COMP 1405Z – Fall 2021

Course Project: Analysis Report

Keaton Lee

Crawler Module (crawler.py)

Overall Design

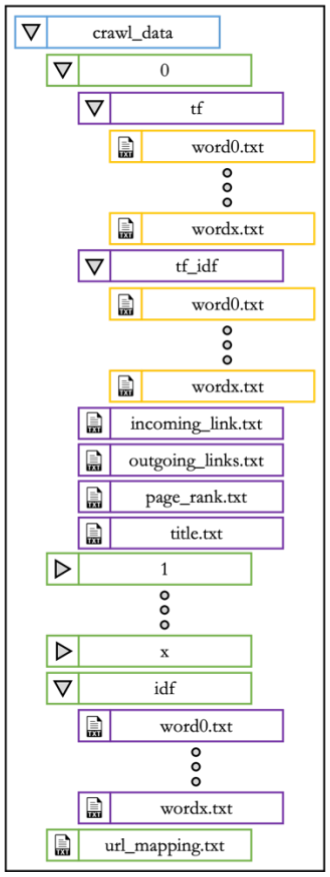
While designing my crawler module, I decided to perform the majority of the data processing necessary for my program within my crawl function in order to reduce the computation required for answering search queries. Executing more calculations during the crawl rather than during each search would increase the total runtime of the crawl but would allow for a reduction in the runtime of each search as the program would store more of the data needed for searching prior to answering search queries. I therefore decided to store the following data after each crawl: URL titles, incoming and outgoing links, inverse document frequencies, term frequencies, tf-idf weights, and PageRank values.

For my crawler module, I decided to implement multiple smaller functions which could then be called from within the 'crawl()' function in order to efficiently complete the crawl process while generating the data required for searching. The use of smaller helper functions allowed me to improve the readability of my 'crawl()' function and reduce much of the repetitive code. This section of the report covers in-depth analysis of the time complexity of each of these smaller functions called from within the crawl, followed by a description of the overall time complexity of the 'crawl()' function. The space complexity of the variables used within my crawler module functions will also be observed, with a focus on those containing a non-constant space complexity.

File Structure

File Organization

For my file organization, I decided to implement a directory structure that would contain many specific directories but reduce the amount of data stored per text file. This would allow me to simplify the file operations and allow data to be accessed simply by using the correct file path. The main directory in which data is stored is called 'crawl_data'. Within this directory, a separate directory exists for each URL which is numbered starting from zero. Within each of these directories, there are four files and two more directories. The text file 'title.txt' contains the title of the URL, 'incoming_links.txt' contains a list the incoming links on a single line, 'outgoing_links.txt' contains a list of the outgoing links on a single line, and 'page_rank.txt' contains the PageRank of the URL. The two directories 'tf' and 'tf_idf' each contain a text file per unique word present in the URL. The text files contain the data corresponding to term frequencies and tf-idf weights. Along with the directories for each URL, 'crawl_data' contains one more directory and file. The directory 'idf' contains a text file for each unique word found in any of the URLs. Within these text files are the inverse document frequencies corresponding to each word. Lastly, the file 'url_mapping.txt' contains a single URL on each line followed by the integer it was mapped to (the name of the folder which contains data on the page that URL links to).



Space Complexity

Let N represent the number of interlinked web pages.
Let M represent the maximum number of words per web page.
Worst case space complexity of 'crawl_data': $O(N \times M)$ or $O(N^2)$

Under the 'crawl_data' directory, there are the following files: the directories containing information on a specific URL, the directory 'idf', and 'url_mapping.txt'.

Since there are N URLs, there are N directories containing information on a specific URL. Each of these N directories contain: the directory 'tf', the directory 'tf_idf', 'incoming_links.txt', 'outgoing_links.txt', 'page_rank.txt', and 'title.txt'. 'title.txt' and 'page_rank.txt' both have a space complexity of O(1) given that they contain a single value. 'incoming_links.txt' and 'outgoing_links.txt' both have a maximum space complexity of O(N) since they can contain at most every URL of the interlinked web pages. The directories 'tf' and 'tf_idf' contain at most one text file for every word on a given page. Since these text files contain a single value, the space complexity of 'tf' and 'tf_idf' is O(M). The worst case space complexity of the N directories containing URL data is therefore either O(N × M) or O(N^2).

The directory 'idf' contains a text file corresponding to each unique word found on any of the interlinked web pages. Each of these text files contains a single line of information, and therefore has a space complexity of O(1). However, given that M represents the maximum number of words per web page and that there are N different web pages, if every word on each web page is unique, there can be a maximum of N × M unique words. Therefore, 'idf' has a worst case space complexity of O(N × M).

The file 'url_mapping.txt' has two values per line. Since there is a line per web page, 'url_mapping.txt' will contain at most 2N values. The worst case space complexity of 'url_mapping.txt' is thus approximately O(N).

Based on the files within the 'crawl_data' directory, the maximum space complexity of this directory structure is either O(N × M) or O(N^2). If N is greater than M, the worst case time complexity is O(N^2). If not, the worst case time complexity can be represented as O(N × M).

Function: delete_directory(dir_name)

My 'delete_directories()' function uses recursion to delete a directory as well as the files contained within such directory. The os module is used to delete these files and directories so that they can be correctly deleted regardless of the operating system of the machine in which the program is run.

Time Complexity

Let N represent the number of files and directories in the given structure.
Worst case time complexity of 'delete_directories()': O(N)

Through the use of recursion, every file and directory within the given directory structure is deleted exactly once. The runtime complexity of this file is therefore O(N). This is the minimum possible time complexity that can occur for this function as each element (file/directory) under a parent directory must be deleted in order for the parent directory to be deleted.

Function: add_to_queue(url, queue, url_map)

My 'add_to_queue()' function checks if a given URL has been added to the 'url_map'. If it has not, the URL is added to the 'url_map' as well as the 'queue'. This ensures the same URL is not repeatedly added to the 'queue' so that each page is parsed exactly once. After being added to the 'url_map' and the 'queue', a directory is created for the given URL. By creating a directory for each new URL, the file containing the incoming links of such URL can be updated directly after the URL has been added to the 'queue'.

Time Complexity

Worst case time complexity of 'add_to_queue()': O(1)

Since 'url_map' is a dictionary, checking if a given URL is in the 'url_map' occurs in constant time. If the URL is not in the 'url_map', the URL is added to the 'queue' in constant time using the append() method. Similarly, the URL can be stored in the 'url_map' by using a new key and assigning it a value, which also occurs in constant time. Lastly, making a single directory for such URL also occurs in constant time. Therefore, 'add_to_queue()' occurs in O(1) time as all the operations that take place within this function are constant time operations.

Function: extract_title(html_string)

My 'extract_title()' function locates the text between the <title> and </title> tags of an HTML string. The function then returns the title of the web page to which this HTML string belongs.

Time Complexity:
Let N represent the length of the HTML string.
Worst case time complexity of 'extract_title()': O(N)

This function uses Python's built-in find() method to locate the index of <title> as well as the index of </title>. Given that find() locates the first occurrence of a given substring, finding the index of each tag occurs in at most O(N) time. Although the theoretical worst case time complexity of this function is O(N), it is important to note that time complexity of find() depends on the position of the first occurrence of a given substring. Therefore, on average, finding the title tags of an HTML string will occur much faster than in O(N) time as <title> and </title> will typically be located closer to the start of the HTML string if the HTML is well-formatted.

Given well-formatted HTML, the length of the string between the <title> and </title> tags will likely be a constant that is relatively small in comparison to the full length of the HTML string. However, even if the title is N characters long, the time complexity of this function is still O(N). After the indices of the title tags are located, Python's [:] operation is used to slice the HTML string. Python's built-in strip() method is then used to remove any leading and trailing spaces. Both of these operations occur in linear time with respect to the length of the sliced HTML string. Therefore, the worst case time complexity of this function is O(N).

Functions: extract_words(html_string) and extract_links(html_string)

The two functions 'extract_words()' and 'extract_links()' follow a very similar structure. My 'extract_words()' function locates the text between the <p> and </p> tags of an HTML string, and returns the words contained in the web page to which this HTML string belongs. Meanwhile, my 'extract_links()' function locates the links indicated by "href" within an HTML string, and returns the links contained in the web page to which this HTML string belongs.

Time Complexity:
Let N represent the length of the HTML string.
Worst case time complexity of 'extract_words()' and 'extract_links()': O(N)

These two functions repeatedly use Python's built-in find() method to locate the indices of <p> and </p> tags (in 'extract_words()') or "href" properties (in 'extract_links()'). Although the time complexity of these functions may appear quadratic as find() is located within a while loop, the total time complexity of all of these find() operations remains linear. The time complexity is kept at O(N) by changing the starting index each time find() is called based on the index in which that last paragraph tag or "href" property was found. The find() method is thus used to check a different portion of the HTML string each time it is called.

After an opening <p> tag and a closing </p> tag or an "href" property is located, several operations occur. In 'extract_words()', Python's [:] operation is used to slice the HTML string based on the indices of the <p> and </p> tags. Each "\n" is then replaced with a space using Python's replace() method. The string is then converted to lowercase using Python's lower() method, and is finally converted into a list using Python's split() method. Similarly, in 'extract_links()', Python's [:] operation is used to slice the HTML string based on the location of the "href" property. After, Python's strip() method is used to remove any leading and trailing spaces in the string. All of these built-in methods from both functions iterate through the length of the sliced string in order to make the appropriate modifications. These methods thus occur in linear time. Given that these methods will be applied to a different portion of the HTML string each time, even if the HTML string is composed of either only words or only links, the combined time complexity of all these methods will be at most O(N) in either function.

Space Complexity:
Let N represent the number of words (contained within paragraph tags) in the HTML string.
Let M represent the number of links (contained within "href" properties) in the HTML string.

My 'extract_words()' function uses the variable 'words' to store the words found in the HTML string. Given that there are N words in the HTML string, this variable has a linear space complexity of O(N). Meanwhile, my 'extract_links()' function uses the variable 'links' to store the links found in the HTML string. Given that there are M links in the HTML string, this variable has a linear space complexity of O(M).

Function: add_to_file(file_path, title, contents)

My 'add_to_file()' function opens a file based on the given 'file_path' and 'title'. Once opened, the given 'contents' are appended to the file.

Time Complexity:
Let N represent the number of elements in 'contents' if 'contents' is a list or the number of keys in 'contents' if 'contents' is a dictionary.
Worst case time complexity of 'add_to_file()' if 'contents' is a list or dictionary: O(N)
Worst case time complexity of 'add_to_file()' if 'contents' is not a list or dictionary: O(1)

The time complexity of this function is dependent on the data type of 'contents'. If 'contents' is a list or dictionary, a for loop will run through every element of 'contents' and append each element into the opened file as a string. This for loop will therefore run in linear time. However, if 'contents' is not a list or dictionary, it will simply be converted into a string and then appended into the file. Therefore, given that write() is a constant time operation, the worst case time complexity of the function where 'contents' is a list or dictionary is O(N). Meanwhile, when 'contents' is of a different data type, the time complexity of the function is O(1) as a single element is written into the file.

Function: calculate_tfs(file_path, words, word_in_url_freqs)

My 'calculate_tfs()' function calculates the term frequency of each unique word from a list of words and stores this data in the given 'file_path'. The function also updates 'word_in_url_freqs' to keep track of the number of URLs each word appears in.

Time Complexity:
Let N represent the number of words in the given list
Worst case time complexity of 'calculate_tfs()': O(N)

This function can be broken down into two main for loops. The first for loop calculates the frequency at which each word appears in the list of words. Given that the loop iterates through every element of 'words' and that checking if a value is in a dictionary occurs in constant time, this for loop has a time complexity of O(N).

The second for loop in this function stores the term frequency value of each unique word and updates the 'word_in_url_freqs'. The calculations required to calculate term frequency value and updating the 'word_in_url_freqs' dictionary occurs in constant time. The for loop runs through every key in the dictionary corresponding to the frequency of each word in 'words'. Therefore, if there are more duplicates in 'words', the runtime will be faster as there will be fewer unique words to iterate through. However, given each word in 'words' could be unique, the for loop will have a maximum time complexity of O(N).

Given that the worst case time complexity of both for loops in 'calculate_tfs()' is O(N), the worst case time complexity of this function is O(N).

Space Complexity:
Let N represent the number of words in the given list.

My 'calculate_tfs()' function uses the variable 'freqs' which corresponds to the frequency at which each word in the list of words appears. Given that every element in the list of words can be unique, the number of keys in 'freqs' can be at most O(N). 'freqs' thus has a worst case space complexity of O(N).

Function: calculate_idfs(num_urls, word_in_url_freqs)

My 'calculate_idfs()' function calculates the inverse document frequency of each unique word contained in a group of URLs, and then stores this data.

Time Complexity:
Let N represent the number of unique words contained on any web page of the interlinked web pages

Worst case time complexity of 'calculate_idfs()': O(N)

The dictionary 'word_in_url_freqs' contains a key for each unique word contained in any web page of the group of interlinked web pages. The function uses a for loop to iterate through each key in this dictionary. As the calculations for inverse document frequency within this for loop occur in constant time, the function has a time complexity of O(N).

Function: calculate_tf_idfs(num_urls)

My 'calculate_tf_idfs()' function calculates the tf-idf weight of each unique word in every URL that has been parsed, and stores this data accordingly.

Time Complexity:
Let N represent the number of interlinked web pages.
Let M represent the maximum number of words per web page.
Worst case time complexity of 'calculate_idfs()': $O(N \times M)$

My 'calculate_tf_idfs()' function obtains tf-idf data by reading in the tf and idf values of each unique word in every file. This data is read using Python's read() method which is a constant time operation. This function uses a nested for loop in order to retrieve the necessary data. Given that one for loop iterates through the number of URLs and the other iterates through the number of words in such URL, this function runs with a worst case time complexity of $O(N \times M)$.

Function: calculate_page_ranks(num_urls, url_map)

My 'calculate_page_ranks()' function calculates the page rank of every URL that has been parsed using the incoming links data stored for every URL. These calculations follow the algorithm for computing PageRank using an alpha value of 0.1.

Time Complexity:
Let N represent the number of interlinked web pages.
Let M represent the number of iterations in which the vector is multiplied by the probability matrix in the second portion of the PageRank calculation
Worst case time complexity of 'calculate_page_ranks()': $O(N^3 \times M)$

The first portion of this function involves a nested for loop which is used to create a probability matrix. The outer for loop iterates through the number of URLs. Preceding the inner for loop, the incoming links of each URL is read, and a list containing N zeros is created. Reading in the string of incoming links and using split() to separate the links into a list has a linear time complexity. Assuming that the length of each link is a constant value, the time complexity will be linear relative to the number of interlinked web pages (as any URL can have at most every URL in its list of incoming links). Similarly, initializing a list of zeros of length N has a linear time complexity relative to the value of N. However, given that reading in the string of incoming links and initializing a list of zeros occurs once per URL, these operations have time complexities of $O(N^2)$. Following these operations, an inner for loop is used to iterate through each URL. If the URL is an incoming link of the current URL, the list that was initialized with zeros is modified accordingly (based on the algorithm for computing PageRank). Modifying these values occurs in constant time but checking if this URL is a member of the list of incoming URLs occurs in at most O(N) time. Since this is in a nested loop, the worst time complexity of these operations is $O(N^3)$.

The second portion of the function involves power iteration and continues until the Euclidean distance between a vector and a previous vector is less than or equal to 0.0001. The number of iterations of this while loop will depend on the data. Given that each iteration involves matrix multiplication which occurs in $O(N^3)$ time, the overall time complexity of these calculations in $O(N^3 \times M)$. Once the threshold of 0.0001 has been met, the calculated PageRank of each URL is stored using a for loop which runs in linear time.

Overall, the Big O time complexity of my 'calculate_page_ranks()' function is $O(N^3 \times M)$. This appears to be greater than my other functions in crawler.py. However, it is important to consider the complexity of the mathematical equations involved in computing PageRank as well as the data generated in this function. This function is called exactly once per crawl and performs the necessary calculations to compute the PageRank of every URL that has been parsed.

In storing the PageRank of each URL before receiving any search queries, the PageRank of each URL does not need to be computed for every search which significantly reduces the runtime for answering search queries.

Space Complexity:
Let N represent the number of interlinked web pages.

My 'calculate_tfs()' function uses a 2D matrix called 'probability_matrix' for the PageRank calculation. The dimensions of this matrix are N by N, so it thus has a space complexity of $O(N^2)$. Meanwhile the variables 't_previous' and 't_current' are both vectors of length N. These variables thus contain exactly N elements, and thus have space complexities of O(N).

Function: crawl(seed)

My 'crawl()' function brings together all of the previous functions in my crawler module, allowing a group of interlinked web pages to be crawled starting from a single seed URL. During the 'crawl()' function, the appropriate helper functions are called so that all the data necessary for answering search queries is stored.

Time Complexity:
At the beginning of my 'crawl' function, my 'delete_directory()' function is called to reset existing data by deleting all files and information from any previous crawl. In using the 'delete_directory()' function, this deletion occurs in linear time based on the number of files and directories currently stored in the 'crawl_data' directory. Following this deletion, a new directory is created to store the new crawl data.

Next, a queue of URLs to visit is created, and the seed URL is added to this queue. The function contains a loop which repeats until this queue is empty (and therefore iterates exactly once per unique URL in the group of interlinked web pages). Each iteration of the loop begins by removing the top element of the queue. This element is removed using Python's pop(0) method which occurs in linear time based on the number of elements in the list. The page the given URL links to is then parsed, and data including the title, words, and links are extracted from the page using the extraction helper functions. This extraction also occurs in linear time. A nested for loop runs through each link the current URL links to in order to convert each URL into an absolute URL, add the URL to the queue, and store data regarding incoming links. Given that 'startswith("./")' compares at most the first two characters of each URL and that the other functions called in this for loop occur in constant time, the time complexity of this inner for loop is linear. The 'add_to_file()' function is then used to store data regarding the URL's title as well as outgoing links (which occurs in at most linear time). Lastly, 'calculate_tfs()' is called to generate the term frequency value of each word found on the page the current URL links to, which also occurs in linear time. Since the operations within this while loop occur in at most linear time and the while loop repeats once per URL, the time complexity of this while loop is quadratic.

Once the queue is empty, a text file is created to store the mapping values of the URLs (the integer each URL is mapped to in the file structure). Creating this text file occurs in linear time. After, several more functions are called to generate the remainder of the data needed for searching. This data includes inverse document frequencies, tf-idf weights, and PageRank values. Of these functions and throughout the crawl, 'calculate_page_ranks()' has the largest time complexity of $O(N^3 \times M)$, where N represent the number of interlinked web pages and M represent the number of iterations in which the vector is multiplied by the probability matrix in the PageRank calculation. The 'crawl' ends by returning the number of unique URLs that were found during the crawl.

Space Complexity:
Let N represent the number of interlinked web pages.
Let M represent the maximum number of words per web page.

My 'crawl()' function uses the variable 'queue' to keep track of the URLs that need to be parsed. Although one element of the 'queue' is removed through every iteration of the function's while loop, the 'queue' can have up to approximately N elements. Therefore, the space complexity of 'queue' is O(N). The 'url_map' dictionary is used to map each URL to an integer value. Once all the URLs in a group of interlinked web pages have been parsed, 'url_map' will contain a key for every URL. The space complexity of 'url_map' is thus O(N). Meanwhile, the 'word_in_doc_freqs' dictionary counts the number of URLs each word appears in. If each page contains M unique words, and no words are repeated throughout any of the N URLs, 'word_in_doc_freqs' will contain N × M keys. The worst case space complexity of 'word_in_doc_freqs' is therefore O(N × M).

Searchdata Module (searchdata.py)

Overall Design

By implementing much of the data processing and calculations in my crawl module, I was able to reduce the time complexity of my searchdata.py functions. All of the data required by these functions is stored during the crawl, so the main role of each of these functions is to retrieve such data. This data retrieval mainly consists of opening the correct file path and reading in data from a text file.

Function: get_url_list()

My 'get_url_list()' function retrieves the list of URLs that were crawled. This list is obtained by reading in the data from 'url_mapping.txt'. The data in this file is formatted such that each line contains a URL followed by the integer it was mapped to. Therefore, reading in the data as a string, replacing the '\n's in the file with spaces, and converting the string to a list, returns a list alternating between URLs and integers. Taking every second element of the list starting at index zero, thus returns a list of the URLs.

Time Complexity:
Let N represent the length of the string that was read from 'url_mapping.txt'.
Worst case time complexity of 'get_url_list(): O(N)

The methods replace() and split() have a time complexity of O(N) as they iterate through each element of the string in order to modify the string accordingly. Given that these methods are used in my 'get_url_list()' function to convert the string from 'url_mapping.txt' into a list of URLs, the overall time complexity of this function is O(N).

Function: get_url_mapping(URL)

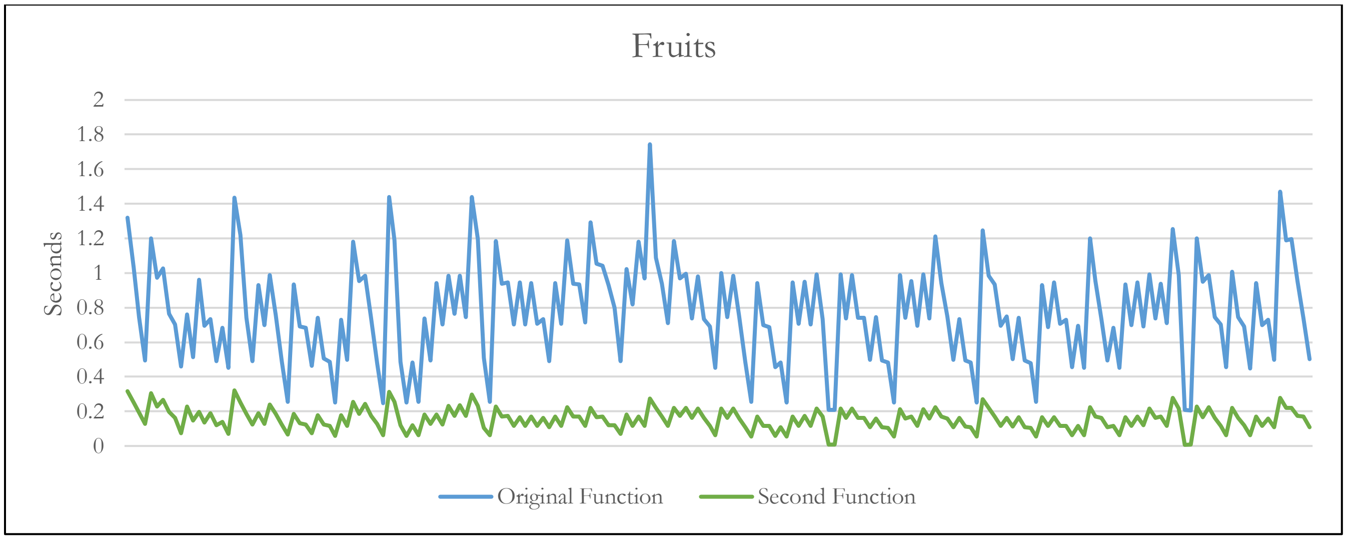
My 'get_url_mapping()' function is essential to many of the other functions in my searchdata module as it returns the integer that a given URL was mapped to in the file structure. This is necessary for accessing any data specific to a given URL. My original design for this function was as follows:

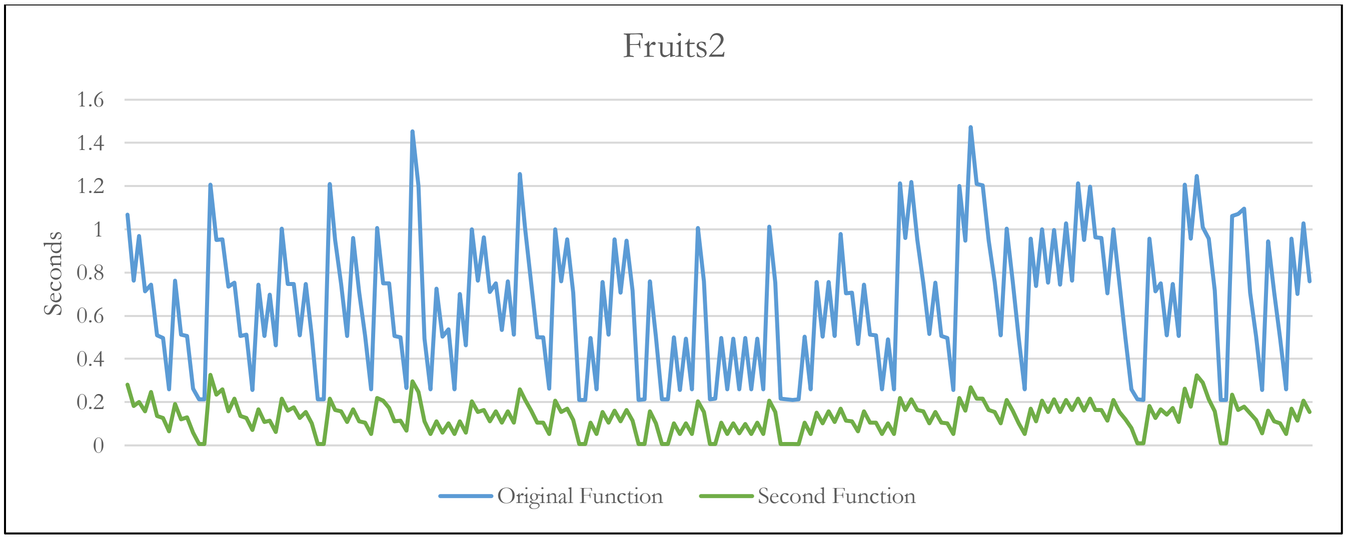
```python
def get_url_mapping(URL):
    with open(os.path.join("search_engine", "url_mapping.txt"), "r") as filein:
        for line in filein:
            current_url, current_url_mapping = line.split()
            if URL == current_url:
                return current_url_mapping
        return -1
```

This function had a worst case time complexity of O(N), where N represents the number of interlinked web pages. Although it was able to generate the correct mapping value for any given URL, after further testing, I realized the majority of my search time was spent getting the integer corresponding to each URL. I therefore decided to implement a different solution which made use of a global variable that could be accessed internally within the searchdata module. This design would still have a worst case time complexity of O(N), but would reduce the average time complexity. By only reading in the mapping values once per search, the first time this function were called, it would have a time complexity of O(N); however, for any subsequent call to 'get_url_mapping()', the function would run in constant time. Although this would decrease the runtime of this function, it would increase the amount of memory required to perform each search given that the global variable would have a space complexity of O(N) in order to store the mapping values of each URL. My second design for this function was as follows:

```
_url_map = {}
def get_url_mapping(URL):
    if not _url_map:
        with open(os.path.join("crawl_data", "url_mapping.txt"), "r") as filein:
            for line in filein:
                current_url, current_url_mapping = line.split()
                _url_map[current_url] = current_url_mapping
    return _url_map.get(URL)
```

Given that the variable '_url_map' would have a space complexity of O(N), I decided to test the difference in runtime between these two versions of the function over several different datasets in order to make a decision on whether I thought the trade-off was worth the extra memory. I timed each of the 200 provided search tests from all five 1000 web page data sets and then compared the average times from my original function to those of my second function (containing the global variable). Below, are two graphs comparing the search test times using the "fruits" and the "fruits2" datasets.





I decided to implement my second solution involving a global variable as my tests revealed an average decrease in runtime of approximately 80.19% with the implementation of the global variable. This is quite a significant decrease, especially if multiple search queries are taking place. Although the use of a global variable may not be suitable for datasets containing extremely large networks of interlinked web pages or in a situation in which memory is limited, I believe this solution is effective given the scale of this assignment.

Time Complexity:
Let N represent the number of interlinked web pages.
Worst case time complexity of 'get_url_list()': O(N)

If the '_url_map' is empty, my 'get_url_mapping()' function will read data from 'url_mapping.txt' in order to find the integer value that each URL maps to so that the file corresponding to any URL can be opened. This function contains a for loop which iterates through every line of 'url_mapping.txt'. Given that there is a different URL on each line, this for loop will iterate N times. Although Python's split() method has a linear time complexity based on the length of a string, assuming that the length of each link is a constant value, each line can be split using Python's split() method in constant time. Assigning a key value pair to the dictionary '_url_map' also occurs in constant time. Therefore, the maximum time complexity of this function is O(N).

Function: get_title(URL)

My 'get_title()' function returns the title of the page with the given URL.

Time Complexity:
Worst case time complexity of 'get_title()': O(1)

My 'get_title()' function calls the 'get_url_mapping()' function to locate the file in which the given URL's data is stored. The 'title.txt' file of this function is opened, and the data is then read and returned. Given that Python's read() method is a constant time operation, this function has a time complexity of O(1).

Functions: get_outgoing_links() and get_incoming_links()

My 'get_outgoing_links()' and 'get_incoming_links()' functions follow a similar structure, but one returns the list of URLs that the page with the given URL links to while the other returns the list of URLs that link to the page with the given URL.

Time Complexity:
Let N represent the length of the string read from a file.
Worst case time complexity of 'get_outgoing_links()' and 'get_incoming_links()': O(N)

These functions call the 'get_url_mapping()' function to locate the file in which the given URL's data is stored. The corresponding file of this function is opened, and the data is then read, split into a list, and returned. Although Python's read() method is a constant time operation, this occurs in linear time as the split() method has a linear time complexity depending on the length of the string. The time complexity of each of these functions is therefore O(N).

Functions: get_page_rank(URL), get_tf(URL, word), and get_tf_idf(URL, word)

The functions 'get_page_rank()', 'get_tf()', and 'get_tf_idf()' follow a similar structure to other functions in the searchdata module. 'get_page_rank()' returns the PageRank value of the page with the given URL, 'get_tf()', returns the term frequency of the given word within the page with the given URL, and 'get_tf_idf()' returns the tf-idf weight for the given word within the page with the given URL.

Time Complexity:
Worst case time complexity of 'get_page_rank()', 'get_tf()', and 'get_tf_idf()': O(1)

These functions call the 'get_url_mapping()' function to locate the file in which the given URL's data is stored. The file corresponding to the function is then opened, so that the data can be read and returned. Since Python's read() method is a constant time operation, these functions have a time complexity of O(1).

Function: get_idf(word)

The function 'get_idf()' is similar to many other functions in the searchdata module, but does not receive a URL as input. This function returns the inverse document frequency of the given word within the crawled pages.

Time Complexity:
Worst case time complexity of 'get_idf()': O(1)

This function opens the file corresponding to the idf of a given word and returns this data. Similarly to previous functions, this occurs in O(1) time given that Python's read() method occurs in constant time.

Search Module (search.py)

Overall Design

My search module utilizes the functions in my searchdata module in order to rank the top ten searches based on a query from highest to lowest. While designing this module, I mainly focused on minimizing the time complexity without sacrificing the readability or organization of the code. Pre-calculated data from the crawl is obtained using the appropriate functions from the searchdata module. Calculations are then performed on this data in order to find which web pages are most relevant based on a given search query.

Function: search(phrase, boost)

Time Complexity:
Let N represent the number of interlinked web pages.
Let M represent the number of words in the query.
Worst case time complexity of 'search()': O(N × M) or O(N log N)

My first section of my 'search()' function converts the given phrase into a list, then calculates the frequency at which each unique word appears. The methods lower() and split() both have a linear time complexity based on the length of the phrase. A for loop iterates through every element in the list of words from the given phrase to calculate the frequency of each unique word in the list. Given that checking if a key is in a dictionary and modifying the value corresponding to a key both occur in constant time, this for loop has a time complexity of O(M).

Next, a for loop iterates through the dictionary containing the frequency in which each unique word of the query appears. Through this for loop, the tf-idf weight of each searched word in the query is calculated as well as the Euclidean norm. The operations within this for loop occur in constant time, so the overall time complexity of this for loop is dependent on the number of words in the frequency dictionary. Given that all words in the query could be unique, the worst case time complexity of this for loop is O(M).

The 'get_url_list()' function from my searchdata module is then called to get the list of URLs in the group of interlinked web pages. This occurs in O(N) time. A for loop is then used to iterate through every URL in this list. Within this for loop, the tf-idf weight of each word that was searched in the query is calculated as well as the Euclidean norm. These values are then used to compute the cosine similarity of that URL with the given query, which is stored in a dictionary. The operations used through these calculations occur in constant time, but the nested for loop which iterates through the unique words of the query runs in linear time. Therefore, the runtime complexity of this portion of the code is O(N × M).

Once a dictionary has been created, Python's built-in sorted() function is used to return the list of keys sorted by their values. The output of this operation is a list containing URLs ordered from that with lowest cosine similarity to that with the highest. I decided to sort the entire list of URLs so that 'search()' would contain the functionality of a typical search engine in that it could easily be adapted to output the next top searches given that this data has already been computed. Although I performed some testing with Merge Sort and Quicksort, I decided to use Python's built-in sorted() function as this produced the most consistent results. This is unsurprising given that sorted() uses the Timsort algorithm, a very efficient hybrid sorting algorithm derived from merge sort and insertion sort. Although Timsort, Merge

Sort, and Quicksort all contain an average time complexity of O(N log N), Timsort is highly optimized to consistently sort real-world data efficiently. In terms of best case time complexity, Timsort outperforms both Merge Sort and Quicksort with a time complexity of O(N). Additionally, Timsort also outperforms Quicksort in terms of worst case time complexity, with a time complexity of O(N log N). Therefore, the worst case time complexity of sort is O(N log N).

Lastly, the data corresponding to the top 10 search results is added to a list then returned. Since the list was ordered from URLs with the lowest cosine similarity to those with the highest, the top URL can be removed from the list using pop() in constant time. Since all the operations within this for loop occur in constant time and 10 is a constant value, the time complexity of these operations is O(1).

Overall, the worst case time complexity of this function is either O(N log N) or O(N × M) depending on the values of N and M. In a large group of interlinked web pages, it is likely that log N will be greater than M if the search query does not contain that many words. In this case, O(N log N) would be the worst case time complexity. However, if there are not many interlinked web pages or the search query contains many unique words, the worst case time complexity could be O(N × M).

Space Complexity:
Let N represent the number of interlinked web pages.
Let M represent the number of words in the query.

Multiple variables were used within my 'search()' function to compute the necessary calculations for the search. The space complexities of 'query_words' and 'query_freqs' are both dependent on the words in the query. 'query_words' will always have a space complexity of O(M), and given that each word in the query can be unique, 'query_freqs' also has a worst case space complexity of O(M). The space complexity of 'query_tf_idf' is also dependent on the number of unique words in the query, and therefore has a worst case space complexity of O(M). Lastly, the number of elements in 'cosine_similarity_dict' and 'ranked_url_list' is dependent on the number of interlinked web pages. Therefore, 'cosine_similarity_dict' and 'ranked_url_list' have a space complexity of O(N).