COMP 1405Z Course Project – Analysis

Eric Desrosiers

October 26, 2022

Everything in my function works.

File structure:

To help keep everything organized I used many nested directories. This helps limit the amount of data needed to be stored in any single file. This simplifies the accessibility of information because I only need the file path. The main directory is *crawl_data* this keeps all data in one place. In *crawl_data* you will find folders with names of urls with the "http:" and "/" replaced by "}". There is a file located in *crawl_data* that stores the original URL and its file path called *filemap*. This keeps a structured format for the program with the benefit of me knowing what folder holds what without having to manually look at the file map. In each URL directory there is 2 other folders: *tf* for term frequencies and *tf_idf* for the *tf_idf* weight of a word. This lets me have multiple files with the same name because they are in different folders. In each URL directory there is a file for outgoing links called "out.txt", incoming links named "in.txt", and page rank labeled "pageRank.txt", they all store their respective data for each URL. Finally, there is an *idf* folder in *crawl_data* that stores the idf of the word in the data set.

Space Complexity

Let N represent the total number of URLs found in crawl.

Let M be the maximum number of words in any given URL.

The worst-case scenario for finding a file in my system is O(NxM). This is because each URL has the *tf* and *tf_idf* directories that have at most M text files, one for each word in the webpage.

Each item in the N directories have a space complexity of O(1) as they contain a single value. There are a few exceptions being “out.txt”, “in.txt” and “filemap.txt”. They can have a maximum space complexity of O(n) because the size of these files is relative to how many URLs are included in each text file as they are a JSON list. This helps with efficiency because there is less processing needed, only to read the data in the file.

Crawler (crawl.py):

In this module I decided to have most of the data processing happen. This significantly decreases the time spent for s*earchdata* and search because they must only look up answers or do minimal computations. The while loop in my crawl does three major tasks: reading and formatting the HTML from the URL, locates the words in the web page and creates the new links from the current web page. For format I decided to remove unnecessary tags like “<html>” which is not required for my program. Locating words entails processing them and compiling them for specific calculations like total occurrence of a specific word in a web page. Doing these calculations while I already have the word cuts back on processing in each calculation.

Function: *url_setup():*

This function takes in a URL and deletes all the folders and files previously stored during a crawl. This resets the folder to be used for the current crawl. This function gets called right after I read a new URL. The time complexity of this function is O(1) because there are no loops in the function. Since the function is in the main crawl loop overall it is O(N), N for the number or URLs in the crawl.

Function: *calculate_page_rank():*

This is a helper function to keep crawl more organized. The function first creates a matrix of size N*N. It then creates a map with each index in matrix correspond to a URL in the dictionary *matrixmap,* fills out each index with a 1 if the URL at index links to it. Then it calls pagerank_matrix() from matmult.py to do all the scalar multiplication steps which include, dividing each row by the number of 1s in that row; multiplying the index by 1- alpha (alpha – 0.01) and adding the adjacency matrix to it (alpha / N). I did all these calculations in one go to save time. The Next step is the convergence multiplication. This is simply taking a 1-D vector that sums to 1 and multiply it into the page rank vector. Then I multiply that into the page rank vector. I keep doing this until the Euclidian distance of the previous iteration and the current iteration is less than 0.0001. After, I take write each answer to a file under the URL directory with name “pageRank.txt”.

The runtime complexity of the function is O(N^2) this is because creating the empty matrix is NxN making it O(N^2). The *pagerank_matrix()* has a nested for loop also making it O(N^2). The convergence multiplication step is O(N^2*M) because I have a nested for loop to get the column and row for the page rank matrix, the M comes from the number of iterations it takes to reach the threshold of 0.0001. Since I always know it’s a 1-D matrix multiplied by a 2-D matrix I only need to go through every column in b and row in a. If I used the original matmult module for this the runtime complexity would have been O(N^3). Finally, writing to the file in O(N) because we write for N URLS. Since none of the different sections’ complexity is over O(N^2) the overall runtime complexity is O(N^2*M).

Function: calculate_tf():

This function calculates the term frequency of each unique word in each document found in crawl. It can be broken up into two main loops:

The first loop is in the main crawl body. This is the loop that processes each webpage. In the '**foundwords**' if statement it counts the total occurrence of each word and the total words The time complexity for this is O(N*M), where n is the total unique URLs and M is the total words.

The second loop is defined in the helper function. It goes through each word found in the given URL and calculates the term frequency of that word. The time complexity for this function is at worst O(N*M) as N is the total number of URLs and M is the number of unique words. The tf of every word in each url is stored in the dictionary *tfwords*.

Function: *calculate_idf():*

This function calculates the idf value for each word in the given data set. It does this by taking the $\log_2$ of the total number of websites divided by one plus the number of websites the word appears in. The time complexity for this function would be O(N) as N is the total number of unique words found in the crawl stored in the dictionary *idfwords*. We only need to calculate the idf value of each word at the end of the crawl.

Function: *calculate_tf_idf():*

This function takes the term frequency of each word in every URL and idf of the word in the data set to calculate the tf_idf of the word. I had the url and word joined by a "a" to keep it only one loop but it is still dependent on how many URLs and word there are. The time complexity of

this function would be an O(N*M) because it must go through every URL in the crawl (N) and every word found at that URL (M)

Function: *outgoing_inoming_tofile():*

This function is used to write all the outgoing and incoming URLs for each URL found in the crawl. It calls the dictionary *in_out* and takes each the incoming and outgoing for each URL and write it to a file. The time complexity for this is O(N) as N is the total number of URLs found during the crawl. My original implementation was doing a JSON dump with all URLs incoming and outgoing; this was less efficient because I had to read all the URLs incoming and outgoing just to get a single URL's outgoing.

Function: *enqueue():*

The function takes the url and checks if the program has already crawled through that URL. If the program hasn't it adds it to queue. This function is O(1) because it does not contain any loops; it simply looks if value in list and appends. The dictionary 'out_in' used also serves the purpose of storing the outgoing and incoming links for that url.

The time complexity for the function would be O(1) as checking and adding the url to the queue is done in constant time. Also adding to the out_in dictionary is also done in O(1) time.

Module: searchdata.py

Since most of the calculations are done in crawler.py the majority of the functions run in O(1) time as it is reading the data from my O(1) filesystem. These functions include: *get_page_rank(), get_tf(), get_idf(), get_tf_idf().*

Function: *get_incoming_links(): get_outgoing_links():*

I included these two together because they function the same way but get different data points.

What make these different from the other functions is that the data is of O(N) space as discussed

in my *File Structure* explanation. This means that it cannot be read in O(1) time. I still used a

JSON list since the program does not need to process the data, only read it and return it.

Module: search.py

General: This module is used to perform a search given a phrase and a optionally a value for page rank. A lot of the data needed was precalculated in my crawl function such as tf_idf of words and the page rank value. This helps decrease runtime of this module by having most of the values needed have a runtime complexity of O(1) because of my file system. My calculations for search score can be broken down into parts.

Part 1 query_vector_setup():

This helper function's main purpose is to set up the query vector for the Cosine Similarity calculation and calculate the left denominator for it. There are two loops involved in this function. The first loop counts how many words there are, number of each word and the total words of the query vector. It also creates a list of words that determine the order of the vectors, I did this so there is consistency in vector indexes. The second loop calculates the tf_idf of each query word and the left denominator for the cosine similarity calculation. The runtime complexity of this function is O(N) where N is the number of words in the query. This is because I have two independent loops that are O(N).

Part 2 returnscores_setup():

This helper functions adds a dictionary entry in the form of the required return format for search, format being: " iteration: { 'url': url, 'title': title, 'score': 0.0}. iteration is what URL we are on since I calculate cosine similarity alongside reading in my tf_idf values. Having the iteration as my main key is very important when I sort the scores.

Part 3 numerator_right_denominator():

This is another helper function that makes the main body of code easier to follow. Its main goal is to get the tf_idf of every word in *wordOrder* to calculate the numerator and right portion of the denominator for the cosine similarity calculation. I calculated the numerator and denominator in the loop instead of building a vector list first and then doing calculations because it makes it

more efficient. The runtime complexity of this function is O(NM) with N being number of pages and M being words in query. The function itself is O(M), since it is inside of another loop it makes it O(NM).

Part 4:

The rest of the module deals with calculating the cosine similarity and returning the 10 pages with the highest score. Before I calculate the score, I check if the left or right denominator equals 0 since anything divided by 0 is 0, if it is zero I append 0 to tuples list and then continue since its pointless to calculate anything at that point. The calculation part is the numerator divided by both denominators. Then multiplying it by the page rank is applicable. After I calculate the cosine similarity, I add it to the *returnscores* dictionary under 'score'. I also create a tuple of iteration and score to help in my sort of the values. The first index in the tuple is the key in the *returnscores* dictionary while the second is the score. I sort the list by the second value in the tuple. Then I return the top pages by searching in the dictionary by the index value as the key.

The runtime complexity of this part is O(N) as N represents the number of URLs found in crawl. This is because the loop to add the top 10 pages to *topScores* always runs 10 times making it O(10), this leads to the calculation portion of this part to be highest factor in complexity. Therefore, the overall runtime complexity for search.py is O(N).

Module: os_crawl.py

Overall, this module handles most of the reading/writing to files. Most functions in this file is either O(1) because there is no loop in the function or O(N) if the function is getting repeatedly called in a loop. The whole module is used for my file structure to keep my program organized.

Function: *check_directory():*

This function is used to check if that directory exists. Returns True if it exists false otherwise.

Function: *create_directory():*

This function is used to create a new directory if there already isn't one with that name.

The *delete_directory():* functions is the inverse

Function: *create_json_file():*

This function takes a directory name and a filename with content and does a JSON dump of the content. The function first checks if the dir_name is valid before creating the file. The runtime complexity of this would be O(N) because the amount of data read to the file is proportional to the size of the data.

Function: *load_json_file():*

This function takes a file that stores a JSON string and reads it. This is used in searchdata.py for *get_outgoing_links():* and *get_incoming_links():* since the data required for these are stored in a JSON format. It is also used to read in the filemap.txt data. The runtime complexity of this function is O(N) for the same reason as the function above.

Function: reset_dir():

This function takes in a directory that contains other directories and deletes everything. It works by going through every item in the original directory and checks if it's a folder; if then does the same thing to the folders found. I created this because I needed a way to do a general reset of crawl_data directory without the previous tests data being an issue. The runtime complexity of this module is O(N*M) with N being number of folders in crawl_data and M being the number of files in tf and tf_idf sub directories.