Project Analysis

Introduction

The project consists of 3 primary components:

1.) The crawler(crawler.py): It accepts a link from a user and crawls all the links and the word data in each subsequent links and stores it locally in respective JSON files. Calculation of various values are done on-off as the crawl happens for future search access. This is done such that the searches are done fast as heavy computations makes the search very slow and affects user experience every time.
2.) The search calculations retrieval(searchData.py): There are various calculations to calculate certain variables that determine the relevancy of the page in search. This file consists of functions that retrieve a specific computed value.
3.) The search engine(search.py): It makes use of the search calculations retrieval file to retrieve required search values and uses them to generate a top 10 list of pages that are most relevant.

The directory structure:

There are 5 specific directories for storing data:

1.) Idf: Has a file words.json which stores a json of all the words and their computed idf values.
2.) pageRank: Has an array of files which store the computed url pagerank value in json format for each url.
3.) Tfidf: Has an array of files which store the computed url tfidf values for unique words found during crawl over all websites in json format for each url.
4.) Tf: Has an array of files which store the computed url tf values for unique words found during crawl over all websites in json format for each url.
5.) webData: Has an array of files which store the computed parsed words, links and page title found during crawl over all websites in json format for each url.

These computed values were broken down like this in multiple files for each url for efficiency as loading all irrelevant values just to extract one piece of information takes a long time and search speed is the main priority for user friendliness. There is a tradeoff in disk space and RAM, but computational memory is of greater priority over hardware space.

Beyond the 3 main files, there are other files as well. The crawl and the search functions have their own helper functions in separate files as well as the generate_pageRank() function. There's also a general helper functions' file which is used by multiple files. This was done to keep the code base clean and readable and make the applicable functions reusable. It also has a CONSTANTS.py file which stores the values of constants which are not to be changed. This makes editing/ updating easier as the constant is only needed to be changed in one part for it to be applied to everywhere its been used. Also, having it in

a separate constants file prevents it from accidental change of value while execution. It also makes certain things clearer for the programmer reading the code.

File crawler.py:

This file, namely the crawl(seed) function is associated with the crawling of data.
- One global variable is used:

1.) linkQueue- A list that stores the links that are to be crawled. The queue keeps on getting filled as new links come in and the links are simultaneously accessed and dequeued as the loop happens.

- There's one function:
    - Crawl(seed):
        - This is the function that carries out the crawl task overall. It accepts a link as a seed from where it starts the crawl. First it clears all data of previous crawls using the 'clearPrevCrawl()' function. Then, it initializes the linkQueue queue by adding the seed to the queue. After that, it starts a loop over the linkQueue and retrieves the html data from the current active url in the linkQueue using the 'read_url()' function in webdev.py file. It parses the html elements to discern the page title, word data, links, searches and adds every unique word it crawls to the 'uniquewords' list, and adds unique links to the active linkQueue using the 'parseHtml()' function and adds all data to a json file using the 'addDataToFile()' function and deletes the current url that was crawled all in one loop. The loop count is incremented in 'linkAccessed' variable to keep track of the number of urls accessed.
        - Then, it generates the idf value for each unique word in the 'uniqueWords' list and stores it in a json file using the 'generateIdf(….)' function. It also generates the tf-idf for every word it crawled and stores it in their own json file using the 'generate_tf_tfIdf' function. It also calculates the pageRanks for each url and again stores then in the urls' own json file separately.
        - The data required to make these final calculations are computed as the loop happens as it makes things much faster. All these calculations make the crawling slower but it is for the greater good as it is not a frequent operation and rather makes the search faster.
        - It returns the total number of links accessed.
- **Time Complexity:**
    - **'N'** be the number of files in the webData directory**,**
    - **'L**' be the number of links accessed for crawl,
    - **'U'** be the number of unique words,
    - **'W'** be the number of words in each url, and
    - **'F'** be the fail count of the url request.
        - The time complexity of the 'clearPrevCrawl()' function is **O((4*L)+1)** as it goes and deletes the files present in the 5 crawled data directories, where 4 directories have L files for the number of links crawled and 1 directory which has 1 file.

- For the first loop,
    - The time complexity of webdev.read_url() function is **O(10)** as in the worst case, there can be 10 failed requests.
    - For the parseHtml() function:
        - The first loop has **O(W+I)** time where W is the number of words and I is the number of links present in the webpage as it goes through each of them to remove their respective html tags.
        - The getLinks(..) function has **O(I)** linear time as it goes through each link whose html tag was removed and creates the final url using the absolute url, also adding the url to the linkQueue.
        - For the last loop, the time complexity is **O(W)** where W is the number of words in the page as it goes through every word and counts the frequency of the word as well as adds it to the uniquewords list if it isn't already in the list.
        - So in the worst case, the time complexity for the function is **O(W+I)**
    - Both the addDataToFile() and the dequeue() functions have **O(1)** complexity as the amount of data passed doesn't make any difference in either of the functions.
- This code block has a linear time complexity of **O(4L+L)** where L is the number of links accessed for crawl.
    - So the overall time complexity for the function is **O(L * (W+I)).**
- For the second loop,
    - The generateIdf(..) function has O(L) linear time where L is the urls accessed for crawl as it has to go through every page to find the idf of a word.
    - The generateIdf(..) function is executed 'U' times, so the time complexity of the overall block would be **O(U*L)**
- For the last function, 'generate_tf_tfidf(…) the time complexity is **O(L* U).** Here as well, every url is gone through and every unique word is checked for every url to generate tfidf value.
- For generate_pageRank():
    - For generate_probabilityTransitionMatrix():
        - It has to go through the list of urls twice: as we need to generate a row for every url in the matrix and then again go through each url again to generate the row. Hence, its time complexity is O(L^2).
    - For generate_scaled_adjacentMatrix(probabilityTransitionMatrix):
        - The time complexity is **O(L)** as there is a row in the probability matrix for every url that was accessed for the first loop. For the second loop, it is **O(U)** as 0/1 value is given to every unique word.
        - Therefore the time complexity is **O(L*U)**
    - For generate_finalMatrix(scaledAdjacentMatrix):
        - The time complexity is **O(L)** as there is a row in the probability matrix for every url that was accessed for the first loop. For the second loop, it is **O(U)** as 0/1 value is is present for every unique word.

        - Therefore the time complexity is **O(L*U).**
    - For mult_matrix(matrixX, matrixY):
        - For two matrices of dimentions [A x B] and [C x D]:
            - It has 3 nested loops:
                - First goes through the rows (A) of matrix X.
                - Second goes through the columns (D) of matrix Y.
                - Third goes through the rows (C) of matrix Y.
                - Hence the time complexity is : **O(A*D*C)**
    - For euclidean_dist(a,b)
        - For two matrices of columns X and Y then the time complexity is O(X) as it does a parallel iteration on X and Y.
    - Since there's so many functions required this has its own helper functions in a separate file: pageRankHelperFunctions.py

- **Space Complexity:**
    - The linkQueue list at a point can store up to N links where N is the number of pages that are linked to each other. So the complexity is O(N)
    - Uniquewords list stores all unique words. So, if U is the number of unique words present in all of the crawled data, then its time complexity is O(U).
    - The Pagesword count stores the frequency of each word of each url in a dictionary. So in worst case if a page has all the uniquewords in it then, the space complexity would be O(U*N) where U is the number of unique words and N is the total number of links accessed for crawl.
    - The urlIndexMap and the urlOutgoings dictionaries both have the time complexity of O(N) N is the total number of links accessed for crawl.

FILE: SearchData.py:

- This file simply consists of a number of functions which specifically retrieves certain values/ computations that was computed by the crawl and uses it in the desired way. Since the computations was done during the crawl, these functions are very fast as they only have to retrieve data stored in files.
- The Functions here are as follows:
    1.) get_outgoing_links(linkstring):
        - It takes a url parameter and returns a list of links that is contained in the url's page.
        - It retrieves the data stored in the 'webData/[URLNAME].json' file.
    2.) get_incoming_links(linkstring):
        - The data is utilized from the 'webData/[URLNAME].json' file.
        - It goes through every url and checks if the url links to the passed link.
        - It returns a list of urls that contains the passed url.
    3.) get_idf(word):
        - It returns the idf value of a passed word.

- returns 0 if the word doesn't exist.
- It retrieves the data stored in the 'idf/words.json' file.

4.) get_tf(url,word):
- It returns the tf value of a passed word in the passed url.
- It retrieves the data stored in the 'tf/[URLNAME].json' file.

5.) get_tf_idf(url,word):
- It returns the tfidf value of a passed word in the passed url.
- It retrieves the data stored in the 'tfidf//[URLNAME].json' file.

6.) get_page_rank(url):
- It returns the pagerank value of a passed url.
- It retrieves the data stored in the 'pageRank//[URLNAME].json' file.

Time Complexity:

- **'N'** be the number of files in the webData directory**,**
- **'L**' be the number of links accessed for crawl,
- **'U'** be the number of unique words,
- **'W'** be the number of words in each url, and
- For get_outgoing_links(linkString):
  - It has constant time complexity **O(1)** as it only has to retrieve data from a json file which stores a dictionary. Accessing items from a dictionary takes constant time always.
- For get_incoming_links(url):
  - Time complexity is **O(L)** as it has to go through every url to find if the passed parameter link is linked to or not.
- For get_idf():
  - Time complexity is **O(1)** as only has to load data from a json file and access the required value from a dictionary which takes constant time always.
- For get_tf(url, word):
  - Time complexity is **O(1)** as only has to load data from a json file and access the required value from a dictionary which takes constant time always.
- For get_tf_idf(url, word):
  - Time complexity is **O(1)** as only has to load data from a json file and access the required value from a dictionary which takes constant time always.
- For get_page_rank(url):
    - This is also O(1) as the required data is already computed and only has to be retrieved from a file.

- **Space complexity of the data files:**
  - words.json:
    - It stores a dictionary where for every unique word crawled from the links, there is one float. So, the space complexity for retrieving data from it is O(N) where N is the number of unique words.
  - 'webData/[URLNAME].json': It stores a dictionary for 1 url.
    - For each url there are 3 keys:

        - Words: a single string of ‘W’ words separated by a space
        - Links: a single string of ‘L’ links separated by a space
        - Title: a single string
        - So the space complexity for it would be O(W+L)
    - Therefore, the space complexity is O(U*(W+L))
- ‘tfidf/[URLNAME].json’:
    - For 1 url there is 1 key:
        - tfidf: a dictionary where each unique word ‘W’ has a its tfidf float value.
        - So the space complexity for it would be O(W)
- ‘tf/[URLNAME].json’:
    - For 1 url there is 1 key:
        - tf: a dictionary where each unique word ‘W’ has a its tf float value.
        - So the space complexity for it would be O(W)
- ‘pageRank/[URLNAME].json’:
    - For 1 url there is 1 key:
        - pageRank: 1 float pageRank Value.
        - So the space complexity for it would be O(1)

- FILE: search.py:

- This file is associated with the actual searching which is done by the function search(query).
    - Search(query):
        - First it generates the tfidf dict for the query input by the user.
        - Then it calculates all the necessary calculations to generate the cosine similarity for each url.
        - If boosting is set, the scores are calculated according to page rank. Otherwise it uses the cosine similarity itself. Then, the scores are sorted and only the top 10 values are selected.
        - After that, the final list of the top 10 with their url, page title and the score is returned.
    - TIME COMPLEXITY:
        - It has 2 nested loops where the first loops ‘L’ times for the total number of links crawled. The second loops for the number of words ‘W’ in the query. So the time complexity is O(L*W)
        - The last loop for calculating the result matrix loops 10 times for the top 10 scores to be created. So it is O(1) all the time.
        - The get_query_tfIdf() function has a complexity of O(W+W+U), ‘W’ for the number of words in the query and ‘U’ for the number of unique words in the query. As there are 3 separate loops.
        - calcScoreByBoost() has a time complexity of O(L) for the L number of pages crawled.
    - SPACE COMPLEXITY:
        - queryVectorDict and pagesWordsCount has a space complexity of O(U) as the space required increases for greater number of unique words U entered by the user.

        - urlList and cosine_similarity has a space complexity of O(L), L for the number of links crawled.
        - Scores also has a space complexity of O(L) as it stores the score for L number of links crawled.
        - Result has a space complexity of O(10) as it stores the results of the Top 10 links max.
        - Since we have computed all the data required for calculations, the search takes a much shorter time.

- **Constants.py:**
    - symbolsMap- it stores all the symbols that are not supported by windows file naming syntax and replaces them with unique symbols correspondingly. Since I have opted to name files with the url of the page for which the data is for, a consistency is required such that these changed url filenames are decoded into actual urls properly.
    - CONST_ALPHA & CONST_DROPTHRESHOLD- These are the constant variables used in pageRank calculations as per the specified requirements. Need to ensure that these constants are not accidentally updated while execution.
    - allDirs specifies all the directories created to store the corresponding data and needed to be cleared/initialized at the start of the crawl process.
- In this project, the latter two parts, search module and searchData are almost completely practically functional since the calculations are all accurate and the computed data retrieval can be done in the same way. However, the crawler function is incomplete as it assumes for a much simplified html page elements and structure which is the contrast in real world pages. So, it has a much limited functionality. Also, due to project restrictions, there are many python modules that haven't been used that can speed-up the speed of execution: such as using orjson instead json for file handling, using numpy modules for the matrix operations which essentially is taking the most time, etc.