Report on Final Project-Search Engine

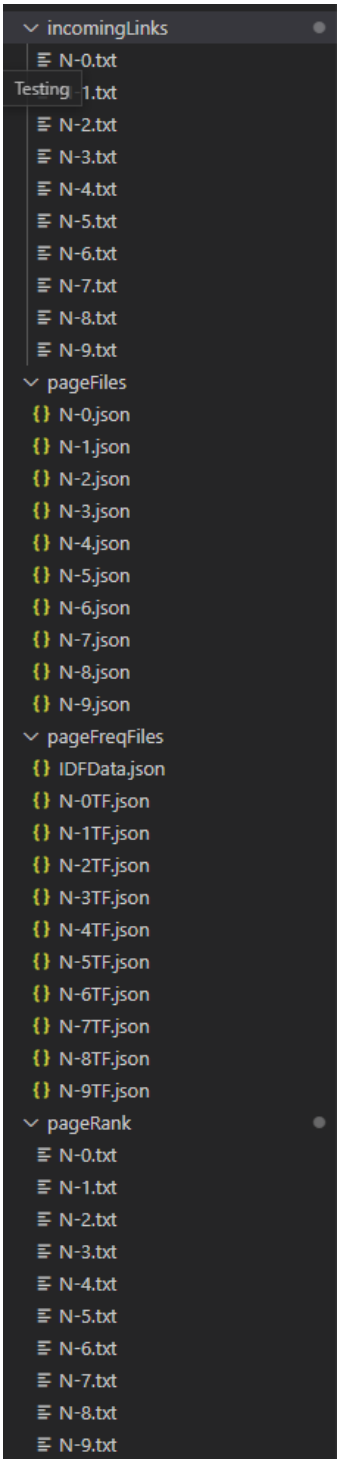
Grant. L.,Venuja P.

COMP1405Z

26/10/2022

**File Structure**

A file structure that optimized for scalability, and readability was used to store all necessary data retrieved by crawler and pagerank.




The following directories are used to store data:

- pageFiles: This directory stores data retrieved by the crawl including the word count, URL, title, and outgoing links for each page. All of this data is stored in a JSON format with the keys representing the type of data and the value representing the specific value of the data for a given file.
- pageFreqFile: This directory contains two special JSON files and JSON files representing each page retrieved in the crawl. The first of these special files contains all of the IDF data for all words seen in the crawl. The second stores data representing the number of documents each word was seen in. The keys in both of these files represent the word in question with the value representing the IDF value and the document frequency respectively. All of the other JSON files follow an identical format. Within these files is data representing the relative frequency in each of the pages retrieved in the crawl as well as the URL and title of the page.
- incomingLinks: This directory stores the incoming links for each page retrieved in the crawl in the form of .txt files. The file name represents the page the data belongs to. Each file contains one line of text with each incoming link separated by a space.
- pageRank: This directory a series of txt files. Each txt file is named after the page it belongs to. Within each file is one line representing the pageRank of that particular file.

**Assumptions:**

- It was assumed that all html files searched were correctly formatted
- It was assumped that all files end in .html
- It was assumed that each file has only one <a> tag, however crawl is capable of reading more than one <p> tag

If any of these are not true the crawler will not be accurate
All required parts of the search engine are fully functional when tested agains the test case

# Crawler Module

## Introduction

The main purpose of the crawl module is to collect all data on a website given its URL, and all other websites that can be reached directly, or indirectly from it. Furthermore, it also calculates all necessary data to be retrieved in other modules.

## Notable Functions

**crawl() -** This function initiates clears/creates all necessary directories used to store page information using **clear_Data()**. It uses a queue system to crawl through a seed webpage and all webpages that can be accessed from it. Also using **get_text()** to process information about each visited webpage**,** **get_TFIDF_Data()** to calculate TFIDF values for each unique word in each web page**,** and **pageRank.pageRank()** to calculate page ranking for each web page.

To reduce run time, the method of crawling interlinked webpages was changed from recursion to interactive. Which reduced our average run time by roughly half( ~150 seconds to ~65 seconds). This is because recursion was implemented when I didn't think of a way to do it with a while loop. But after seeing the high run times for testing, I decided to spend time to change it.

**get_text() -** First gets the title of the webpage from the HTML, then calls **get_wordCount()** to get a dictionary with information about all unique words. It then gets all outgoing links from **get_links()** and stores them along with the URL, title, and total word count in the dictionary. The dictionary is then saved as a JSON file in /pageFile. A list of outgoing URLs is also returned to keep crawling in **crawl()**.

**write_term_frequency() -** Uses the webpage dictionary from **get_text()** and the total word count of a page to calculate term frequencies for each word of that webpage using a for loop to process the term frequency of each word in the dictionary. Then it stores the term frequency information in a JSON file in the directory: /pageFreqFiles

**addIncoming() -** given a URL representing a webpage and one of its outgoing URLs. The function opens the outgoing URL's webpage file in the /incomingLinks directory and then adds it to the txt file.

Originally this function would write to a JSON file. However this increased the time complexity of this function to O(n) as it required the JSON file to be loaded every time. Thus, the final program uses text files which do not need to be loaded prior to writing to them.

**clearData() -** First checks and/or creates the necessary directories using **checkFileDir()**. Then it resets all global variables and empties out all existing directories of their files using individual for loops.

**get_TFIDF_Data() -** First calls **get_IDF_Data()** to return a dictionary of IDF values for every unique word found during the crawl, this dictionary is later stored in /pageFreqFiles. Using this dictionary, it loops through every JSON file in /pageFiles containing information about each visited webpage in the crawl. In a loop, a new dictionary is created to store TFIDF values of each unique word on that webpage. An inner loop loops through every TF value of that webpage to create TFIDF values to put in the new

dictionary. The JSON file for that webpage is then updated to have TFIDF values of each word from the new dictionary.

**get_IDF_Data() -** First it loops through all the files in /pageFiles, and if a unique word exists in a given file, it is put into a new dictionary as a key with a count of 1. All subsequent instances of this word found in the other files of /pageFiles will increment its count in the dictionary. It then loops through the dictionary to calculate the IDF values of each unique word found during the crawl. This dictionary is then stored as a JSON file in /pageFreqFiles.

**Time Complexity Table**

| Function | Description | Worst Case Time Complexity | Space Complexity |
|---|---|---|---|
| crawl() | Initiates all data collecting and processing. | O(x(c*r*v)). Where x is the number of repetitions to convergence. Where r is the number of rows, c is the number of columns in the matrix, and v is the length of row r. This is because pageRank runs inside of crawl. | O(n) |
| get_links() | Returns an array of all outgoing links. | O(n). Where N is the number of links in the HTML given. | O(n), variables used are created in a single while loop |
| get_text() | Finds and processes text information in a webpage. | O(n*m), where n is the number of <p> blocks and m is number of words in the p block. Because of word_count(). | O(n), not including the return dictonary, where n is the size of the words array. |
| get_wordCount() | Returns numerical data based on words in a webpage. | O(n*m), where n is the number of <p> blocks and m is number of words in the p block | O(n), not including the return dictonary, where n is the size of the words array. |
| write_term_frequency() | Processes and stores term frequency data for each webpage. | O(n) where N is the number of unique words on a page | O(n), where n is the size of the tf dictonary |
| addIncoming() | Adds an incoming link for a particular page | O(1) | O(1) |
| clearData() | Deletes all of the files in data storage directories | O(n), where n is the number of files in the directory | O(1) |
| checkFileDir() | Checks if all of the file directories needed for storage exist and creates them if they do not | O(1) | O(1) |

| | | | |
|---|---|---|---|
| get_TFIDF_Data() | Calculates TFIDF data for the crawl | O(n*m) where n is the number of files in /pageFiles, and m is the number of unique words in a file | O(n), where n is the size of idfData |
| get_IDF_Data() | Calculates IDF data for the crawl | O(n*m) where n is the number of files in /pageFiles, and m is the number of unique words in a file | O(n), where n is the size of itemPages |

# Page Rank Module

## Introduction

The page rank module takes the data crawled by the crawler module and returns the page rank values for each of the discovered pages. All of the data produced by this module is saved in the pageRank directory in text files. The main function of this module to be run only if the crawl has completed.

This module is designed in a modular fashion such that the page rank function calls a handful of helper function in order to accomplish its final goal. The modular design also greatly improves code readability and organzation thus improving scalability considerably. It also allows for quick pivots in system design such as the switch from json data for testing to txt files for the final application.

## Notable Functions

**createMap() -** This is a crucial function that creates a map based on the number of pages discovered in the crawl. It creates two distinct dictionaries that allow for the easy reference of an index within the matrix to the corresponding page and vice versa. The url-index map is used to create the initial matrix. The index-url map is used to record the page rank data once the final vector has been computed.

**createMatrix() -** This function creates a zero matrix of nxc size, where n is the number of pages and c equals n. Each of the columns represent an incoming link from the cth page and each of the rows represent an outgoing link from the nth page. It then populates this matrix based on the connections between pages. It does so by going through all of the files in the pageFiles directory and grabbing all of the outgoing links for a particular page. It will update the values in the nth row if there is a connection between the nth page and any other page. I will also update the values in the nth row if the cth page has a link to the nth page.

**randomProbability() -** This function traverses the matrix and modifies the elements in a particular row based on predefined specifications. If all of the values in a particular row are 0, then each of the values are changed to 1/the number of pages. Otherwise, each of the ones in the row is divided by the number of ones in the row.

**piMultiplication() -** This function multiplies a vector in the form [1 0 0 … 0n] by the final matrix until the euclidan distance between the resultant vector and the previous vector is equal to or less than .001.

**altSaveData() -** This function saves all of the data to text files inside of the pageRank directory. It does this using the index-url map that maps every index in the final vector to the corrapsonding page. Originally, a single JSON was used but this required either the time complexity or space complexity of any function using page rank to be O(n). Using text files allows any function accessing pageRank data to reduce this access time to O(1) space and time complexity.

**Time Complexity Table**

| Function | Description | Worst Case Time Complexity | Space Complexity |
|---|---|---|---|
| resetData | Resets all of the global variables | O(1) | O(1) |
| multScalar | Multiplies a matrix(2D array) by an integer | O(r*c), where r is the number of rows and c is the number of columns in the matrix. | O(1), not including the return matrix |
| multMatrix | Multiplies a matrix by another matrix | O(c*r*v), where r is the number of rows, c is the number of columns in the matrix, and v is the length of row r. | O(1), not including the return matrix |
| euclid_dist | Returns the euclidean distance between two arrays | O(n), where n is the length of the array | O(1), not including return value |
| fetchURL | Gets the url of a page given its url | O(n) | O(n), where n is the size of the dictonary |
| createMap | Creates a mapping of indices to urls and urls to indices. | O(n), where n is the number of pages | O(n), where n is the sum of the sizes of the mapping. This function updates a global variable, each global variable takes O(n) space. |
| createMatrix | Creates the matrix with nxn dimensions | O(n*l), where n is the number of pages and l is the number of outgoing links in the nth page. | O(n), where n is the size of the matrix being stored |
| randomProbability | Modifies the matrix based on the predefined specifications. Adds random probabilities if necessary. | O(r*c), where r is the number of the rows in the matrix and c is the number of columns. | O(1), the function itself does not use or add any extra space. |
| modAlpha | Modifies the matrix based on the predefined specifications. Multiplies every value in the matrix by 1-alpha. Also adds alpha+n to each value in the matrix. | O(r*c), where r is the number of the rows in the matrix and c is the number of columns. | O(1), the function itself does not use or add any extra space. |
| addCurrVector | Adds 0s to end of an array. | O(n), where n is the number of pages-1 | O(1), not including return array |

| | | | |
|---|---|---|---|
| piMultiplication | Multiplies a vector of n length by the matrix, where n is the number of pages, until the eculidian distance of the vector is less than .001. | O(x(c*r*v)). Where x is the number of repetitions to convergence. Where r is the number of rows, c is the number of columns in the matrix, and v is the length of row r. | O(n). Where n is the number of pages multiplied by two. |
| altSaveData | Saves the data to text files. | O(n). Where n is the number of files | O(1). Function does not use or add extra space in RAM. |
| Overall | | O(x(c*r*v)). Where x is the number of repetitions to convergence. Where r is the number of rows, c is the number of columns in the matrix, and v is the length of row r. | O(n) |

## searchData Module

### Introduction

This module contains a series of functions that retrieve a given piece of crawled data based on varying parameters.

Time complexity was of greater priority when compared to space complexity, for example getting the idf data of a crawl is loaded once into a global variable. This allows for a much lower get_idf function time complexity at the expense of space complexity.

### Notable Functions

**fetchIDFData()** - This function loads in the IDF data for all of the words found in the crawl from the JSON file in pageFreqData.

**get_outgoing_links()** - This function loads in the JSON file of for the corresponding page and returns the value of outgoingLinks within it

**get_incoming_links()** - This function grabs the data inside of the the corresponding text file in the incomingFiles directory and returns an array form of that data.

**get_page_rank()** - This function takes in a url and loads the corresponding text file in the pageRank directory.

**Time Complexity Table**

| Function | Description | Worst Case Time Complexity | Space Complexity |
|---|---|---|---|
| fetchIFData | Loads in a the idf data as a dictionary | O(n), where n is the number of elements in the JSON file | O(n),where n is the number of elements in the JSON file. Function sets a global variable of n size. |
| get_page | Gets the page name based on the url | O(n), where n is the length of the url | O(1) |
| checkURL | Checks if a file was found in the crawl based on the url | O(1) | O(1) |
| get_outgoing_links | Gets all of the outgoing links from a particular page | O(n), where n is the length of the JSON file | O(n) |
| get_incoming_links | Gets all of the incoming links for a particular page | O(n), where n is the number of characters in the txt file | O(1) |
| get_page_rank | Gets the page rank value for a particular page. | O(1) | O(1) |
| get_idf | Gets the idf value for a particular word | O(1) | O(1), function itself does not use any extra space. |
| get_tf_data, get_tf, get_tf_idf | Gets the term frequency or tfidf data for a particular word page combination | O(n),where n is the length of the JSON file | O(n),where n is the size of the JSON file |

# Search Module

## Introduction

The search module is responsible for returning the most relevant pages based on a query string. It does this by performing a predefined algorithm that assigns a score to each page found in the crawl. It then finds the best 10 pages and returns these 10 pages along with their URLs, titles, and scores.

Similar to the pageRank Module, the search module is designed in a modular fashion such that the program can easilby be diciphered and expanded as needed.

This search module is not always perfect due to the number of files being searched and relaitivly small and close distribution of some of the page scores.

## Notable Functions

**insert_List() -** This is a crucial helper function that takes in a page data and determines where it fits within the final rankings

**calc_CS() -** This function calculates the cosine similarity score of a given page given the page vector. It does this by comparing the query vector to the page vector and calculating the euclidian distance between these two. It also multiples this score by the page rank value of the page if boost is true.

**populateBasisVector() -** Traverses the query phrase and adds every word to the basis vector. Also counts the frequency of each word in the query string and saves it as a dictionary.

**populateQueryVector() -** Traverses the query dictionary and adds the value of every entry(the frequency of a word) to the query vector, an array, and divides every value of the query vector by the length of the query. This then traverses the query vector and takes the $\log_2$ of each entry in the vector plus one and multiplies it by the IDF value of the corresponding enetry in the basis vector.

**compareScore() -** Takes in a dictionary containing the data for the page and compares it to lowest score entry in the return array. If the length of the return array is less than 10, it automatically adds the new page to the return array using the insert_List function. Otherwise, the new page will only be added to the return vector if it's score is greater than the lowest score entry in the return array, also uses insert_List to insert the new page.

**calculateScore() -** Opens the JSON file of each page and pulls the title and url. Score is calculated using the calc_CS function and compared to the existing return array using compareScore().

**Time Complexity Table**

| Function | Description | Worst Case Time Complexity | Space Complexity |
|---|---|---|---|
| insert_List | Takes in a dictionary and inserts it into the return array depending on scores | O(n), where n is the length of the original return list | O(n), where n is the size of the original return list |
| calc_CS | Caluclates the cosine similarity of the basis vector and query vector | O(n), where n is the length of the query vector | O(1) |
| init | Resets all global variables, splits the string entered by the user into an array, loads IDF data from the JSON file. | O(n), where n is the length of IDF data | O(n), where n is the size of IDF data |
| populateBasisVector | Searches the query vector and adds each element, along with the element frequency to the basis vector and query dictionary respectivly | O(n), where n is the length of the query | O(n), function updates a global variable. N is the sum of the size of the query dictionary and basis vector. |
| populateQueryVector | Preforms necessary calculations to create the query vector. | O(n), where n is the sum of the lengths of the query dictionary and basis vector | O(n), where n is the size of the query vector |
| compareScore | Takes in a dictionary and determines if it should be inserted into the return array | O(n), where n is the length of the original return list | O(n) worst case. Where n is the size of the original return list. |
| calculateScore | Calculates the score of a given page | O(n * m), where n is the number of pages discovered in the crawl, and m is the length of the basis vector | O(n), where n is the size of the page dictionary |
| Overall | | O(n * m), where n is the number of pages discovered in the crawl, and m is the length of the basis vector | O(n), where n is the size of the page dictionary |