COMP 1405 Z

Course Project: Search Engine - Analysis

Alvina Han (Kayeon), Omar Shah

Crawler.py

This module is responsible for performing crawling and extracting all significant data from a web-page. It then processes that data and saves it as a JSON file, which will be used later in the program. This module is also responsible for managing the creation and removal of folders.

Function: `crawl(seed)`

Using a seed (url) this function navigates through all connecting urls by extracting them from the page's html. Furthermore, by also extracting the page's words, the crawl is able to process information and find values such as the tf, twf and incoming links. That information is stored into JSON files to be sent to "searchdata.py" which, although increases the space complexity, decreases the time complexity which my partner and I prioritized in this search engine project. Processing the data once in `crawl(seed)` as opposed to processing each time a function in "searchdata.py" is called would be more time efficient. The space complexity of this function would be O(n) and the time complexity would be O(n^3) due to the nested loops.

File Organization

Function: `manageFolder()`

The purpose of this function is to create a folder, "crawl" in which all url data extracted from the web crawl will be stored. If the function detects that a crawl folder already exists, presumably from a previous crawl, it deletes each file in the crawl folder to leave it empty for the next crawl to run. This function also creates six JSON files to hold data that is going to be passed into different modules. These fields are titled with the prefix "0_" to ensure that they remain at the top of the crawl folder and are quickly accessible. The time complexity of running this function would be O(n) for deleting all the previous files, and O(1) for creating the folder and each JSON file. This function would have an O(1) space complexity.

Function: `createFiles(currentFile)`

The `createFiles(currentFile)` is a helper function which takes in the name of a url, creates an empty text file and returns its file path. This function operates at a space and time complexity of O(1)

Highlights

- The six JSON files could be merged into a single file but that was decided against because six files improves code readability and organization
- Storing the words from the crawler into a file was decided against due to the processing of tf and twf during the crawl which rendered the word-storage files unnecessary

Searchdata.py

Most of the data has been collected in the "crawler.py", reducing the runtime and space complexity in this module. Other than the main function module, `get_page_rank(URL)`, the other functions focus on retrieving the data from the JSON files made in "crawler.py" and returning their respective outputs.

List of functionality that is complete:

`get_outgoing_links(URL), get_incoming_links(URL), get_page_rank(URL), get_idf(word), get_tf(URL, word), get_tf_idf(URL, word)`

Function: `get_outgoing_links(URL)`

This function returns a list of URLs that the given URL links to. All the outgoing links of a specific URL have been saved to a text file with a name of a unique number_(title of the page). Retrieving the name of the text file from the 0_dict.json file has a runtime complexity of O(1) and searching for the file with all the outgoing links runs at O(n).

Runtime complexity: O(n) Space complexity: O(n)

Function: `get_incoming_links(URL)`

This function returns a list of URLs for pages that link to the page of the given URL. All the incoming links of a specific URL have been collected during crawl and added to a JSON file. 0_dict.json file is used to retrieve only the unique number of the given URL; .split() method is used for the retrieval. Using the unique number in the 0_incoming.json file, the function returns a list of all incoming links for the given URL.

Runtime complexity: O(1) Space complexity: O(n)

Function: `get_idf(word)`

This function returns the inverse document frequency of the given word of the pages that were crawled. During the crawling process, the number of documents of all the words that appear in have been calculated and were saved to a 0_twf.json file. This function accesses the twf.json file and calculates the idf of the given word.

Runtime complexity: O(1) Space complexity: O(1)

Function: `get_tf(URL,word)`

This function returns the term frequency of the given word within the page of the given URL. The tf value of all words in every URL have been calculated and saved to the 0_tf.json file. 0_dict.json file is used to retrieve only the unique number of the given URL; .split() method is used for the retrieval. Using the unique number and the given word, the function returns the calculated value.

Runtime complexity: O(1) Space complexity: O(1)

Function: `get_tf_idf(URL, word)`

This function returns the tf-idf weight for the given word of the given URL. It calculates the idf and tf values using the get_idf(word) and get_tf(url, word) functions. Then it calculates and returns the tf-idf weight.

Runtime complexity: O(1) Space complexity: O(1)

Function: `get_page_rank(URL)`

This function returns the page rank value of the given URL. It uses a nested list called adjacencyMat where it stores the values right up to the power iteration. During the power iteration, it calls matmult.py and calculates the Euclidean Distance until it is below 0.0001. After the page rank of all urls have been calculated, it dumps the list into a 0_pageRank.json file (empty file created during crawl). After the first time page rank function is called, it will use the 0_pageRank.json file to return the page rank values.

Runtime complexity: $O(n^2)$ Space complexity: $O(n^2)$

Highlights

- 0_pageRank.json file is filled once `get_page_rank(url)` is called for the first time. When the function is called again, it will not go through the calculation rather return the json file, greatly reducing the time complexity
- Most calculations are done in one line to reduce space complexity

Search.py

The search module returns the top ten search results relating to the phrase that was given. It uses functions from searchdata.py and the code is separated into two parts to help readability and optimize organization; part 1: vector space model and part 2: cosine similarity. We tried our best to minimize the code for time and space complexity.

This module was created using two different methods, each differing in the process used to obtain the top ten search results. A complexity analysis was performed on each method to determine which solution was more optimal and would be implemented into the final project.

Part 1: Vector Space Model

This part finds unique words in the parameter, phrase and keeps count of duplicates. For the document vector, it uses a nested list to store the tf-idf weight of all unique words of every page found during the crawl process. The tf-idf weights are found using the `get_tf_idf(url, word)` function from searchdata.py. The time complexity and space complexity for the document vector are O(n*m) with n being the total number of pages and m being the total of unique words.

For the query vector, it is a one-dimensional list that stores tf-idf weights of all unique words in the phrase that was given. It uses the `get_idf(word)` function from searchdata.py for idf value and tf-idf is calculated manually. The time complexity and space complexity for the document vector are O(m) with m being the total of unique words.

Part 2: Cosine Similarity

This part has a temporary dictionary called cosDict, which contains the url, title, and cosine similarity value that was calculated inside a loop that will loop for the number of documents. Following this code, two different methods were made differing in the process used to obtain the top ten search results.

Method One - "Value Mapping"

In this method the cosine similarity values for every document was stored in a list which was sorted using the built-in, list.sort(), method. The built-in method was used because it utilizes Tim Sort, which is a combination of merge and insertion sort running at O(nlogn) time, making it optimal for a list that can be potentially very large. Using a dictionary, each cosine similarity value is mapped to a list of urls that have it: {cosineVal: [url1, url2] }. After the list of values has been sorted, the top ten values are retrieved and using the mappings, the corresponding urls and titles are obtained as well. This information is then formatted into a dictionary and returned by the `search(phrase, boost)` function.

Method Two - “Storing only Top 10”

This method only stores top ten cosine similarity values in a list. Everytime a cosine similarity value is calculated, it checks its value with the tenth(smallest) cosine similarity value and if it is bigger, the tenth dictionary in the list is removed and the new dictionary, cosDict is added. The new dictionary is added by going through every value in the list to be placed at the right index. The modified sort method has a time complexity of O(n) but since it is being sorted everytime a new dictionary is created it has a time complexity of O(n*10) because the list will not pass the size of 10. This method prioritized space complexity as it will have a list of size 10 with each index having a dictionary of three values.

Which Method and Why?

We finalized on Method Two because of its superior space complexity. Throughout the project, priority was given to minimizing runtime complexity over space complexity, but both methods had similar output in runtime. This result shifted the focus to space-complexity and method two was more optimal in that aspect. Method One made use of a list (with a potential to become very large) and a dictionary which both took up space, whereas Method Two made use of a list (max size of 10) which took up less space overall. Due to these distinctions, the final version of the project implements Method Two due to its superior space complexity.

The worst time complexity is $O(n^2)$ and the space complexity is O(1).

Highlights

- Instead of using the built-in sort() method as everything is sorted except for the current dictionary in the top ten list, we modified the bubble sort algorithm to find the right place for the current dictionary, cosDict. This method is used to optimize the runtime
    - Sorting the whole list is done only once, which is when the list hits the size of 10 for the first time. Other times the method above is used
- The list that will be returned will never exceed a size of ten, meaning that it will only store the top ten search results. This method was chosen for space complexity

- Usage of the integer, topNumber:
    - To rid magic numbers within the code
    - To change how much of the top ranked results print out
        - ex: if one wanted to output the top 25 search results, change topNumber = 25
- The calculation of left denominator for cosine similarity was done outside of any loop as it will be the same for all calculations - reduce time complexity