COMP 1405Z – Fall 2023 Term
Final Project Analysis
Jonathan Pasco-Arnone & Aidan Lalonde-Novales

Project Overview

The purpose of the project, and a brief description of the modules and data structure.

Our program acts as a basic search engine, which presents users with relevant webpages based on a search term, as well as information surrounding those webpages. This is done through the implementation of various modules (e.g., crawler.py, searchdata.py) and methods (e.g., crawl(seed), get_page_rank(URL)), which provide the necessary functions of a search engine. We decided to focus on optimizing time complexity over space complexity in our search engine, as users will likely want the results of their search at the quickest speed possible, even if that comes at the cost of higher memory usage.

The crawler module (crawler.py) downloads and indexes a wide variety of content (e.g., title, links, words) from a provided 'seed' URL. This content is then stored within the 'crawler_data' folder, which is used by other modules to derive information from certain webpages.

The searchdata module (searchdata.py) analyzes data in the 'crawler_data' folder to produce needed information about a webpage and related webpages. This includes the incoming and outgoing links from a URL, the significance of a URL (specifically a 'page rank'), the inverse document frequency of a word, term frequency of a word, and the combined 'tf-idf' weight of a given word.

The search module (search.py) determines the ten most relevant webpages to display to the user. The 'score' of a webpage determines how relevant it is, which is derived based on data like term frequency of the search word in a document or through the page rank algorithm in the searchdata module.

All these modules are found under the 'testing-resources' folder, along with various test files (e.g., fruits-all-test.py) and the results of those test files (e.g., fruits-all-idf-passed.txt). The 'testing-resources' folder also contains the 'crawler_data' folder, which stores data from webpages in numerical-ordered folders (i.e., 1, 2, 3, …), the file 'idf.txt', which stores how many times certain words appear in a network of files, the file 'page_rank_file.txt', which is a matrix that stores values pertaining to how important a file is, and the file 'link_locations.txt', which is a dictionary between crawler_data folders and the link of the webpage they represent.

File Structure

Overall file hierarchy, locations of key files and folder, the purpose of those folders, the files they contain, the significance of those files

Our file structure is split into two main directories, the 'docs' folder and the 'testing-resources' folder. The 'docs' folder stores our readme file, which contains the authors of the program and instructions for running the program from the command line. It also contains this analysis document, and the course project specifications for quick reference. The 'testing-resources' folder contains the main components of the program, like the four modules used to derive data and assign importance to webpages, as well as the test files, results to those test files, and data storage like the crawler_data folder and text files like 'idf.txt' and 'link_locations.py'. The essential modules in the testing-resources folder are crawler.py, searchdata.py, search.py, and matmult.py, which are stored here as they have easy access to the test files, due to being in the same directory.

A key folder worth touching on in the testing-resources folder is the 'crawler_data' folder, which stores all the data derived in the crawler. This data is stored in a list of folders in ascending numerical order, which contains four key files which provide information about a webpage. These files are 'file_text.txt', which contains the words in a webpage, 'incoming_links.txt', which contains the links which point to a given webpage, 'page_links.txt', which contain the outgoing links in each a given webpage, and 'title_and_link.txt', which contains the title and the link of a given webpage. We decided to store our data like this as it provides easy organization and data access from the crawler, so that it may be accessed by other modules with ease (e.g., searchdata.py) at an efficient space complexity of O(N), where N is a webpage. The only downside to this is that you cannot locate a specific webpage without indexing through each crawler_data folder's 'title_and_link.txt', which would harm our time complexity. We got around this by implementing a 'link_locations.txt' folder, which contains pointers for each webpage within the crawler_data directory. This saves on time complexity by avoiding indexing N folders and N files within the folder, instead just indexing a single file, bringing the time complexity of finding a webpage from $O(N^2)$ down to O(N), at the cost of an additional O(N) space complexity, where N are the total webpages in the link_locations.txt file.

Another file worth mentioning in the testing-resources directory is 'idf.txt', which tells us which words are contained in a network of webpages. This file is essential for reducing the time complexity of the search(phrase, boost) method in the search.py module, as the search method relies on accessing the idf values of a word to determine which webpages are more relevant to the user. This allows the search(phrase, boost) method to have much lower time complexity, with O(N), where N is the number of words in the idf file, as opposed to $O(N^2)$, which would be accessing N webpages with N words in them to find idf.

Lastly, the file 'page_rank_file.txt' within the testing-resources directory is very similar to 'idf.txt', telling us the page rank of each webpage in a network of webpages. This file also serves to reduce time complexity at the cost of space, essentially being a more accessible pointer for the page rank of every webpage in the crawler_data folder. This file allows us to have a time complexity of O(N), where N is number of webpages in a network, while without this file, each directory would have to be accessed one at a time, giving us a suboptimal time complexity of $O(N^2)$, where N files page_ranks are being accessed in N directories.

Every other file not mentioned above in the testing-resources directory falls under one of two categories. It is either a testing file, used to run through test cases and validate our code, or an already prewritten module to help streamline our code. Examples of these testing files are the python testing files (e.g., fruits-all-test.py) and the text-based result

files. Examples of the prewritten modules include the webdev.py file, streamlining reading URLs in a webpage, and the matmult.py file, which allows to make matrix operations (imperative for the page_rank method in searchdata.py).

Crawler Module: crawler.py

In-depth purpose and explanation of the module
All methods explained, with time and space complexity provided

The crawler module (crawler.py) has one method, called 'crawl(seed)', with one input variable, 'seed', which is a string value that represents a webpage. The crawl method creates a network of webpages that link to and from the seed webpage, and downloads and indexes webpage data like the title, link, outgoing links, incoming links, and page text of every webpage in this network. These are stored in the 'crawler_data/(number)/' directory, under the files 'file_text.txt', 'incoming_links.txt', 'page_links.txt', and 'title_and_link.txt'. It is essential that this module does this as this data is needed by other modules for the fulfillment of a user's search.

crawl(seed):
This method begins by removing every previous folder and file in the crawler_data directory, allowing for a new webpage crawl. It then creates a new directory for each webpage in the crawl, as well as four files for the titles, link of the webpage, incoming links, outgoing links, and the text on the webpage. Webpage titles are grabbed from webpages by looking at the string between the html tags <title> and </title>, text in a webpage is grabbed by looking at the string between the html tags and , and links are grabbed by looking at the string between the html tags <a href=\" and \">. The webpages being downloaded and indexed are those that have outgoing links pointing to the seed webpage, or incoming links from the seed webpage. The seed webpage is provided by the input variable 'seed', which represents a webpage in string form.

This algorithm gives a space complexity of O(NxM) or $O(N^2)$, where N is number of webpages and M is the number of words in a webpage. This allows us to save on time complexity of other methods in other modules, as all the required data will be easily accessible within the crawler_data directory. The time complexity of downloading and indexing all this data is O(N), where N is the number is webpages.

Search Data Module: searchdata.py

In-depth purpose and explanation of the module
All methods explained, with time and space complexity provided

The searchdata module (searchdata.py) analyzes data in the 'crawler_data' folder to calculate the incoming and outgoing links for a webpage, the page_rank of that webpage (i.e., it's importance among other webpages), the term frequency of a word in a specific webpage, the inverse document frequency of a word (i.e., which websites in the

crawl contain a word and which do not), and the tf_idf weight of a word in a webpage (i.e., logarithmic combination of the term frequency and inverse document frequency of a word).

To minimize the amount of time the program takes to run we have chosen to do the vast majority of computation during the original crawl, making get_outgoing_links, get_incoming_links, get_tf, get_idf, and get_tf_idf all have a big O notation of O(1). The only function in this file that does not have a big O of 1 is the get_page_rank function. This is because the first time running the program it must generate all the values making the complexity O(n). Although, after this first run, all the page ranks will be generated. Making the big O notation O(1). Unfortunately, the lower runtime comes at the cost of a higher space complexity by having multiple files and folders to store all this data. However, these files take up an insignificant amount of space for the drastic increase in runtime making this choice more optimal for speed.

get_outgoing_links(URL):
This method takes an input variable, 'URL' (a string representing the URL of a webpage) and returns a list of the outgoing webpages on the webpage that pertains to the input URL. This is done by looking through a webpage's 'page_links.txt' file in the crawler_data directory. This has a time complexity of O(1), as there is no looping through input data, making this method very time efficient.

get_incoming_links(URL):
This method takes an input variable, 'URL' (a string representing the URL of a webpage) and returns a list of the webpages that have links pointing to the webpage that pertains to the input URL. This is done by looking through a webpage's 'incoming_links.txt' file in the crawler_data directory. This has a time complexity of O(1), as once again, there is no looping through input data, making this method very time efficient.

get_page_rank(URL):
This method takes an input variable, 'URL', and determines the importance of the webpage pertaining to the URL by considering how often it links to other webpages compared to how often other webpages link to it. This is done in our code by creating a probability matrix which represents a network of webpages, with each webpage being assigned a weight. If a webpage is linked to often but doesn't link to other webpages as often, it will have a higher weight than that of other webpages that aren't linked to often but link to other webpages often. These weights are then scaled by 1 – an alpha value (0.1) and iterated through by steady state probabilities, which will steadily change weights into their page ranks, ending once the euclidean distance between a basic vector and a vector representing the weights of each webpage is low enough (> 0.0001).

This method will have a time complexity of O(N) for its first run, where N is the size of matrix, which is determined by the size of the network of webpages (i.e., more webpages, more time to run). However, after this first run, all the page ranks will be generated in the 'page_ranks_file.txt', making the big O notation O(1) because determining page rank can be done by simply pointing to the index of the webpage in page_ranks_file.txt. By creating a file with all the page ranks in it after the initial use of the get_page_rank() method, we can save time for future runs.

get_idf(word):
This method gets the inverse document frequency of a word (string input value) in the web crawl. This returns a value based on how many times a word appears in each 'file_text.txt' file in the crawler_data directory. The 'idf' value is

calculated by returning the base 2 logarithm of the number of webpages in the crawl divided by number of webpages containing the word in the input variable, plus one. This method has a time complexity of O(1) as there is no looping of the input, the algorithm simply finds the 'idf' value by getting the number of webpages in crawler_data and the inverse document frequency from the 'idf.txt' file. By looking through files in memory, we save time.

get_tf(URL, word):
This method gets the term frequency of a word in a URL, with the input values representing the string of the word and the URL. To check how many times a word appears in a webpage, this method loads the webpage pertaining to the URL into memory and checks how many times the input word appears in the 'file_text.txt' file pertaining to the webpage. This method then returns the time the word appears divided by the total number of words in the file. This has a time complexity of O(1), as the input values are not looped through, instead the values are calculated by simply looking through the words in an associated 'file_text.txt' file. By looking through a file in memory, we save time.

get_tf_idf(URL, word):
This method gets the 'tf_idf' weight of a URL and word that are provided as input variables. This is done by just running the get_idf() and get_tf() methods while passing the URL and word into them (if needed by the method) and getting the base 2 logarithmic value of the 'tf' weight plus one, and multiplying this by the 'idf' weight. This gives us a time complexity of O(1), as the most complex thing taking place are the get_idf() and get_tf() methods running, which have time complexities of O(1).

Search Module: search.py

In-depth purpose and explanation of the module
All methods explained, with time and space complexity provided

The search module (search.py) contains one method, being search(phrase, boost), which determines the ten most relevant webpages to display to the user based on a score. The score is either determined by the most optimal 'tf_idf' weights or the most optimal page ranks in the crawl. These values are obtained in the searchdata.py module and can be toggled between using the 'boost' boolean input.

search(phrase, boost):
This method takes a word as a string for its phrase input, and a boolean value for its boost input. If boost is true, then the score of a webpage will be based on its page rank, while if boost is false, the score of a webpage will be based on its tf_idf value. These values are converted to cosine similarity values through 'q' and 'd' values calculated in the method, which represent the score of a particular webpage. These scores, represented through cosine similarity, determine the placement location of a webpage when it comes to being searched, and the URL, title, and score of the top ten websites in the crawl are placed into a dictionary for data storage. The method then returns the top ten results.

The time complexity of this method is O(NxM) or $O(N^2)$ where N is the number of webpages and M is the number of search queries. Since the most search queries requested is about 10 words and the most webpages are 1000, the M

variable is rather insignificant. Because of this, the time complexity really comes out as the big O notation equivalent of O(n), due to the M variable being obsolete. This method primarily just follows the instructions provided to us, as there weren't many opportunities to optimize the time of looking through up to 1000 webpages through storage.

Additional Information

All important information that does not fit under the above categories.

- The matmult module (matmult.py) allows various matrix operations to take place, which are particularly important for calculating 'page_rank' values. This module can multiply a matrix by a scalar (mult_scalar(matrix, scale)), calculate the dot product of two matrices (mult_matrix(a, b)), and calculate the Euclidean distance between two points in a matrix (euclidean_dist(a, b)). Only the functions mult_matrix() and euclidean_dist() are used within our program, with a time complexity of O(NxM) or $O(N^2)$ for mult_matrix(), where N is the collective rows and M is the collective columns of two matrices, and a time complexity of O(1) for euclidian_dist().