COMP 1405Z Intro to Computer Science I Fall 2025
Final Project: Implementation of Search Engine

Sophie Gu and Alice Jiang
Department of Computer Science, Carleton University
COMP 1405Z
Ava Mckenney
October 17, 2025

I. crawler.py Functions:

crawler(seed):
This function takes one seed link and crawls the web starting from said seed. It writes relevant data from each web page to its own text file and then adds that website to a list containing all visited sites to avoid repetition. Once all sites have been visited, it writes the total number of sites as well as all the sites from the crawl to a file called "readsites.txt." Then, it calls other functions to perform relevant calculations (all calculations for part 2) and the results are each stored in their own text file. This is done to keep the space and time complexity of all the functions in searchdata.py smaller. Due to it looping through all the sites and all the data in each site, this function has a runtime complexity of $O(n^2)$. Since this function uses 2 lists to store the websites, as well as one list to store data for each site, the space complexity is also $O(n^2)$.

readsites_data():
This function opens the "readsites.txt" file and returns the total number of websites as well as all the raw data in the file. Since all operations (opening the file, reading it, closing it, and returning the data) take constant time, the time complexity is O(1). There are 2 variables used to store data in this function, thus the space complexity is also constant, meaning O(1).

txt_name(link):
This function takes in a link and returns the corresponding text file name. Since all the operations are in constant time and there is only one memory location, space and time complexity are O(1).

find_outgoing_links(URL):
The use of this function is to find all outgoing links for the input URL. I chose to put this function in crawler.py to ensure the get_outgoing_links(URL) function (in searchdata.py) would have low space and time complexity. Since this function uses a list to store data and only opens one file, it has a space complexity of O(n). Similarly, the time complexity is also O(n) as it loops through the one file to find outgoing links.

find_incoming_links(URL):
This function finds all incoming links to the input URL and returns a list of them. Similarly to find_outgoing_links(URL), this is run in the crawl to avoid space and time complexity in searchdata.py. The use of a list to store data as well as opening files within a loop means the space complexity is $O(n^2)$, and because this loops through all the site files to find all the ones that outgo to the URL, the runtime complexity is O(n).

mult_scalar(matrix, scale), mult_matrix(a, b), euclidian_dist(a, b)
All these functions were implemented to support page rank calculations while keeping the code readable and modular. The functions mult_scalar(matrix, scale) and euclidian_dist(a, b) run in

$O(n^2)$ as they loop twice to access all values in the 2D lists. mult_matrix(a, b) runs in $O(n^3)$ due to it needing to access rows and columns differently for matrices a and b. As for space complexity, mult_scalar(matrix, scale) has a space complexity of O(n) as it uses 2 lists for storage. mult_matrix(a, b) uses $O(n^2)$ due to it storing rows before adding them to a larger 2D list. Finally, euclidian_dist(a, b) has a space complexity of O(n) since it uses only constants other than the initial matrix.

calc_page_ranks():
Unlike find_outgoing_links(URL) and find_incoming_links(URL), this function does not calculate page rank for one page because to find the page rank for one page, all the page ranks need to be calculated. Thus, it's much more efficient to calculate all page ranks at once. To do this, the function loops through the sites in "readsites.txt" (found using the readsites_data() function) to make a row for each. Within that, it finds the outgoing links for the site using find_outgoing_links(URL) and loops again through all the sites in "readsites.txt" and adds 1 or 0 to the row based on whether that site is in the "outgoing" list. After that, more calculations are performed to find the final page rank. However, all those calculations are of equal or lower order. Thus, since there is a nested loop, the runtime complexity of this function is $O(n^2)$. The space complexity is also $O(n^2)$ for similar reasons. A 2D list is made using the nested for loop, and the other space used is of equal or lower degree.

II. searchdata.py Functions:

find_index(URL):
This function opens "readsites.txt" and finds the index of the input URL. It then returns that number minus 1 to accommodate the extra line at the beginning of "readsites.txt" (which contains the total number of sites). Since it loops through the file by making a list using readlines(), and the other storage and operations are of lower order, it has a time and space complexity of O(n).

get_outgoing_links(URL), get_incoming_links(URL), get_page_rank(URL):
These three functions all operate using the same logic. They open the file containing the calculations from the crawl, extract the data into a list, and return the element at the index of the input URL (found using find_index(URL)). Thus, all of them have a space and time complexity of O(n) as all other storage and operations are of equal or lower order.

word_count(word, URL):
This is an additional function made to count the number of words in a document by searching through a list one at a time in O(n) runtime complexity, which would similarly run in O(n) space complexity. It efficiently goes through the entire document only once and counts the occurrence of a specific word since the document is unsorted.

doc_freq_of_word(word):
This function goes through every link that's been crawled through at O(n) runtime and space complexity and calculates the word count each time at O(n) runtime and space complexity, giving it a final runtime complexity and space complexity of $O(n^2)$. It saves time and space by only going through each document once for every word that it checks.

total_docs(), get_total_words(URL):
These additional functions both run on O(1) runtime and space complexity for using a constant number of operations. Both operate by opening a document to read the first line, then processing and returning the information in an operation. This is the most efficient execution of the functions that's possible due to the structured format of all the files.

get_idf(word):
This function uses doc_freq_of_word(word) and total_docs() once each, which gives it a runtime and space complexity of $O(n^2)$.

get_tf(URL, word):
The runtime and space complexity of this function is determined by its use of the previously made find_index(URL) to check if the URL is valid on O(n) runtime complexity. By looping through every URL, it can check off all the options without needing excessive repeats or a wide range of storage that would be required for a more efficient runtime. In the same vein, it uses an O(n) space complexity which contains the list of items it will search through.

get_tf_idf(URL, word):
Although this function only contains one operation, the operation uses the previously stated functions of get_tf(URL, word) and get_idf(word), which take $O(n^2)$ and O(n) runtime and space complexity respectively. While it has a large runtime and space complexity, it computes the data from those functions in O(1) runtime and space complexity since it only requires one equation.

III. search.py Functions:

sort_highest(list_2d):
This sorting algorithm uses selection sort to sort from highest to lowest. Due to the nested for loop, it has a runtime complexity of $O(n^2)$. Similarly, the space complexity of the function is $O(n^2)$ because it requires going through n items for a variable amount of times.

cos_similarity(query, link):
For this function, the runtime and space complexity are O(n) since it must iterate through an get_tf(URL, word) and the list of query word scores to calculate and output a number.

search(phrase, boost):
This function uses many of the above functions, giving it a much larger runtime and space complexity. It iterates through many lists and ends up running the get_idf(word) function in a for loop, which results in an $O(n^3)$ runtime and space complexity. It opens, writes in, and closes many files, which doesn't affect the $O(n^3)$ complexity if n were a larger number, but can be significant if the n is a small amount.