Assignment 1 Report

Monday 13th February 2023

Group Members

- **Aidan Traboulay** 200115590 - trab5590@mylaurier.ca
- **Mobina Tooranisama** 200296720 - toor6720@mylaurier.ca
- **Nausher Rao** 190906250 - raox6250@mylaurier.ca

Contributions

Aidan Traboulay 200115590

- In `utils.py`: I wrote the `session_handler()`, set up the user-agents and proxies. Wrote the initial iteration of the `get_content(url)` function, later updated by Nausher to pass in params. Additionally, I wrote the `get_page(base_url, params)` function. I also initially wrote the `write_raw_data(content, url)` function and the `hash_url(url)` function.
- Also wrote the loading animation in the terminal with the cute giraffe (for fun).
- In `webcrawler2.py`: I built the entirety of the Google Scholar crawler.
- Wrote the report for `webcrawler2.py` and for `utils.py`.

Mobina Tooranisama 200296720

- In `webcrawler1.py`: I wrote the `crawl_urls(url, max_depth, rewrite, verbose)` function. I aso wrote the `get_dt()` function to retreive the current datetime.

Nausher Rao 190906250

- In `utils.py`: I wrote the `parse_url(url)` function, which takes a fully-valid URL, and dissasembles it into a base URL and a dictionary of parameters to be used by the `requests` library. I also edited the `get_page(baseUrl, params)` function to pass in the parameters, as well as edited the `write_raw_data(content, url)` to write the raw HTML to a data folder properly.
- In `webcrawler3.py`: I built the entirety of the Tag & Graph crawler.
- Wrote the report for `webcrawler3.py` and for `utils.py`.

Explanations

All three crawlers were written using `Python 3.10`, utilizing the `BeautifulSoup`, `matplotlib`, `numpy`, `requests`, `sys`, `re`, and `json` libraries, as well as the utility file, which contained all the set up and hashing functions for each crawler.

Utilities (`utils.py`)

The purpose of this file is to handle the session of the crawler using the `requests` library, this is contained within the `session_handler()` function. Moreover, it creates a `BeautifulSoup` object and returns the soup, within the `get_content(url)` function. A `parse_url(url)` function was also created so that when a user enters a URL argument, it fixes the error that would arise with the special characters. Another solution would have been to wrap the URL in quotations, however, this would force the user to always remember to make the URL a string. The next major function created would be the `get_page(base_url, params)` function which would send a GET request to the URL, passing in the user-agent headers, any extra parameters and the proxy of the current user's OS, it returns the response if the response is in the

range of `200-299` (a successful or ok response). Otherwise, it will return a message with the non-ok or non-successful status code. Additionally, we have a file handler which writes the raw HTML to a data folder, called `write_raw_data(content, url)`. Finally we have the hashing function called `hash_url(url)` which returns a SHA256 encoding of any given url.

Depth & Logger Crawler (`webcrawler1.py`)

The `crawl_urls` function is a web crawler that retrieves HTML pages, extracts all the URLs from the page, and recursively continues the process to a certain maximum depth.

It first calls the `get_page` function to retrieve the HTML content of the page at the specified URL. If the response is not None, it uses the BeautifulSoup library to parse the HTML content and extract all the hyperlinks on the page.

It then creates a hashed version of the URL using the `hash_url` function and a datetime stamp using the `get_dt` function. It uses these two values to create a filename for the page content and writes the parsed HTML content to a file with that name. It also writes a log entry to the crawler1.log file, recording the hashed URL, the original URL, the datetime stamp, and the HTTP response.

If the maximum depth of the crawl is not reached, the function then iterates over all the extracted links and calls itself recursively, with the link as the new URL and a decremented maximum depth. If the verbose flag is set to True, the function will print the URL and its depth as it is crawled.

It first calls the `get_page` function to retrieve the HTML content of the page at the specified URL. If the response is not None, it uses the BeautifulSoup library to parse the HTML content and extract all the hyperlinks on the page. It then creates a hashed version of the URL using the `hash_url` function and a datetime stamp using the `get_dt` function. It uses these two values to create a filename for the page content and writes the parsed HTML content to a file with that name. It also writes a log entry to the crawler1.log file, recording the hashed URL, the original URL, the datetime stamp, and the HTTP response. If the maximum depth of the crawl is not reached, the function then iterates over all the extracted links and calls itself recursively, with the link as the new URL and a decremented maximum depth. If the verbose flag is set to True, the function will print the URL and its depth as it is crawled.

The `main` function is the entry point of the script, it first processes the command line arguments and retrieves the URL and the maximum depth of the crawl. If either of these values is not provided, an error message is displayed. It then starts a session and displays a giraffe and a loading animation, and finally calls the `crawl_urls` function with the specified parameters. In summary, the script is a simple implementation of a web crawler that retrieves pages, extracts all links from those pages, and writes the content of the pages to files. It also writes log entries for each page it crawls.

Google Scholar Profile Crawler (`webcrawler2.py`)

The purpose of this crawler was to scan and parse through all the relevant data on any given Google Scholar profile. The structure of the program is as follows: `def get_parsed_content(url)` - this function handles all the content of the raw HTML from the BeautifulSoup object. Majority of the parsing is done using the `.find()` or `.find_all()` methods. In the event that data of the same type needs to be collected from one search I utilized Python's list comprehension methodology; an example of this would be where I would need to find the researcher's keywords:

```
researcher_keywords = [keywords.get_text() for keywords in soup.find_all("a", class_="gsc_prf_inta gs_ibl")]
```

Additionally, in the case where I would need to get various information for one person or paper, I created an empty array where a dictionary of data will be stored and linked together. An example of this would be the `researcher_coauthor_dict` which stored the a dictionary of the coauthor's name, title and relevant link together:

```
researcher_coauthor_dict = []
for coauthor in researcher_coauthor_content:
        researcher_coauthor_dict.append({
                "coauthor_name": coauthor.find("a").get_text(),
                "coauthor_title": coauthor.find("span", class_="gsc_rsb_a_ext").get_text(),
                "coauthor_link": coauthor.find("a", href=True)["href"]
        })
```

Tag & Graph Crawler (`webcrawler3.py`)

The purpose of this crawler is to visually quantify the different parts of a webpage, and what sections are more important than others. The crawler first calls the `get_page` function to retrieve the HTML content of the page at the specified URL. If the response is not None, it uses the BeautifulSoup library to parse the HTML content.

These contents are then ran through three different regular expression substitutions to convert all non-HTML tags into 0's and all HTML tags to 1's. This can be seen by the below substitutions:

```
HTML_CONTENT_REGEX = r"\b\w+\b(?![^<]*>)"
```

```
HTML_TAGS_REGEX = r"<[^<]+?>"
```

There is also a third regular expression that is used to remove any outlying characters that aren't 0's or 1's, and weren't caught by the first two regexes. This can be seen by the below substitution:

```
NON_ZERO_ONE_REGEX = r"[^0-1]"
```

After this is performed, the string of bits is turned into an integer-array of bits to make performing mathematical operations on the string easier.

This bit-array is then used in two different functions; `optimize_webpage` and `generate_heatmap`. The first function uses the following mathematical function to determine the most important section of the webpage: $f(i,j) = \sum_{n=0}^{i - 1}{b_n} + \sum_{n=i}^{j}{(1-b_n)} + \sum_{n=j}^{N-1}{b_n}$

The function then returns the index of the bytes that represent the start and end of that section, represented by i^* and j^* respectively. The second function uses the `matplotlib` library to generate a heatmap of the webpage, where the darker the color, the more important that section of the webpage is.

The second function, `generate_heatmap`, uses a similar formula to the one used in the first function to generate a heatmap of the webpage, where the colours represent the importance of the section of the webpage.

Lastly, the program uses the values attained from the `optimize_webpage` function, and grabs the most important content, printing it to the hashed-URL named file.