Get_RGB(int i, int j); //获取图像第i行第j列的rgb值
    //直接调用cvGet2D的方法获取该位置rgb值
    double Get_RGB(int i, int j, int k); //获取图像第i行第j列的rgb值
    void Set_RGB(int i, int j, double *value); //将图像第i行第j列的像素设置为value参数:
    void Set_RGB(int i, int j, CvScalar value); //将图像第i行第j列的像素设置为value参数:
    void Set_Gray(int i, int j, double value); //设置图像第i行第j列的灰度值为value
    double getGray(int i, int j, bool isDefault); //获取图像第i行第j列的灰度值
    void Blurring(int n, int m = 1); //以均值滤波器将图像作平滑模糊处理
    void Sobel(double scale = 4); //以sobel算子处理图像，取得各位置的梯度值
    void Save_Image(); //保存图像
    void Save_Image(char *savePath); //保存图像至其它路径
    void Save_ImageGray(); //保存灰度图像
    void Save_ImageGray(char *savePath); //保存灰度图像至其他路径

private:
    void Load_RGB(); //遍历图像的每一个像素，并调用Get_RGB方法获取各位置的像素值，并记录至RGB_Array矩阵中
    void Save_RGB(); //遍历RGB_Array矩阵并实际更新到实际图像的每一个像素
    void Load_Gray(bool isDefault = false); //遍历图像的每一个像素，并调用getGray方法获取各位置的灰度值，并记录至Gray_Array矩阵中
    void Save_Gray(); //遍历Gray_Array矩阵并实际更新到实际灰度图像的每一个像素
    void Sum_Average_RGB(double *rgb, int ci, int cj, int n); //计算图像第ci行第cj列像素周围n*n个像素的rgb平均值，并记录到rgb向量中
    void Mid_Average_RGB(double *rgb, int ci, int cj, int n); //计算图像第ci行第cj列像素周围n*n个像素的中值，用于中值滤波器实现
    double Gxy(int ci, int cj); //利用sobel算子计算ci行cj列处灰度值的梯度，并返回
    double Sobel_Thresh(double scale); //计算sobel阈值
};
```

其中构造函数负责 `PixelFormat` 类的初始化处理

`fileName` 顾名思义，就是所打开图片的路径即名称

`isGrayDefault` 表示是否以 `opencv` 默认的灰度获取方法获取图像灰度矩阵

通过构造函数，获取待处理图像的长宽，并且利用 `Load_RGB` 和 `Load_Gray` 这两个函数将图片的色彩和灰度信息存储在自定义 `RGB_Array` 和 `Gray_Array` 中

```
public:
    PixelImage(char *fileName, bool isGrayDefault = false)
        :img(cvLoadImage(fileName)), imgGray(cvLoadImage(fileName, 0))
    {
        if (!img)
        {
            cout << "图像打开错误" << endl;
            return;
        }
        //功能:利用opencv库进行图像的读取操作以及一些成员变量的初始化.
        //isGrayDefault:是否以opencv默认的灰度获取方法获取图像灰度矩阵.
        grayDefault = isGrayDefault;
        //获取图片名
        if (strrchr(fileName, '\\'))
            ImageName = strrchr(fileName, '\\') + 1;
        else
            ImageName = fileName;

        if (img->height != imgGray->height || img->width != imgGray->width)
        {
            cout << "RGB或者灰度读取错误" << endl;
            return;
        }

        ImageHeight = img->height; //获取图像高度
        ImageWidth = img->width; //获取图像宽度
        Load_RGB(); //将图像RGB值读取到RGB_Array中
        Load_Gray(grayDefault); //将图像灰度值读取到Gray_Array中
    }
```

其中，灰度的处理在实现的过程中发生了一些情况

在知乎上可以查到,RGB 和灰度是有一定的转换关系的:



Avatar Ye

游戏 话题的优秀回答者

33 人赞同了该回答

因为人眼对RGB颜色的感知并不相同，所以转换的时候需要给予不同的权重。事实上这个原理普遍应用于计算机图像处理系统，比方说DXT压缩的时候，会给予G通道多一个比特，R : G : B为5 : 6 : 5。

另外需要指出的是，这个公式适用于sRGB空间的图像。基本上Photoshop等软件默认都是使用sRGB空间存储图像。另外没有记错的话，Windows操作系统也使用这个公式进行转换。

$$Y' = 0.299 R' + 0.587 G' + 0.114 B'$$

如果是线性空间的图像，则需要使用这个公式。

$$Y' = 0.2126 R' + 0.7152 G' + 0.0722 B'$$

具体参看这个Wiki页面：

[Grayscale](#)。

而 opencv 也自带有读取灰度的函数

用一张色彩鲜艳对比明显的原图作为参考:



使用 opencv 自带的函数得到的图
isGrayDefault=true 的图:



通过灰度计算公式得到的图
isGrayDefault=false:



通过比较可以发现，这两幅图的灰度细节有些不一致
因此为了保险起见，还是选择用 opencv 的基本功能

2. PixImage.cpp

下面结合代码说明图像处理的基本思路和实现方法

（一）模糊处理

（1）读取图像信息

在构造函数中便通过 Load_RGB 函数从图像中读取图片各个像素点的 RGB 值信息，将图像 RGB 值读取到 RGB_Array 中

```
void PixImage::Load_RGB()
{
    for (int i = 0; i < ImageHeight; i++)
    {
        for (int j = 0; j < ImageWidth; j++)
        {
            CvScalar rgb = Get_RGB(i, j);
            RGB_Array[i][j][0] = rgb.val[0];
            RGB_Array[i][j][1] = rgb.val[1];
            RGB_Array[i][j][2] = rgb.val[2];
        }
    }
}
```

其中用到了 Get_RGB 函数，获取该位置 RGB 值。根据其形参的不同，有一个函数重载。其中 CvScalar 是 opencv 提供的一个结构体类型可以利用 cvGet2D 返回该点的 RGB 值，用返回类型为 double 的重载函数可以根据 k 的取值返回特定的 R/G/B 值。

```
CvScalar PixImage::Get_RGB(int i, int j)
{
    return cvGet2D(img, i, j);
}

double PixImage::Get_RGB(int i, int j, int k)
{
    return cvGet2D(img, i, j).val[k];
}
```


(2) 处理图像

图片模糊处理的主函数是 `Blurring` 函数
其中 `n` 是均值的范围，`m` 为迭代的次数

```
void PixImage::Blurring(int n, int m) //n:均值像素所取的范围n方,m:处理的迭代次数
{
    if (!img)
        return;
    double *rgb = new double[3];

    while (m > 0) //将均值滤波器迭代m次
    {
        for (int i = 0; i < ImageHeight; i++)
        {
            for (int j = 0; j < ImageWidth; j++)
            {
                Sum_Average_RGB(rgb, i, j, n); //分别获取第i行第j列像素周围n*n个像素的rgb平均值
                RGB_Array[i][j][0] = rgb[0];
                RGB_Array[i][j][1] = rgb[1];
                RGB_Array[i][j][2] = rgb[2];
            }
        }
        Save_RGB(); //调用Save_RGB方法更新图像
        m--;
    }
}
```

均值滤波器的基本原理是：对于任意一个像素，使用其周围 $n \times n$ 像素范围内的平均值来置换该像素值。因此我们使用 `Sum_Average_RGB` 函数来计算其周围 $n \times n$ 像素范围内的平均值，这里处理上的细节中定义了一个 `double *rgb = new double[3]`，用指针直接的数组直接将平均值记录在 `rgb` 中。

处理时对于超出边缘的像素点也要排除，使得平均得到的是真实值。

```
void PixImage::Sum_Average_RGB(double * rgb, int ci, int cj, int n)
{
    for (int i = sy; i <= ey; i++) {
        for (int j = sx; j <= ex; j++) {
            if (i < 0 || i >= ImageHeight || j < 0 || j >= ImageWidth)
            {
                cnt++; //记录超出图像范围的位置的数量
                continue;
            }
            r += Get_RGB(i, j, 0);
            g += Get_RGB(i, j, 1);
            b += Get_RGB(i, j, 2);
        }
    }
    int div = n*n - cnt; //获取实际计算在内的位置数量
                        //求取平均值
    rgb[0] = r / div;
    rgb[1] = g / div;
    rgb[2] = b / div;
}
```

(3) 保存图像信息

图片的保存用的是 `Save_Image` 函数，如果未指定路径则默认保存在当前目录，否则保存在制定的路径中。

```
void PixImage::Save_Image()
{
    if (!img)
        return;
    Save_RGB(); //调用Save_RGB的方法将RGB_Array中的值全部更新到实际图像中
    cvSaveImage(ImageName, img); //调用opencv自带的cvSaveImage将图像保存至相应位置
}

void PixImage::Save_Image(char * savePath)
{
    if (!img)
        return;
    Save_RGB();

    char *s = new char[strlen(savePath) + strlen(ImageName) + 1];

    strcpy(s, savePath);
    strcat(s, ImageName);
    cvSaveImage(s, img);
}
```

保存图像的原理是，通过 `Save_RGB`，调用 `Set_RGB`，把修改后的 `RGB_Array` 赋值给 `*img` 指向的图片，然后调用 `cvSaveImage` 保存图片。

```
void PixImage::Save_RGB()
{
    for (int i = 0; i < ImageHeight; i++)
    {
        for (int j = 0; j < ImageWidth; j++)
        {
            Set_RGB(i, j, RGB_Array[i][j]);
        }
    }
}
```

```
void PixImage::Set_RGB(int i, int j, double * value)
{
    CvScalar rgb; //声明一个CvScalar向量,并分别取出rgb值然后赋值
    rgb.val[0] = value[0];
    rgb.val[1] = value[1];
    rgb.val[2] = value[2];
    cvSet2D(img, i, j, rgb); //opencv自带方法,用以设置img图像中第i行第j列处的像素为
}

void PixImage::Set_RGB(int i, int j, CvScalar value)
{
    cvSet2D(img, i, j, value);
}
```

（二）边缘化检测

（1）读取图像信息

在构造函数中便通过 Load_Gray 函数从图像中读取图片各个像素点的灰度值信息，将图像灰度值读取到 Gray_Array 中

```
void PixImage::Load_Gray(bool isDefault)
{
    for (int i = 0; i < ImageHeight; i++)
    {
        for (int j = 0; j < ImageWidth; j++)
        {
            Gray_Array[i][j] = getGray(i, j, isDefault);
        }
    }
}
```

其中用到了 Get_Gray 函数，获取该位置灰度值。根据其 isDefault 的不同，选择是采用 opencv 自带的灰度获取方式还是根据 RGB 值根据公式来计算，为了保险起见，在这次处理中都使用 opencv 自带的函数读取灰度值。

```
double PixImage::getGray(int i, int j, bool isDefault) //isDefault:是否以opencv默认的灰度获取方法获取图像灰度值
{
    if (!isDefault)
        return (RGB_Array[i][j][0] * 299 + RGB_Array[i][j][1] * 587 + RGB_Array[i][j][2] * 114 + 500) / 1000; //
    else
        return cvGet2D(imgGray, i, j).val[0]; //opencv以isColor=0的方式读取到的灰度值
}
```

（2）处理图像

图片模糊处理的主函数是 Sobel 函数
其中 scale 为 scale 系数

```
void PixImage::Sobel(double scale)//scale:scale系数
{
    if (!img)
        return;

    double t = Sobel_Thresh(scale); //根据给出的公式计算sobel阈值

    for (int i = 1; i < ImageHeight - 1; i++)
    {
        for (int j = 1; j < ImageWidth - 1; j++)
        {
            if (Gxy(i, j) >= t)//若梯度大于等于阈值，则置为255
                Gray_Array[i][j] = 255;
            else //若梯度小于阈值，则置为0
                Gray_Array[i][j] = 0;
        }
    }
}
```

其原理是

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。

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

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。

索贝尔边缘检测中，首先利用 Sobel_Thresh 根据给出的公式计算 sobel 阈值

```
double PixImage::Sobel_Thresh(double scale)
{
    double mean = 0;
    //遍历图像的每一个位置的灰度值, 求取灰度平均值
    for (int i = 0; i < ImageHeight; i++)
    {
        for (int j = 0; j < ImageWidth; j++)
        {
            mean += Gray_Array[i][j];
        }
    }
    mean /= (ImageHeight*ImageWidth);
    return sqrt(scale*mean); //根据公式计算sobel阈值
}
```

然后通过公式计算梯度:

```
double PixImage::Gxy(int ci, int cj)
{
    double Gx = 0, Gy = 0; //Gx Gy初始化为0, 用以记录x和y两个方向的梯度值
    int li, lj;
    for (int i = -1; i <= 1; i++)
    {
        for (int j = -1; j <= 1; j++)
        {
            li = i + ci, lj = j + cj;
            if (li < 0 || li >= ImageHeight || lj <= 0 || lj >= ImageWidth) //记录超出图像范围的位置的数量
                continue;
            Gx += Sobel_OperatorX[i + 1][j + 1] * getGray(li, lj, grayDefault); //两矩阵对应位置相乘并求和
            Gy += Sobel_OperatorY[i + 1][j + 1] * getGray(li, lj, grayDefault); //两矩阵对应位置相乘并求和
        }
    }
    return sqrt(Gx*Gx + Gy*Gy); //返回最终的梯度估计值
}
```

最后将梯度与阈值做比较，若大于阈值则置 255，否则置 0。由于索贝尔算子是 3*3 的矩阵，因此这里循环从 1 开始，对边缘的情况进行忽略。

```
for (int i = 1; i < ImageHeight - 1; i++)
{
    for (int j = 1; j < ImageWidth - 1; j++)
    {
        if (Gxy(i, j) >= t) //若梯度大于等于阈值，则置为255
            Gray_Array[i][j] = 255;
        else //若梯度小于阈值，则置为0
            Gray_Array[i][j] = 0;
    }
}
```

(3) 保存图像信息

图片的保存用的是 `Save_ImageGray` 函数，如果未指定路径则默认保存在当前目录，否则保存在制定的路径中。

```
void PixImage::Save_ImageGray()
{
    if (!img) return;
    Save_Gray(); //通过Save_Gray方法将Gray_Array中的值更新到实际图像中
    cvSaveImage(ImageName, imgGray); //调用cvSaveImage方法保存图像至相应目录
}

void PixImage::Save_ImageGray(char * savePath)
{
    if (!img) return;
    Save_Gray();
    char *s = new char[strlen(savePath) + strlen(ImageName) + 1];
    strcpy(s, savePath);
    strcat(s, ImageName);
    cvSaveImage(s, imgGray);
}
```

保存图像的原理是，通过 `Save_Gray`，调用 `Set_Gray`，把修改后的 `GrayArray` 赋值给 `*imgGray` 指向的图片，然后调用 `cvSaveImage` 保存图片。

```
void PixImage::Save_Gray()
{
    for (int i = 0; i < ImageHeight; i++)
    {
        for (int j = 0; j < ImageWidth; j++)
        {
            Set_Gray(i, j, Gray_Array[i][j]);
        }
    }
}
```

```
void PixImage::Set_Gray(int i, int j, double value)
{
    CvScalar s;
    s.val[0] = value;
    s.val[1] = s.val[2] = 0;
    cvSet2D(imgGray, i, j, s);
}
```

3. Image.cpp

Image.cpp 是 main 函数所在的文件，用于与用户交互。
用户根据从键盘键入数据，选择想要实现的功能。

```
int main()
{
    PixImage img("4.jpg", true);
    int swi = 0;
    int m = 0, n = 0, scale = 0;
    cout << "请输入功能选择: 0代表模糊, 1代表索贝尔: " << endl;
    while (cin >> swi)
    {
        if (swi == 0)
        {
            cout << "选择模糊功能:" << endl;
            cout << "请依次输入模糊范围 n 和迭代次数 m: " << endl;
            cin >> n >> m;
            img.Blurring(n, m);
            img.Save_Image("处理后\\");
            cout << "模糊成功" << endl;
        }
        else if (swi == 1)
        {
            cout << "边缘化检测功能:" << endl;
            cout << "请输入索贝尔系数scale: " << endl;
            cin >> scale;
            img.Sobel(scale);
            img.Save_ImageGray("处理后\\");
            cout << "边缘化检测成功" << endl;
        }
    }
}
```

四、最终项目达到的效果

(1) 网络爬虫

选取 <http://www.gamersky.com/> 为起始网站

设置最大爬取图片数为 10050（有些图片可能下载失败）

设置最大爬取网址数为 1000

```
import queue
from lib import imaScript

htmlStart="http://www.gamersky.com/"

script = imaScript.imaScript(htmlStart,'image\\',imaMaxCount=10050,htmlMaxCount=1000)
script.run(66)
```

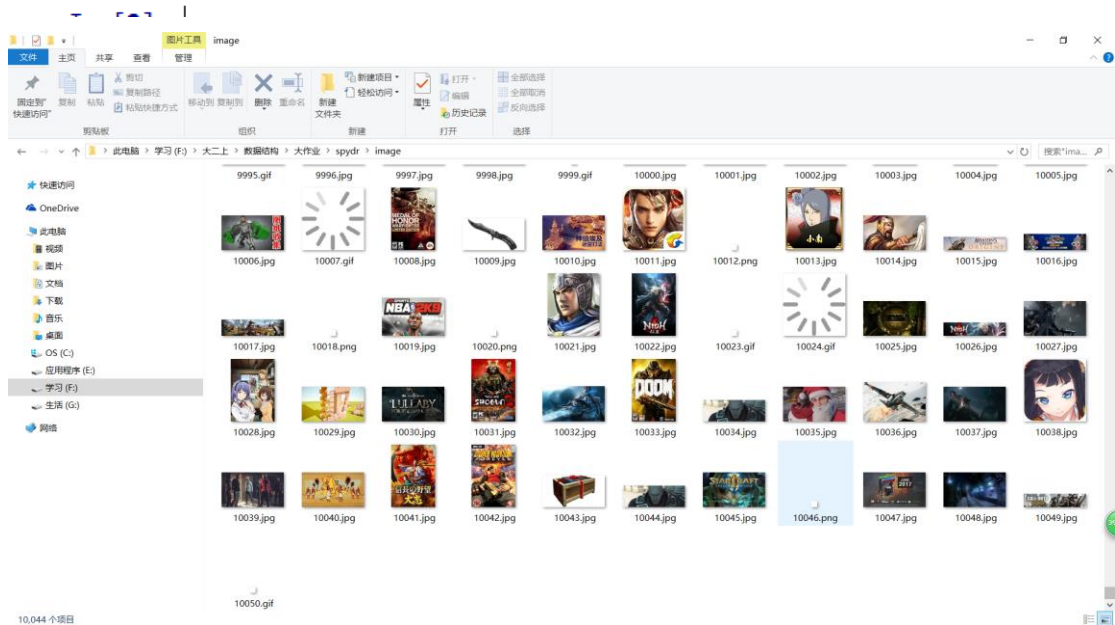
运行爬虫



```
Console 1/A
捕获到过滤后的网址http://www.gamersky.com/news/
201801/998201.shtml
捕获到过滤后的网址http://www.gamersky.com/news/
201801/998203.shtml
捕获到过滤后的网址http://www.gamersky.com/news/
201801/998076.shtml
捕获到过滤后的网址http://www.gamersky.com/news/
201801/998154.shtml
捕获到过滤后的网址http://acg.gamersky.com/news/
201801/997917.shtml
捕获到过滤后的网址http://www.gamersky.com/news/
201801/998046.shtml
捕获到过滤后的网址http://ol.gamersky.com/news/201801/997942.s
捕获到过滤后的网址http://www.gamersky.com/news/
201801/998029.shtml
捕获到过滤后的网址http://www.gamersky.com/news/
爬取网站:http://www.gamersky.com/z/Sangokushi12/
爬取网站:http://www.gamersky.com/z/Homefront/
爬取网站:http://www.gamersky.com/z/moh/
爬取网站:http://www.gamersky.com/z/sanguo7/
爬取网站:http://www.gamersky.com/z/homeworld2/
爬取网站:http://www.gamersky.com/z/starcraft/
爬取网站:http://www.gamersky.com/z/sangokushi10/
爬取网站:http://www.gamersky.com/z/ageofempires2/
爬取网站:http://www.gamersky.com/z/war3/
爬取网站:http://www.gamersky.com/z/taikorisshiden5/
爬取网站:http://www.gamersky.com/z/re7/
爬取网站:http://www.gamersky.com/z/doom/
爬取网站:http://www.gamersky.com/z/deadbydaylight/
爬取网站:http://www.gamersky.com/z/soma/
爬取网站:http://www.gamersky.com/z/bloodborne/
爬取网站:http://ku.gamersky.com/2013/Outlast/
爬取网站:http://www.gamersky.com/z/deadspace3/
```


最终能够实现下载至少 10000 张图片的功能

爬取网站:<http://www.gamersky.com/z/re7/>
爬取网站:<http://www.gamersky.com/z/deadrising4/>
爬取网站:<http://www.gamersky.com/z/doom/>
爬取网站:<http://www.gamersky.com/z/deadbydaylight/>
爬取网站:<http://www.gamersky.com/z/untildawn/>
爬取网站:<http://www.gamersky.com/z/f13game/>
爬取网站:<http://www.gamersky.com/z/salt/>
爬取网站:<http://www.gamersky.com/z/freddy4/>
爬取网站:<http://www.gamersky.com/z/thenonarygames/>
爬取网站:<http://www.gamersky.com/z/dyinglighttf/>
爬虫结束,总耗时311秒,信息如下;
爬取网站1000个,其中6个爬取失败;
其中爬取图片10050张,其中6张下载失败;
原始图片已全部保存至目录image\下;



(2) 图像处理

1. 图片模糊

```
请输入功能选择：0代表模糊，1代表索贝尔：  
0  
选择模糊功能：  
请依次输入模糊范围 n 和迭代次数 m：  
3 5  
模糊成功  
_
```



2.边缘检测

```
1
边缘化检测功能:
请输入索贝尔系数scale:
100
边缘化检测成功
```



可以看出实验的结果符合预期和要求。

五、附录



原图



模糊处理

在处理灰太狼图像时，发现很难达到样例的效果

查阅资料后发现，对于这类有白色噪点的图片，用中值滤波器可能好一些，于是试着把程序改了一下。其中用了 `vector` 和 `<algorithm>` 里的 `sort` 函数，但是这样一次处理图片需要很长时间，因为每次的排序的时间还是蛮长的。估计是对于这个中值的获取，有着更为简单的方法。

```
vector<double> rvc;
vector<double> gvc;
vector<double> bvc;

for (int i = sy; i <= ey; i++) {
    for (int j = sx; j <= ex; j++) {
        if (i < 0 || i >= ImageHeight || j < 0 || j >= ImageWidth)
        {
            cnt++; //记录超出图像范围的位置的数量
            continue;
        }
        rvc.push_back(Get_RGB(i, j, 0));
        gvc.push_back(Get_RGB(i, j, 1));
        bvc.push_back(Get_RGB(i, j, 2));
    }
}

int div = n*n - cnt; //获取实际计算在内的位置数量

sort(rvc.begin(), rvc.end());
sort(gvc.begin(), gvc.end());
sort(bvc.begin(), bvc.end());
```