Experiment 2/3

517030910356 谢知晖

目录

1 实验准备

1.1 实验环境

本次实验环境依旧为 Ubuntu 平台 (版本号 18.04) 下的 Python(版本号 2.7)。

1.2 实验目的

实验二中，我们在 Python 上完成了模拟浏览器发送 POST 请求的操作，对爬虫时怎么进行请求有了初步的认识。其次，了解爬虫基本概念，掌握两种搜索策略, 为之后的爬虫做准备。最后，复现基础的网页爬虫，进一步熟悉爬虫的流程操作。

实验三中，为了进一步优化我们的爬虫系统，我们探讨了 BloomFIlter 以及并发编程技术的运用，分别提高了爬虫系统中遍历已爬网页和爬取过程的效率。

1.3 实验原理

我们主要介绍实验中运用在爬虫系统中的三种技术的实现原理。

1.3.1 BFS/DFS

BFS(广度优先搜索策略)/DFS(深度优先搜索策略)，是爬虫抓取时所用的两种抓取策略。针对爬虫系统的不同使用目的，我们可以选择相应的策略来实现爬虫。

由于爬虫的搜索基于队列这一结构，两种策略的主要区别便在于爬取得到的网页在队列中的存放顺序。

BFS 广度优先搜索策略沿着树的宽度遍历节点。它每次将爬取到的链接放在待爬取队列之首。每次爬取队尾的网页，即先被爬取的是队列中先被放入的网页，且拥有同个父节点的网页将依次被爬取，这样即保证了只有爬完同一层的网页才会开始爬下一层。

DFS 与广度优先搜索策略不同，深度优先搜索策略沿着树的深度遍历树的节点。它将爬取的链接放在队尾，这样只有当当前节点下的所有边都被探寻过，爬虫才会开始爬取同一层的其他节点。

1.3.2 BloomFilter

BloomFilter 是为了解决空间占用过大而诞生的一种数据结构。它能够以一定的错误率为代价，有效地检查一个集合中是否包含某个元素，将检索过程的时间复杂度降为 O(1)。并且，我们可以通过控制相应的条件以降低错误率。这无疑为我们的爬取过程提供了很大的便利。

它的实现原理如下：首先我们为集合配备一个唯一的有 m 位的 BitSet，并将所有位设为 0。对于每一个待存储的 keyword，我们使用 k 个 hash function 来映射到 BitSet 中相应的位上，相应的位置 1。这样，当我们存储了所有 n 个 keyword 后，我们得到了一个部分位为 1、其余位为 0 的 BitSet。在之后的检查过程中，我们只需对待检查的输入依次计算 k 个 hash function 的值。只要有一个值对应的位为 0，我们即可断定该输入不在集合中，否则我们能预测其有很大可能在集合中。

并且通过数学上的计算 (BloomFilters-the math)，我们能够确定当 $k = \frac{ln2*m}{n}$ 时出错概率最小 [1]。

1.3.3 并发编程

并发编程是一种为了充分利用计算机线程的一种技术。其核心内容为计算机能够以任何顺序来执行的独立计算，从串行代码中提取出并发工作可以分配到多个线程上 (或者相互协作的进程上)，并且可以再多核处理器中的任何一个核上运行 (或者在单核处理器上运行，此时需要将不同的独立计算换入/换出处理器核，从而形成一种并行执行的假象)[2]。

值得一提的是，并发与并行并不相同。如果某个系统支持两个或者多个动作同时存在，那么这个系统就是一个并发系统。如果某个系统支持两个或者多个动作同时执行，那么这个系统就是一个并行系统。两者定义间的关键差异在于" 存在" 一词。

在本次实验中，我们主要利用 Python 中的 threading 模块实现。我们将在之后详细讨论。

2 实验过程

2.1 Exp2-1

2.1.1 实验步骤

bbs-set 函数'bbs-set' 接受三个参数 id, password, text，没有返回值。

我们首先利用"cookielib" 模块初始化我们的 Cookie，并将初始化后的 Cookie 加入到 opener 中。这样，我们在实验开始时能够注销可能存在的账号。

```
    cj = cookielib.CookieJar()
    opener = urllib2.build_opener(urllib2.HTTPCookieProcessor(cj))
    urllib2.install_opener(opener)
```

然后，我们创建登录信息。它是通过"urllib" 中的函数'urlencode'，将一个字典编码成一个 URL 查询字符串。通过模块"urllib2" 中的相应函数向 BBS 的登录页面发送登录请求。

```
    postdata1 = urllib.urlencode({
        'id': id,
        'pw': pw,
        'submit': 'login'
    })
    req1 = urllib2.Request(url='https://bbs.sjtu.edu.cn/bbslogin', data=postdata1)
    urllib2.urlopen(req1)
```

同样的，我们将需要修改的个人说明档信息进行编码，并发送请求。

```
    postdata2 = urllib.urlencode({
    'type': 'update',
    'text': text
})
    req2 = urllib2.Request(url='https://bbs.sjtu.edu.cn/bbsplan', data=postdata2)
    urllib2.urlopen(req2)
```

最后，我们可以利用 BeautifulSoup 来解析个人主页，查找相应的'textarea' 内容，以判断个人说明档是否被修改。

```
    content = urllib2.urlopen('https://bbs.sjtu.edu.cn/bbsplan').read()
    soup = BeautifulSoup(content, features='html.parser')
    print str(soup.find('textarea').string.strip().encode('utf-8'))
```

main 值得一提的是，在 main 函数中，我们对输入的中文文本做了 decode 和 encode 的操作。

```
id = sys.argv[1]
pw = sys.argv[2]
text = sys.argv[3].decode('utf-8').encode('gbk')

bbs_set(id, pw, text)
```

我们已将 python 的编码方式设为 utf-8，为什么还需要重新将中文字符以 gbk 方式译码呢？通过查阅相关文档，我们得知 urllib 中的 urlencode 方法的默认编码方式为 gbk，因此我们需要先 decode 输入的中文字符，再以 gbk 方式 encode。

2.1.2 实验结果

程序能够通过命令行接收需要修改个人说明档的 BBS 账号、密码以及相应的文本。

2.2 Exp2-2

2.2.1 实验步骤

这次我们只关注函数'union_bfs' 的实现。它接收两个 list，并将 b 中的元素不重复地插在 a 之前。

```
def union_bfs(a,b):
    order = 0
    for i in reversed(b):
        if i in a:
            continue
        else:
            a.insert(order, i)
            order += 1
```

2.2.2 实验结果

该函数能够如要求的那样实现两个 list 的合并。

```
>>> a = [1, 2, 3]
>>> b = [2, 4, 4, 5]
>>> union_bfs(a, b)
>>> a
[5, 4, 1, 2, 3]
>>> b
[2, 4, 4, 5]
```

2.3 Exp2-3

2.3.1 实验步骤

本次练习我们修改爬虫的关键函数'crawl'。该函数接收种子 seed 以及爬取方式 method，并返回图的结构以及爬取过的网页。

首先我们初始化爬虫系统，包括设置种子，并将图的结构及爬取过的网页设为空。

```
tocrawl = [seed]
crawled = []
graph = {}
```

然后我们通过一个 while 循环控制爬取的进行。

```
while tocrawl:
    page = tocrawl.pop()
    if page not in crawled:
        content = get_page(page)
        outlinks = get_all_links(content)

        graph[page] = outlinks

        globals()['union_%s' % method](tocrawl, outlinks)
        crawled.append(page)
```

在循环中，我们首先判断是否已经爬完所有内容。若还有网页待爬取，我们 pop 出 tocrawl 中的网页，并再次判断该网页是否已被爬取过。若没被爬取过，我们便通过函数'get_page' 以及'get_all_links' 得到该网页下的所有 url。

接着，我们只需向添加字典 graph 键值对来实现图的结构，key 为网页，value 为该网页下的所有 url。

最后我们通过调用相应的爬取方式函数来将这些 url 添加到待爬取的网页 list 中，并把当前网页标记为已爬取网页。

2.3.2 实验结果

我们可以利用该函数进行爬取并得到爬取结果。

```
graph_dfs: {'A': ['B', 'C', 'D'], 'C': [], 'B': ['E', 'F'], 'E': ['I', 'J'], 'D': ['G', 'H'], 'G': ['K', 'L'], 'F': [], 'I': [], 'H': [], 'K': []}
crawled_dfs: ['A', 'D', 'H', 'G', 'L', 'K', 'C', 'B', 'F', 'E', 'J', 'I']
graph_bfs: {'A': ['B', 'C', 'D'], 'C': [], 'B': ['E', 'F'], 'E': ['I', 'J'], 'D': ['G', 'H'], 'G': ['K', 'L'], 'F': [], 'I': [], 'H': [], 'K': []}
crawled_bfs: ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L']
```

2.4 Exp2-4

2.4.1 实验步骤

本次我们需要完成整个网页爬虫。为此，我们需要修改三个函数'get_all_links'、'get_page'、'crawl'。

get_all_links 我们利用正则表达式来匹配所有有效格式的 url，并用"urlparse" 中的'urljoin' 来处理其格式。

```
def get_all_links(content, page):
    links = []
    soup = BeautifulSoup(content, features='html.parser')
    for link in soup.findAll('a', {'href': re.compile('^http|^/')}):
        links.append(urlparse.urljoin(page, link['href']))
    return links
```

函数中，我们首先创建一个 BeautifulSoup 对象 soup，并用函数'findAll' 查找属性 href 的值满足特定正则表达式的所有 <a> 标签，并将修正格式后标签的 href 值添加到集合中。

get_page 函数'get_page' 中，我们需要做适当的异常处理，以防止网页无法访问卡死的情况。

函数的输入为网址 page，返回网页内容 content。

```
def get_page(page):
    try:
        req = urllib2.Request(page, None, {'User-agent': 'Custom User Agent'})
        content = urllib2.urlopen(req, timeout=10).read()
    except Exception, err:
        print err, page
        return None
    else:
        return content
```

我们利用 python 的 try/except 语句来完成异常处理。

在 try 语句中，我们将浏览器信息模拟放入 header 发出，并打开网页内容。这里我们设置了函数'urlopen' 的参数'timeout' 为 10s，以防爬取卡死。

而在之后的 except 语句中，我们接收报错信息。如果异常没有抛出，将返回 content。否则 print 出异常内容，并返回 None 值，这有利于我们在之后的 crawl 中进行异常处理。

crawl 和前一练习中的函数相比，我们仅添加了参数 max_page。若爬取的网页数达到要求，即可返回。

```
def crawl(seed, method, max_page):
    tocrawl = [seed]
    crawled = []
    graph = {}
    count = 0

    while tocrawl:
        page = tocrawl.pop()
        if page not in crawled:
```

```
            content = get_page(page)
            if content == None:
                print "Overtime!"
                continue
            else:
                count += 1
                print count, page
                add_page_to_folder(page, content)
                outlinks = get_all_links(content, page)
                globals()['union_%s' % method](tocrawl, outlinks)
                crawled.append(page)

                graph[page] = outlinks

        if count == max_page:
            break
    return graph, crawled
```

2.4.2 实验结果

我们现在能够实现初步的爬虫系统。以爬取知乎网站为例：

2.5 Exp3-1

2.5.1 实验步骤

我们这次通过类 Bitarray 来实现 Bitset，并选用 GeneralHashFunction 中的几个表现较好的函数来实现我们的 BloomFilter。

首先我们确定实现的一些定量。根据实验原理，我们先确定好预想的输入字符串个数为 110000 个，然后挑选出 5 个表现较好的 HashFunction，并由公式最后确定 BitSet 的大小约为 800000。

接着我们定义了三个辅助函数: 'create_random_string'，'add_keyword'，'check'。

create_random_string 利用 python 模块"random" 以及"string"，函数可以生成一个长度在 a 到 b 之间的随机字符串。

```
def create_random_string(a, b):
    size = random.randint(a, b)
    ran_str = ''.join(random.sample(string.ascii_letters + string.digits, size))
    return ran_str
```

add_keyword 此函数用以将 keyword 映射到 BitSet 中。它接受字符串 word，BitSet 本身，以及一个用字符串表示的 HashFunction 的列表，我们将用这个列表中的函数进行 hash。

```
def add_keyword(word, bitset, funcs):
    for func in funcs:
        bitset.set(eval(func)(word) % bitset.get_size())
```

check 函数用来检查所需要检索的字符串是否在 BitSet 中。若返回 False，表示一定不在，否则表示有可能在。它接受的参数与函数'add_keyword' 相同。

```
def add_keyword(word, bitset, funcs):
    for func in funcs:
        bitset.set(eval(func)(word) % bitset.get_size())
```

main 我们已经在全局中导入了"Bitarray" 以及"GeneralHashFunction"。

主函数中，我们首先选择用在 BloomFilter 中的 HashFunctions，并创建一个初始化的 BitSet。

```
funcs = ['BKDRHash', 'RSHash', 'JSHash', 'SDBMHash', 'DEKHash']
bit_obj = Bitarray(800000)
```

接着，我们生成 110000 个不重复的、长度在 1-10 之间的字符串，映射到 BitSet 中。

```
words = []
train_num = 110000
while train_num > 0:
    ran_str = create_random_string(1, 10)
    if ran_str not in words:
        words.append(ran_str)
        train_num -= 1

for word in words:
    add_keyword(word, bit_obj, funcs)
```

最后，为了检测我们的 BloomFilter 的错误率，我们生成 100000 个不在 BitSet 中的字符串，并检验其是否被判断错误。最终计算测试样本的错误率。

```
count = 0
test_num = 100000

while test_num > 0:
    ran_str = create_random_string(1, 10)
    if ran_str not in words:
        train_num -= 1
        if check(ran_str, bit_obj, funcs):
            count += 1

print float(count) / test_num
```

2.5.2 实验结果

最终，我们得到的 BloomFilter 模型的错误率为 0.04 左右。

```
/home/fffffarmer/miniconda2/bin/python /home/fffffarmer/PycharmProjects/Exp3/bloomfilter.py
0.04002

Process finished with exit code 0
```

2.6 Exp3-2

2.6.1 实验步骤

working 这是我们实现并发的主要函数。它指导每一个线程从 Queue 中执行相应的爬取任务。

```
def working(depth):
    global count
    while count < depth:
        page = q.get()
        if varLock.acquire():
            if page not in crawled:
                content = get_page(page)
                if not content:
                    print "Overtime!"
                    continue
                else:
                    count += 1
                    print count, page
                    add_page_to_folder(page, content)
                    outlinks = get_all_links(content, page)
                    for link in outlinks:
                        q.put(link)
                    graph[page] = outlinks
                    crawled.append(page)
            varLock.release()
            q.task_done()
```

我们首先设定循环条件。若已达到预设的爬取深度，则不再爬取。

然后，我们从 Queue 中得到需要爬取的网页，并获取线程锁。这保证了在整个任务执行当中，全局的变量不会被多个线程同时占用，从而防止了重复地爬取工作。

函数的剩余部分和之前类似。我们在最后解除线程锁，并用"Queue" 的'task_done' 方法告知任务结束。

crawl 我们同时改写了'crawl'，使其可以在一定深度下爬取用多线程爬取。

```
def crawl(seed, thread_num, depth):
    q.put(seed)
    for i in range(thread_num):
        t = threading.Thread(target=working, args=(depth, ))
        t.setDaemon(True)
        t.start()
```

```
    q.join()
```

函数首先将种子放入队列中，并设置一定数量的线程。最后，我们堵塞队列，直到所有任务都完成，才结束程序。

2.6.2 实验结果

这一次，我们能以较快速度运行爬虫。在 4 线程左右爬虫的运行时间约为原来的 1/3。爬取每个网站时间约在 0.2s 左右。

3 实验感想

3.1 BloomFilter

在实验中，我们的 BloomFilter 所使用的 HashFunction 仅有 5 个，这局限了我们 BloomFilter 的性能，错误率也比较高。为此，改进的方法为添加其他 HasuFunction，并结合它们的特点进行选择。

3.2 并发编程

我们早已发现并发编程并没有那么容易实现。在处理多线程时，我们不得不考虑到线程之间的冲突问题以及可能遇到的其他错误。

为此，我希望在接下来更深入地了解并发编程的一些原理与技巧，并实现性能调优。

References

[1] Bloom Filters - the math

[2] Clay Breshears. The Art of Concurrency: A Thread Monkey's Guide to Writing Parallel Applications