目录

1 实验介绍

1.1 实验内容

本次报告涵盖了第二次和第三次实验, 其内容分别如下:

1.1.1 实验二

1. 使用自己的账号模拟登陆 BBS 后, 修改个人说明档 (修改 bbs_set_sample.py).
2. 修改 crawler_sample.py 中的 union_bfs 函数, 完成 BFS 搜索.
3. 修改 crawler_sample.py 中的 crawl 函数, 返回图的结构.
4. 进一步修改函数, 完成网页爬虫 (将练习 2,3 中修改的部分加入 crawler.py, 注意做异常处理).

1.1.2 实验三

1. 实现 BloomFilter, 并设计一个实验统计你的 BloomFilter 的错误率 (false positive rate).
2. 实现一个并行的爬虫, 将实验二中的 crawler.py 改为并行化实现.

1.2 实验环境

操作系统: Ubuntu 18.04 LTS
Python 版本: 3.7

1.3 所需技能

实验二主要涉及运用 Python 模拟网络请求, 这需要具备一定的网络知识. 而其后对网页内容进行解析, 并根据不同策略进行爬取, 则需要基本的算法知识.

实验三涉及到 Hash Function, Hash Table, BitMap 以及 Bloom Filter 的原理以及实现. 在编写多线程并行程序时, 还需要对操作系统, 阻塞以及锁等概念有些了解.

2 实验过程

2.1 Basic Crawler

2.1.1 BBS 修改个人说明档

该练习主要在于了解用 POST 请求提交数据的流程, 较为简单.

实验提供的指导是用 urllib 库来模拟网络请求,cookielib 库来操作 cookie, 操作较为繁琐. 事实上, 由于本练习仅需要保持登录状态, 并不需要涉及直接操作 cookie 的细节, 因此可以使用 requests 库提供的 Session() 对象. 如下:

```
def bbs_set(id, pw, text):

    s = requests.Session()
    login_data = {
        "id": id,
        "pw": pw,
        "submit": "login",
    }
    login_url = "https://bbs.sjtu.edu.cn/bbslogin"
    login_request = s.post(login_url, login_data)
    assert login_request.status_code == 200, "Failed to log in."
    update_data = {
        "text": text,
        "type": "update",
    }
    update_url = "https://bbs.sjtu.edu.cn/bbsplan"
    update_request = s.post(update_url, update_data)
    content = s.get('https://bbs.sjtu.edu.cn/bbsplan').content
    soup = BeautifulSoup(content, features="html.parser")
    print(str(soup.find('textarea').string).strip())
```

值得注意的是对提交本文的编码, 需要进行 GBK 编码.

2.1.2 BFS

由于此处未考虑运行的效率,BFS 函数的实现相当简单:

```
def union_bfs(a, b):
    for e in b:
        if e not in a:
            a.insert(0, e)
```

2.1.3 爬取与解析页面

在爬取页面时, 为了避免程序中途因为网络原因而崩溃, 需要捕获异常以增强健壮性:

```
def get_page(page):
    try:
        r = requests.get(page, timeout=10)
    except:
        return None
    content = r.content
    return content
```

而在解析页面时, 可以复用实验一时所写的 `BaseParser` 类, 而无需重复编码:

```
from parsers import BaseParser
def get_all_links(content, page):
    parser = BaseParser()
    url_set = parser.parse_url(content, page)
    links = [u for u in url_set]
    return links
```

2.1.4 网页编码自动检测

在实验一的报告中, 我将网页编码的问题暂时搁置. 而在这次实验中, 由于要爬取的目标不再是单一的网站, 而是互联网上各式各样的网站, 其编码方式各异, 网页编码的问题便无法回避了.

`add_page_to_folder` 函数将爬取的网页保存到本地文件 (UTF-8 编码), 其间涉及到两次编码解码: 一是对网页内容解码, 二是在写入文件时编码. 第二步并无大碍, 但第一步, 如何对不同编码的网页解码呢?

事实上,Python 提供了 chardet 库专门用以检测文本的编码, 只需在解码前先检测出可能的编码, 便可以从容应对大多数的情况. 至于少数检测不出编码方式的网页文本, 只能按照默认的 UTF-8 来处理, 并忽略解码时的错误. 如下:

```
import chardet

def add_page_to_folder(page, content):
    #some code omitted
    #...
    f = open(os.path.join(folder, filename), 'w', encoding="utf8")
    encode = chardet.detect(content)
    if not encode["encoding"]:
        print("Failed to detect encoding.({url},{encode})".format(url=page,
        ↪  encode=encode))
    f.write(content.decode(encode["encoding"] or "utf8", errors="ignore"))
    f.close()
```

2.2 Bloom Filter

2.2.1 Hash Function

网络上有许多种公开可用的哈希函数, 像本次实验提供的 General Purpose Hash Function Algorithms Library 中就已有 10 余种哈希函数, 除此之外, 更是有诸如 MD5,SHA1 等常用于加密的哈希函数.

对于 Bloom Filter, 哈希函数的选用十分关键, 关涉到运行的效率与检测的正确性. 考虑到我们的过滤器仅仅用于爬虫的网址去重, 相较于安全性, 更注重运行的速度.

那么, 如何找到一个高效率的哈希函数呢? 诚然, 可以从算法角度进行分析, 但既然我们手中已有现成的代码实现, 设计一个简单进行测试是最直接的做法. 为此, 我编写了 HashTest.py, 用以比较不同的哈希函数的运行效率:

```
from GeneralHashFunctions import *
import mmh3
import random
import time
from string import ascii_letters as letters

def test_hash_efficiency(hash_function, test_strings):
    start_time = time.time()
    for s in test_strings:
        hash_function(s)
    finish_time = time.time()
    return finish_time - start_time

test_strings = ["".join(random.sample(letters, 32)) for i in range(100000)]
hash_functions = [hash, mmh3.hash, JSHash, PJWHash, ELFHash, BKDRHash,
↪  SDBMHash, DJBHash, DEKHash, BPHash, FNVHash, APHash]
result = []

for f in hash_functions:
    result.append((f.__name__, test_hash_efficiency(f, test_strings)))
    print("{0:<9}: {1:.8f}s".format(*result[-1]))
print("*********Result Sorted By Time**********")
result.sort(key=lambda x: x[1])
for r in result:
    print("{0:<9}: {1:.8f}s({2:.3f}x)".format(*r, r[1]/result[0][1]))
```

随机生成了 100000 个长度为 32 的字符串, 然后对其用不同的哈希函数进行散列处理, 比较运行时间, 最终得到的结果如下:

```
*********Result Sorted By Time**********
hash     : 0.00897479s(1.000x)  # Built-in Hash
hash     : 0.02493358s(2.778x)  # mmh3.hash
BPHash   : 0.73630977s(82.042x)
BKDRHash : 0.83501196s(93.040x)
DJBHash  : 0.88225102s(98.303x)
FNVHash  : 0.99035144s(110.348x)
DEKHash  : 1.01107359s(112.657x)
SDBMHash : 1.23072314s(137.131x)
PJWHash  : 1.63085604s(181.715x)
JSHash   : 2.06598854s(230.199x)
APHash   : 2.06879592s(230.512x)
ELFHash  : 2.11861658s(236.063x)
```

可见, 从运行效率上来说, 不同哈希函数的差距巨大 (相差最大 236 倍), 其中,Python 自带的 `hash()` 最优, murmurhash3 紧随其后. 但是, 考虑到在 Bloom Filter 中需要对同一个哈希函数加以不同的种子 (当然也可以用不同的哈希函数, 但那样较为繁琐), 而 Python 自带的 `hash()` 并不支持这么做, 故最终我选择了 murmurhash3, 近年兴起的一种高效率哈希函数.

2.2.2 Bit Array

在实验提供的 Python 中, 已实现了 `Bitarray` 类, 但由于其编写于 Python 2.x 环境下, 并且存在一个严重的 Bug, 需要对其进行一些修正. 修改后的 `Bitarray` 类如下:

```python
class Bitarray:
    def __init__(self, size):
        """ Create a bit array of a specific size """
        self.size = size
        self.bitarray = bytearray(size//8 + 1) # Attention Here!!!

    def set(self, n):
        """ Sets the nth element of the bitarray """
        index = n // 8
        position = n % 8
        self.bitarray[index] = self.bitarray[index] | 1 << (7 - position)

    def get(self, n):
        """ Gets the nth element of the bitarray """
        index = n // 8
        position = n % 8
        return (self.bitarray[index] & (1 << (7 - position))) > 0
```

需要注意的是, 原程序对 `bytearray` 的大小设置有误, 运行时有概率导致下标越界.

2.2.3 BloomFilter 类实现

对于哈希函数个数 k、位数组大小 m、加入的字符串数量 n, 根据一定的数学推导, 可知当 k = ln(2) * m/n 时出错的概率是最小的. 而设定 m = 20 * n, 则出错的概率为 0.0000889, 对于爬虫而言完全可以接受.

因此, 在类初始化时, 既允许同时指定 n,m,k, 也可以允许只指定 n, 其余值按照上述规则计算.

而在对哈希函数列表初始化时, 用到了 Python 具有的函数式语言特点, 通过自行实现的类的私有方法 `__hash_function_wrapper` 来对哈希函数进行处理, 方便之后的调用:

```
import mmh3     # third-party library for murmurhash3
from functools import partial
import math

class BloomFilter(object):

    def __hash_function_wrapper(self, hash_func, seed, mod):
        def f(s):
            return partial(hash_func, seed=seed)(s) % mod
        return f

    def __init__(self, potential_num, bitarray_size=None, hash_num=None):
        self.potential_num = potential_num
        self.bitarray_size = bitarray_size or 20 * potential_num
        self.hash_num = hash_num or
        ↪  int(math.log(2)*self.bitarray_size/self.potential_num)
        self.hash_functions = [self.__hash_function_wrapper(
            mmh3.hash, seed=i, mod=self.bitarray_size
            ) for i in range(self.hash_num)]
        self.bitarray = Bitarray(self.bitarray_size)
        self.count = 0
```

在初始化完成之后,Bloom Filter 所需实现的操作为加入元素 `add` 与检查元素是否已加入 `check`, 至于 `__len__` 与 `parameters`, 是为了方便外界获知 Bloom Filter 的状态.

```
def add(self, string):
    for hash_func in self.hash_functions:
        self.bitarray.set(hash_func(string))
    self.count += 1

def check(self, string):
    return all(self.bitarray.get(hash_func(string))
               for hash_func in self.hash_functions)
```

```
def __len__(self):
    return self.count

def parameters(self):
    return self.potential_num, self.bitarray_size, self.hash_num
```

2.3 Multi-threading

在原有的 crawler.py 程序基础上, 为了使其能够以多线程的方式并发爬取, 需要修改 `crawl` 函数.

为了不引入过多的全局变量污染程序, 利用 Python 的函数式特性: 闭包, 我在 `crawl` 函数内部定义了 `crawl_single_page` 函数作为多线程的 worker. 大致如下:

```
def crawl(seed, max_page, thread_num=4):

    def crawl_single_page():
        pass
    url_queue = Queue()
    lock = Lock()
    crawled = BloomFilter(max_page)
    graph = {}
    global count
    count = 0
    url_queue.put(seed)
    for i in range(thread_num):
        t = Thread(target=crawl_single_page)
        t.setDaemon(True)
        t.start()
    url_queue.join()
    return graph, crawled
```

需要注意的是, 不同于实验二, 这里的 `crawled` 不是列表, 而是我们所实现的 Bloom Filter.

至于 `crawl_single_page` 的具体实现, 与实验二基本无异, 不同之处在于锁的引入, 避免多线程导致变量读写的异常.

```
def crawl_single_page():
    global count
    while count < max_page:
        url = url_queue.get()
        if not crawled.check(url):
            print("#{0:<4} {1}".format(count+1, url))
            content = get_page(url)
```

```
            outlinks = get_all_links(content, url)
            with lock:
                if count >= max_page: #attention here
                    print("Drop!")
                    break
                graph[url] = outlinks
                for l in outlinks:
                    url_queue.put(l)
                crawled.add(url)
                count += 1
            add_page_to_folder(url, content)
        url_queue.task_done()
```

留意以上代码, 为了确保最终爬取的页面数量与设定一致, 在爬取完页面后, 需要丢弃由于多线程导致的冗余页面 (如目标是 100 个页面, 但在爬取完 99 个页面后,4 个线程从队列中取出一个页面进行爬取).

然而, 由于我们在爬取到足够数量页面之后就停止爬取, 队列中往往会存留尚未爬取的页面, 这就导致了在线程结束后依旧阻塞, 程序无法终止. 因此, 在主程序中, 应对每个 Thread 进行 join 操作, 而非对 Queue.

```
threads = []
for i in range(thread_num):
    t = Thread(target=crawl_single_page)
    threads.append(t)
    t.setDaemon(True)
    t.start()
for t in threads:
    t.join()
```

3 总结与分析

经过实验二与实验三, 相较于实验一中简单的爬虫, 已有不少提升, 如 Bloom Filter 和多线程对运行效率的显著提升, 以及许多诸如网页编码自动检测等细节的优化. 在进行实验的过程中, 课程提供的指引起到了很大的作用, 但在某些情形下, 还是需要自己查找资料并动手实践来解决问题, 这样让我学到了不少新的内容.

此外, 每次实现一个功能, 总是有不止一种可供选择的方案. 有的方案简单而粗暴, 却缺乏美感 (譬如引入一大堆全局变量), 我总是试图避免出现这样的情况, 因而或许写出了一些不太常规的代码.

虽然多费了时间, 但挺有趣的.