COMP 1405Z – Course Project – Jalal Bouri

Part 1: Web Crawler (crawler.py)

Overall Design

The web crawler uses the 'crawl()' function as a master function that facilitates all other functions, initializes specific variables, and manages data uploads to the file system. It's important to note that the 'crawl()' function itself doesn't handle the actual crawling or computations. Before delving into the crawling process, it's essential to outline some key aspects of the function.

- *The 'parse()' function is responsible for the actual crawling, operating recursively. It is called once from the 'crawl()' function with the seed link passed as a parameter.*
- *My program uses only one dictionary ('crawl_dict') throughout the crawl and uploads all data to a single JSON file.*

File Structure

As previously mentioned, I've opted to use only one dictionary and one JSON file to store all the data collected from the crawl. The basic structure of this dictionary is outlined on the right. The first sub-dictionary encompasses all word IDFs from all URLs. The second sub-dictionary comprises all the URLs, each with its own sub-dictionary. These URL sub-dictionaries encompass the title, page rank value, and lists containing both incoming and outgoing URLs. Additionally, there's a sub-dictionary that contains every word found within the URL. Each word has its own sub-dictionary containing its TF and TF-IDF values for the URL.

```
"word_idfs": {
    "word": 1
},
"urls": {
    "website.com": {
        "title": "N-0",
        "words": {
            "word": {
                "tf": 1.0,
                "tfidf": 1.0
            }
        },
        "outgoing_links": [
            "website_other.com"
        ],
        "incoming_links": [
            "website_other.com"
        ],
        "page_rank": 1.0
    }
}
```

Why? We learned that dictionaries allow for O(1) time complexity for searches. Drawing from previous experience with JSON files, I understand that conversion between JSON and a dictionary is relatively straightforward. This approach also simplifies the storage of data and enhances comprehensibility. The fact that we were permitted to use the JSON module further convinced me that this was the optimal choice.

Functions

Function	Description	Time/Space Complexity
create_url(line, url)	**What?** This function takes a string containing an 'href' and a parent URL, and it returns the child URL. **How?** There are two possible cases. In the first case, if the URL is absolute (determined by checking whether 'http' is in the string), and in the second case, if the URL is relative. In both cases, string operations are performed on the input parameters to create the new URL, which is then returned. **Why?** I chose to create this as a separate function because manually constructing a new URL can be inherently messy. Keeping it separate from the 'parse()' function enhances code organization and facilitates the clean and efficient creation of new URLs when needed.	The time complexity of this function depends on the lengths of the strings passed into it. Since the function involves several string operations that rely on the lengths of the strings being manipulated, it runs in O(n) time, where 'n' represents the lengths of the input strings. The space complexity is the same, O(n).
parse(url)	**What?** This function extracts a URL's title, words, term frequencies, and outgoing links, storing this data in the appropriate places within the URL's sub-dictionary. **How?** The function is recursive, initially called from the 'crawl()' function with the original URL. Here's the step-by-step breakdown of the process: 1. The function begins with 'is_words' set to false and initializes an empty 'words' list to store all words from the URL. The 'read_url' function from the 'webdev' module is called on the URL, and the result is split into a list. The URL's unique index is based on the length of the 'visited_urls' list, and the URL is appended to that list. The function then iterates through every line in the URL's HTML string,	While this function is complex, its time complexity is primarily influenced by recursive calls, looping through each line in the HTML code, and the number of words in each URL. Therefore, if 'n' is the number of interlinked webpages, 'l' is the average number of lines in all HTML strings, and 'w' is the number of words in each URL, the function operates in O(n * l * w) time. This is because the function is recursively called based on the number of interlinked webpages ('n'), each interlinked webpage has its HTML code looped through line by line ('l'), and within that loop, every word in the URL is also looped through ('w'). The space complexity of this function is primarily influenced by the call stack caused by recursion ('r'), the length of the HTML code

	checking for specific elements in each line. 2. It first checks if "<title>" is in the current line. If it is, string operations are performed to extract the title, which is then added to the URL sub-dictionary. 3. The next check determines if the current line contains text between ' ' and ' ' tags. If ' ' is in a line, 'is_words' is set to true. If 'is_words' is true, string operations are done to extract words, which are added to the 'words' list using list comprehension. If ' ' is in a line, 'is_words' is set to false. A 'words' sub-dictionary is created in the URL sub-dictionary, and each word in the 'words' list is looped through. In each iteration, a sub-dictionary is created within the 'words' sub-dictionary for each word, if it doesn't exist already. A 'tf' key is then added to that sub-dictionary with a default value of 0, which increases by 1 for each occurrence of the word. After the 'word' list loop is completed, another loop iterates through the set of the 'word' list (the list without duplicates). In each iteration of this loop, the value associated with each word in the 'word_idfs' sub-dictionary is increased by 1, and the 'tf' value for each word (which previously held the number of occurrences) is divided by the length of the 'words' list to obtain the true TF value. 4. The last check examines if 'href' is in the line. If it is, a new URL is created using the	('h'), and the size of the 'words' list ('w'). As a result, the space complexity can be expressed as O(r + h + w).

	'create_url()' function. An 'outgoing_links' key in the current URL's sub-dictionary is initialized as an empty list if it doesn't exist already. If the new URL isn't already in the current URL's 'outgoing_links' list, it's appended. A sub-dictionary is created for the new URL if it doesn't exist already, and the current URL is added to the new URL's 'incoming_links' list if it's not already there. If the new URL isn't in the 'visited_links' list, the 'parse()' function is called recursively on the new URL. **Why?** The use of recursion in the 'parse()' function was driven by the repetitive nature of web crawling tasks. Since the same data extraction and calculations were to be performed on every webpage, recursion offered an elegant solution. Loops weren't ideal because the number of webpages to crawl was unknown, and there wasn't a clear condition for a 'while' loop. The structure of webpages linked to other webpages also resembled the nested dictionary practice problem, which worked well with a recursive approach. Despite the success of this approach (all tests were passed, and relatively quickly), there are potential issues, which will be discussed further.	
get_euclidean_dist(a, b)	**What?** This function takes two lists/matrices and calculates the Euclidean distance between them. **How?** It iterates through the indices of one of the matrices and uses 'distance' as an accumulator. 'Distance' is the sum of the squares of the differences between corresponding elements of the two matrices. After the loop is completed, the square root of the 'distance' is computed and returned.	The time complexity of this function is dependent on the length of the input lists, as it iterates through each element in them. It operates in O(n) time, where n is the length of the input lists. The space complexity is the same, O(n).

	Why? This function is exclusively called from the 'get_page_ranks()' function and is utilized to determine the Euclidean distance between the two most recent probability vectors. Creating a separate function was the preferred approach, as it enhances code organization and simplifies the process of computing with different parameters.	
create_t2(t1, adjacency_matrix)	**What?** This function takes an existing probability vector and an adjacency matrix and returns a new probability vector. **How?** Given that 't1' is a 1D list and 'adjacency_matrix' is a 2D list, the function iterates through each column of the adjacency matrix. Within this loop, it further iterates through each element of 't1,' utilizing an accumulator to calculate the sum of each corresponding element in 't1' multiplied by each corresponding element in the current column of the 'adjacency_matrix.' After each iteration of the columns, the result is appended to a result list. Upon completing these loops, the result list is returned. **Why?** This function serves a similar purpose to 'get_euclidean_dist(a, b),' and I opted to create a dedicated function to manage the creation of probability vectors. This approach leads to cleaner code and minimizes confusion with parameters.	The time complexity of this function depends on the number of rows and columns in the adjacency matrix. It loops through each column of the adjacency matrix, and within each of those loops, it iterates through the individual elements in each column. Therefore, it runs in O(r * c) time, where 'r' represents the length of each row, and 'c' represents the length of each column. The space complexity is the same, O(r * c).
get_page_ranks()	**What?** This function computes the PageRank value for every crawled URL and adds the value to each URL's sub-dictionary. **How?** 1. Creating the Adjacency Matrix: The first step involves constructing an adjacency matrix. It begins by iterating	The time complexity of this function is primarily determined by the convergence loop. During each iteration, it calls 'create_t2()', which takes O(r * c) time, where 'r' is the length of each row and 'c' is the length of each column of the adjacency matrix. Therefore, this function runs in O(n * c * i) time, where 'i' is the number of

	through each URL sub-dictionary. In each iteration, a list is created with 'n' elements, all set to 0. Then, it loops through each of the outgoing links from the current URL, determines the index of that outgoing URL, and changes the corresponding element in the list to 1. After processing all outgoing links, some additional calculations are performed on each number in the list. Finally, the list is appended as a row to the adjacency matrix. Once all URLs have been processed, the adjacency matrix is complete, and the 'index' keys in each URL sub-dictionaries are deleted. 2. Creating Probability Vectors: The second step is the creation of the initial two probability vectors. The first list, 't1,' is created with 'n' elements, each set to 1/n. The other list, 't2,' is generated by invoking the 'create_t2()' function using 't1' and the adjacency matrix. Then, the Euclidean distance between these two vectors is calculated by calling 'get_euclidean_dist()' with 't1' and 't2.' These vectors, along with their initial distance, serve as the starting point. 3. Convergence Loop: The third step involves running a convergence loop until a steady-state probability vector is reached. This is achieved using a 'while' loop that continues as long as the Euclidean distance between the two most recent vectors is greater than 0.0001. Within the loop, 't1' is updated to 't2,' 't2' is created using	iterations required to reach a steady-state probability vector. The space complexity of this function is primarily determined by the creation of the adjacency matrix, which has a space complexity of O(r * c), where 'r' represents the length of the rows and 'c' represents the length of the columns.

	'create_t2(),' and the distance is checked using 'get_euclidean_dist().' Once the loop concludes, a 'for' loop is utilized to iterate through each element in the steady-state probability vector and add them to the corresponding URL sub-dictionary. **Why?** Initially, PageRank calculations may seem complex, but by breaking them down into three distinct steps, the process becomes more manageable and straightforward. The use of functions like 'create_t2()' and 'get_euclidean_dist()' significantly simplifies the initialization of the initial probability vectors and the convergence loop.	
get_idfs()	**What?** This function calculates the IDF for every word found in all documents and adds the values to the corresponding sub-dictionary. **How?** When this function is called, the 'word_idfs' sub-dictionary already contains every unique word found in all URLs as keys, along with the number of URLs in which each word appears as values. The function iterates through each word/count pair in this sub-dictionary, computes the IDF for each word, and updates the current value of each key with the IDF value. **Why?** Looping through the URL counts for each word was the most efficient and straightforward approach to compute the IDFs. This is because the two required parameters for calculating an IDF are the number of documents a word appears in and the total number of documents. I chose to temporarily store the URL counts for each word in the 'word_idfs' sub-dictionary, so I can repurpose the same sub-dictionary instead of creating two	The time complexity of this function depends on the size of the 'word_idfs' sub-dictionary, as it iterates through each element within this sub-dictionary. It operates in O(n) time, where 'n' represents the number of elements in the 'word_idfs' sub-dictionary. The space complexity is the same, O(n).

	separate ones. Additionally, the URL count serves no purpose after the IDFs have been calculated.	
get_tfidfs()	**What?** This function calculates the TF-IDF values for sets of words within each URL and appends these values to the respective sub-dictionary. **How?** The function iterates through each URL's sub-dictionary. Within each iteration, it further iterates through the words contained in the URL. During these nested iterations, the TF-IDF value for each word is computed and added to the sub-dictionary associated with that word within the 'words' sub-dictionary. **Why?** Storing each TF-IDF value within the word sub-dictionary, which is located within the 'words' sub-dictionary of each URL's sub-dictionary, offers an organized and clean way to store the data. This approach also facilitates easy access to the TF of each word within each document since this value is also located within each word's sub-dictionary. The implementation of this approach required the use of nested loops, which were implemented accordingly.	The time complexity of this function depends on both the size of the 'urls' sub-dictionary and the average size of the 'words' sub-dictionaries within each URL sub-dictionary. This is because it iterates through each element of the URL sub-dictionary, and within each iteration, it loops through each element of the corresponding 'words' sub-dictionary. As a result, it operates in O(n * m) time, where 'n' represents the number of elements in the 'urls' sub-dictionary, and 'm' represents the average length of all 'words' sub-dictionaries. The space complexity of this function relies on the size of the 'urls' sub-dictionary, making the space complexity O(n).
crawl(seed)	**What?** This function acts as the central function of the program. It orchestrates the crawling of a network of webpages and stores all gathered information in a JSON file. **How?** The function begins by initializing the 'crawl_dict' sub-dictionary, and the 'visited_urls' list. After this setup, it calls the 'parse()' function on the seed URL. Following the parsing, it calls 'get_page_ranks()', 'get_idfs()', and 'get_tfidfs()'. Finally, the 'crawl_dict' is saved to a JSON file, and the number of crawled webpages is determined by checking the length of the 'visited_urls' list.	Four functions are called from this function, and their respective time complexities have been previously discussed. Among them, 'parse()' and 'get_page_ranks()' are the most significant contributors to the overall time complexity. 'parse()' operates in O(n * l * w) time, where 'n' represents the number of interlinked webpages, 'l' is the number of lines in each HTML code, and 'w' is the number of words in each URL. 'get_page_ranks()' runs in O(r * c * i) time, where 'r' and 'c' signify the length of the rows and columns of the adjacency matrix, and 'i'

	Why? Breaking down the web crawling process into distinct functions enabled me to address each task separately and establish the necessary order of execution (e.g., IDFs must be calculated before TF-IDFs). It also greatly aided in assessing the time and space complexity of the program since I could analyze the complexities of each function called and determine the most intensive component(s).	represents the number of iterations needed to reach a steady-state probability vector. Since 'r' and 'c' are both equivalent to the number of interlinked webpages, the time complexity of 'get_page_ranks' can be represented as $O(n^2 * i)$. Now the key question is whether '$n^2 * i$' or 'n * l * w' dominates in terms of time complexity. In tests with practice links, it was found that 'get_page_rank()' executes roughly 20 times faster than 'parse()'. Consequently, for the provided test links, the time complexity is O(n * l * w). However, it's important to recognize that different webpage networks may cause 'get_page_ranks()' to be the dominant factor. Therefore, the comprehensive answer is $O(\max(n * l * w, n^2 * i))$. The space complexity of all helper functions has been previously discussed, and two functions carry the most significance. Firstly, 'parse()' has a space complexity of O(s + h + w), where 's' represents the recursive call stack, 'h' is the length of the HTML code, and 'w' is the number of words in the URL. Secondly, 'get_page_ranks()' holds a space complexity of O(r * c), with 'r' and 'c' being the lengths of the rows and columns of the adjacency matrix. As 's,' 'r,' and 'c' can all be redefined as 'n,' where 'n' represents the number of interlinked webpages, the space complexity of 'crawl()' is $O(\max(n + h + w, n^2))$.

Part 2: Data Required for Search (searchdata.py)

Overall Design

Since there is only one file, the search functions are all relatively straightforward and share similar structures. They open the file in read mode, extract the dictionary, and return the requested value. To handle invalid inputs, try/except blocks are employed. Additionally, several functions accept a dictionary as an optional argument. This approach prevents the file from being opened and closed multiple times within the search function, significantly enhancing its efficiency.

Functions

Functions	**Description**	**Time/Space Complexity**
get_outgoing_links(URL) get_incoming_links(URL) get_tf(URL, word) get_idf(word, dic=None) get_tf_idf(URL, word, dic=None) get_page_rank(URL, dic=None)	**What?** These functions return the requested value or None/0/-1 if the value doesn't exist. **How?** The functions open the file, extract the dictionary, and then attempt to retrieve the requested value from the dictionary using try/except to handle errors appropriately. Additionally, some functions allow an optional dictionary parameter. When a dictionary is provided, the file isn't opened, and the provided dictionary is searched instead. **Why?** This approach is the most straightforward way to retrieve values from a dictionary, and using try/except ensures clean error handling. The optional dictionary parameter was essential to ensure reasonable search function performance. Before its implementation, opening, reading, and closing the file repeatedly resulted in extensive search times, especially for larger networks. With this enhancement, the file is opened once in the main search function, and subsequent searches are performed on the dictionary from that file.	These functions have a time complexity of O(1) because dictionary searches occur in constant time. Their space complexity is O(n), with 'n' representing the size of the file or dictionary.

Part 3: Search (search.py)

Description

What? This program accepts a search query and a Boolean variable and provides a list of the top 10 most relevant results.

How? This program contains a single function, 'search(phrase, boost)'. The process begins by initializing the 'all_results' list, which serves as the container for all search results. From there, the file is opened, and the dictionary is extracted. The search query is then split into a list of individual words, with duplicates removed through the use of a set. A query vector is constructed by iterating through each unique word in the set and performing calculations during each iteration. Next, the program loops through each URL sub-dictionary. For each URL, a URL vector is generated by iterating through the set of unique words and appending the corresponding TF-IDF value for each word. Following this, calculations commence with the initialization of 'numerator,' 'right_denominator,' and 'left_denominator' to 0. The query vector is then iterated through at each index, and during each iteration, calculations are performed and added to the accumulator variables. Once the loop completes, additional calculations are done to determine the cosine similarity. If 'boost' is set to true, the cosine similarity is adjusted by multiplying it by the PageRank value. Afterwards, the 'all_results' list is looped through, starting from index 0. If the URL's score exceeds the score of the URL at the current index, the URL's condensed dictionary is inserted at that index. If it doesn't surpass any of the existing scores in the list, the URL's condensed dictionary is added to the end. After processing all URLs, the first 10 elements of the 'all_results' list are returned.

Why? The creation of the query vector, URL vector, and cosine similarity required the use of loops, lists, and accumulator variables to efficiently produce results. Regarding the storage of results, the decision to store them within a single list was made since list slicing would provide a quick and efficient method to return the top X results. This design flexibility allows for easily adjusting the program to return a different number of results if necessary.