Experiment 2

Name: Wang Jialu

Class: F1503023

ID: 515030910558

This is the continuous section for last week's report.

Hash

Time Complextity

An interesting problem is that how long it takes to run the program. A crawler often needs to deal with large of data so that it costs too much time. Here's a simple method to calculate the runtime.

```
import time
start_time = time.time()
main()
print "%s seconds" % (time.time() - start_time)
```

Make it Faster!

We use list to store crawled pages.

```
tocrawl = [seed]
crawled = []         #已爬序列，list方式存储
while tocrawl:
    page = tocrawl.pop()
    if page not in crawled: #查看page是否在已爬序列中
        …            #抓取page
        crawled.append(page)    #将page加入已爬序列中
```

It has the time complexity *O(n)*, of low efficiency. To greatly improve it, we use hash table instead of list.

First we put strings in b bucket according to hash function and look up in relevant bucket.Thus the search range reduce to *1/b*.

Here's an simple hash function example.

```
def simple_hash_string(keyword, b):
    if keyword != '':
        return ord(keyword[0]) % b
    else:
    return 0
```

This function let string mapped to [0..1] numbers. But it also has a big problem that the frequency of each alpha is difference. So we have a big space to improve it.

Then we need to initialize a hash table.

```
def make_hashtable(b):
    table = []
    for i in range(b):
        table.append([])
    return table
```

To find an element in hash table, we need to acquire the bucket according to the hash function.

```
def hashtable_get_bucket(table, keyword):
    index = simple_hash_string(keyword)
    return table[index]
```

Thus, `hashtable_get_bucket(table, keyword)` represent the bucket. Then we can finish the function to check if a keyword is in hash table.

```
def hashtable_lookup(table, keyword):
    if keyword in hashtable_get_bucket(table, keyword):
        return true
    else:
        return false
```

Finally, if the keyword is not in the table, we should add it.

```
def hashtable_add(table, keyword):
    table[hashtable_get_bucket(table, keyword)].append(keyword)
```

We should look up the keyword outside `hashtable_add` because it is widely used to check if a keyword is in the hash table.

BloomFilter

However, considering our crawler will deal with a large amount of webpage content and fetch tens of thousands of URLs, hash table will cover too large space. So we use bloomfilter to do with it ultimately.

Principle

A Bloom filter is a data structure designed to tell you, rapidly and memory-efficiently, whether an element is present in a set.

The price paid for this efficiency is that a Bloom filter is a **probabilistic** data structure: it tells us that the element either *definitely* is not in the set or *may be* in the set.

The base data structure of a Bloom filter is a Bit Vector. Each empty cell in that table represents a bit, and the number below it its index. To add an element to the Bloom filter, we simply hash it a few times and set the bits in the bit vector at the index of those hashes to 1.

To reduce the probability of conflict, we use k hash function rather than one.

Implementation

Let's define the Bitset at first:

```
class Bitarray:
    def __init__(self, size):
        """ Create a bit array of a specific size """
        self.size = size
        self.bitarray = bytearray(size/8)

    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
```

Here we just use bit operation to change the value of a bit to 1. The expression `1 << (7 - position)` represent the bit of the *position*. And we use the **BitArray of size** as the base data structure.

```
bit_map = BitArray(size)
```

The hash functions used in a Bloom filter should be independent and uniformly distributed. They should

also be as fast as possible. We use k hash function, so we should calculate each value hash function for the keyword. If all of them are 1, *may be* it exists. But one of them is not 1, it's definitely not stored.

```
def BloomFilter(s):
    h1 = hash1(s)
    h2 = hash2(s)
    h3 = hash3(s)
    h4 = hash4(s)
    if not(bit_map.get(h1) and bit_map.get(h2) and bit_map.get(h3) and bit_get(h4)
):
        bit_map.set(h1)
        bit_map.set(h2)
        bit_map.set(h3)
        bit_map.set(h4)
        return True
    else:
        return False
```

The code above is a simple implementation. It takes 4 hash function.

Tuning

We can change hash function to improve the result.

Also, k, the number of hash functions makes a difference. Gnerally, `k = ln(2) * m/n`.

The complete code of exercise 1 is in attachment.

Check the correct rate

When you finish all the things, you should desigh a experiment to check your **correct rate** of BloomFilter.

I consider comparing the numbers of words in a text between using *directionary* and *BloomFilter*. Here's my code.

```
def merge(s, l):
    for i in l:
        if i not in s:
            s.append(i)


def calc1():
    fo = open('pg1661.txt', 'r')
    reads = []
    for i in range(10000):
        s = fo.readline()
        l = s.split(' ')
        merge(reads, l)
    return len(reads)


def calc2():
    fo = open('pg1661.txt', 'r')
    count = 0
    reads = BloomFilter(32000)
    for i in range(10000):
        s = fo.readline()
        l = s.split(' ')
        for i in l:
            count += reads.BloomFilter(i)
    return count
```

To my delight, two results are 14312 and 11816, about 80%. I think it is a little low. But when I add the BloomFilter's size up to 3200000, two results are equal. This time it spent too much space. Finally BloomFilter of size 320000 is perfect. Two results are 14312 and 11816. It is fast and don't waste space.

Concurrent Programming

We can use concurrent programming to decrease the time cost. We should use library **Threading** and **Queue**. An example is like this.

```
for i in range(NUM):
    t = Thread(target=working)
    t.setDaemon(True)
    t.start()

for i in range(JOBS):
    q.put(i)

for i in range(NUM):
    threads[i].join()
```

To use **Threading** we should define a class:

```
import threading

class MyThread(threading.Thread):
    def __init__(self,func,args,name='')
        threading.Thread.__init__(self)
        self.name=name
        self.func=func
        self.args=args

    def getResult(self):
        return self.res

    def run(self):
        print 'starting', self.name
        self.res = apply(self.func, self,args)
        print self.name, 'finished'
```

And the use of some functions in **Queue** are listed:

```
Queue.qsize()    # return the size of queue
Queue.empty()    # return true if the queue is empty, else return false
Queue.put(item)      # put item into queue
Queue.get()          # get out of an item from queue
```

You should remember Queue is **FIFO**.

One important problem bothering me for a long time is that when to kill the thread. The easiest method is to count the pages the crawler fetched. But what if the pages in the queue are all crawled? So if the queue is empty and it is not the first page considering the seed URL, also kill the thread.

When I run my crawler, a *UnicodeEncodeError* came up. I think it was because of the irregular website.

BeautifulSoup is easy but too slow. I recommend **lxml** to parse url, which is 100 times faster. More brutely, you can use **regex** directly, 1000 times faster than BeautifulSoup.

The exercise 2 code can be seen in the attachment.