上机实验二 & 三
实验报告

刘瀚文
517030910294

2018 年 10 月 10 日

目录

Part I

实验二

Chapter 1

实验准备

1.1 背景知识

1.1.1 HTTP 协议

HTTP 协议——HTTP 请求

本质上说，一个 HTTP 请求起始于用户端向 HTTP 服务器发送的一个 URL 请求。一个标准的 HTTP 请求由以下几个部分组成：

<request-line> 请求行，用来说明请求类型、要访问的资源（URL）以及使用的 HTTP 版本

<headers> 头部信息，用来说明服务器要使用的附加信息

<CRLF> 回车换行符 (Carriage-Return Line-Feed)(/r/n)，用于标明头部信息的结束

[<request-body><CRLF>] 主体数据（可不添加）

HTTP 协议——HTML 表单

目标地址（URL）action 标签属性指定了表单提交的目标地址。

发送方式 method 标签属性指定了表单的发送方式，发送方式只有两种：GET 及 POST。

输入类型 type 标签属性指定了输入类型。这里 text 类型为文本输入框，submit 为提交按钮。

HTTP 协议——Python 模拟 GET

urllib2 是 python 自带的网络接口，使用 urllibe2 来模拟 GET 请求。

HTTP 协议——Python 模拟 header

某些网站只让浏览器访问，浏览器信息存放在 headers 中，可以将浏览器信息模拟放入 headers 中发出。

HTTP 协议——Python 模拟 POST

使用 urllib2、cookie，模拟 POST。

1.1.2 爬虫

爬虫是一种按照一定的规则，自动抓取万维网信息的程序或者脚本。

爬虫的基本概念

robots 协议 robots 协议也就是 robots.txt，网站通过 robots 协议告诉搜索引擎哪些页面可以抓取，哪些页面不能抓取。Robots 协议是网站国际互联网界通行的道德规范，其目的是保护网站数据和敏感信息、确保用户个人信息和隐私不被侵犯。因其不是命令，故需要搜索引擎自觉遵守。

抓取策略

广度优先搜索策略 BFS（Breadth First Search） BFS 是从根节点开始，沿着树的宽度遍历树的节点。如果所有节点均被访问，则算法中止。

深度优先搜索策略 DFS（Depth First Search） DFS 是沿着树的深度遍历树的节点，尽可能深的搜索树的分支。当节点 v 的所有边都已被探寻过，搜索将回溯到发现节点 v 的那条边的起始节点。这一过程一直进行到已发现从源节点可达的所有节点为止。

1.2 环境配置准备

注册饮水思源 bbs 账号。

1.3 实验的目的及原理

实验 1：学习使用 Python 模拟 POST，完成登陆 BBS 后，修改个人说明档的操作。

实验 2、3、4：对已有的代码进行调整和完善，实现网络爬虫。

Chapter 2

实验过程

2.1 Practice 1

2.1.1 实验步骤

Cookie 的初期设置

初始化 cookie，并将 cookie 加入 opener。

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

根据发出数据模拟 request-body

使用 Python 模拟 POST 登陆至 bbs。

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

使用 Python 模拟 POST 修改 bbs 签名档。

```
text = 'hellobbs12345'
postdata = urllib.urlencode({
  'text': text,
  'type': 'update'
})

req = urllib2.Request(url='https://bbs.sjtu.edu.cn/bbsplan', data=
                                    postdata)
response = urllib2.urlopen(req)
```

2.1.2 实验结果展示

```
davidstark (David Stark) 共上站 23 次  网龄[5]天  [处女座]
上 次 在: [2018年10月10日20:38:32 星期三] 从 [10.162.146.122] 到本站一游。
目前在线: [讯息器: (打开) 呼叫器: (打开)] 生命力:[30] 文章:[0] 信箱:[  ]
目前 davidstark 状态如下:
Web 浏览 Web 浏览 Web 浏览

hellobbs12345
```

2.2 Practice 2

2.2.1 实验步骤

修改 crawler_sample.py 中的 union_bfs 函数，完成 BFS 搜索。

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

2.2.2 实验结果展示

```
crawled_bfs: ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L']
```

2.3 Practice 3

2.3.1 实验步骤

修改 crawler_sample.py 中的 crawl 函数，返回图的结构。

```
def crawl(seed, method):
tocrawl = [seed]
crawled = []
graph = {}
while tocrawl:
  page = tocrawl.pop()
  if page not in crawled:
    content = get_page(page)
      if graph.get(page) == None:
        graph[page] = []
      for element in content:
        graph[page].append(element)
      outlinks = get_all_links(content)

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

2.3.2 实验结果展示

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

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

2.4 Practice 4

2.4.1 实验步骤

进一步修改函数，完成网页爬虫。

get_all_links 函数

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

return links
```

get_page 函数

```
  def get_page(page):
    try:
      content = urllib2.urlopen(page, timeout=100).read()
    except:
      return None
```

2.4.2 实验结果展示

```
(1, 'http://www.sjtu.edu.cn')
(2, u'http://bbs.sjtu.edu.cn/bbscon?board=TrafficInfo&file=M%2E1273596232%
                                    2EA')
(3, u'http://bbs.sjtu.edu.cn/bbscon?board=TrafficInfo&file=M%2E1273256175%
                                    2EA')(4, u'http://bbs.sjtu.edu.cn/file
                                    /TrafficInfo/1273306347249714.rar')
(5, u'http://bbs.sjtu.edu.cn/bbscon?board=TrafficInfo&file=M%2E1251084684%
                                    2EA')
(6, u'http://bbs.sjtu.edu.cn/bbscon?board=TrafficInfo&file=M%2E1251084678%
                                    2EA')
(7, u'http://bbs.sjtu.edu.cn/bbscon?board=TrafficInfo&file=M%2E1251084672%
                                    2EA')
(8, u'http://bbs.sjtu.edu.cn/bbscon?board=TrafficInfo&file=M%2E1251084690%
                                    2EA')
(9, u'http://page.renren.com/600993176')
(10, u'http://www.miibeian.gov.cn/')
(11, u'http://sq.ccm.gov.cn/ccnt/sczr/service/business/emark/toDetail/
                                    DFB957BAEB8B417882539C9B9F9547E6')
```

Part II

实验三

Chapter 3

实验准备

3.1 背景知识

3.1.1 哈希散列

时间复杂度

如果一个问题的规模是 n,解这一问题的某一算法所需要的时间为 T(n)，它是 n 的某一函数 T(n)，称为这一算法的“时间复杂性”。

常用大 O 表示法表示时间复杂性，在这种描述中使用的基本参数是 n，即问题实例的规模，把复杂性或运行时间表达为 n 的函数。

爬虫代码中的时间复杂度

按 list 方式存储时，判断 page 是否在 crawled 中，程序需要逐个比对 crawled 中的元素，直到找到为止。在实际应用中，已爬序列中的网页通常非常多，遍历方式的判断效率不高。

Hash Table

我们可以将字符串根据规则（hash function）放在 b 个不同的 bucket 里，这样查找时只要在对应的 bucket 中进行查找。这样可以缩小搜索范围。理想情况下可以将搜索范围减小到原来的 1/b。

Hash Function 我们定义 hash function 的输入为字符串 keyword，输出为一个数，该数字告诉我们应该在第几个 bucket 中查找 keyword。

初始化 Hash Table 输入为 bucket 数 b，输出为空的 Hash Table。

从 Hash Table 中得到 bucket 输入为 Hash Table，keyword, 输出为 bucket。keyword 不在 table 中，也应返回 hash function 计算的 bucket。

查找 HashTable 中的元素 输入为 Hash Table，keyword。keyword 在 Table 中输出 True，否则输出 False

添加 keyword 到 HashTable 中 输入为 HashTable，keyword, 将 keyword 添加到 Table 中。

3.1.2 BloomFilter

Bit-Map

建立一个 BitSet，将每个 URL 经过一个哈希函数映射到某一位。

初始状态时，BitSet 是一个包含 m 位的位数组，每一位都置 0。添加元素 x 时，通过哈希函数将 x 映射到 0 m-1 的某个位置 h(x)，将该位置置 1。

查找元素 y 时，对 y 用哈希函数，如果 h(y) 位置为 1，则很有可能 y 属于集合。如果 h(y) 位置为 0，则 y 肯定不属于集合。

实现

BitMap 缺点是冲突概率高，为了降低冲突的概念，Bloom Filter 使用了 k 个哈希函数，而不是一个。假设有 k 个哈希函数，位数组大小为 m，加入 keyword 的数量为 n。

哈希函数的选择 简单的方法是选择一个哈希函数，然后送入 k 个不同的参数。选择 BKDRHsh()。

哈希函数个数 k、位数组大小 m 选择 文献证明：对于给定的 m、n，当 k = ln(2)* m/n 时出错的概率是最小的。

哈希函数个数 k 取 10，位数组大小 m 设为字符串个数 n 的 20 倍时，false positive 发生的概率是 0.0000889 ，这个概率基本能满足爬虫的需求。

3.1.3 并发编程

Queue

Queue 是一个先入先出的队列。
Queue 用来维护任务队列。

Thread

Thread 在爬取的过程中不停的往队列中添加待爬 URL。

Chapter 4

实验过程

4.1 Practice 1：实现 BloomFilter

实现一个简单的 BloomFilter。设计一个实验统计你的 BloomFilter 的错误率 (false positive rate)。

4.1.1 实验步骤

定义随机字符串函数

```
def get_random_string():
  return ''.join(random.sample([chr(i) for i in range(48, 123)], 6))
```

生成 1 基础数据集 2 测试集 3 测试结果集 4 理论结果集

```
strlist = [get_random_string() for i in xrange(1600)]
testlist = [get_random_string() for i in xrange(20230)]
notinit = []
truenotinit = []
```

定义传入 k(k=10) 个参数的哈希函数

```
def BKDRHash(seed, key):
hash = 0
for i in range(len(key)):
  hash = (hash * seed) + ord(key[i])
return hash


seedlist = [31, 131, 1313, 13131, 131313, 1313131, 13131313, 131313131,
                                    1313131313, 13131313131]
```

使用基础数据集进行初始化

```
for i in strlist:
  for j in seedlist:
    num = BKDRHash(j, i) % 32000
    bitarray_obj.set(num)
```

对测试集进行测试，并把结果传入测试结果集

```
for i in testlist:
  l = 1
  for j in seedlist:
    num = BKDRHash(j, i) % 32000
    l *= bitarray_obj.get(num)
  if l == 0:
    notinit.append(i)
```

对基础数据集和测试集进行比对，得出理论结果集

```
for j in testlist:
  count = 1
  for i in strlist:
    if j == i:
      count = 0
  if count == 1:
    truenotinit.append(j)
```

比较测试结果和理论结果，得出正确率数据

```
from __future__ import division
print(len(notinit) / len(truenotinit))
```

4.1.2 实验结果展示

在使用 k=10 个参数的哈希函数的情况下，在使用 1600 个随机生成的 6 位长度字符串进行初始化的情况下：当测试集为 200 时：正确率为 1.0 当测试集为 2000 时：正确率为 0.9985 当测试集为 20000 时：正确率为 0.99825 当测试集为 200000 时：正确率为 0.998175 当测试集为 2000000 时：正确率为 0.998137

4.2 Practice 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。

4.2.1 实验步骤

定义新的 get_page 函数

```
def get_page(page):
  try:
    content = urllib2.urlopen(page, timeout=100).read()
    print 'downloading page %s' % page
    time.sleep(0.5)
  except(urllib2.URLError):
    print("Error!")
  else:
    return content
```

定义 working 函数

```
def working():
  while True and len(crawled) <= 10:
    page = q.get()
    if page not in crawled:
      content = get_page(page)
        if content != None:
```

```
            add_page_to_folder(page, content)
            outlinks = get_all_links(content, page)
            for link in outlinks:
              q.put(link)
            if varLock.acquire():
              graph[page] = outlinks
              crawled.append(page)
              varLock.release()
            q.task_done()
```

主程序进行参数的初始化

```
start = time.clock()
NUM = 2
crawled = []
graph = {}
varLock = threading.Lock()
q = Queue.Queue()
q.put('http://www.sjtu.edu.cn')
```

建立子进程列表进行操作

```
thread_list = []
for i in range(NUM):
  t = threading.Thread(target=working)
  thread_list.append(t)

print(thread_list)
for t in thread_list:
  t.setDaemon(True)
  t.start()
for t in thread_list:
  t.join()
```

4.2.2 实验结果展示

```
[<Thread(Thread-1, initial)>, <Thread(Thread-2, initial)>]
downloading page http://www.sjtu.edu.cn
downloading page http://alumni.sjtu.edu.cn/newalu/
downloading page http://info.sjtu.edu.cn/index.aspx?jatkt=rejected
downloading page http://3dcampus.sjtu.edu.cn/
downloading page http://gk.sjtu.edu.cn/
```

```
downloading page http://www.jwc.sjtu.edu.cn/web/sjtu/198109.htm
downloading page http://mail.sjtu.edu.cn
downloading page http://www.sjtu.edu.cn/opinion.html
downloading page http://news.sjtu.edu.cn/jdyw/20180405/11199.html
downloading page http://en.sjtu.edu.cn/
downloading page http://foundation.sjtu.edu.cn/
downloading page http://join.sjtu.edu.cn/
1.481351
```

Chapter 5

实验感想

通过本次实验，更深入地了解了 HTML 的知识，初步实现了网络爬虫（实验二）。同时在实验三中，通过对 BloomFilter 的使用，促使我更深入的了解了哈希函数的原理和使用。并且最后的实验让我初步掌握并发编程的方法及优势。