1405Z Course Project - Analysis

Note: Before we start, we just wanted to let you know that this analysis was written jointly by my partner and I. Our analysis will be extremely similar so you don't have to read them both if you don't want to. In our implementations, there are some slight differences but the core ideas are the same. Function names have been changed to match our respective code.

`crawler.py`

Global Variables

When it comes to efficiency, it's difficult to balance both runtime and space. In some cases, one must be prioritized over the other. For this project, we decided to prioritize runtime efficiency over space in certain instances. When dealing with a search engine, no one has the patience to wait for their search results. Since we expect results almost instantaneously, we put the most effort into minimizing the runtime complexity whenever we could (even if it was at the expense of space). This can be seen in some of our global variables, such as `numToURL` (the list) and `urlToNum` (the dictionary). Although these dictionaries and lists are big in size and take up a lot of memory, they allowed us to create a search program that has a big-O notation of O(n) (where n is the pages) instead of $O(n^2)$. This will be discussed in further detail in the `search.py` section below. Another global variable we used was the `wordsAppear` dictionary - this will also be discussed later.

`getUrlFromNum()` and `getNumFromUrl()`

We used global variables to map the numbers to a list and a dictionary at the same time in our `mapID()` function. Every time we would map a URL to a number, it would be reflected in both our list and dictionary as the same number. The list is ordered so the index of a value in a list would be the ID number (eg. `numToURL = [url0, url1, url2]`). In the dictionary, we set the keys to be the URL and the values to be the ID number (eg. `urlToNum = {url0 : 0, url1: 1, url2: 2}`).

When we wanted to retrieve the number from a URL, we would call the value associated with the URL key (eg. `urlToNum[url]`) from our dictionary. This process is O(1).

When we want to retrieve the URL from a number, we would call the value at that index (the number) in the list (eg. `numToUrl[num]`) and that would retrieve the URL associated with that number. This process is also O(1).

We originally implemented a file that listed all the ID numbers and accessed it through the code in *Figure 1*. However, we realized that every time we wanted to get the ID or URL, the worst case would be to look through every line in this file which would be O(n). Although we could have stored the ID and URLs in a file to minimize space complexity (no global variables that take up memory), we chose to improve the time complexity instead.

```
def getNumFromUrl(url):
     idMap = open('id-map.txt', 'r')
     for line in idMap:
          line = line.strip()
          index = line.find(url)
     if index != -1:
          idFromUrl = line[:index - 1]
          idMap.close()
          return idFromUrl
     return -1
```

Figure 1. Old code that was mentioned above

`savetf()` and `saveIdf()`

When starting the project, we saved three pieces of information in files - the page title, its links (incoming and outgoing) and the page's contents. However, we realized we would have to go through all the pages' contents after the crawl was complete to calculate tf and idf. Instead of crawling and then revisiting the page contents to perform calculations, we decided to do as much of these calculations during the crawl.

Since each word on each page has its own term frequency value, we calculate each word's tf value after every page. We do this by calling the `savetf()` function after we store the current page's contents in a list. At the same time, we update the information needed (how many pages a word appears in) to calculate idf in our `wordsAppear` global dictionary. This way, when the crawl is done, this dictionary has all the information we need to calculate each word's idf value before storing it in a file. By doing it this way, we don't need to store each page's contents in a file or go back and evaluate the data after the crawl is complete.

`searchdata.py`

`get_tf_idf(URL, word)`

Originally, we considered storing all the tf-idf values in a file. However, we realized that if we just calculated it in `searchdata.py` using the `get_tf()` and `get_idf()`

functions, the entire process would be O(1) anyway. In `get_tf()`, we're retrieving the number from the URL (which, as discussed earlier, was O(1)). All the other logic in this function is considered O(1) as well (ie. no loops and no functions that are called that contain loops). Since we generalized the functions for `get_tf()` and `get_idf()`, they're both O(1) (they open a file, retrieve information from the file, close it and don't have any loops). In `get_tf_idf()` we're just doing a simple calculation which is also O(1). Because of this, it doesn't really matter if we stored them in files or if we just calculated it on the spot. We decided not to store the values in files to avoid taking up more room on the hard drive than the project needed to.

`search.py`

Let n represent the number of pages found during the crawl. Let m represent the number of words in the search query.

To start, when a phrase is searched for, our search program does all of the calculations that are independent of the crawl data first. This includes calculating the tf-idf value of each unique word in the query, saving these values to the query's vector and calculating the left denominator of the cosine similarity formula.

The for loop that follows iterates through each page found in the crawl (this would be O(n)). Since it contains a nested for loop iterating through the query vector, this section would technically be O(nm). However, we can assume that m is a constant since in reality, the number of words in a query usually won't exceed ~10. Therefore, when looking at the big picture, m doesn't have a significant impact on the big-O notation.

Note that there are other functions inside this for loop (ie. `crawler.getUrlFromNum()`, `searchdata.get_tf_idf()`, `searchdata.get_page_rank()`, and `searchdata.getTitle()`). As we've discussed earlier, everything from `searchdata.py` is opening a file and extracting information which is O(1). Additionally, `crawler.getUrlFromNum()`, as discussed earlier, is also O(1). Therefore the functions don't have too much of an impact on the runtime complexity.

When all the pieces are put together, it results in a runtime complexity of O(n) for `search.py`. This is ideal, since theoretically, as n increases, the search time taken increases linearly as opposed to quadtracially. This quadratic increase was the case with our original implementation of `getUrlFromNum()` and `getNumFromUrl()`.

Table 1 below outlines the big-O notation of all the functions in the three programs.

Table 1. Runtime Complexity of Functions

File Name	Function Name	Runtime Complexity	Explanation
crawler.py	enqueue	O(1)	Searching in dict is constant time
	dequeue	O(1)	Simple if statement and then using `.pop()` which is O(1)
	saveTitleText	O(n)	Where n represents the length of the HTML string where string.find() is being used (`saveToFiles()` is called but it's O(1) so it doesn't have an impact)
	analyzeParagraphText	O(nm)	Where n represents the number of <p> tags in a page and m represents the length of the HTML string where string.find() is being used (`savetf()` is called but it's a lower-order term that doesn't need to be considered)
	saveUrls	O(nm)	Where n represents the number of <a> tags in a page and m represents the length of the HTML string where string.find() is being used (`saveToFiles()` is called but it's a lower-order term that doesn't need to be considered)
	savetf	O(n)	Where n in the total number of words found on a page (The second for loop goes through the dictionary of the unique words (which has length k) where k <= n, and since the loop isn't nested, the complexity is still O(n))
	saveIdf	O(n)	Where n is the number of unique words that appear throughout all the pages
	savePageRank	$O(nm^3)$	Where n represents the number of times the *while euclidDist > 0.0001* loop runs and m represents the number of

			pages found in the crawl (Note that we made the `matmult` module for Tutorial 4 before we knew anything about runtime complexity)
	saveToFiles	O(1)	Only checks to see if a directory exists, makes a directory if not, opens a file and writes a single line → all O(1)
	deleteDir	O(n)	Where n represents the number of files in the directory being deleted. Deletes all the files in the directory before deleting the directory itself which is O(n)
	mapId	O(1)	Calls `getNumFromUrl()` which is O(1), appends URL to global list (which is O(1)) and creates a dictionary key/value pair (which is O(1))
	getNumFromUrl	O(1)	See explanation above
	getUrlFromNum	O(1)	See explanation above
	crawl	$O(nm^3 + m(op))$	Where m is the number of pages found in the crawl, and thus nm^3 is the big-O notation for `calculate_page_ranks` and op is the big-O notation for `get_page_links()/get_page_contents()`
searchdata.py	returnInfoLinks	O(n)	Let n represent the number of incoming or outgoing links there are for each page.
	returnInfo	O(1)	Open file, read value, close file, return value → all O(1)
	get_outgoing_links	O(n)	A one-liner that uses returnInfoLinks
	get_incoming_links		
	get_page_rank	O(1)	A one-liner that uses returnInfo
	get_idf		
	get_tf		

	get_tf_idf		
	getTitle	O(1)	Open file, read value, close file, return value → all O(1)
search.py	search	O(n)	See explanation above
	partDenomCosSim	O(n)	Where n represents the length of the tf idf vector