**Project Analysis**

**Introduction**

The project consists of 3 primary components:

1. The crawler: 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 a JSON file. 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: 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: 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:**

Since the project has only a few files, I have placed all the code files in the same directory and only a specific directory (webData) for the computed values in json (urlPageData.json, uniquewordsIdf.json). In a large scale project, the overall directory would be divided separately for each search, searchData, crawler functions, a separate directory for helper functions and another for the test files. There also exists a readme file that expresses the basic information. Each primary component (crawl, searchData, and search) has its own helper functions file which contain a number of functions that carry out a specific task. This was done to keep the code base clean and readable, and make the 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.

**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 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 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 the file with the crawled data using the ‘generate\_tf\_tfIdf’ function.
    - 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(N)** as it goes and deletes the files present in the ‘webData’ directory one at a time.
    - 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.
      * Since there’s only one loop, this code block has a linear time complexity of **O(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.
    - So finally, putting things together, the maximum time complexity would be O(N+ (U\*L)+(L\*[W+I]))
* **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.

**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, most of these functions are very fast as they only have to retrieve data stored in files. The functions here utilize functions in the searchDataHelper.py file for certain tasks as well.
* 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/urlPageData.json’ file.
2. get\_incoming\_links(linkstring):  
   - The data is utilized from the ‘webData/urlPageData.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 ‘webData/uniqueWordsIdf.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 ‘webData/urlPageData.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 ‘webData/urlPageData.json’ file.
6. get\_page\_rank(url):  
   - It returns the pagerank value of a passed url.  
   - Many helper functions are used from the searchDataHelper.py as follows:
   1. generateUrlIndexMap():  
      - This function generates a map for every url assigning it an index. Such that it can be easily accessed in matrix calculations.
   2. generate\_probabilityTransitionMatrix(urlIndexMap, urlOutgoings):  
      - It generates and returns a link transition matrix from the given urlIndexMap.  
      - Instead of generating adjacency matrix first and only then the transition matrix, we directly calculate the transition matrix for efficiency.
   3. generate\_scaled\_adjacentMatrix(probabilityTransitionMatrix):  
      - It generates and returns the scaled adjacency matrix after scalar multiplication with the previously computed probability transition matrix.
   4. generate\_finalMatrix(scaledAdjacentMatrix):  
      - It generates and returns the final matrix after considering the random transport value and adds it to the scaled adjacent matrix.
   5. mult\_matrix():  
      - It multiplies two matrices and returns the result.
   6. euclidean\_dist():  
      - It returns the Euclidian distance of two matrices.
   * First it checks whether the passed url is present in the url list. If not, then -1 is returned.If yes, it generates an index map and the outgoing links for each url. The indexMap makes it easier to access the link from a matrix from its index value. The outgoing links are essential for further calculation. The probability transition matrix is calculated, followed by the scaled matrix, and then the final matrix after adding in the transport value. Next, a loop is used to generate a vector ‘PI’ such that the Euclidian distance between the finalVector and the previous vector is less than or equal to the constant drop threshold. Then it returns the pagerank value of the required url that is present in the previously calculated vector ‘PI’ by using its index in the urlIndexMap.

* **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):
    - get\_url\_list() takes **O(1)** constant time as it’s just retrieving data from a json file
    - generateUrlIndexMap():
      * Since it goes through every url to map its index, its time complexity is **O(L)**
    - For the first loop,
      * The time complexity is **O(L)** as it has to go through every link to find each of its outgoing links.
    - 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 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, if X>Y then the time complexity is O(X) and O(Y) if it is the inverse as it does a parallel iteration on X and Y.