Isabella Nguyen
COMP 1405Z Fall 2021

COMP 1405Z Course Project Analysis Report

When implementing a web crawler and search engine, it is crucial to pay attention to the runtime and space complexity of your program. Using dictionaries, different sorting algorithms, and reusing code are just a few examples of ways you can reduce runtime and/or space complexity.

Web Crawler Analysis (crawler.py)

The web crawler module is responsible for retrieving data from the web, processing that data, and storing the required information into files for further use. The web crawler works by extracting the HTML information from the seed URL using webdev.py and string manipulation to find the necessary information from the HTML string. This information is then used to calculate the necessary data (page rank, outgoing links, incoming links, tf, idf, tf-idf) to be stored in files.

Improved queue

To extract data from each URL, a queue was used to keep track of all the URLs that still needed to be parsed while ensuring that no URL was revisited. To do this, the program would need to check if the URL found was already in the queue or if a directory for that URL has already been made. The issue with this is searching in the queue to find the URL is O(n) time. For larger queues, this will take a toll on the runtime complexity. Therefore, a dictionary was added to keep track of all the URLs in the queue. This way, searching for a URL in the queue will only take O(1) time, increasing the efficiency of our queue.

Dealing with files and directories

To keep organized and quickly fetch the required data later on in the project, saving the information in an organized manner was key. A directory was made for the idf values, and a directory for each URL containing the tf, tf-idf, page rank, incoming and outgoing information all in separate files so that each file only had one number or the correct set of links, depending on what data the file contained. Although this increases the space required for the crawl, it was an understandable sacrifice to decrease runtime complexity for later, as iterating through one file to find the correct data is much less efficient and harder to implement.

Regarding the names of the directories, the names came from the relative link instead of the title in the case of duplicate titles, as all the links must be unique. “url” was added before each directory name to easily distinguish the directories from other files in the folder. This is useful when determining which files/directories to delete at the start of the crawl. Unfortunately, concatenating “url” every time the directory name was needed may increase runtime complexity as concatenating strings is $O(n^2)$ time. Therefore, I tried minimizing this as much as possible by storing the directory name into a variable to use that variable wherever it was needed, instead of doing “url”+X each time.

Searching for a key in a dictionary is much easier and quicker than searching for something in a file. This is why a dictionary (`dirNameDic`) was made, containing the absolute link as the key and the directory name as the value. `dirNameDic` was stored in a file so that the other modules (searchdata.py and search.py) could easily determine which directory belonged to which absolute link in O(1) time.

Extracting links from a URL

At first, the basic code for extracting the outgoing links from a certain URL looked like this:

```
startLink=contents.find("<a href=\".")+len("<a href=\".") #Find the first link
links=contents[startLink:].split("<a href=\".") #make a list of all the links
```

This code finds the end index of the first `"<a href=\"."` in the html string, then makes a list of the rest of the code by splitting wherever `"<a href=\"."` shows up again. Later, string manipulation would be done on each value to extract the relative url. The problem with this implementation is that it makes many different assumptions. For example, this assumes that "`href=`" comes right after "`<a`". It also assumes that there will be no "`href=`" outside of the "`<a>`" tags. To make a more generalized code that doesn't assume too many things, the implementation was changed to:

```
links=[]
start=0
startLink=contents.find("href=\".")+len("href=\".")
while start!=-1:
    endLink=contents.find("\"",startLink)
    links.append(contents[startLink:endLink])
    start=contents.find("href=\".",endLink)
    startLink=start+len("href=\".")
```

This new implementation only assumes that the outgoing links come after "`href=`" and is within quotation marks (an assumption directly made in the project specifications). The code works by finding the end index of the first `"href=\"."` without the "`<a`". This fixes the assumption that every "`href=\"."` comes after a "`<a`". Then, while there are still "`href=\"."` to be found, find the quotation mark that follows the "`href=\".`". These indexes will give us the relative URL to append to a list. We continue this pattern by finding the next start and end indexes until there are no more "`href=\"."` tags left to be found, indicated when the `.find()` function returns -1.

Reusing code

To minimize code, some functions were used twice for different purposes. For example, the `getFreq(dic, words)` function was used for determining the frequency of each word in a link and keeping track of how many documents a word appears in. The function `saveWordInfo(dic, dir_name)` was also reused for saving data in a directory from a dictionary into separate files with the title (keys) and the content (values). This was used for saving tf and idf values.

Saving calculation results in files

At first, all the calculations were implemented in the searchdata.py module. But calculating everything each time someone entered a new query was unnecessary and would increase runtime complexity when the runtime complexity of the search module matters the most. So everything was moved to the crawler where all the information needed would be stored into files for further use.

Page Rank algorithm

To remember which row in the adjacency matrix corresponds with which URL, a dictionary (`URLNum`) was made that stores the URL as the key and the corresponding number as the value. This way, finding the URL's number can be done in O(1) time, as opposed to, for example, storing all the URLs in a list and using the indexes of each URL as the corresponding number which would take O(n) time.

```
Ex. URLNum={URL7:0, URL3:1, URL4:2, URL8:3, …}
```

To efficiently search for the correct set of outgoing links, a dictionary (`allOutgoing`) was made to store a dictionary of outgoing links for each URL. The sub dictionary also decreases runtime complexity when determining whether a URL links to a certain URL or not. This is especially useful for URLs with a large number of outgoing links, as a linear search to determine if a link is present in a list would be O(n) time, while searching in a dictionary is O(1) time.

```
Ex. allOutgoing={URL1:{out1:1, out2:1, …}, URL2:{out1:1, out2:1, …},
…}
```

The steps in the page rank algorithm show filling the adjacency matrix with 1's and 0's, and then dividing by the total number of outgoing links. These steps were combined into 1 to reduce the code.

```
Ex. if link in outgoing:
      adjMatrix[URLNum[page]].append(1/numOutgoing)
```

This example is adding 1 divided by the total number of outgoing links to the correct row in the matrix instead of simply adding 1 and then later on making another for loop to revise the matrix. `numOutgoing` comes from the length of the sub dictionary containing all the outgoing links from `allOutgoing`.

Similar to the example above, instead of replacing all the 0s to 1/N if the whole row is 0, 1/N was added for every value in the row right away if the sub dictionary (in `allOutgoing`) corresponding to that row was empty. This minimizes the code and computation needed to produce the adjacency matrix.

At first, for the step where alpha/N is added to each number in the adjacency matrix, this was implemented by creating a new matrix and then adding the new value into that matrix.

```
Ex. for row in range(len(adjMatrix)):
      newMatrix.append([])
      for num in adjMatrix[row]:
            newMatrix[row].append(num+(alpha/N))
```

However, this implementation will unnecessarily take up more space than necessary as the program is creating a whole new matrix which may contain a large amount of values. Creating this large extra matrix may also increase runtime complexity with very large matrices. Therefore,

this step needed to be improved by simply changing the values within the original matrix (`adjMatrix`), decreasing space required while also decreasing runtime complexity.

```
Ex. for row in range(len(adjMatrix)):
      for num in range(len(adjMatrix[row])):
            adjMatrix[row][num]+=alpha/N
```

The page rank algorithm produces a vector that contains the page ranks for all the URLs. I made sure to produce that vector only once and then extract each page rank and save it to files instead of doing all that computation to produce the same vector each time the page rank for a URL was needed. That would be a huge amount of extra computation, negatively affecting the runtime complexity of the web crawler.

Search Data Analysis (searchdata.py)

The search data module is responsible for extracting the necessary data from files stored by the web crawler to calculate the resulting score in the search module.

Reusing code

`get_outgoing_links(URL)` and `get_incoming_links(URL)` do the exact same thing except for different files. Therefore, the function `getInfoList(URL, filename)` was made as a general function used in both the outgoing and incoming functions. `getInfoList(URL, filename)` accepts a URL to tell it what directory to search in, and a filename to tell it what file to read into a list.

Similarly, `get_page_rank(URL)`, `get_idf(word)`, `get_tf(URL, word)`, and `get_tf_idf(URL, word)` all return a number from a file within a directory. `getInfoNum(dir_name, filename)` is generalized code that accepts a directory name and a filename to tell the function what file to read. The function will then return a float.

These generalized functions greatly minimize code, make the code easier to follow and more efficient.

Initializing `dirNames`

As explained in the web crawler analysis, the dictionary `dirNameDic` contained the URLs and corresponding directory names that were stored in a file. In this module, searchdata.py, that dictionary is necessary to extract the correct information. But, since it will never be changed, the dictionary, `dirNames` in this case, only needs to be initialized once by extracting the dictionary from the file. The initializing occurs in a function called `getDirNames()`. But since this is a module, the functions will be called on from another python file. Therefore, it is unknown which function will be called first. So, `initializeDirNames()` was made to check if `dirNames` had already been initialized. If it was not empty, then the program will not reinitialize the dictionary for no reason. This improves runtime complexity as checking one if statement is much quicker than extracting a dictionary (that may contain 1000 keys) from a file each time a function is called when it is not necessary to do so.

Search Analysis (search.py)

The search module is responsible for receiving a query and outputting the top ten search results according to the cosine similarity algorithm and the data stored in files by the crawler.

Recursion for determining cosine similarity

To determine the numerator and denominator for the cosine similarity of a link, two functions using recursion were used. One for the numerator (`get_numerator(vectorA, vectorB, index)`), and the other for calculating the euclidean norm (`get_euclidean_norm(vector, index)`). Although recursion tends to take up more space than normal iterations like a for loop due to stacking, it minimizes code and makes the code easier to write.

Keeping track of which cosine similarity score corresponds with which URL

The dictionary `cosSimURL` contains the cosine similarity score as the key and a list of all the URLs with that score as the value.

> `Ex. cosSimURL={0.94383748:[http://x1.html, http://x2.html], 0.38494:[http://x3.html],...}`

This way, after sorting the list of cosine similarities, the corresponding absolute links will still be known. In addition, it only takes O(1) time to find the top 10 links from the dictionary. The process of extracting the links from the lists is done by a method similar to a queue. The first item is taken and removed from the list to add to the dictionary of the top 10 search results. Therefore there is no searching through lists to find scores or links, greatly reducing runtime complexity; especially with a large amount of links.

Quicksort

To efficiently sort the list containing all the cosine similarity scores, the quicksort algorithm was used. Quicksort is able to sort a list in O(n log n) time, which is very efficient, especially for sorting very large lists. This algorithm was chosen instead of merge sort in consideration of the space complexity, as merge sort requires the program to create a large amount of lists, while quicksort sorts the list at hand. Quicksort is also known to be generally faster than merge sort due to the same reason. With all the dictionaries and files used in this project, it seemed like a good idea to pay attention to the space complexity while also decreasing the runtime complexity. Therefore, quicksort was the best algorithm to use for sorting the large list of scores.

This algorithm was slightly modified in this project to sort the list from highest to lowest scores instead of lowest to highest.

Checking if boost was true or false

Looking at the code, it seems as if it would be much easier to check if boost was true in the for loop that calculates the cosine similarity. But, that would mean checking if boost was true for every link. That is unnecessary as boost is never changing. Therefore, after all the cosine

similarities are calculated, the program checks if boost is true only once. This minimizes the amount of computation necessary for the boost feature.

```
Ex. pseudocode for checking for boost inside the for loop
for url in links:
      cosSimList.append(cosineSimilarity)
      if boost:
            Multiply the value at index in cosSimList by its page rank
```

This requires checking the value of boost for each link shown in purple.

```
Ex. pseudocode for checking for boost outside the for loop
for url in links:
      cosSimList.append(cosineSimilarity)
if boost:
      for url in links:
            Multiply the value at index in cosSimList by its page rank
```

This method requires checking if boost is true only once throughout the whole search, as opposed to potentially 1000 in the previous pseudocode (making this O(n)). This seemingly small difference can greatly reduce the amount of computation needed to calculate the final scores to therefore reduce the runtime complexity of this specific issue to O(1).