COMP 1405Z - Course Project

Search Engine

Benjamin Le - 101036759

Professor David McKenney

October 23rd, 2022

List of Functionality

Within the searchdata.py file

- get_outgoing_links: retrieves the outgoing links from the provided link, while disregarding links that don't exist in the crawl or at all. Worst case scenario is O(n) runtime complexity since the last link in the list of directories requires iterating over the contents of the entire list until it finds the directory of the provided link.
- get_incoming _links: retrieves all the links of pages that refer to the provided link, again while disregarding links that don't exist in the crawl or at all. Worst and even best case scenario is O(n^2) runtime complexity since this requires the entire contents of the list to be iterated over to check if the link exists in the outgoing links test file, which also needs to be iterated over as well.
- get_page_rank: retrieves the contents of the page rank text file for the provided link. Worst case is O(n) runtime complexity since the list of directories needs to be iterated over until the directory of the provided link is found.
- get_idf: Counts number of documents that the given word shows up and finds the inverse document frequency (idf) value of the word. Worst case scenario is O(n^2) due to the need to use os.listdir to find all the directory names and then iterating over that list of directories to access the files within.
- get_tf: Goes and reads a single text file with the same name as the entered word of a given url. Since it's just using os.path.exists to find the exact file, the time complexity of this is O(1). Possibility that it could be O(n^2) due to the way that the title finding function is written, as the split method for strings needs to both find the specific string and then actually perform the split operation.
- get_tf_idf: Retrieves the idf and tf values and does the requisite calculation to combine them for the tfidf value. Similar to get_tf, this is probably running in O(n^2) due to the use of the built in split method in Python.

Within the crawler.py file

- This crawler file has only one function, which is the crawl. This crawl does a combination of reading and recording page information (page contents, title, references) as well as doing the calculations for term frequency, inverse document frequency and page rank ahead of time to avoid placing the burden on the search function, allowing it to just read the text files containing the relevant information and find the values it needs. Due to the complexity of how the parsing is done, the worst case runtime seems to be O(n^3) as there are several nested for loops within the crawl function as well as the parse_function module used.
- The main reason I believe it to be running in O(n^3) time is during the initial HTML parsing for the title, page contents and references. The entire process for these 3 pieces of info are done in a for loop, which has a for loop within and uses a function from the parse_functions module that uses a for loop to read each word of the document to check if that word has been recorded before or not.

File Structure

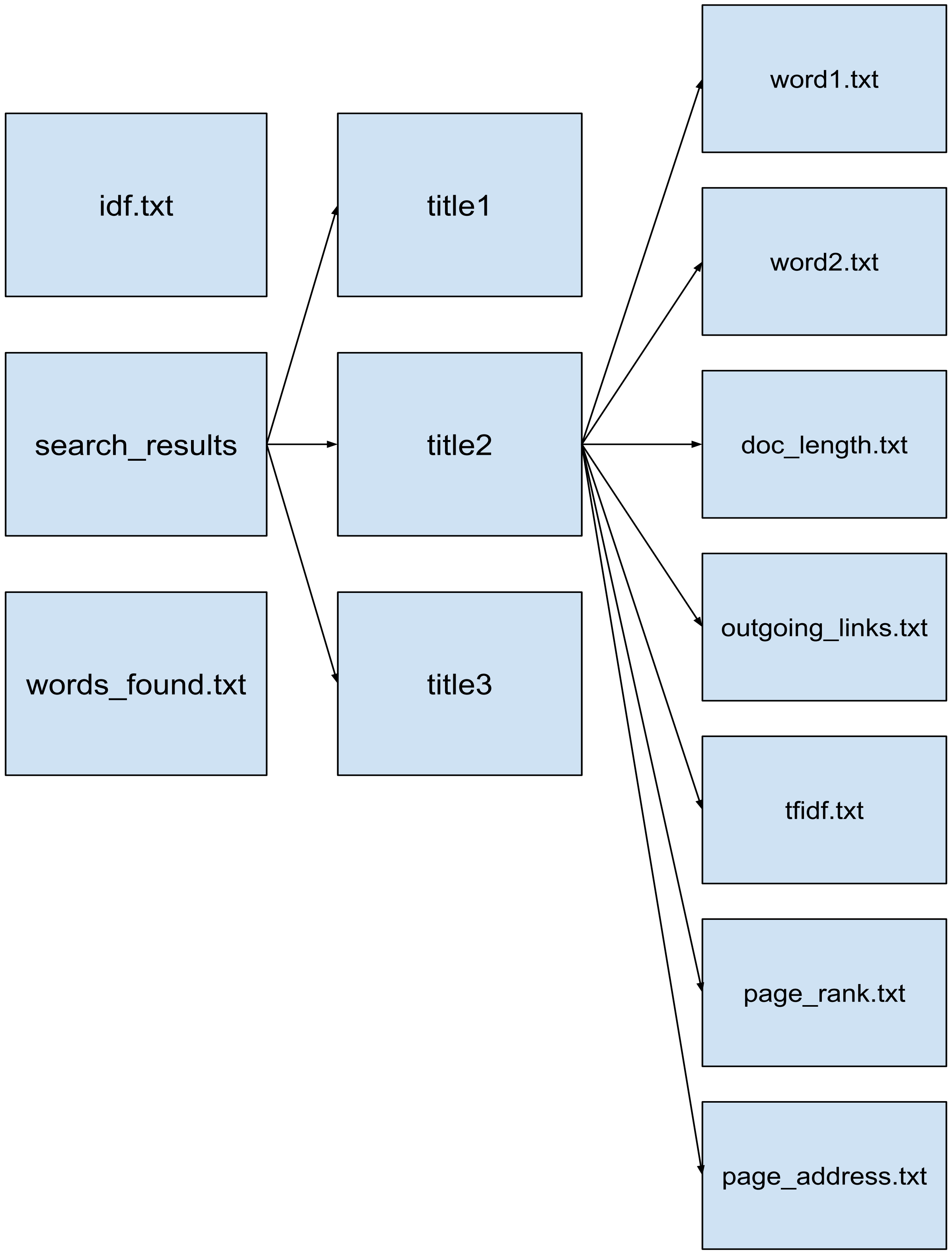

Figure 1: Basic representation of the file structure for the search algorithm

The figure on the previous page shows a flow chart style figure of how the file structure was made for the search algorithm that I made. To separate the files from the actual Python files, all of the data except for idf.txt and words_found.txt were put into a directory called search_results. Within the search_results directory are individual folders with the title of the webpage as the directory name. This was what I found to be the easiest way to separate all the pages, as they all had individual titles and as such were easily referenced. Within these directories were text files that each held a specific set of information. There are text files for each individual word in the page contents, which house the term frequencies for that specific URL. The other files are more descriptive, as their names directly tell what the contents inside are. The outgoing_links.txt file has a list of links that are referenced in the html of the directory's URL. The page_address.txt file is just the URL of the directory. The page_rank.txt holds the page rank value that was calculated in the crawl, and is there to just be read in the event that the boost value in the search is True. The final file is the tfidf.txt file, which has the tfidf scores of the URL for each unique word found in the crawl. These are ordered in the same way that the words_found.txt file is ordered, so the indexes can be related through either using the .index() method in python or through a dictionary.

In an effort to try and reduce the search function's complexity, I chose to load most of the calculations for term frequencies, inverse document frequencies and page rank values into the crawl so that the search could be done in O(n) time. This was further reinforced by storing individual values into single text files and separating data into the simplest forms possible to avoid having to iterate over an entire text file across a possible 1000 URL's. An issue that was made apparent in early versions of the crawl algorithm was that the creation and writing of thousands of text files can take a long period of time, as each URL had at minimum 6 text files to create. If the page contents has multiple unique words, this number can rise up to hundreds in the worst case. One way around this would be to use JSON and while I considered it in the initial draft of the project, I ultimately decided against it as the lack of familiarity with JSON coupled with my limited experience in programming would most likely lead to integration problems.

Design Choices and Future Additions

Crawler.py

For the crawler, there were a number of decisions that I made for the sake of having a crawl that finished in a reasonable timeframe. The first choice was using a queue based design to get the crawl started and generating all the text files. The one difference in the queue is that there is no removing of elements from the queue, which would normally be done with the .pop() method. The reason I chose to not remove elements from the queue was due to a timing issue in the execution. Originally I had made the queue before iterating over the entire queue, which would lead to large amounts of time wasted generating the queue. Instead, I elected to generate the queue as I was parsing the URL's by using the references from the current page to append onto the queue. This reduced the crawl time by about a minute and a half, which was a great improvement.

Another deliberate choice I made in the crawler was to not use the parse_functions module I had created to make and append the tfidf text files for each directory. This was due to the amount of overhead required to go into a different module and do all the work in there to then output back into the crawler module. By instead using the os module's path.join() method and the open() method for text files, I greatly reduced the runtime complexity of the tfidf portion of the crawler. The original version using the parse_functions module took around 15 minutes to complete that portion, while the current version that omits those functions takes about 4 seconds. This was the largest improvement to the efficiency of the crawl, as the average time for a 1000 page crawl came out to roughly 5 minutes.

In general, the crawl function I made wrote as much information to files to avoid consuming large amounts of RAM during operation and used as few function calls as it could get away with to reduce the overall time taken in an effort to get a reasonable crawl time. In a future version of this crawl, I believe using a binary search to check if an element is already in a list would make a few portions of the algorithm run faster as opposed to using the in method for lists. Another addition would be to completely integrate the parse_functions module into the crawl, or not even make them functions in the first place. This would reduce time needed to call them and use them, which would reduce the overall parse section of the crawl.

Searchdata.py

In the searchdata functions I wrote them in a way to just read the relevant text file to get the information needed. As such these are just simple "find the directory, find the file, return file contents" functions. This was to reduce the amount of iteration needed to a minimum, resulting in mostly O(n) runtimes. The exceptions for that are the get_idf and get_tf_idf functions, which I believe run in O(n^2). This is due to the nested for loop needed to get the inverse document frequency value, as it needs to iterate over the entire directory list to check the total number of documents a word shows up in. In the end I left it as is since the total time taken wasn't actually that long and redoing the entire thing wouldn't have saved a noticeable amount of time. If it came down to a point where the idf and tfidf function were the bottleneck, one addition that could be made would be to move the calculation of the idf to the crawl instead of the calculation being handled by a function in searchdata.py.

File Structure

As mentioned in the file structure section, the reason I chose to structure my files as such was to ease my understanding of the whole search as well as to break the pieces of data down into single bits of information that can easily be searched up. Using JSON would be favorable in a version 2.0 of sorts to help reduce the sheer number of files created, as less files created should help reduce runtime and require less operations as a whole to get to the same result. I used the titles as directory names as opposed to ID's so that I could sort them easier, since they are automatically sorted by Windows in alphanumeric order anyways. Other than changing to JSON the file structure would be pretty similar, as individually storing bits of information is a faster method of accessing than storing a master list of information per URL and having to nest loops within loops.

Conclusion

Overall I believe that the search algorithm was a success in getting to the end goal, as it completed the crawl in a reasonable time and the search was done quickly as well. There are small optimizations to be done to tighten up the overall algorithm, as mentioned in the design choices section and further work needs to be done on the crawl to make it not run in O(n^3) time.