Web Crawler Report

Utkarsh Pant : 22B0914

Contents

1 Introduction

Web Crawler is a computer program that is used to search and automatically index website content and other information over the internet. These programs, or bots, are most commonly used to create entries for a search engine index. This project performs web crawling and link analysis on a given website or set of websites. It takes command-line arguments to specify the target URL(s), recursion depth, output file name, and various options for link analysis.

2 Methodology

The following procedure was followed by me to implement a web crawler.

1. Constructing a function to extract the links into a given set using python libraries. This function also converts all relative hyperlinks into absolute hyperlinks before storing them.
2. Filtering out all the internal links from the set in which the links were extracted. These links are used in further levels of recursions.
3. Constructing a function which segregates the extracted links by extensions and then printed if called.
4. Creation of a recursive function which recursively extract links and prints them at various depths using previously specified functions.
5. Implementing and constructing the code for infinite depth web crawler using previous functions.
6. Using standard python libraries to take command line arguments as inputs and handling errors. Further, addition of new command line arguments as customisations.

3 Implementation

To implement the above stated procedure I used a lot of python libraries and modules for doing various functions.

- For the extraction of links I used **get** function from **requests** library. This code sends an HTTP GET request to the specified url and stores the response in the response variable and **BeautifulSoup** function from **bs4** library parses it. Then all links in tags a, link, script and img were extracted and stored in links set. **urljoin** function from **urlib.parse** library was used by me to convert relative to absolute links.
- In order to filter out the internal links for further stages of recursion I used **urlparse** function from **urlib.parse** library which parses a url into its individual components. Then I compared the **netloc** attribute which stands for net location for filtering out the urls.
- To get the extensions of the URLs, I used **urlparse** and **os.path** modules to parse a URL, extract the **path** component, and retrieve the file extension from the **path.get**. Then, I created a dictionary with keys as extensions and values as set of urls with respective extensions. If specified by the user there is a provision of not sorting them as well.
- In order to crawl a site for finite threshold. I took the help of recursion. The **crawl()** function takes in a set of urls and depth as input and returns the function if depth is zero. Else it iterates through the url set and prints the output as required. It also contains functions to print all urls or to print only the count depending on the output requirement. Finally it filters out the internal links and calls the function again this time depth decreased by one.

- To take in command line inputs and arguments I used **ArgumentParser** class from **argparse** library. In order to add new arguments I used **add_argument** method from the given class . I also used this library for error handling and specifying conditions of argument input. For example I set *Required = True* for "−**url**" which displays an error message is its requirement in not satisfied.
- I used **sys** library to print error error statements in the standard error stream using **sys.stderr** as well as exit the code using **sys.exit(1)**. I also used this for file handling as well by changing the **sys.stdout** to given output file and then changing it back to **sys.__stdout__**. Hence the code *sys.stdout = file* is always seen in the source code when file is opened.
- Finally I used **warnings** library to suppress all command line warnings and **time** library to add additional functionality as discussed ahead.

4 Understanding the Code

4.1 Importing Libraries

The script begins by importing the necessary libraries: requests for making HTTP requests, **BeautifulSoup** from **bs4** for parsing HTML content, **urlparse** and **urljoin** from **urllib.parse** for URL manipulation, **os.path** for working with file paths, **argparse** for parsing command-line arguments, **sys** for system-related operations, time for measuring execution time, and warnings for suppressing warning messages.

4.2 Command Line Inputs

The script defines the command-line arguments using **arg_parser.add_argument()**. The **-u** or −**url** argument expects one or more website links to crawl and is required. The **-t** or −**threshold** argument specifies the recursion depth threshold. The **-o** or −**output** argument specifies the output file name. The **-c** or −**count** argument is a flag that indicates whether to count the number of each link extension. The **-d** or −**desegregate** argument is a flag that indicates whether to desegregate the links. The **-i** or −**internal** argument is a flag that indicates whether to print only internal links. The **-T** or −**time** argument is a flag that indicates whether to print the execution time. The **iurl** variable is set to a set of the URLs provided as command-line arguments. The **threshold** variable is set to the value of the threshold argument. The **output_file** variable is set to the value of the output argument.

4.3 Error Handling

Error handling is performed to check if the **iurl** set is empty. If it is empty, an error message is printed, and the script exits with a status of 1. Additional error handling is performed to check if the **threshold** value is invalid (less than or equal to zero). If it is invalid, an error message is printed, and the script exits with a status of 1. If an output_file is specified, it is opened in write mode to create or overwrite the file

4.4 Functions

The **extract_links()** function is defined. It takes a URL and parses the response content using **BeautifulSoup**, and extracts the links from the specified tags. The extracted links are added to the links set.
The **extension_segregator()** function is defined. It takes a set of URLs as input, iterates over each URL, extracts the file extension using **os.path.splitext**, and categorizes the URLs based on their extensions. The categorized URLs are stored in a dictionary **ext_seg** where the keys are the extensions and the values are sets of URLs with that extension.
filter_internal_links() takes a set of links and a base URL as input, extracts the netloc (domain) component from the base URL, and filters the links to include only those with the same netloc.

The **seg_print()** function is defined. It takes a set of extracted links and the extension segregation dictionary as input. The function prints the total number of links found and iterates over the extensions in the dictionary. For each extension, it prints the count of URLs with that extension and the URLs themselves. Function **c_print** only prints total number of links found.// The **crawl()** function is defined. It takes a set of URLs, a depth value, and a recursion level as input. If the depth is zero, the function returns.
Otherwise, it extracts links from the given URLs using **extract_links()**, applies optional filtering based on the command-line arguments (**args.internal** and **args.desegregate**), and performs extension segregation using **extension_segregator()**. If an output file is specified, the output is redirected to the file. The link analysis results are printed using **c_print()** or **seg_print()**, depending on the **args.count** flag. Internal links are filtered using **filter_internal_links()**, and the **crawl()** function is recursively called with the new set of internal links and decreased depth.

4.5 Main

The **start** variable is set to the current time using **time.time()** to measure the execution time. If the **threshold** is None, it means the recursion depth is set to infinite. The **print_links_recursive** function is called to extract links recursively for each URL in the **iurl** set. Filtering and link analysis are performed similar to the previous case.The **end** variable is set to the current time to mark the end of the execution. If the **args.time** flag is set, the total execution time is printed.

5 Additional Features

The following features were also added by me

- Multiple URL arguments : If multiple urls are present besides -u the code crawls both of the sites simultaneously and gives combined output.
- count argument : Typing **-c** in the command line argument outputs only the numbers of total and individual extension types.
- desegregation argument : Typing **-d** in command line results in the output not being segregated by extension types and instead as a list of all links.
- internal links argument : Typing **-i** in the command line prints only the internal links of the given iteration.
- execution time argument : Typing **-T** in the command line prints execution time of the given command in the terminal shell
- help argument : Typing **-h** in the shell displays all arguments and methods as well as their usage.

6 Source Code

```
#importing required libraries
import requests
from bs4 import BeautifulSoup
from urllib.parse import urlparse, urljoin
import os.path
import argparse
import sys
import time
import warnings
warnings.filterwarnings("ignore") #suppress warnings

#defining all the command line arguments
arg_parser = argparse.ArgumentParser()
```

```
arg_parser.add_argument("-u", "--url",nargs="+", help="Website link to crawl",
                                        required=True)
arg_parser.add_argument("-t", "--threshold", help="Threshold of recursion depth",
                                        const=None, type=int)
arg_parser.add_argument("-o", "--output", nargs="?", const=None, help="output file
                                        name")
arg_parser.add_argument("-c", "--count", help="Number of Each extension",action="
                                        store_true")
arg_parser.add_argument("-d", "--desegregate", help="Desegregates the links",action
                                        ="store_true")
arg_parser.add_argument("-i", "--internal", help="prints only internal links",
                                        action="store_true")
arg_parser.add_argument("-T", "--time", help="prints execution time",action="
                                        store_true")
args = arg_parser.parse_args()

iurl = set(args.url)
threshold = args.threshold
output_file = args.output

#error handling
if not iurl:
    print("URL is required!", file=sys.stderr)
    sys.exit(1)

if threshold is not None:
    if threshold <= 0:
        print("Invalid threshold!", file=sys.stderr)
        sys.exit(1)

if output_file:
    with open(output_file, "w") as file:
        file.close()
    #Empties the output_file if it was non empty


tags = ['a', 'link', 'script', 'img']#attributes to be checked for scraping

# Function to extract links from a URL into a set
def extract_links(url,links):

    try:
        response = requests.get(url)
        soup = BeautifulSoup(response.content, 'lxml')

        for tag in soup.find_all(tags):

            href = tag.get('href')
            if href:
                links.add(urljoin(url, href))
            src = tag.get('src')
            if src:
                links.add(urljoin(url, src))
    except requests.exceptions.RequestException:
        pass

#Function to segregate links based on their extensions into a dicrionary
def extension_segregator(url_set):
    try:
        ext_seg = {}
        for link in url_set:
            parsed_link = urlparse(link)

            path = parsed_link.path
            extension = os.path.splitext(path)[1]

            if extension in ext_seg.keys():
                ext_seg[extension].add(link)
            else:
```

```
                ext_seg[extension] = set()
                ext_seg[extension].add(link)

        return ext_seg

    except requests.exceptions.RequestException:
        pass

# Function to filter internal links based on command line URLs
def filter_internal_links(links, base_url = iurl):

    base_netloc= set()
    for base_link in base_url:
        base_netloc.add(urlparse(base_link).netloc)
    internal_links = set()

    for link in links:

        parsed_link = urlparse(link)
        if parsed_link.netloc in base_netloc:
            #compares net  location of links in input set and links
            internal_links.add(link)
    return internal_links

# Function to print segregated links and their extensions
def seg_print(extracted_links,ext_seg):


    sorted_dict = sorted(ext_seg.items(), key = lambda x:-len(x[1]))#sorting by
                                                    number of files per extension
    converted_dict = dict(sorted_dict)
    print("Total Links found:",len(extracted_links))

    for extension in converted_dict.keys():
        if extension == "":
            print("Miscellaneous : {}".format(len(converted_dict[extension])))
        elif extension == "Desgregate Links":
            pass
        else:
            print("{} : {}".format(extension,len(converted_dict[extension])))
        for url in converted_dict[extension]:
            print(url)
    print()

# Function to print link counts based on extensions
def c_print(links,ext_seg):
    sorted_dict = sorted(ext_seg.items(), key = lambda x:-len(x[1])) #sorting
    converted_dict = dict(sorted_dict)
    print("Total Links found:",len(links))
    for extension in converted_dict.keys():
        if extension == "":
            print("Miscellaneous : {}".format(len(converted_dict[extension])))
        elif extension == "Desgregate Links":
            continue
        else:
            print("{} : {}".format(extension,len(converted_dict[extension])))
    print()

# Function to recursively print links particular for infinite depth search
def print_links_recursive(url, visited_links):

    if url in visited_links:
        #base case
        return

    visited_links.add(url)
    links = set()
    extract_links(url, links)
    internal_links = filter_internal_links(links)
```

```
    visited_links.update(links - internal_links) #adding external links without
                                                  iterating through them
    for link in internal_links:
        #iterating through all internal links
        print_links_recursive(link, visited_links)

# Function to crawl and analyze links
def crawl(url_set,depth,rec_lvl = 1):

    if depth == 0:
        #base case
        return

    links = set()
    for link in url_set:
        extract_links(link,links)
    if args.internal:
        links = filter_internal_links(links)
    if args.desegregate:
        ext_seg = {"Desgregate Links":links}
    else:
        ext_seg = extension_segregator(links)
    if output_file:
        with open(output_file, "a") as file:
            sys.stdout = file
            if args.count:
                print("At recursion level",rec_lvl)
                c_print(links,ext_seg)
            else:
                print("At recursion level",rec_lvl)
                seg_print(links,ext_seg)
            sys.stdout = sys.__stdout__
    else:
        if args.count:
            print("At recursion level",rec_lvl)
            c_print(links,ext_seg)
        else:
            print("At recursion level",rec_lvl)
            seg_print(links,ext_seg)

    new_urlset = filter_internal_links(links)
    crawl(new_urlset,depth-1,rec_lvl+1)


start = time.time()

if threshold is None:
    #infinite depth search

    links =set()

    for url in iurl:
        #iterating through all input urls
        print_links_recursive(url,links)
    if args.internal:
        links = filter_internal_links(links)
    if args.desegregate:
        ext_seg = {"Desgregate Links":links}
    else:
        ext_seg = extension_segregator(links)

    if output_file:
        with open(output_file, "a") as file:
            sys.stdout = file
            if args.count:
                print("At recursion level infinite")
                c_print(links,ext_seg)
            else:
                print("At recursion level infinite")
```

```
                seg_print(links,ext_seg)
            sys.stdout = sys.__stdout__
    else:

        if args.count:
            print("At recursion level infinite")
            c_print(links,ext_seg)
        else:
            print("At recursion level infinite")
            seg_print(links,ext_seg)

else:

    crawl(iurl,threshold)

end = time.time()

if args.time:
    print(f"Time taken: {(end-start)*1000:.03f} ms")
```

7 Gallery

Figure 1: Infinite Depth Search

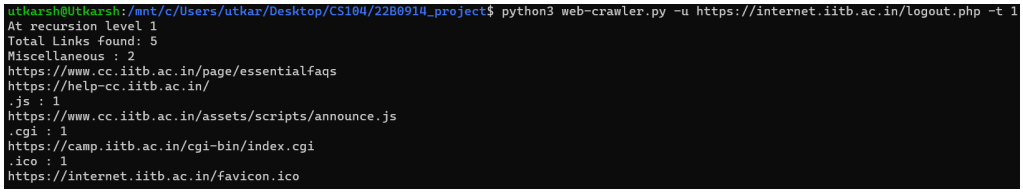

Figure 2: Segregation by Extension

Figure 3: python3 web-crawler.py -u https://moodi.org/ -t 1 -o output.txt

References

[1] OpenAI. Chatgpt. https://www.openai.com/chatgpt, 2023.

[2] Stack Exchange Inc. Stack overflow. https://stackoverflow.com, 2023.

[2] [1]