COMP 1405
Summative Project - Search Engine

Hassan Ali, Kunwar Nir

I. File Structure

The general file structure of the source directory is as follows:

```
1405Z-Course-Project/
├── Idfs/
│   ├── apple.txt
│   ├── banana.txt
│   ├── coconut.txt
│   └── ...
├── pages/
│   ├── http}{{people(scs(carleton(ca{d̃avidmckenney{fruits2{N-0(html/
│   │   ├── countAll/
│   │   │   ├── apple.txt
│   │   │   ├── banana.txt
│   │   │   ├── coconut.txt
│   │   │   └── ...
│   │   ├── incominglinks.txt
│   │   ├── outgoinglinks.txt
│   │   ├── pageRank.txt
│   │   ├── title.txt
│   │   └── wordCount.txt
│   └── ...
├── testing-resources/
├── crawler.py
├── improvedqueue.py
├── matmult.py
├── search.py
├── searchdata.py
├── testingtools.py
└── webdev.py
```

The above file structure uses “fruits2” as an example.

We did not initially have this file structure as we wanted to use a JSON to hold all the information, but we quickly recognized that this would not be optimal for both space complexity (in reference to ram) and the time complexity of unpacking the JSON into a dictionary. After this realization, we decided to emulate our dictionary using folders as keys to other dictionaries and files as keys to values. For reference, the layout of our dictionary is as follows:

```
data ={
   <url>: {
    outgoinglinks : [],
    incominglinks : [],
     countAll  : {
       <word>: <occurrences within the url>
   },
     wordCount  : <total wordcount of the page>,
      title  : <title of the page>,
     pageRank  : <the page rank>
   },
   . . .
}
```

Having the layout of the dictionary mirror the file structure increased the efficiency of the code. We also wanted a place to hold the idfs of all the words that have ever been found so that we can get the inverse document frequency (idf) of a word in the query directly. That is the purpose of the "Idfs" folder. It reduces the computations needed to retrieve the idf of word later on.

II. CRAWLER.PY

A. *Planning*

As outlined by point 2 under the "Constraints and Assumptions" heading in the project specification, we decided to try and do as much of the processing as possible before search.py so that the searches themselves are fast and optimal. Although this would increase the crawl's time complexity, the priority for this project is to minimize the time and space complexity of the search as the crawl only runs once while the search may be executed multiple times. Thus, we decided to store all the data necessary for searchdata.py (outgoing links, incoming links, the occurrences of each word, the total word count, the page rank, term frequencies, tf-idf values) by URL, as well as any extra data we may need in the search (the title of the page and the inverse document frequencies).

We first started with decomposing the overall crawler process into smaller chunks that can run on their own. These smaller chunks turned into their own separate functions that could run freely without being dependent on the other small functions. This allowed us to test different parts of the crawler without necessarily knowing how another process needed to work as those functions would be abstracted by what they would return. It also helped to make the code more readable and less redundant.

Function: *`createSubString(str, start, end)`*

The createSubString function takes in a string and returns the substring in between two strings, the start and the end. For example, if it were called `createSubString("hello world", "e", "r")`, it would do the following:

```
print(createSubString("hello world", "e", "r"))
>>> "llo wo"
```

Where n is the number of characters in the string, the worst-case time complexity for the `createSubString(str, start, end)` function is $O(n)$, since it calls pythons built-in `.index()` function which has a worst case time complexity of $O(n)$. This is because, in order to find an index of a given character in a string, in the worst case the `.index()` function would have to iterate over every character in that string.

Function: *`get_idf(word, totalUrls, IDFs)`*

The `get_idf()` function returns the inverse document frequency (idf) of a given word in relation to the number of total URLS (`totalUrls`) and its presence in the dictionary `IDFs`.

The function has a time complexity of $O(1)$. This is because the function uses pythons built in `in` operator which has to check whether the word is present in the dictionary `IDFs`. It then returns the value of the idf based on the value of that key in the dictionary.

Function: *`get_tf(URL, word, data)`*

The `get_tf()` function gets the term frequency (tf) of a given word and a given URL in relation to the general word counts and specific word counts for a url, both of which are stored in the dictionary `data`.

The `get_tf` function has a time complexity of $O(1)$ since it also uses pythons built in `in` operator to check whether the word is present in the given file, since a dictionary is used this is $O(1)$. The function then returns the term frequency calculated by retrieving the frequency of that specific word from the dictionary (which has time complexity $O(1)$) and the total word count from the dictionary which has time complexity $O(1)$.

Function: *`get_tf_idf(URL, word, data, IDFs)`*

The `get_tf_idf)(` function gets the tf-idf of word.

The function calls both the `get_tf` and the `get_idf` functions, and returns the calculation of tf-idf from the values that they return as follows:

```
tfidf =log(1+ get_tf(URL, word, data), 2)* get_idf(word, len(data), IDFs)
return tfidf
```

Since both of the functions called within this function have time complexities of $O(1)$, this function also has a time complexity of $O(1)$.

Function: *`createPageRanks(data)`*

This function creates the page ranks for every url in the dictionary that is passed in. It uses the given values for alpha and the distance threshold which is 0.1 and 0.0001, respectively.

Where n is the number of links and where m is the number of convergence iterations, the time complexity for this function is $O(n^2 \times m)$. At the start of the function, we loop through all the urls to create a mapping, and then create a 2D matrix for the probability matrix. In the loop of creating the matrix, we do all the calculations for probability such as multiplying the contents by 1-alpha and adding alpha/n. This means we avoided more loops. To show what it looks like, we made both functions.

The current function with the reduced amount of loops:

```
for i in range(length):
      matrix.append([])
      for j in range(length):
         if len(ogIndexes:=data[mapping[i]]["outgoinglinks"]) ==0:
            matrix[i].append( ((1/length) *(1-ALPHA)) +(ALPHA/length))
         else:
            matrix[i].append( ((1/len(ogIndexes)) *(1-ALPHA)) +(ALPHA/length) if mapping[j] in ogIndexes
            else (ALPHA/length))
```

The function with more loops:

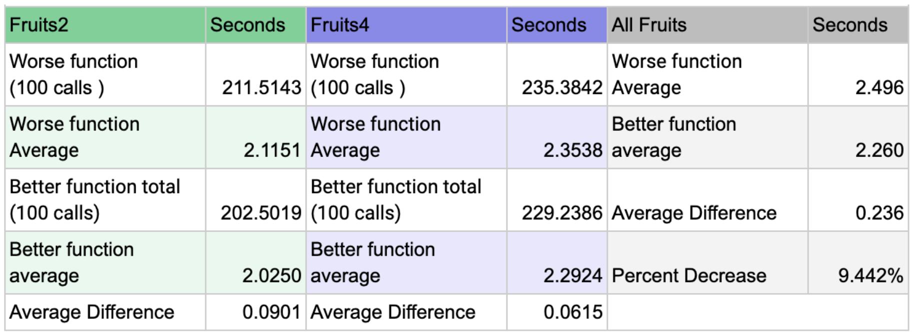
```
for i in range(length):
   matrix.append([])
   for j in range(length):
      if len(ogIndexes:=data[mapping[i]]["outgoinglinks"]) ==0:
         matrix[i].append(1/length)
      else:
         matrix[i].append(1/len(ogIndexes) if mapping[j] in ogIndexes else 0)

matrix =mult_scalar(matrix, 1-ALPHA) #Multiply matrix by 1-alpha

# Add alpha/N to each of the elements in the matrix
for i in range(length):
   for j in range(length):
```

To compare the difference in time taken, the table below was constructed:

Fruits	Seconds	Fruits3	Seconds	Fruits5	Seconds
Worse function (100 calls)	259.5618	Worse function (100 calls)	302.5242	Worse function (100 calls)	238.8903
Worse function Average	2.5956	Worse function Average	3.0252	Worse function Average	2.3889
Better function total (100 calls)	234.4153	Better function total (100 calls)	232.0048	Better function total (100 calls)	231.8909
Better function average	2.3442	Better function average	2.3200	Better function average	2.3189
Average Difference	0.2515	Average Difference	0.7052	Average Difference	0.0700



Fruits2	Seconds	Fruits4	Seconds	All Fruits	Seconds
Worse function (100 calls)	211.5143	Worse function (100 calls)	235.3842	Worse function Average	2.496
Worse function Average	2.1151	Worse function Average	2.3538	Better function average	2.260
Better function total (100 calls)	202.5019	Better function total (100 calls)	229.2386	Average Difference	0.236
Better function average	2.0250	Better function average	2.2924	Percent Decrease	9.442%
Average Difference	0.0901	Average Difference	0.0615		

Fig. 1: Comparison Between Times Taken For Both Functions

The table clearly indicates that having the function with less loops increases the run time efficiency of the function. This was an important reminder to us that, despite the fact that the main factor of the time complexity came from the convergence, it is important to try and reduce the other factors as well as they can add up to a meaningful amount of time (even 10%!). We tried to continue to make smaller things more optimal throughout the entire project after this realization.

After this point, the function uses the matrix to find the stable state. This is the main factor of the time complexity where it loops through the length of the matrix (which is the number of urls) and then loops through each column (which also adds up to the number of urls) for the dot product. This process is encased in another while loop that keeps going until the distance between the old probability vector and the new probability vector is almost non-existent. This makes the time complexity for this function is $O(n^2 \times m)$.

Function: *`dotProduct(pi, b)`*

The `dotProduct()` function return the dot product of two vectors as follows:

```
sum =0
for i in range(len(pi)):
    sum+=pi[i]*b[i]
return sum
```

Where n is the length of the first vector, the function has a time complexity of $O(n)$.

Function: *`createFiles(data, IDFs)`*

This function creates files for all the values in our data dictionary so that they can be accessed later by `searchdata.py` with a small time complexity.

Where n represents the number of URLs in the dictionary and where m represents the number of unique words in those urls, the function has a time complexity of $O(n \times m)$.

This is because the function iterates through each url and inside that loop, there is another loop that iterates through each word in `data[url]["countAll"]`. Therefore, this function has $O(n \times m)$ time complexity.

Function: *`deleteFolder(folder)`*

This function opens a folder and recursively deletes each file and subfolder within that folder, and then deletes the main folder that was given.

Where n is the number of files and subdirectories in a directory, the worst case time complexity would be $O(n)$.

Since this function is recursive, the time complexity will be $O(n)$ as it passes over every file once to delete it. This is the lowest possible time complexity for a process like this as every file has to be passed over to be deleted.

Function: *`parse(url, data)`*

The `parse()` function uses the webdev module to read in a URL and then saves all of the necessary information, including the title, the outgoing links, the overall word count, and the individual word counts.

The `parse()` function uses the `read_url()` function from the webdev module and saves the returned string. It then splits the string into a list using pythons built in `.split()` function. The function then has one for loop which loops through every element in the list. During every iteration, it checks to see whether the element contains the file title, a link, or the words of the webpage. In each case, it saves the corresponding data in to the correct place of the dictionary, `data`. Furthermore, in the case of the title and the links, the parse function uses the `createSubString()` function in order to save only the necessary peices of the string for the title and the links. The `createSubString()` function has a worst case time complexity of $O(m)$, where m is the number of characters in the string.

Since the loop takes $O(n)$ time and the `createSubString()` function is called within the loop, the overall parse function takes $O(n \times m)$ time. The m is negligible however, since the numbers of characters in the string is much smaller than to the number of lines in the webpage, n, therefore it has a worst case time complexity of $O(n)$.

Function: *`crawl(seed)`*

The `crawl(seed)` function ties everything from `crawler.py` all together. It calls all of the other functions in the file, either directly or indirectly through `parse()` and `createFiles()`.

The function takes in an initial url, the seed, and then searches through every link referenced in the seed and its links. This continues until the tree of links is exhausted. The function then returns the number of links it has crawled through.

The function uses a `improvedqueue` to keep track of all the urls. This is done for two reasons; firstly, for efficiency (the lookup is $O(1)$); and secondly, in order to not search the same url twice.

Where n is the number of files, the time complexity for this function is $O(n^2 + nm)$. This is because there is a while loop which continues until all of the files are searched, resulting in n time. The `parse()` function is called within the while loop; it has a time complexity of $O(n)$, thereby increasing the overall time complexity to $O(n^2)$. After the while loop has finished, the `createFiles()` function is called. The `createFiles()` function has a time complexity of $O(nm)$, where m is the number of unique words in all of the urls. Since the `createFiles()` function is called after the while loop and is adjadacent to it, the overall time complexity increases to $O(n^2 + nm)$. Therefore, the time complexity for the `crawl()` function is $O(n^2 + nm)$.

All of the helper functions in crawler.py are used, directly or indirectly, in `crawl()`. For this reason, in reference to space complexity, we'll only be talking about the `crawl()` function as every other function is called and returned in the `crawl()` function at the end. The space complexity for `crawl()` is $O(n^3)$, where n is the number of urls. The variable taking up all this space is the "data" dictionary which holds all the urls, their content, and their unique words. This means that as n increases, the base dictionary holds n dictionaries (which is already n^2), and each of those n dictionaries holds two lists (outgoinglinks and incominglinks) which increase as n increases. This makes the space complexity $O(n^3)$. We decided to do this because although we are using up a lot of memory, the trade-off is that all the processing is extremely fast due to O(1) lookups. This is the reason that our crawler is efficient. Almost all of the other functions in `crawler.py` require 'data' as a parameter, and so this dictionary gets used a lot. We believe that the trade-off is worth it as it makes our searches very fast, and our crawls fairly efficient. We also considered using files from the very start instead of only creating the files at the end, but we realized that the creation of files and constantly reading and writing the processed data to them made the crawler too slow.

III. SEARCHDATA.PY

*A. **Planning***

Since search.py will use searchdata, we planned to make as many of the searchdata functions $O(1)$ by making these functions access the files created by the crawler. Almost all of these functions are exactly the same where they take the url, find the value in the correct file, and then return that value.

Function: *`readFile(path)`*

This is a helper function that opens up a file and returns the contents. We created this to avoid redundancy in our code.

Where n is the number of lines in the file, the worst case time complexity would be $O(n)$. This is because this function returns all the lines in the file as a list.

Functions: *`get_outgoing_links(URL) and get_incoming_links(URL)`*

These functions open the outgoinglinks.txt and incominglinks.txt files, respectively, to return the outgoing and incoming links of that file.

Where n is the number of URLs, the functions have a worst case time complexity of $O(n)$.

The files which are being read each include a list of urls, and so when there are more URLs, there are bound to be more incoming and outgoing links. Therefore the time complexity is $O(n)$.

Functions: *`get_page_rank(URL)`*

This function opens the pageRank file within the directory of the corresponding URL and returns the value in the file.

The time complexity of this function is $O(1)$. This is due to the fact that the function only has to ever open one file and that file always has only one value in the file regardless of the number of links or files and so the time complexity is $O(1)$.

Functions: *`get_title(URL)`*

This is a helper function for search.py and it opens the title file of the corresponding URL and returns the value in the file

The time complexity of this function is $O(1)$. The function only needs to open file and that file always has only one value in the file regardless of the number of links or files and so the time complexity is $O(1)$.

Functions: *`get_idf(word)`*

This function opens the text file of the specific word within the IDFs folder to return the idf value which is stored in that file.

The time complexity of this function is $O(1)$. The function only searches one file and that file always has only one value in the file regardless of the number of links or files and so the time complexity is $O(1)$.

Functions: *`get_tf(URL,word) and get_tf_idf(URL, word)`*

These functions open the text file of the specified word to return the tf and idf value.

The time complexity of this function is $O(1)$. The function only searches one file and although the file has a list, it will always be a list of length 3 regardless of the number of links or files and so the time complexity is $O(1)$.

IV. SEARCH.PY

A. *Planning*

As referenced in the design of crawler.py, our goal is to make the search's time complexity as small as possible. We accomplished this by making most of the searchdata.py functions $O(1)$. We thought it would be simple to make the search function take very little time, but after implementing the cosine similarities we realized that it will always be $O(n^2)$ at least due to the fact that we have to loop cosine similarities for every url. We still tried to make the runtime as low as possible, and used helper functions to decompose different parts of the search. We knew that at some point we would have to sort the URLs into a "top 10" list. Instead of sorting the urls after making the large list, we planned to insert the url in the correct position every single time we found a URL to avoid more time complexity.

Function: `search(phrase, boost)`

Where n is the number of urls and where m is the number of unique words in the phrase, the worst case time complexity of the function is $O(n^2 + nm)$.

At first we split the phrase and call the `phraseVector()` function. This automatically creates a `phraseVector()` and also returns a dictionary of the unique words in the phrase and their occurrences. The next part is the cosine similarity calculation and the sorting. We decided to do both at the same time. First we loop through every url in the network, and then loop through every unique word in the phrase to get their document vector (which is $O(nm)$). Then we process its cosine similarity using the `cosineSim()` function (which is $O(n)$). Instead of having to sort through a list of a thousand URLs, we decide to create a list of 11 urls and each new url compares itself to those 11. This means that each url only compares itself to other urls a set amount of times regardless of the amount of urls, making the process $O(1)$. At the end of comparing through the 11 urls, the new url will either insert itself in the right place, or be discarded. At the end of this for loop of all the urls, we pop out the last url as the 11th one is a buffer URL and for testing. We then format the cosine similarity list and the urlSort list into a list of dictionaries so that it fits the tester.

If there are many unique words, the nm term in the time complexity can be an important factor. The average case will usually have a low amount of unique words, and so the average case would be $O(n^2)$.

All of the helper functions in `search.py` and the functions in `searchdata.py` are used in `search()`. For this reason, in reference to space complexity, we'll only be talking about the search function as every other function is called and returned in the search function. The space complexity for `search()` is $O(n)$. The largest variable that increases as something else increases is 'files' which increases linearly as the number of urls in the network increase. There are some other variables that also increase as the number of words in the query increases, but this is likely to be far smaller than the number of files. Every other variable that looks large or is nested (like the 'results' dictionary or the cosineSimilarities list) is $O(1)$ as they stay constant under the assumption that we are only looking for the top 10 urls. All the searchdata functions (aside from the outgoing and incoming links functions) are $O(1)$ space complexity as they are constant size regardless of the number of urls. The `cosineSim()` and `euclidean_norm()` functions both return O(1) values, and `getPhraseVector()` returns 2 $O(n)$ values. This means that the space complexity of the main function (search) is $O(n)$.

Function: `cosineSim(a, b)`

We return the dot product of both vectors divided by both of the euclidean norms of the vectors multiplied by each other.

Where n is the length of the vectors (a and b), the worst case time complexity is $O(n)$. This is because the `dotProduct()` and the `euclidean_norm()` functions are both $O(n)$, and so the `cosineSim()` complexity is $O(n)$.

Function: `euclidean_norm(a)`

We loop through the vector and sum the elements of those vectors squared. We then square root the sum.

Where n is the length of the vector given, the worst case time complexity is $O(n)$. This is because the function only loops once and there are no nested loops, so it is $O(n)$.

Function: `getPhraseVector(phraseWords)`

We first get rid of any duplicate words in the phrase and then find the IDFs of each of those words. After that, we find the term frequency of those words in the query and calculate the tf-idf of each of those words, generating the query vector.

Where n is the number of words in the query, the worst case time complexity is $O(n^2)$. This is due to the fact that the process of getting rid of any duplicate words is $O(n^2)$, and so the total time complexity is $O(n^2)$.

V. Functionality List

- Implemented tf, idf, and tf-idf calculations
- Implemented incoming and outgoing links
- Implemented pageRank
- Implemented search
- Handles html attributes
- Handles both absolute and relative urls
- Handles different page names
- Handles multiple paragraphs