

Assignment 1  
Introduction to Information Retrieval  
CS734/834

John Berlin

September 19, 2016

## Q.1 Question 1.2

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

### Q.1 Answer

To compare the four searches I performed the first three using Google and base query “async redux”. The search results shown are limited to the first three in order to have presentable figures. For the site search I used the site *github.com* as the context for the base query is developing web pages in the React.js ecosystem. As seen in figure 1 the results are for libraries that do async actions in redux. The first two is from the redux library itself whereas the third is from a library that enhances that functionality.

The web searches top three results figure 2, featured two results from the online documentation of the library itself and one code example which the first result of the site search. These results were as expected as the query was missing the context given to search engine through the “site:github.com” as is done in site search. The context is key when comparing site search to web search as it effects the precision of these queries rather than recall. For example, I made this exact query recently when looking into how to better handle the async problems faced by WAIL<sup>1</sup> which is using flux (the predecessor to redux). I performed the web search first as I wanted to see in the documentation on how redux handled async operations rather than diving into the code bases of the libraries themselves. Google understood the likely hood of this happening for the base query and was precise in its top three results given the lack of context which is present in the site search.

Vertical search is done when only results of a certain type are desired which is another contextual based query. The results of this search, using Google Videos figure 3, in comparison to the site search can be described as combination of the web search and the site search. The first two results of this query are a combination of all three results from the web search plus the first and second of the site search. The last result of the vertical search features content from the third result from the site search.

In order to compare enterprise search to site search I will use two examples: the first will be the classical example using my personal computers file search. The second will be using more targeted search on my personal computer within the context of the previous two comparisons. As I do not have a large corporate intranet to search I use *Desktop Search* in this example. Desktop search, as defined in our book, is the personal version of enterprise search [?, pp. 3]. I felt this is a correct substitution because the previous two comparisons used examples about how I personally used the them for WAIL. Furthermore if a team of developers was looking to find out where on their computers which libraries are currently being used for async operations figure 4(a) and then where in the repository is the functionality implemented figure 4(b).

Enterprise Search (*Desktop Search*) returned results that had the search term in them not necessarily the most relevant figure 4(a). The search term async can be found in directory names as well as many duplicates. The duplicates in these results are a product of the search being done on the file system where duplicates are acceptable. Unlike site search which returned unique results where duplicates are not as common. Whereas if you were to use the Linux utility *grep*, you can perform the same kind of query but limited to lines contained in the files in a specific part of the file system figure 4(b), much like site search. It is clear to see that enterprise search is a broader type of search yet is done a specific site(file system).

---

<sup>1</sup><https://github.com/N0taN3rd/wail>

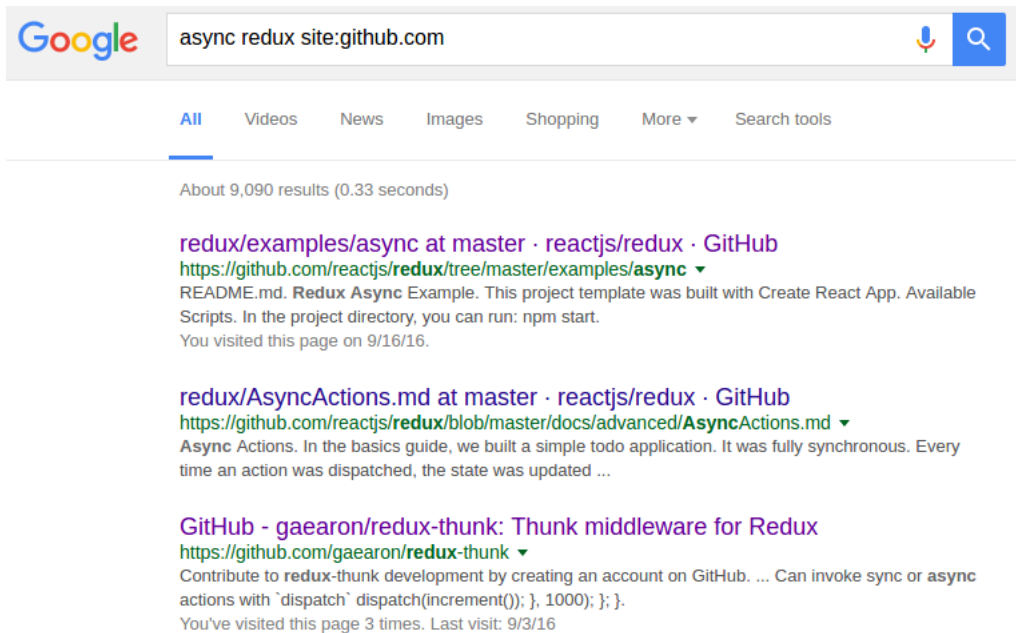


Figure 1: Site Search

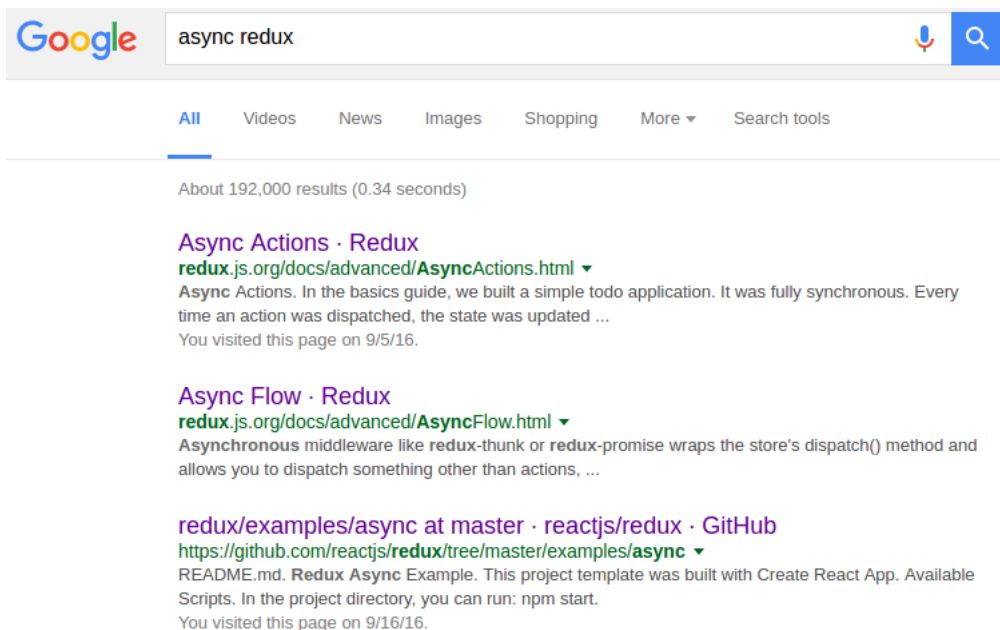


Figure 2: Web Search

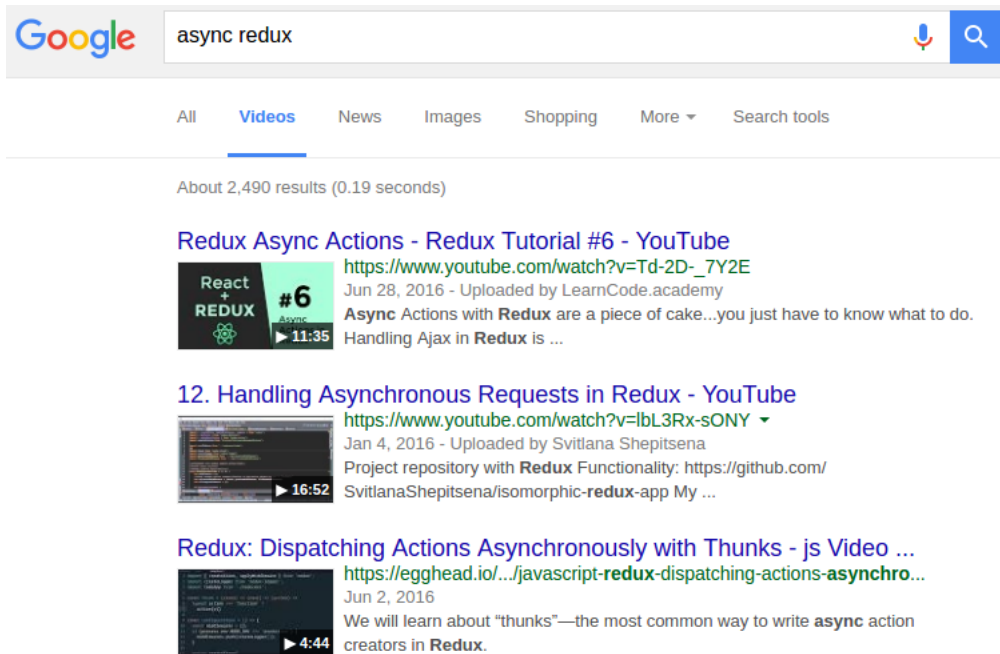
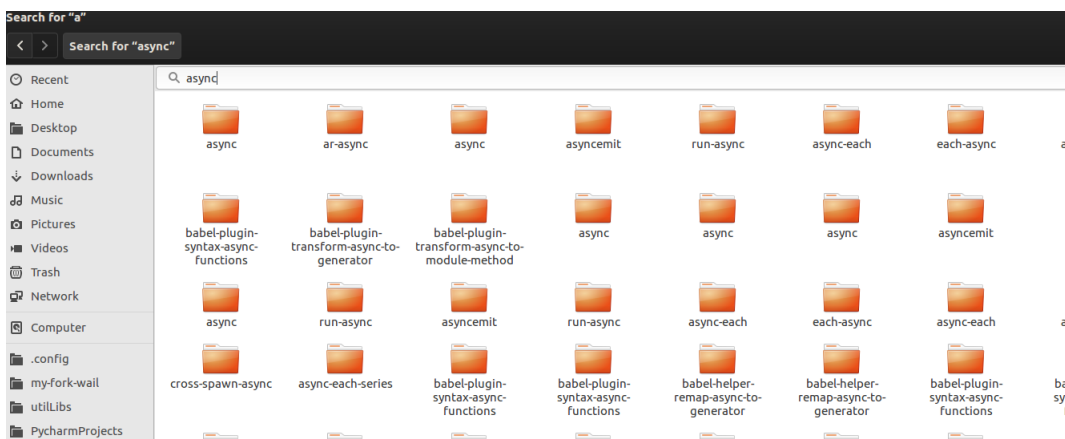
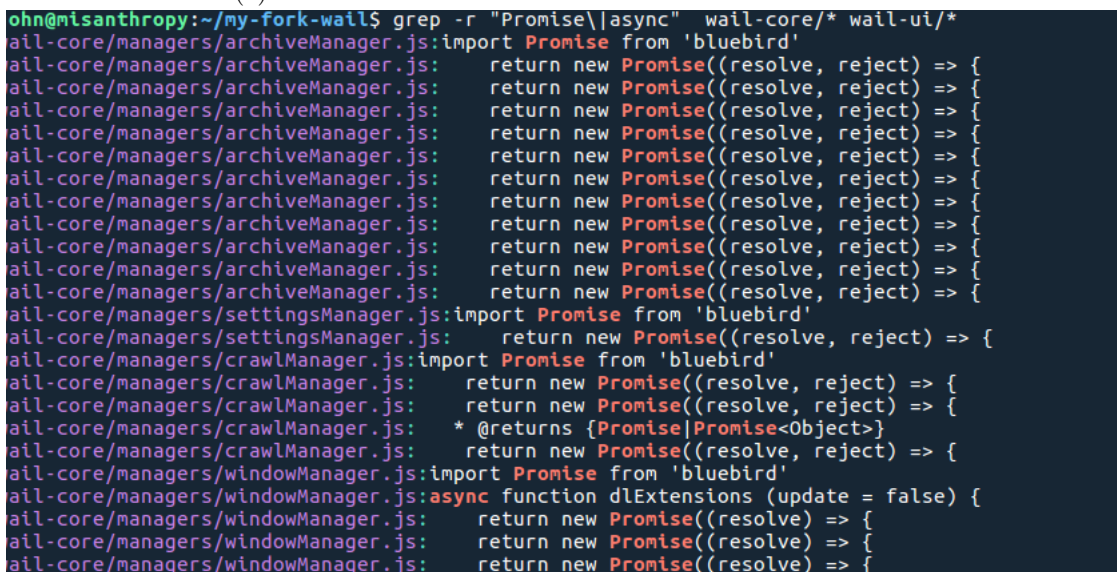


Figure 3: Vertical Search



(a) Classic



(b) Targeted

Figure 4: Enterprise Search

## Q.2 Question 3.7

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.

### Q.2 Answer

The code for this question can be seen in the source code listing 1. The program utilizes three libraries which made this question easy: `PyFilesystem` [?] for its combination of listing the contents of a directory and file stats in a single function call, `Arrow` [?] the python version of `Momentjs` and `Yattag`[?] pythonic document creation. Using python's argument parser library to get the directory for which to create the sitemap, the program lists the contents of the directory and if the name of the file is acceptable, determined by “-dots” (include . files or not) argument create a new xml node where the last modified time is converted into utc and finally after the files of the directory have been read output the created site to stdout.

---

```
from urllib.parse import quote_plus as quote
from argparse import ArgumentParser, ArgumentError, Action
import os
from arrow import Arrow
from fs.osfs import OSFS
from yattag import Doc, indent

dots = False

class FullPaths(Action):
    """Expand user- and relative-paths"""
    def __call__(self, parser, namespace, values, option_string=None):
        setattr(namespace, self.dest, os.path.abspath(os.path.expanduser(values)))

def is_dir(dirname):
    """Checks if a path is an actual directory"""
    if not os.path.isdir(dirname):
        msg = "{0} is not a directory".format(dirname)
        raise ArgumentError(msg)
    else:
        return dirname

def accept(name, moreDots=dots):
    if moreDots:
        return True
    else:
        return name[0] != '.'

if __name__ == '__main__':
    parser = ArgumentParser(description="Create a sitemap of a directory on your local file system",
        ↪ prog='sitemap', usage='%(prog)s [options]')
    parser.add_argument('-dir', '--directory', help='directory to use', action=FullPaths, type=is_dir)
    parser.add_argument('-dots', help='include dot files', action='store_true')
    args = parser.parse_args()
    dir = OSFS(args.directory)
    dots = args.dots
    urlSet = []
    doc, tag, text = Doc().tagtext()
    with tag('urlSet', xmlns="http://www.sitemaps.org/schemas/sitemap/0.9"):
        for name, stats in dir.listdirinfo(files_only=True):
            if accept(name):
                with tag('url'):
```

```
        with tag('loc'):
            text('http://www.example.com/%s' % quote(name))
        with tag('lastmod'):
            text(str(Arrow.utctimestamp(stats['modified_time']).timestamp()))
    dir.close()
    result = indent(
        doc.getvalue(),
        indentation=' ' * 4,
        newline='\r\n'
    )

print(result)
```

---

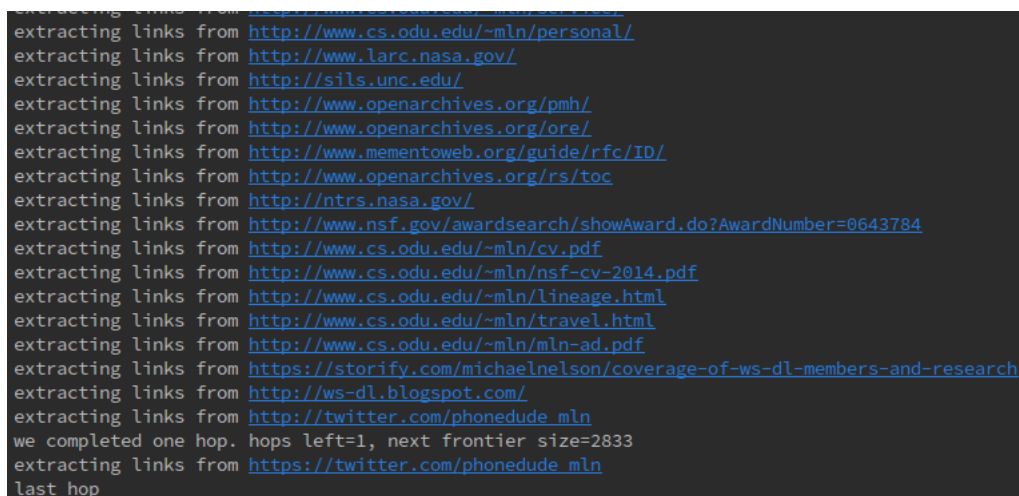
Source Code 1: Python Directory Sitemap Generation

### Q.3 Question 3.9

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.

### Q.3 Answer

The solution to this question is a modified version of the solution submitted for assignment 1 for cs532 and the source can be seen in source code listing 2. The modifications made to the original were minor instead of looking for links that resolve to pdf files the code now extracts all links from a uri that resolves to a 200 and puts them into a queue the frontier\_gen function. The first uri or seed is placed in the queue first and the first call to frontier\_gen is made. Then after waiting for five sections the next uri is popped from the queue and the links contained in it are added. I have added an additional part to this assignment which is a stopping criteria hops. Hops is the number links away from seed url and the reason for this stipulation can be seen in figure 5. After the first hope the frontier has reached a size of 2833 uris waiting to be processed after visiting links one hope away from the seed url <http://cs.odu.edu/~mln>. I also did this from experience gained while working with Heritrix which will download the entire internet if it is not given explicit hop bounds. The current hop the program is at is determined by decrementing the frontier count gotten at the start of next hop by taking the length of the queue. When it reaches zero the last uri for the final\_hop has its links extracted and the remaining uris are printed.



```
extracting links from http://www.cs.odu.edu/~mln/personal/
extracting links from http://www.larc.nasa.gov/
extracting links from http://sils.unc.edu/
extracting links from http://www.openarchives.org/pmh/
extracting links from http://www.openarchives.org/ore/
extracting links from http://www.mementoweb.org/guide/rfc/ID/
extracting links from http://www.openarchives.org/rs/toc
extracting links from http://ntrs.nasa.gov/
extracting links from http://www.nsf.gov/awardsearch/showAward.do?AwardNumber=0643784
extracting links from http://www.cs.odu.edu/~mln/cv.pdf
extracting links from http://www.cs.odu.edu/~mln/nsf-cv-2014.pdf
extracting links from http://www.cs.odu.edu/~mln/lineage.html
extracting links from http://www.cs.odu.edu/~mln/travel.html
extracting links from http://www.cs.odu.edu/~mln/mln-ad.pdf
extracting links from https://storify.com/michaelnelson/coverage-of-ws-dl-members-and-research
extracting links from http://ws-dl.blogspot.com/
extracting links from http://twitter.com/phonedude_mln
we completed one hop. hops left=1, next frontier size=2833
extracting links from https://twitter.com/phonedude_mln
last hop
```

Figure 5: Frontier Size

```
import re
import requests
from urllib.parse import urljoin
from bs4 import BeautifulSoup
from collections import deque
from argparse import ArgumentParser
import time

reg_s = "((([A-Za-z]{3,9}:(?!(\\|\\/)?)(?:[\\-;:&=+\\$,\\w]+@)?[A-Za-z0-9\\.\\-]+|\" + \\
\"(?:www\\.|[\\-;:&=+\\$,\\w]+@)[A-Za-z0-9\\.\\-]+)((?:\\/\\[+~%\\|\\.|\\w\\-]*)?)\" + \\
\"\\?(?:[\\-\\+&=%@\\.\\w]*)#(?:[\\.|\\/\\|\\w]*)?)?)\"
# my standard url regex found a while ago
url_re = re.compile(reg_s, re.IGNORECASE)

relative = re.compile("(?!www\\.|(?:(http|ftp)s?:\\/\\/[A-Za-z]:\\|\\/)).*")
```

```

def gen_frontier(uri, session, q, seen):
    """
        Simplistic frontier generation function
        Download the current uri and if it resolves to a 200
        extract all the a elements.
        If we have seen the link contained in an href of the current
        a element skip it otherwise add it to our queue.
        A sanity check uri regex is utalized
        if it fails check if the uri is relative
        otherwise skip it
    """
    r = session.get(uri) # type: requests.Response
    if r.ok:
        try:
            s = BeautifulSoup(r.text, 'html5lib')
        except:
            # just because if this fails there are problems
            s = BeautifulSoup(r.text)
        all_a = s.find_all('a', href=True)

        for link in map(lambda a: a['href'], all_a):
            if link not in seen:
                print("extracting links from %s" % uri)
                if url_re.match(link):
                    q.append(link)
                else:
                    if relative.match(link):
                        link = urljoin(r.url, link)
                        q.append(link)
                    else:
                        print("The input uri %s failed to pass my regex " % link, reg_s)
            else:
                print("We have a link that does not resolves to a 200: %s, %d" % (uri, r.status_code))

if __name__ == '__main__':
    parser = ArgumentParser(description="Single Threaded Crawler", prog='crawler', usage='% (prog)s
    ↪ [options]')
    parser.add_argument('-s', '--seed', help='seed to start crawling', type=str,
    ↪ default='http://cs.odu.edu/~mln')
    # ok I am adding this to the program as having worked with Heritrix an unbounded crawler will
    ↪ download the internet
    parser.add_argument('-hops', help='how many hops from the start should the crawler crawl',
    ↪ type=int,
    ↪ default=2)
    args = parser.parse_args()
    useragent = 'Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:44.0) Gecko/20100101 Firefox/44.01'
    hops = args.hops
    q = deque()
    # add the initial seed url to queue
    q.append(args.seed)
    # simple attempt at avoiding crawler traps i.e. links that reference the page containing them
    seen = set()

    with requests.Session() as session:
        session.headers.update({'User-Agent': useragent})
        uri = q.popleft()
        # extract all uris from the seed and add it to seen
        gen_frontier(uri, session, q, seen)
        seen.add(uri)
        # get the number of uris in the frontier for hop 1
        frontierCur = len(q)
        # continue crawling until we are at the last hop
        while hops > 0:

```



```

time.sleep(5)
uri = q.popleft()
frontierCur -= 1
# when the frontier for the current hop has been exhausted
if frontierCur == 0:
    # get the new frontier size
    frontierCur = len(q)
    # decrement hops as we are now one curHop+1 away from the seed uri
    hops -= 1
    print('we completed one hop. hops left=%d, next frontier size=%d' % (hops,
        ↪ frontierCur))
    # visit this uri and extract its uris
    gen_frontier(uri, session, q, seen)
# we are now at the last hop exhaust the queue
print("printing uris at hope %d away from the seed"%args.hops)
while True:
    try:
        uri = q.popleft()
        print('uri gotten from last hop'%uri)
    except IndexError:
        break

```

---

Source Code 2: Python Single Threaded Crawler