Assignment 1

Fall 2016
CS834 Introduction to Information Retrieval
Dr. Michael Nelson

Mathew Chaney

September 21, 2016

Contents

List of Figures

Listings

List of Tables

1 Question 1.1

1.1 Question

Think up and write down a small number of queries for a web search engine. Make sure that the queries vary in length (i.e., they are not all one word). Try to specify exactly what information you are looking for in some of the queries. Run these queries on two commercial web search engines and compare the top 10 results for each query by doing relevance judgments. Write a report that answers at least the following questions: What is the precision of the results? What is the overlap between the results for the two search engines? Is one search engine clearly better than the other? If so, by how much? How do short queries perform compared to long queries?

1.2 Resources

The search engines Google [1] and DuckDuckGo [2] were used to obtain the results.

1.3 Methodology

The following search queries were issued to each of the two mentioned search engines:

1. professional skateboarder song
2. daewoo song
3. rollerblade song korea
4. daewon song

The expected relevant pages will contain information regarding the professional skateboarder named Daewon Song. A score is assigned to each search engine for each query using the following equation:

$$\frac{R}{10}$$

Where R the number of relevant documents returned for the given query. Overlap is calculated by using the following equation:

$$\frac{\|G_i \cap D_i\|}{10}$$

Where G_i is the set of pages returned by Google for query i and D_i is the set of pages returned by DuckDuckGo for query i. This result is divided by the size of the overall result set for each query per search engine, ten.

A brief description of the results is provided, with any items that stand out to the reviewer given extra measure. After the detailed listing of results, Table 1 lists the relevancy scores and overlap for each search engine per query.

First Query: “professional skateboarder song”

The relevancy scores from the first query were evenly matched for both search engines, five out of ten relevant documents each. Interestingly, Google provided more social media related results, whereas DuckDuckGo provided a couple of celebrity net worth results.

Second Query: “daewoo song”

Trying to throw a curve ball at the engines with this one, it is intentionally misspelled. Results for both engines show active error correction and query suggestion functionality, which helps to increase relevance when a user perhaps does not know the correct spelling of an item they wish to find. Google wins this round with eight out of ten versus DuckDuckGo’s result of 6 out of ten.

Third Query: “rollerblade song korea”

Somewhat of another curve ball, although this is *close* to a “correct” query, since rollerblading and skateboarding are similar “extreme” sports. The results, however, show that this level of judgment is not available to a search engine, as both websites returned zero relevant documents pertaining to Daewon Song. Perhaps the error was simply between the chair and the keyboard.

Fourth Query: “daewon song”

This is a control, as it is exactly the name of the target of the search query for this question. Both search engines provided a perfect relevancy score, which was not surprising considering that the information need was embodied in a person with a unique name, making narrowing down the relevant results quite easy based on the name alone.

1.4 Results

Table 1 shows the relevancy scores and overlap for each search engine per query.

Query	Google Relevance	DuckDuckGo Relevance	Overlap
professional skateboarder song	0.5	0.5	0.1
daewoo song	0.8	0.6	0.2
rollerblade song korea	0.0	0.0	0.0
daewon song	1.0	1.0	0.7

Table 1: Relevance Scores

Both search engines performed relatively similarly to each other for the given queries. Neither one provided a *significantly* higher number of relevant results for any of the queries, although Google did win out overall. When given the ideal query, overlap was high, although not at 1.0, while also providing a “perfect” relevancy score of 1.0. The length of the query did not seem to be a strong controlling factor, term accuracy within the query seemed to decide the relevancy of the top ten results for a particular topic.

2 Question 1.2

2.1 Question

Site search is another common application of search engines. In this case, search is restricted to the web pages at a given website. Compare site search to web search, vertical search, and enterprise search.

2.2 Resources

The textbook *Search Engines: Information Retrieval in Practice* [3] was used to answer this question.

2.3 Answer

Site search, as stated in the question, refers to a general search query, with results limited to pages within a particular host domain, or website. Web search is defined as a general search over all the pages the web has to offer. On one hand, site search is similar to web search because the query in unstructured. On the other, site search is a far simpler beast than a general web search. Being constrained to a particular host domain has a number of traits that make it an easier task to handle.

Firstly, the number of documents is orders of magnitude smaller for site search compared to web search, allowing the search results to be extremely fresh while not representing a daunting task of crawling the entire document collection. The indexing process can also be completed much faster due to the drastically decreased number of documents. Also, site search never has to worry about document *spam*, items that should be excluded from search results as they are not intended to meet an information need, because the website maintaining the documents has complete control over which documents are to be indexed, leaving the decision made before the indexing process begins.

There are many examples of websites with site-specific search features. For example, *The Internet Movie Database* (IMDB) [4], is an informational website in the domain of film and television. IMDB provides users with a site search feature to make finding a movie, TV show, or actor a simple task. A vertical search engine may be about movies, but it is not necessarily going to have it's own repository of information within the domain of interest.

The key difference between a vertical search and a site-specific search is the search agent, or who is conducting the search. With a vertical search the agent is retrieving results from many websites which are all outside the control of the vertical search engine itself. For example, Google image search returns results that are all images, but they are not all contained on one of Google's own servers. Thus, to build the search index on these documents, the vertical search provider must crawl the websites pertaining to their search topic, a task which requires care if the information is to be kept on the topic of interest.

This differs from a site-specific search in that this type of search engine builds its index using documents controlled by the site administrators; the search itself is a service offered by a website to allow users more easy access to its content. This gives the proprietors of such a service an advantage in that web crawling is not required, eliminating many problems that come along with crawling the web – non-uniformity of document models, possible network issues, storage concerns (the number of results are likely higher considering the vertical search engine must index on the entire web whereas the site-specific search engine only indexes on the documents contained in that website).

Enterprise search is probably most similar to site search. This is due to the same reasons site search is dissimilar to web search and vertical search – the owner of the documents over which the search is made is also in control of the search engine itself. The owner of the enterprise search engine is likely a company with a large repository of corporate documentation, thus the need for an enterprise search solution in the first place. This allows the enterprise search provider to have the same benefits of site search described above: a smaller document collection (compared to web and vertical search) leads to faster indexing, query processing, and more fresh indexes.

3 Question 3.7

3.1 Question

Write a program that can create a valid sitemap based on the contents of a directory on your computer's hard disk. Assume that the files are accessible from a website at the URL http://www.example.com. For instance, if there is a file in your directory called homework.pdf, this would be available at http://www.example.com/homework.pdf. Use the real modification date on the file as the last modified time in the sitemap, and to help estimate the change frequency.

3.2 Resources

With the Python programming language [5], the script `sitemapgen.py` was created to perform the necessary tasks to complete this question. This script can be found in Listing 5. The output matches the format found at Sitemaps.org [6].

The script uses the last modified time of each file to estimate the change frequency and also escapes special characters in the output to ensure valid output. The DOM is built while the script traverses the file system and when this process is complete the document is printed to standard out.

3.3 Answer

Starting with with the folder structure found in Listing 1 as input script produced the output shown in Listing 2.

```
smgen
+---sitemapgen.py
+---testdir
    +---dir1
    |   +---testfile
    +---dir2
        +---dir3
        |   +---testfile
        +---testfile
```

Listing 1: test directory structure

```
<?xml version="1.0" ?>
<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">
    <url>
        <loc>http://www.example.com/testdir/dir1/testfile</loc>
        <lastmod>2016-09-18</lastmod>
        <changefreq>daily</changefreq>
        <priority>0.5</priority>
    </url>
    <url>
        <loc>http://www.example.com/testdir/dir2/testfile</loc>
        <lastmod>2016-09-18</lastmod>
        <changefreq>daily</changefreq>
        <priority>0.5</priority>
    </url>
    <url>
        <loc>http://www.example.com/testdir/dir2/dir3/testfile</loc>
        <lastmod>2016-09-18</lastmod>
        <changefreq>daily</changefreq>
        <priority>0.5</priority>
    </url>
    <url>
        <loc>http://www.example.com/sitemapgen.py</loc>
        <lastmod>2016-09-19</lastmod>
        <changefreq>always</changefreq>
        <priority>0.5</priority>
    </url>
</urlset>
```

Listing 2: site map generator output

4 Question 3.8

4.1 Question

Suppose that, in an effort to crawl web pages faster, you set up two crawling machines with different starting seed URLs. Is this an effective strategy for distributed crawling? Why or why not?

4.2 Resources

The textbook *Search Engines: Information Retrieval in Practice* [3], *Parallel Crawlers* [7], and *Efficient URL Caching for World Wide Web Crawling* [8] were used to answer this question.

4.3 Answer

I would say that the answer to this question is "it depends". To maximize effectiveness a solid goal for the designers of this system would be to ensure it downloads each unique page once and only once, minimizing overlap. Overlap is when a parallel crawler downloads the same page more than once. This issue can be a problem if there is no communication among the crawling processes and if there are system limitations that cannot be overcome, such as limited bandwidth, maximum hard disk space, or time constraints. In one of these scenarios overlap can be a debilitating problem, wasting precious resources on repeatedly downloading duplicate web pages.

In order to address the overlap issue, each crawling process would need a way to identify which URLs belong to it, so that it would not crawl URLs that belong to the other crawling machine. One way of doing this is by partitioning the web prior to the start of the crawling process. There are a few ways to go about this. First, the crawlers could receive a static set of seed URLs to start from and then go from there. This is the simplest route, and is also the one employed by Cho and Garcia-Molina in [7]. Another method is to use a central controller to divide the web into partitions and then, as these URLs are discovered by the crawling processes, the central controller sends them to the responsible crawling process. If this issue is not addressed by the parallel crawling machines then overlap will become a problem.

In addition to the partitioning scheme, the two crawlers would benefit from using some inter-process communication to inform each other which URLs have been visited, which mitigates the overlap issue described above. This communication would cause some additional processing and network bandwidth consumption, but can be minimized by using batched dispatching [7] and a URL caching scheme [8].

Another issue for parallel crawlers is coverage. Coverage is how much of the web is downloaded. If the crawler is downloading pages to build an index for a search engine coverage can be an important issue. If the crawlers do not cooperate with each other coverage can suffer under certain circumstances. For example, in Figure 1, S_1 is a partition of websites assigned to Crawler C_1, and S_2 is a partition of websites assigned to Crawler C_2. If the crawlers do not communicate with each other and also do not traverse the inter-partition links, Crawler C_1 will not discover pages d or e, and so coverage will suffer. If coverage is important then the parallel crawling machines should pass URLs to each other to minimize the pages missed due to these limitations.

These are but a few items to ponder when going about designing a parallel crawler. They can be effective at their job if they are designed to mitigate these issues, but if they are not addressed then the crawler will likely not be very effective at all.

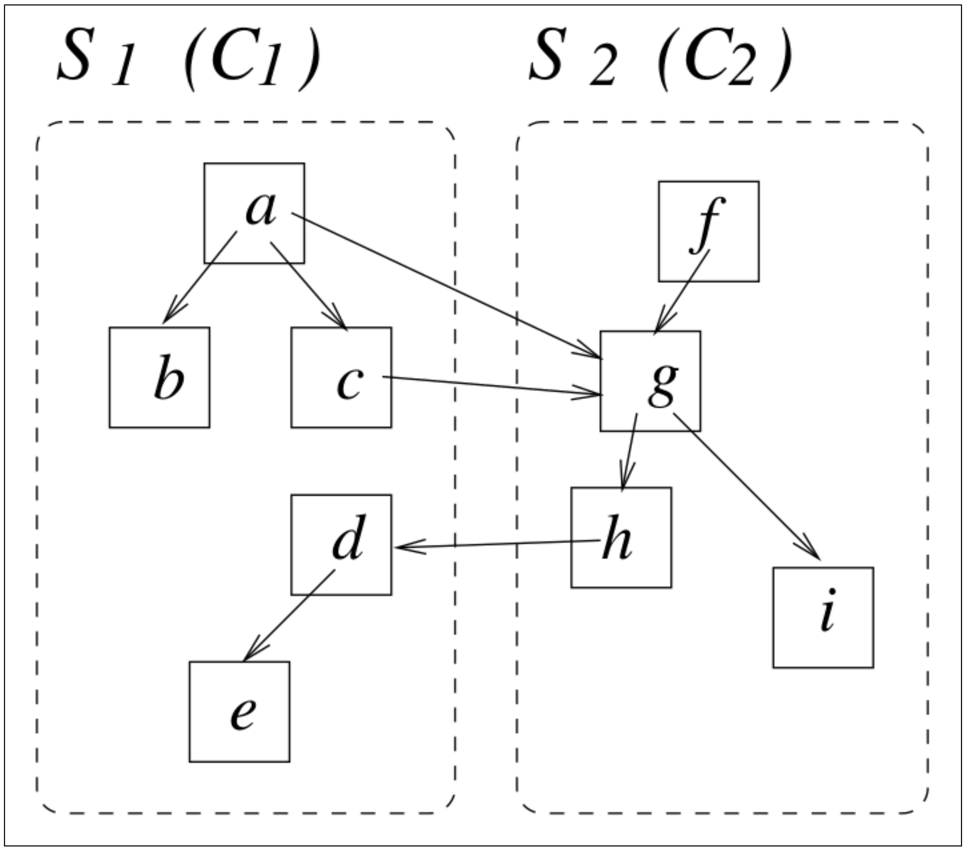

Figure 1: Two site example[1]

[1]Source: Junghoo Cho and Hector Garcia-Molina. Parallel Crawlers. In *Proceedings of the 11th International Conference on World Wide Web*, WWW '02, page 126, New York, NY, USA, 2002. ACM.

5 Question 3.9

5.1 Question

Write a simple single-threaded web crawler. Starting from a single input URL (perhaps a professor's web page), the crawler should download a page and then wait at least five seconds before downloading the next page. Your program should find other pages to crawl by parsing link tags found in previously crawled documents.

5.2 Resources

The `crawler.py` script, found in Listing 4, was written to accomplish this task. It was written in the Python programming language [5]. The Requests [9] and BeautifulSoup [10] modules were particularly useful in completing this task. Some sample output can be found in Listing 3.

5.3 Answer

```
retrieving http://cs.odu.edu/~mln
retrieving http://www.cs.odu.edu/~mln/
retrieving http://www.cs.odu.edu/
retrieving http://www.odu.edu
retrieving http://www.cs.odu.edu/~mln/research/
retrieving http://www.cs.odu.edu/~mln/pubs/
retrieving http://www.cs.odu.edu/~mln/teaching/
retrieving http://www.cs.odu.edu/~mln/service/
retrieving http://www.cs.odu.edu/~mln/personal/
retrieving http://www.larc.nasa.gov/
retrieving http://sils.unc.edu/
retrieving http://www.openarchives.org/pmh/
retrieving http://www.openarchives.org/ore/
retrieving http://www.mementoweb.org/guide/rfc/ID/
retrieving http://www.openarchives.org/rs/toc
retrieving http://ntrs.nasa.gov/
retrieving http://www.nsf.gov/awardsearch/showAward.do?AwardNumber=0643784
```

Listing 3: crawler.py output

6 Appendix A

```python
import argparse
import requests
from time import sleep
from collections import deque
from bs4 import BeautifulSoup


def crawl():
    while True:
        url = frontier.popleft()
        if url not in visited:
            print 'retrieving', url
            try:
                res = requests.get(url)
            except:
                continue
            if res.ok:
                visited.add(url)
                soup = BeautifulSoup(res.text, 'html.parser')
                links = soup.find_all('a')
                for link in links:
                    try:
                        linkurl = link['href'].strip()
                        if linkurl not in visited:
                            frontier.append(linkurl)
                    except KeyError:
                        continue
        sleep(5)


if __name__ == '__main__':
    parser = argparse.ArgumentParser('web crawler')
    parser.add_argument('--host', default='http://cs.odu.edu/~mln')
    args = parser.parse_args()
    visited = set()
    frontier = deque()
    frontier.append(args.host)
    crawl()
```

Listing 4: crawler.py

```python
import sys
import os
import argparse
import datetime
import time
import urllib

from os.path import getmtime, isdir, isfile

import xml.dom.minidom as md


def append(doc, urlset, loc, lastmod, changefreq, priority='0.5'):
    url = doc.createElement('url')
    urlset.appendChild(url)

    loc_element = doc.createElement('loc')
    loc_element.appendChild(doc.createTextNode(loc))
    url.appendChild(loc_element)

    lastmod_element = doc.createElement('lastmod')
    lastmod_element.appendChild(doc.createTextNode(lastmod))
    url.appendChild(lastmod_element)

    changefreq_element = doc.createElement('changefreq')
    changefreq_element.appendChild(doc.createTextNode(changefreq))
    url.appendChild(changefreq_element)

    priority_element = doc.createElement('priority')
    priority_element.appendChild(doc.createTextNode(priority))
    url.appendChild(priority_element)


YEAR = 3.154e+7
MONTH = 2.592e+6
WEEK = 604800.0
DAY = 86400
HOUR = 3600
MINUTE = 60

def estimate_changefreq(posixtime):
    timenow = time.time()
    delta = timenow - posixtime
    if delta > YEAR:
        return 'never'
    elif delta > MONTH:
        return 'yearly'
    elif delta > WEEK:
        return 'monthly'
    elif delta > DAY:
        return 'weekly'
    elif delta > HOUR:
        return 'daily'
    elif delta > MINUTE:
        return 'hourly'
    else:
        return 'always'


def convertdate(posixtime):
    return datetime.datetime.utcfromtimestamp(posixtime).strftime('%Y-%m-%d')


def delve(root, folder, doc, urlset):
    items = os.listdir(root + folder)
    for item in items:
        filepath = root + folder + item
        if isfile(filepath):
            loc = args.host + urllib.quote(folder + item)
            lastmodsecs = getmtime(filepath)
            lastmod = convertdate(lastmodsecs)
            changefreq = estimate_changefreq(lastmodsecs)
            append(doc, urlset, loc, lastmod, changefreq)
        elif isdir(filepath):
            delve(root, folder + item + os.sep, doc, urlset)

```

```

def test(doc, urlset):
    append(doc, urlset, 'www.google.com', '2016-09-17', 'always', '0.8')
    append(doc, urlset, 'www.duckduckgo.com', '2016-09-17', 'daily')
    print doc.toprettyxml()


if __name__ == '__main__':
    # parse arguments
    parser = argparse.ArgumentParser('site map generator')
    parser.add_argument(
        '-test',
        '-t',
        action='store_true')
    parser.add_argument(
        '--path',
        '-p',
        default='.')
    parser.add_argument(
        '--host',
        default='http://www.example.com/')
    args = parser.parse_args()

    # create a document
    doc = md.Document()
    urlset = doc.createElement('urlset')
    urlset.setAttribute('xmlns', 'http://www.sitemaps.org/schemas/sitemap/0.9')
    doc.appendChild(urlset)

    # if desired, perform simple test and return
    if args.test:
        test(doc, urlset)
        sys.exit(0)

    # parse path from args
    path = args.path
    if path[len(path)-1] != os.sep:
        path = path + os.sep

    # iterate over all items in doc
    delve(path, '', doc, urlset)

    print doc.toprettyxml()
```

Listing 5: sitemapgen.py

7 References

[1] Google. Available at: http://www.google.com. Accessed: 2016/09/17.

[2] DuckDuckGo. Available at: http://www.duckduckgo.com. Accessed: 2016/09/17.

[3] Bruce Croft, Donald Metzler, and Trevor Strohman. *Search Engines: Information Retrieval in Practice*. Pearson, first edition, February 2009.

[4] The Internet Movie Database. Available at: http://www.imdb.com/. Accessed: 2016/09/17.

[5] The Python Programming Language. Available at: https://www.python.org/. Accessed: 2016/09/17.

[6] Sitemaps XML format. Available at: http://www.sitemaps.org/protocol.html. Accessed: 2016/09/17.

[7] Junghoo Cho and Hector Garcia-Molina. Parallel Crawlers. In *Proceedings of the 11th International Conference on World Wide Web*, WWW '02, pages 124–135, New York, NY, USA, 2002. ACM.

[8] Andrei Z. Broder, Marc Najork, and Janet L. Wiener. Efficient URL Caching for World Wide Web Crawling. In *Proceedings of the 12th International Conference on World Wide Web*, WWW '03, pages 679–689, New York, NY, USA, 2003. ACM.

[9] Kenneth Reitz. Requests: HTTP for Humans. Available at http://docs.python-requests.org/en/master/. Accessed: 2016/09/20.

[10] Leonard Richardson. Beautiful Soup. Available at: https://www.crummy.com/software/beautifulsoup/. Accessed: 2016/09/20.