HW 1 - Webscience Intro

Jacob Conner

September 20, 2020

Q1

Consider the ”bow-tie” structure of the web in the Broder et al. paper (http://snap.stanford.edu/class/cs224w-readings/broder00bowtie.pdf) that was described in Week 1. Now consider the following links (1) Draw the resulting graph (either sketch on paper or use another tool) and include an image in your report.)

```
A --> B
B --> A
B --> C
C --> D
C --> G
D --> A
D --> H
E --> F
E --> O
F --> G
G --> P
H --> L
J --> N
K --> I
M --> A
N --> L
O --> J
P --> C
```

Listing 1: Links

For the above graph, list the nodes (in alphabetical order) that are each of the following categories:

- IN:
- SCC:
- OUT:
- Tendrils:
  - indicate if the tendril is reachable from IN or can reach OUT
- Tubes:

  - explain how the nodes serve as tubes

- Disconnected:

Answer

All figures must have a caption and must be referenced in the text. Example below.

Figure 1 was created using the *Dia* diagramming tool [1] and shows the relationships between each node.

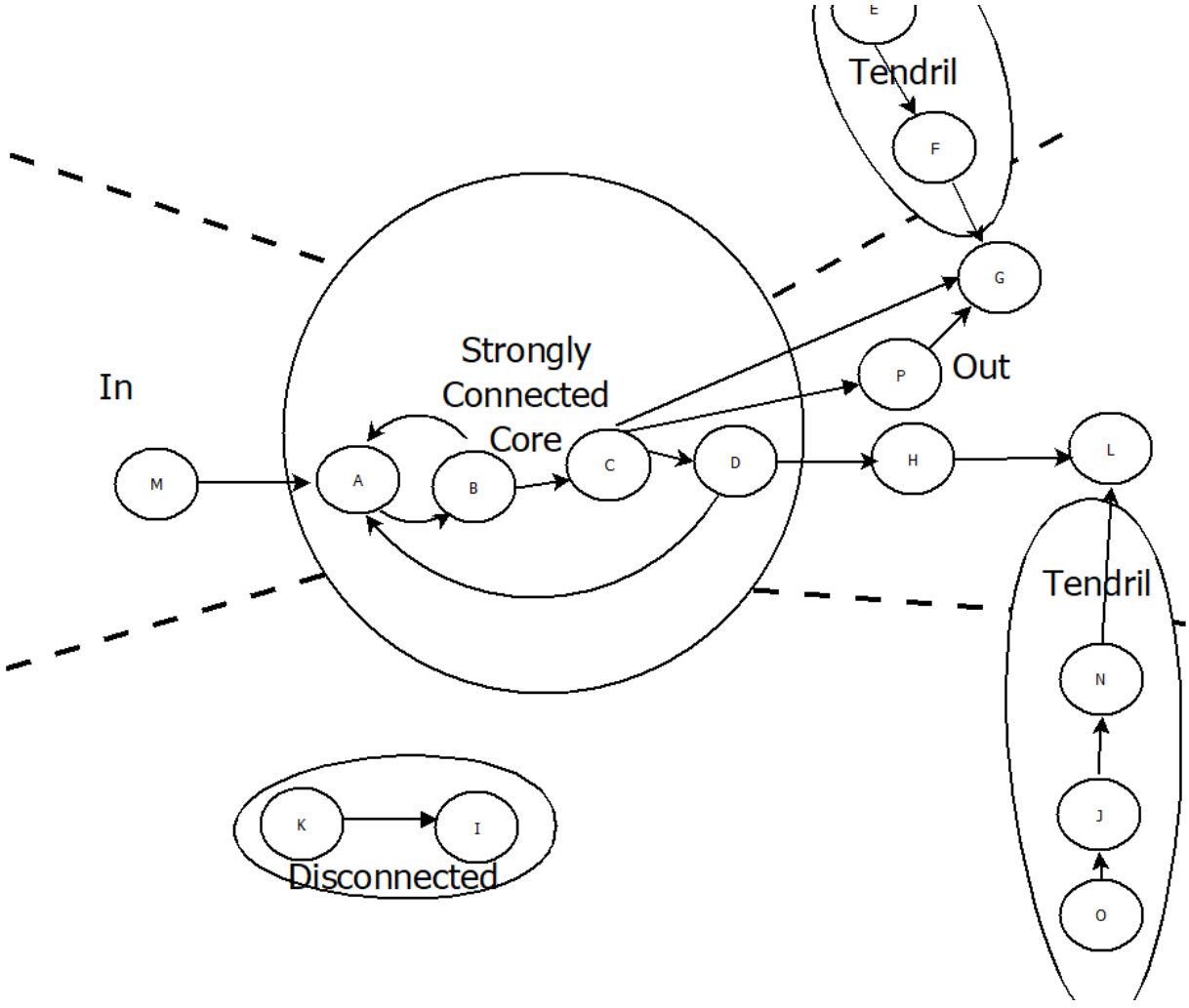

Figure 1: Bow-Tie Graph representing relationship between nodes

- IN: M

- SCC: A, B, C, D

- OUT: G, H, L,
- Tendrils : NONE
  - Reaching outward (FROM IN): NONE
  - : Reaching in (to OUT): E, F, J, N, O
- Tubes: NONE
- Disconnected: K, I

Discussion

You must provide some discussion of every answer. Discuss how you arrived at the answer and the tools you used. Discuss the implications of your answer. In this graph, the strongly connected component (SCC) consists of the sub-graph where all nodes are capable of reaching each other. Table lists the paths. The in portionn of the graph was detrmined by identifying those nodes that only linked in to another in node or to the SCC. Similarly the OUT nodes were identified by find those nodes that only linked out from another SCC node or to another out node. Tendrils are the groups of nodes that either link out from an IN node or link inward to an OUT node. In this graph, there were not any nodes linking outward from the IN nodes, but there were three nodes (J, N, O) that reached in to the OUT Node L. Tubes are the nodes that that link between IN and OUT nodes without going through the SCC. Since there were no nodes linking from IN to the OUT nodes, there were no tubes. Then disconnected nodes were the nodes that do not connect to the SCC, IN, OUT are the disconnected nodes. In this graph, there were two disconnected nodes K and I.. Node K can connect to Node I but cannot connect to anything else and Node I serves only as a destination for Node K and does not link to anything else.

A few interesting observations from this graph is that there are several things that will prevent webcrawlers from discovering all websites. First some nodes are disconnected and are not linked to from any other website. It is simply not possible to uncover these websites by following links. However, even if all nodes were somehow connected, it may be impossible to find websites based on where the webcrawler is searching at from the graph. If I started crawling at the SCC node A, I would be able to find most of the nodes in this graph, but I would never be able to reach node M since it is merely linking to A and A does not link back to M. A webcrawl would also never be able to uncover any of the tendrils since they link to out nodes, but the out nodes do not link back to them.If by some chance a webcrawler started on an out node then the number of websites that it could uncover would be quite small.

Q2

Demonstrate that you know how to use curl and are familiar with the available options.

Table 1: SCC Links

col	A	B	C	D
A	X	A,B	A,B,C	A,B,C,D
B	B,A	X	B,C	B,C,D
C	C,D,A	C,D,A,B	X	C,D
D	D,A	D,A,B	D,A,B,C	X

a) In a single curl command, request the URI, show the HTTP response headers, follow any redirects, and change the User-Agent HTTP request field to ”CS432/532”. Show command you used and the result of your execution on the command line. (Either take a screenshot of your terminal or copy/paste into a code segment.)

b) Then make the same request again, but without showing the HTTP response headers and with saving the HTML output to a file. Show the command you used and the result of your execution on the command line. View that file in a browser and take a screenshot.

c) Finally, load the URI directly from your browser and take a screenshot.

Explain the results you get for each of these steps.

Answer

a)

```
curl -i -L -A "CS432/532" http://www.cs.odu.edu/~mweigle/courses/cs532/
   ua_echo.php
```

Listing 2: Command

```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time
    Current
                                 Dload  Upload   Total   Spent    Left
    Speed
100   178  100   178    0     0   2282      0 --:--:-- --:--:--
   --:--:--  2825
100   114  100   114    0     0    561      0 --:--:-- --:--:--
   --:--:--  1461HTTP/1.1 301 Moved Permanently
Server: nginx
Date: Sat, 29 Aug 2020 15:08:24 GMT
Content-Type: text/html
Content-Length: 178
Connection: keep-alive
Location: https://www.cs.odu.edu/~mweigle/courses/cs532/ua_echo.php

HTTP/1.1 200 OK
```

```
Server: nginx
Date: Sat, 29 Aug 2020 15:08:24 GMT
Content-Type: text/html; charset=UTF-8
Content-Length: 114
Connection: keep-alive
Vary: Accept-Encoding
X-Powered-By: PHP/5.6.40

<!DOCTYPE html>
<html>
<body>

<br/>USER AGENT ECHO
<br/><br/>
<b>User-Agent:</b> CS432/532<br/>

</body>
</html>
```

Listing 3: Response

The curl command is used to make HTTP GET Requests. the -i or –include flag is used to specify curl to include the headers in the response. The -L or –location flag is used to request the header and response for all redirects. The -A or –user-agent flag specifies the User-Agent property in the header that is sends to the string next to it. IN this case the User-Agent sent is ”CS432/5432”. Then the last part of the curl command is the URI(s) for the webserver(s) that we are making a HTTP GET request to /citecurlManPage.

b)

```
curl -o response.html -L -A "CS432/532" http://www.cs.odu.edu/~mweigle/
   courses/cs532/ua_echo.php
```

Listing 4: Command

```
1  <!DOCTYPE html>
2  <html>
3  <body>
4
5  <br/>USER AGENT ECHO
6  <br/><br/>
7  <b>User-Agent:</b> CS432/532<br/>
8
9  </body>
10 </html>
```

Listing 5: response.html

This command is similar to the previous command in part a. However since we do not need the header information, the -i –include flag has been omitted. I have also added the -o output flag to

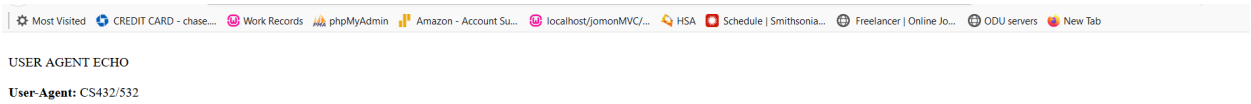

Figure 2: Response 2

copy the resulting response from the server into the file response.html. /reflst:responseHtml shows the result output from the HTTP request. The response.html file is a fairly simple html file with a body tag enclosing one set of bold tags surrounding the words User-Agent, and the value of our User-Agent property (CS432/532) in the GET request header. The browser representation of the response is displayed in 2 c)

If a browser navigates to https://www.cs.odu.edu/ mweigle/courses/cs532/ua_echo.php, the browser sends the GET HTTP request in the background and receives the Response and creates a representation of the response which will look similar to /reffig:g2CResponse. It looks very similar to the previous response.html retrieved from the curl command, but in the browser the User-Agent shows a different string. One the computer I am currently using, I am running Windows 10 64-bit and running Firefox, so the Browser uses that information to send a string to the User-Agent in the GET request to help the webserver identify how I am accessing this page. If I were to use a different browser such as Chrome or if I ran firefox on a linux distribution, I would get a different string. On every GET Request submitted by the browser, this same User-Agent string is submitted.

Discussion

In this section, different types of GET Requests were examined. In the first two questions, the GET HTTP Request was created using the curl command and the headers had to be specified using User-Agent. Then in the last question, a browser was used to issue a GET request for https://www.cs.odu.edu/ mweigle/courses/cs532/ua_echo.php and the browser automatically created a User-Agent string for the header. One interesting takeaway from this exercise is that a webserver can very easily identify if a browser or another program is issuing a GET request because the browser sends a User-Agent string and other programs do not by default.

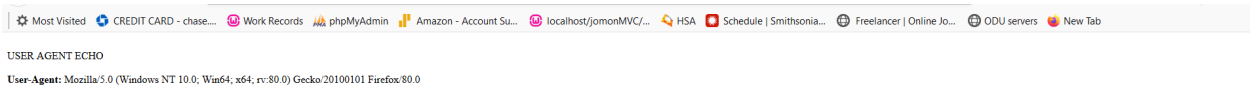

Figure 3: Response 3

Q3

Write a Python program to find links to PDFs in a webpage.

Your program must do the following:

- take the URI of a webpage as a command-line argument
- extract all the links from the page
- for each link, use the Content-Type HTTP response header to determine if the link references a PDF file
- for all links that reference a PDF file, print the original URI (found in the source of the original HTML), the final URI (after any redirects), and the number of bytes in the PDF file. (Hint: Content-Length HTTP response header)

Show that the program works on 3 different URIs, one of which must be https://www.cs.odu.edu/ mweigle/courses/cs532/pdfs.html Here is a snippet of the expected operation:

```
% python3 findPDFs.py https://www.cs.odu.edu/~mweigle/courses/cs532/
   pdfs.html

URI: http://www.cs.odu.edu/~mln/pubs/ht-2015/hypertext-2015-temporal-
   violations.pdf
Final URI: https://www.cs.odu.edu/~mln/pubs/ht-2015/hypertext-2015-
   temporal-violations.pdf
```

```
Content Length: 2,184,076 bytes

URI: http://www.cs.odu.edu/~mln/pubs/tpdl-2015/tpdl-2015-annotations.
   pdf
Final URI: https://www.cs.odu.edu/~mln/pubs/tpdl-2015/tpdl-2015-
   annotations.pdf
Content Length: 622,981 bytes
```

Listing 6: Expected Output

Answer

Listing 7 is the imported code I used to scrape pdf files from webpages.

```
1  from bs4 import BeautifulSoup #for parsing html
2  import requests #for getting final uri
3  import random # used for User Agent Function
4  import sys #for command line argments
5  import re # for regex method of validating PDFs
6  from urllib.parse import urljoin  # for converting relative to absolute
       paths
7
8  #code for handling user-agent to deal with some webservers
9  #GET_UA() code derived from
10 # https://www.jcchouinard.com/random-user-agent-with-python-and-
       beautifulsoup/
11 # User Agent Strings obtained using https://www.whatismybrowser.com/
12 def GET_UA():
13     uastrings = ["Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:80.0)
       Gecko/20100101 Firefox/80.0",
14                  "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit
       /537.36 (KHTML, like Gecko) Chrome/85.0.4183.83 Safari/537.36",
15                  "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit
       /537.36 (KHTML, like Gecko) Chrome/85.0.4183.83 Safari/537.36 Edg
       /85.0.564.44",
16                  "Mozilla/5.0 (Windows NT 10.0; WOW64; Trident/7.0;
       Touch; rv:11.0) like Gecko",
17                  "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit
       /537.36 (KHTML, like Gecko) Chrome/84.0.4147.135 Safari/537.36 OPR
       /70.0.3728.160",
18                 ]
19     return random.choice(uastrings)
20
21 #converts relative links to absolute paths
22 # absolutePath code derived from
23 # https://stackoverflow.com/questions/476511/resolving-a-relative-url-
       path-to-its-absolute-path
```

```
24 def absolutePath(webpage, url):
25     absoluteUrl = url
26     if not url.startswith('http://www.'):
27         absoluteUrl = requests.compat.urljoin(webpage, url)
28     elif not url.startswith('https://www.'):
29         absoluteUrl = requests.compat.urljoin(webpage, url)
30     else:
31         absoluteUrl = url
32     return absoluteUrl
33 
34 # code for handling system arguments
35 def checkForArguments():
36     #code for argument handling
37     try:
38         webpage = sys.argv[1]
39     except Exception as error:
40         print("Error: A url is a required argument to find pdf links")
41     return webpage
42 
43 # gets response from http request or returns none
44 def validateRequest(url):
45     headers = {'User-Agent': GET_UA()}
46     response = None
47     try:
48         response = requests.get(url, headers=headers)
49     except Exception as error:
50         print("Unable to receive response from ", url)
51         print("Try another webpage address")
52     return response
53 
54 def checkIfPDFRegEx(url):
55     #use regex to check if link ends in .pdf
56         m = re.search('.*(\.pdf)$', url)
57         # anything matches, then must be a pdf
58         if m != None:
59             return validateRequest(url)
60         return None
61 
62 def printResults(response, link):
63     print("URI:", link) #link from anchor tag
64     print("Final URI:", response.url) #obtained from response
65     #check for content-length
66     if 'Content-length' in response.headers:
67         print(response.headers['Content-length'], ' bytes')
68     else:
69         print('File Size Unknown')
70     print() # print a new line for aesthetics
```

```
71
72 def main():
73     webpage = checkForArguments()
74     if webpage == '': return
75     #get headers and text for url
76     response = validateRequest(webpage)
77     # parse response text using Beautiful soup to get links
78     links = BeautifulSoup(response.text, 'html.parser').findAll('a')
79
80     for link in links:
81         #check if relative path and convert to absolute path
82         url = absolutePath(webpage, link['href'])
83         #check if pdf, and get http request
84         response = checkIfPDFRegEx(url)
85         if (response):
86             #print results
87             printResults(response, url)
88
89 if __name__ == '__main__':
90     main()
```

Listing 7: FindPDFs.py

Below I have provided screenshots of various sample output from the findPDFs.py scraper. 4 is the output I received after running the findPDFs.py scraper on https://www.cs.odu.edu/ mweigle/courses/cs532/pdfs.html

5 is the output I received after running the findPDFs.py scraper on the Ubuntu manual page (https://help.ubuntu.com/)

6 and 7 are screenshots of the output I received after running the findPDFs.py scraper on the CRAN-R R programming manuals page (https://cran.r-project.org/manuals.html)

8 and **??** are screenshots of the output I received after running the findPDFs.py scraper on the Society of East Asian Archaeology's 2018 bulletin page (https://seaa-web.org/publications/bseaa/bulletin-society-east-asian-archaeology)

Discussion

The code above generates a console program that accepts a url parameter from the user . If no url is provided, the program exits. Next a HTTP GET request is made to the URL sent with a User-Agent header to allow the code to run on more webservers. Once the response from the webserver that the URL points to is received, the resulting text from the response is parsed using the BeautifulSoup library to retrieve all of the anchor ('a') tags in the document. Then every anchor's 'href' attribute is tested to see if it is an absolute path and if the url is a relative path then

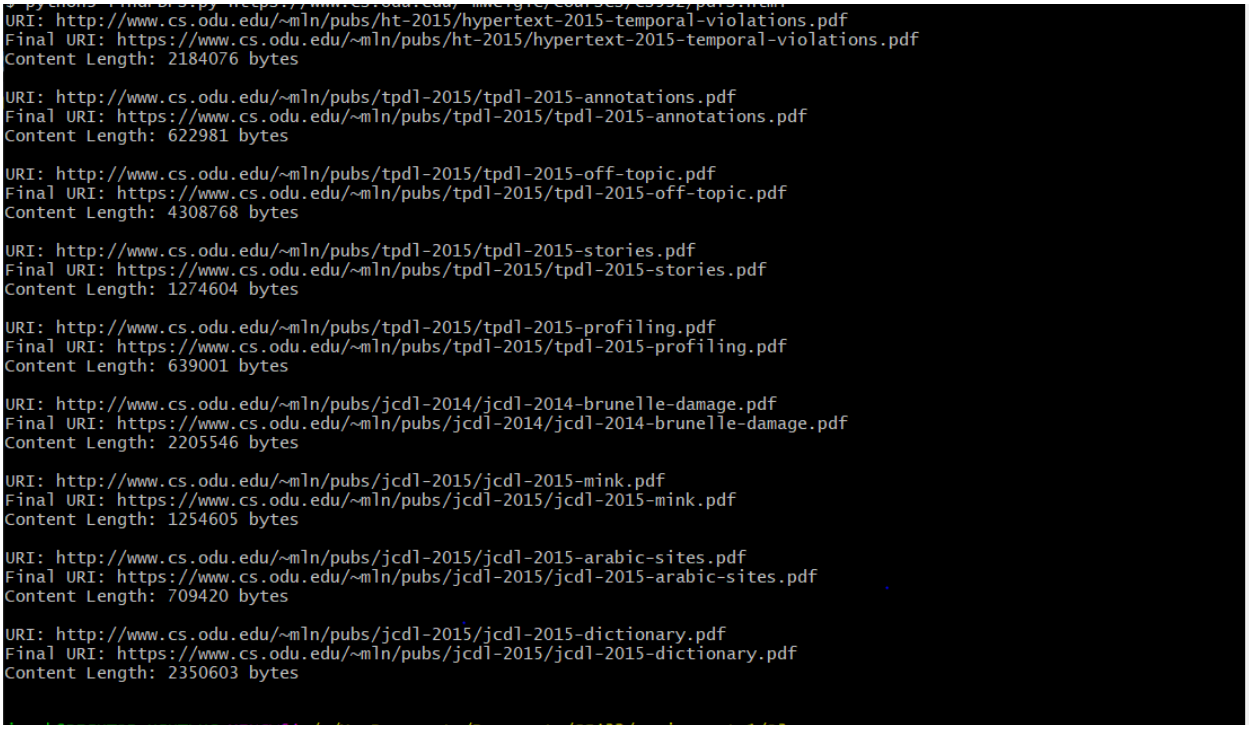

Figure 4: findPDFs.py results for https://www.cs.odu.edu/ mwei- gle/courses/cs532/pdfs.html

it is converted to be an absolute path. After the link is in the correct form, it is checked to see if it is a pdf and another HTTP Request is made for each URL. The resulting response from the HTTP request is used to print the URI, Final URI and file size output to the console.

In order to accomplish task, I have created a main function with 6 other functions. The The main function begins with retrieving the url from the user input. THe user input is handled using the checkForArgumentsfunction. This function uses Python's built in sys library's argv function to get the first command from the user. A conditional is used to check if the the argv variable exists and if not, it exits the program. This function only cares that there is a second parameter after the findPDFs.py command and does not evaluate if it is a valid url or not.

The validateRequest function validates the user's url and any other uri, we provide it. Since some webservers do not like webscrapers and will reject their GET requests, the validateRequest function begins with building a header and generates a User_Agent string to mimic a browser's User-Agent string using a modified version of the GET_UA function.[2]. TheThe GET_UA function is simply a list of User Agents that I created using 5 different browserson Windows 10 (Firefox, Chrome, Eduge, Opera, and Internet Explorer) and obtained by visiting whatismyip.com [3] to get the User Agent. Then this function use Python's built in random library to pick one of these User_Agent strings at random. The validateRequest function uses the resulting User_Agent to build a header. A try catch block is used to try a HTTP GET request using the requests library. If a response is received, the validateRequest function returns it. If it fails, an Exception is generated and the function returns None.

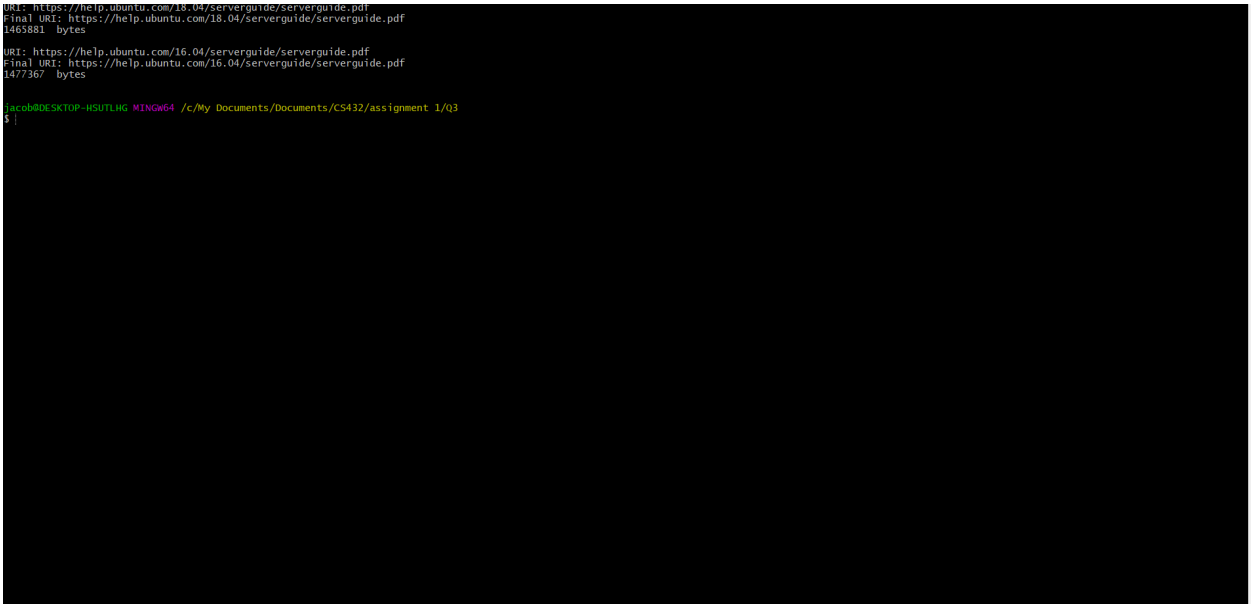

Figure 5: findPDFs.py results for https://help.ubuntu.com/

After retrieving the response, the /emphmain function uses the BeautifulSoup library to parse the HTM to retrieve all anchor tags. There are numerous HTML parsers available including python's built in HTMLParser library, but in order to make use of that library to get all of the anchor tags, a subclass of HTMLParser has to be created and various methods have to be overriden. the BeautifulSoup library is able to do this with a single line that initializes the BeautifulSoup object with the HTTP GET request's response's text and then calls the FindAll method to retrieve all anchor tags. So BeautifulSoup was used in this case for the sake of conciseness. BeautifulSoup returns a list-like object that is iterated through using a for loop in order to check each anchor tag's 'href' attribute.

Now that I have a list of links, I need to format them so they are ready for the response requests. The first modification made is ensure the url is an absolute path. Since many websites make use of relative URLs to make the code easier to port between testing and production environments, I have modified an absolutePath function [4] to check if a url starts with "http://www" or "http://www" and if not, the function assumes the link found was a relative url. In order to convert the relative url to an absolute url, I made use of python's urllib library to combine the two urls using the parse.urljoin method.

Next the The /emphcheckIfPDFRegEx function is used to check if the link is a pdf. This function uses the re (regular expression) libary to check if a url ends in '.pdf' and if it ends in .pdf, the function assumes the link is a pdf. This will find all pdfs as long as the file has the extension. I found a website [5] that has several PDFs that cannot be retrieved by this due to not having the .pdf extension. An alternative approach to checking for PDFs is to make a GET or HEAD request using the Request library. The resulting response has a headers method that can be called that may contain the property Content-Type. If the property exists, you can check if it is is application/pdf and if so, the link must be a pdf. Unfortunately, the Content-Type property does not always exist.

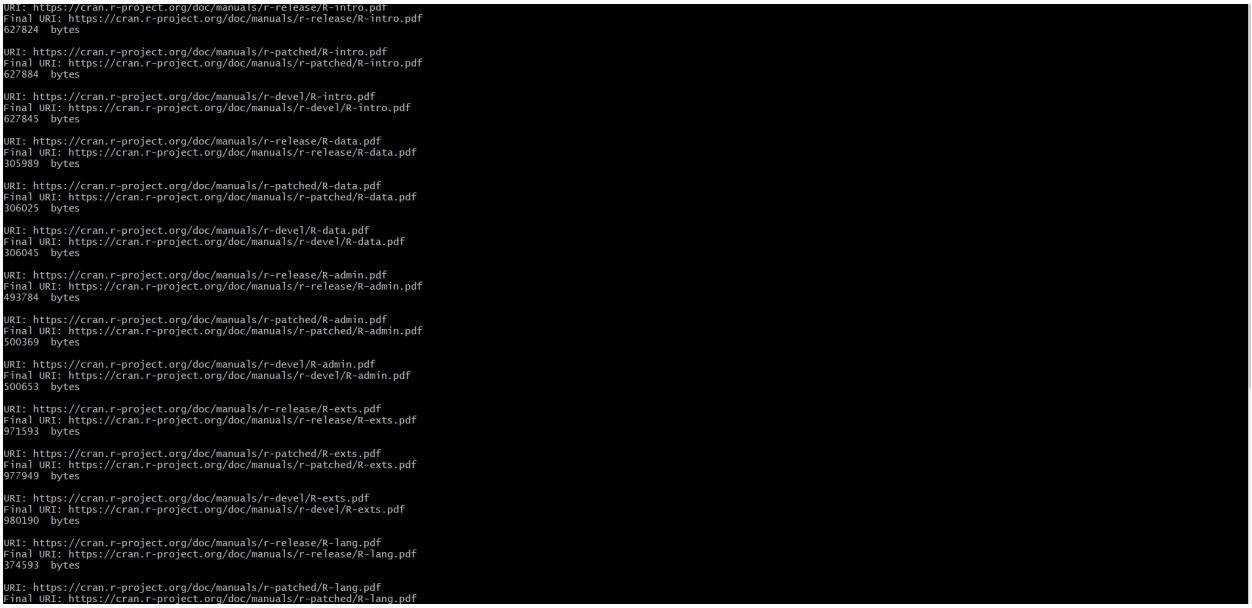

Figure 6: findPDFs.py results for https://cran.r-project.org/manuals.html

Figure 7: findPDFs.py results for https://cran.r-project.org/manuals.html

Since thiis approaach does not always work and results in numerous GET or HEAD requests that could result in my webscraper being blocked by the webserver, I found the Header approach to identify pdfs to be less desirable than the regular expression approach. After verifying the url is a pdf link, the checkIfPDFRegEx function calls the validateRequest function to generate a new GET request and returns the response. If a link is not a PDF, then the function returns None.

The main function ends by calling a the printResult function on all of the responses returned from checkIfPDFRegEx. This function displays the original URI, the final URI (after any redirects) and the size of the pdf file. The URI is created using the original link url (after converting to an absolute path). The final URI is obtained using the GET Request response's url property. The Request library by default will set the url to the final redirect unless the GET Request is sent with an additional argument of allow-redirects=False. If allow-redirects is set to false, then GET requests would have to be made on the resulting response.url until the response.status_code is no longer 300, 301, 302, 303, 307, or 308. This is quite inefficient and the more HTTP requests to a server in a short period of time, increases the likleihood of being the server blocking the webscraper. Instead, a normal GET request using the requests library can be used and the url property of the response is the final URI after any redirects. If a history of redirects is needed,

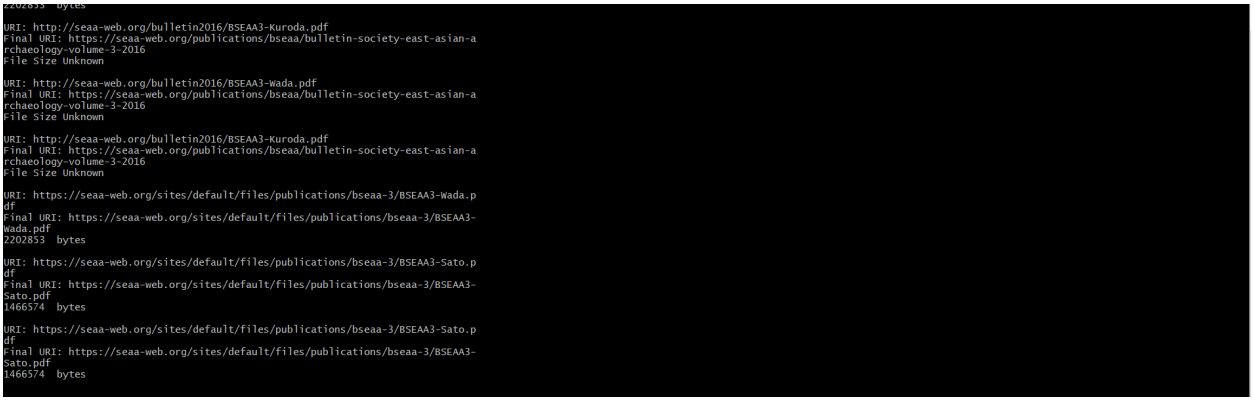

Figure 8: findPDFs.py results for https://seaa-web.org/publications/bseaa/bulletin-society-east-asian-archaeology

the history property of the response can be looped through to get all URIs inbetween [6]. The file size of the PDF is found using the 'Content-length' found in the response's header. Since header properties can be absent on certain sites, I have made use of a loop to check if it exists, prior to displying the file size to avoid an exception. I rand into this issue when running the program on the Soceity for East Asian Archaeology's site [7]. Technically everything used in the printResults can be used using a HEAD Request, however many webservers in an attempt to avoid scrapers do not allow HEAD requests, so a GET request was used. Once all links to pdfs have been displayed, the program ends.

This code is a fairly decent general purpose webscraping tool for pdfs on most web servers. However I found a few limitations. Some web servers will generate the "'Unable to receive response from javascript:void(0)". For example I attempted to run the code on Google Scholar search results for Jomon (a prehistoric culture in Japan) [8] and the program generated this exception. In cases like this, Google's server has blocked my webscraper from running the GET Requests because it is able to identify this is a scraper and not a regular browser. On clever servers like this, I can't make the requests library work and a headless browser approach using a library like Selenium would have to be used. Another limitation I came across and was suprised to see was that some web servers will load files without the file extension. This again is hard to identify. The only ways around this is to check the content-type in the header if it is available or to tailor your scraper to a single website. Websites with lots of pds tend to follow patterns for CSS formatting nd javascript purposes, so it is possible all anchor tags linking to pdfs may be stored uner a div with a unique class. In cases like these a specialized approach to PDF scraping must be used.

References

[1] The Dia Developers. Dia diagram editor. `http://dia-installer.de/index.html.en`. Accessed on 2020-9-07.

[2] JR Oakes. Randomize user-agent with python and beauti-

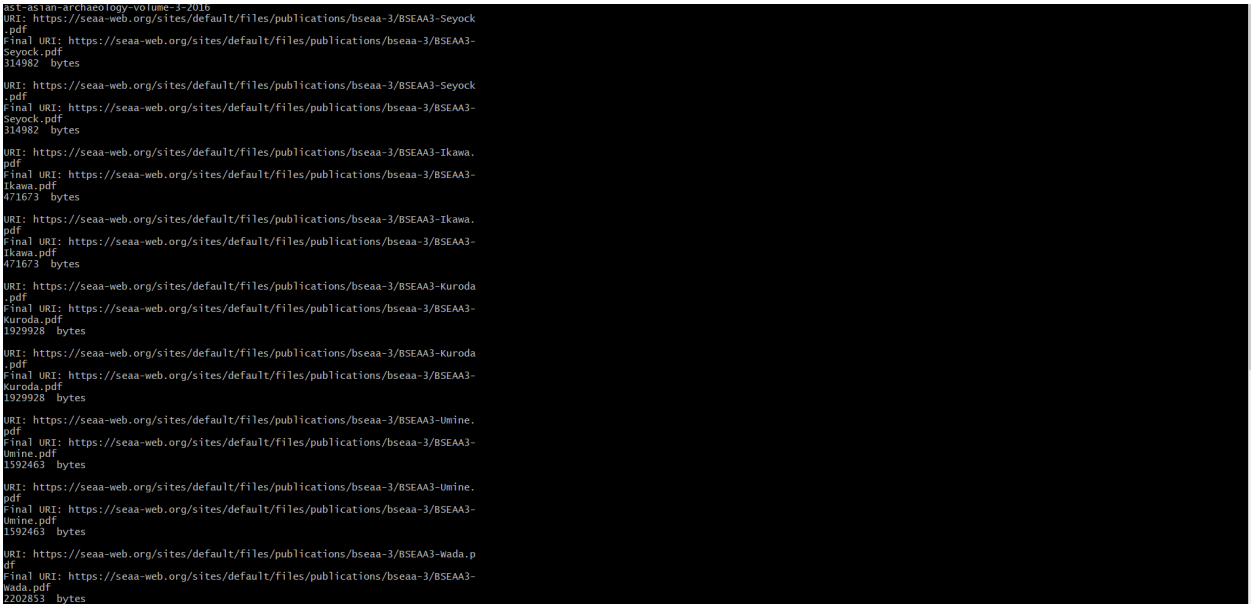

Figure 9: findPDFs.py results for https://seaa-web.org/publications/bseaa/bulletin-society-east-asian-archaeology

fulsoup (by jr oakes). https://www.jcchouinard.com/random-user-agent-with-python-and-beautifulsoup/, February 2020. Accessed on 2020-9-07.

[3] Whatismyip.com user agent info. https://www.whatismyip.com/user-agent-info/. Accessed on 2020-9-07.

[4] StackOverflow. Resolving a relative url path to its absolute path. https://stackoverflow.com/questions/476511/resolving-a-relative-url-path-to-its-absolute-path. Accessed on 2020-9-07.

[5] The Anthropological Society of Nippon. Anthropological science volume 101, issue 1. https://www.jstage.jst.go.jp/browse/ase/101/1/_contents/-char/en. Accessed on 2020-9-07.

[6] StackOverflow. Python requests library redirect new url. https://stackoverflow.com/questions/20475552/python-requests-library-redirect-new-url. Accessed on 2020-9-07.

[7] Bulletin of the society for east asian archaeology volume 3 (2016). https://seaa-web.org/publications/bseaa/bulletin-society-east-asian-archaeology-volume-3-2016. Accessed on 2020-9-07.

[8] Google. Google scholar. https://scholar.google.com/scholar?hl=en&as_sdt=0%2C47&q=jomon&btnG=. Accessed on 2020-9-07.