Analysis Report for COMP1405Z Course Project

Jiayu Hu

2022/10/20

README.txt

To run the crawler on a seed page:

1) Let the “seed” variable in “crawler-start.py” equals the URL of the seed page

Eg.

```
seed = "http://people.scs.carleton.ca/~davidmckenney/fruits/N-0.html"
```

2) enter this line in the command line

```
python3 crawler-start.py
```

Then the crawler.py will crawl the URLs and save the required data in the current directory.

To run any test file:

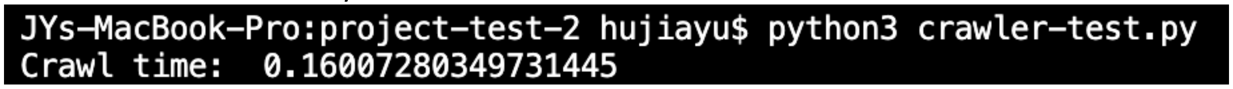
1) run the crawler on a seed page from that database, since I’ve commented out the crawler.crawl() line to prevent duplicate crawls
2) run the test file

Introduction

In this Project, I want to optimize the runtime efficiency, which has to sacrifice space efficiency. Specifically, in order to shorten the runtime for finding an item in a list/doing the calculation for page score and page rank/etc., I used plenty of dictionaries and JSON files to store the data. Thus, runtimes have been improved from O(n) to O(1), while the space complexity has been increased a lot by using a dictionary or writing in a JSON file.

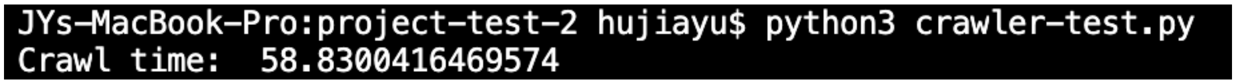
Final Results of Time Complexity

Runtime of crawler for tinyfruit database:



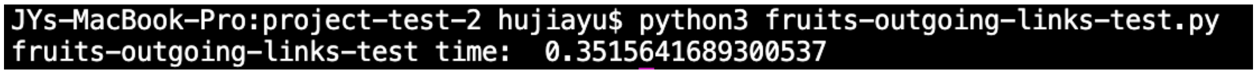
```
JYs-MacBook-Pro:project-test-2 hujiayu$ python3 crawler-test.py
Crawl time:  0.16007280349731445
```

Runtime of crawler for fruit database:



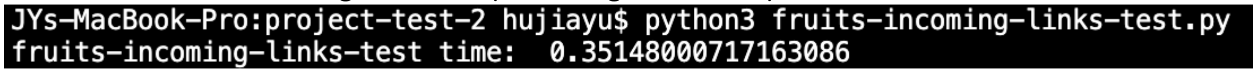
```
JYs-MacBook-Pro:project-test-2 hujiayu$ python3 crawler-test.py
Crawl time:  58.8300416469574
```

Runtime of fruits-outgoing-links-test (excluding crawl time):



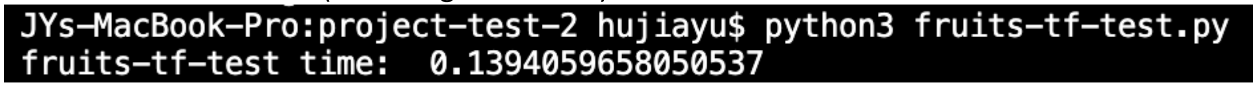
```
JYs-MacBook-Pro:project-test-2 hujiayu$ python3 fruits-outgoing-links-test.py
fruits-outgoing-links-test time:  0.3515641689300537
```

Runtime of fruits-incoming-links-test (excluding crawl time):



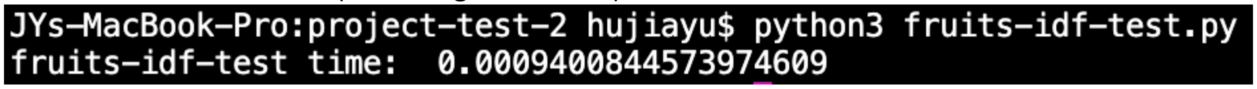
```
JYs-MacBook-Pro:project-test-2 hujiayu$ python3 fruits-incoming-links-test.py
fruits-incoming-links-test time:  0.35148000717163086
```

Runtime of fruits-tf-test (excluding crawl time):



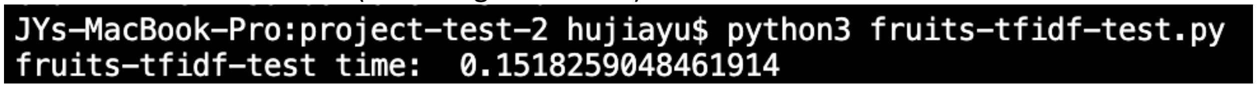
```
JYs-MacBook-Pro:project-test-2 hujiayu$ python3 fruits-tf-test.py
fruits-tf-test time:  0.1394059658050537
```

Runtime of fruits-idf-test (excluding crawl time):



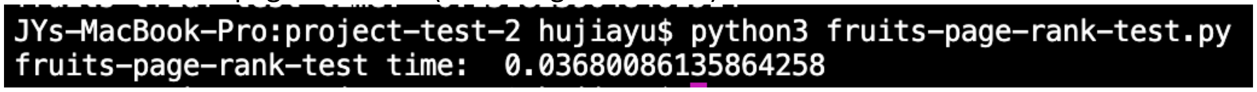
```
JYs-MacBook-Pro:project-test-2 hujiayu$ python3 fruits-idf-test.py
fruits-idf-test time:  0.0009400844573974609
```

Runtime of fruits-tfidf-test (excluding crawl time):



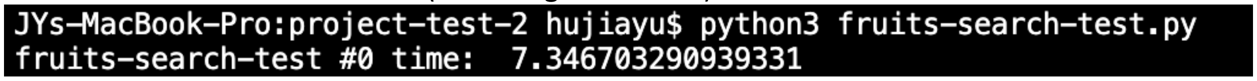
```
JYs-MacBook-Pro:project-test-2 hujiayu$ python3 fruits-tfidf-test.py
fruits-tfidf-test time:  0.1518259048461914
```

Runtime of fruits-page-rank-test (excluding crawl time):



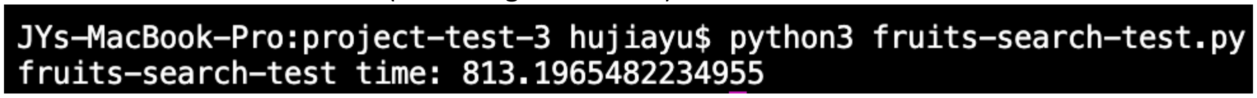
```
JYs-MacBook-Pro:project-test-2 hujiayu$ python3 fruits-page-rank-test.py
fruits-page-rank-test time:  0.03680086135864258
```

Runtime of fruits-search-test #1 (excluding crawl time):



```
JYs-MacBook-Pro:project-test-2 hujiayu$ python3 fruits-search-test.py
fruits-search-test #0 time:  7.346703290939331
```

Runtime of fruits-search-test (excluding crawl time):



```
JYs-MacBook-Pro:project-test-3 hujiayu$ python3 fruits-search-test.py
fruits-search-test time: 813.1965482234955
```

File Structure to Store the Data

JSON file name	Type in Python	File structure

crawled-pages.json	dictionary	{
		"http://people.scs.carleton.ca/~davidmckenney/fruits/N-0.html": {
		"link":
		"http://people.scs.carleton.ca/~davidmckenney/fruits/N-0.html",
		"title": "N-0",
		"words": ["banana", ……],
		"links": [

		"http://people.scs.carleton.ca/~davidmckenney/fruits/N-3.html", ……] }, …… }
crawled-links-list.json	list	["http://people.scs.carleton.ca/~davidmckenney/fruits/N-0.html", ……]
crawled-links-hash.json	dictionary	{"http://people.scs.carleton.ca/~davidmckenney/fruits/N-0.html", ……}
all-words.json	dictionary	{ "http://people.scs.carleton.ca/~davidmckenney/fruits/N-0.html": ["banana", ……], …… }
crawled-words.json	list	["banana", "apple", "coconut", "peach", "pear"]
word-idf.json	dictionary	{ "banana": 0.066427361738976, "apple": 0.06039727964395631, "coconut": 0.05889368905356862, "peach": 0.05439229681862769, "pear": 0.06039727964395631 }
word-tf.json	dictionary	{ "http://people.scs.carleton.ca/~davidmckenney/fruits/N-0.html": { "banana": 0.28735632183908044, "apple": 0.2413793103448276, "coconut": 0.11494252873563218, "peach": 0.19540229885057472, "pear": 0.16091954022988506 }, …… }
urls-if-connect.json	dictionary	{ "http://people.scs.carleton.ca/~davidmckenney/fruits/N-0.html": { "http://people.scs.carleton.ca/~davidmckenney/fruits/N-3.html": true, "http://people.scs.carleton.ca/~davidmckenney/fruits/N-39.html": true, ……} ……

		}

pageranks.json	dictionary	{
		"http://people.scs.carleton.ca/~davidmckenney/fruits/N-0.html": 0.010463140002251388,
		"http://people.scs.carleton.ca/~davidmckenney/fruits/N-3.html": 0.013385889918798107,
		……
		}

crawler.py

Main function: crawl(seed)

1. Reset any existing data

   1.1 delete all the JSON files in the current directory

```python
for filename in os.listdir(curr_dir):
    if filename.endswith("json"):
        os.remove(filename)
```

# Time complexity: O(n), depending on the file numbers in the current directory
# Space complexity: O(1) (for any liner search the space complexity is O(1), which is the same for all of the functions below, so it will not be mentioned again)

1.2 clean cache

```python
for file in os.listdir(cache_dir):
    os.remove(os.path.join(cache_dir, file))
os.rmdir(cache_dir)
```

# Time complexity: O(n), depending on the file numbers in the current directory

2. Perform web crawls starting at the seed URL

   For most of the lists, I used the `improved_queue` module from tutorial 4 to search for a specific item in a queue, which is able to insert, remove, and determine if an item is present in the queue in **O(1) time**, while also maintaining the order of item insertion

   2.1 create a list of URLs that we need to visit and a list of URLs we've already crawled

```python
queue_list = []
queue_hash = {}

already_crawled_list = []
already_crawled_hash = {}
```

# Space complexity: **twice** that of the space complexity using liner search, because we also create a same-size dictionary with every list

2.2 add the initial URL to the queue

2.3 repeat until the queue is empty

2.3.1 get the next URL from the queue

```
link = improved_queue.removestart(queue_list, queue_hash)
```

# Time complexity: **O(1)**

2.3.2 if the page has already crawled, skip it

```
if improved_queue.containshash(already_crawled_hash, link):
    continue
```

# Time complexity: **O(1)**

2.3.3 read the current page (using webdev module)

- create a dictionary to save the data of a page
- each page has a dictionary, with “link”, “title”, “words” and “links” as its keys

```
{
    "link": "http://people.scs.carleton.ca/~davidmckenney/tinyfruits/N-2.html",
    "title": "N-2",
    "words": [
        "coconut",
        "apple",
        "orange",
        "cherry",
        "orange",
        "papaya",
        "coconut"
    ],
    "links": [
        "http://people.scs.carleton.ca/~davidmckenney/tinyfruits/N-0.html"
    ]
}
```

- add those outgoing links to the “queue”
- add the current page to the "already_crawled"

2.4 repeat until the queue is empty

3. save crawled data to JSON files

JSON file	**Python object**	**Python object’s type**
crawled-pages.json	crawled_pages	dictionary
crawled-links-list.json	crawled_links_list	list

crawled-links-hash.json	crawled_links_hash	dictionary
all-words.json	all_words	dictionary
crawled-words.json	crawled_words	list
urls-if-connect.json	urls_if_connect	dictionary

# Time complexity: I used dictionary structure to save most of the data because it will save a lot of runtime for searchdata.py and search.py, and I can improve the time complexity from O(n) to O(1); I didn't use it on the crawled-words list because it is relatively small (as least in this case; maybe I still need to create a hash for crawled-words list as well when applying it to a larger database that has more specific crawled words); after that, when I need to use those data during searchdata.py and search.py, I just need to read the information from the JSON files instead of doing the crawl and calculation again. Doing this during crawling can save a lot of times and space then doing it during searching.

4. calculate pageranks of all crawled pages using helper function calculate_page_ranks (crawled_links_list, crawled_pages, int_links, urls_if_connect), save pageranks as a dictionary (object) into a JSON file
5. calculate idfs for each crawled word using helper function calculate_idfs(crawled_pages, crawled_words, all_words), save them as a dictionary (object) into a JSON file
6. calculate tfs for each crawled word using helper function calculate_tfs(crawled_pages, crawled_words, all_words), save them as a dictionary (object) into a JSON file

JSON file	**Python object**	**Python object's type**
pageranks.json	pageranks_object	dictionary
word-idf.json	word_idf_object	dictionary
word-tf.json	word_tf_object	dictionary

# Time complexity and space complexity: I saved every data to JSON files after looping through the query so that if the searchdata.py or search.py needs to use the data, they just need to read the information from the file instead of doing the crawl again, saving a lot of operating times and space

7. return the number of total pages found during the crawl

Helper functions:

find_title(str): input an HTML, output a string of title included in the <title> tag (in our case it only has one title)

# Time complexity: O(n), depending on the length of the HTML and <title> tag

find_words(str): input an HTML, output a list of words included in the <p> tag

# Time complexity: O(n), depending on the length of the HTML and <p> tags

rel_or_abs(link, seed): input a seed URL which is an absolute URL, and a URL that could be a relative or absolute URL, output a string of the link as an absolute URL

# Time complexity: O(n), depending on the length of the seed URL string

find_links(html, seed): input an HTML of a specific page and the seed page, output a list of absolute links that could be found in the specific page
# Time complexity: O(n), depending on the length of the HTML and <a> tags

get_crawled_links(crawled_pages): return **a list and a hash**(dictionary) of all URLs that have been crawled
# Time complexity: O(n), depending on the number of pages crawled
# Space complexity: **twice** that of the space complexity using liner searching, because we also create a same-size dictionary with the list

get_all_words(crawled_pages): return a dictionary including all of the words in crawled pages, key -> URL, value -> words
# Time complexity: O(1), since crawled_pages is a dictionary, and the value of each key in crawled_pages is also a dictionary

get_crawled_words(all_words): return a list including all words crawled (no duplication), save it into a JSON file
# Time complexity: O(1), since all_words is a dictionary

map_int_link(crawled_links_list): maps each URL that has been crawled to a specific int, and creates a dictionary to contain those ints and URLs
# Time complexity: O(n), depending on the length of crawled_links_list (the number of pages crawled)

get_url(integer, int_links): turn integer into the original URL
# Time complexity: O(n), depending on the number of outgoing links included in the page

get_outgoing_links(URL, crawled_links_hash): return a **dictionary** of other URLs that the page with the given URL links to; this is used for the if_connect() function
# Time complexity: O(1), since crawled_links_hash is a dictionary

get_incoming_links(URL, crawled_links_hash, crawled_pages): return a **dictionary** of URLs for pages that link to the page with the given URL; this is used for the if_connect() function
# Time complexity: O(n), depending on the number of pages crawled

if_connect(URL_1, URL_2, crawled_links_hash, crawled_pages): figure out if two links have connections
# Time complexity: O(1), since URL_1_outgoing_links and URL_1_incoming_links are two dictionaries, we don’t need to find URL in a list and spend O(n) times

get_url1_url2_connect(crawled_links_list, crawled_links_hash, crawled_pages): get the information about whether every two pages are connected

# Time complexity: O(n^2), because it has a nested loop to go through every URL with each other, and we have #n URLs
# Space complexity: O(n^2), because it is going to connect every URL to each other and save the data as a dictionary
calculate_page_ranks(URL): calculate the pageranks for all pages; do this in the crawler.py once to save time for searchdata.py and search.py

1 create an N*N matrix
# Time complexity: O(n^2), as we need to loop through all items in the N*N matrix
2 for the number at [i, j] = 1, if node i links to node j, then [i, j] = 1; otherwise = 0
2.1 figure out if two links have connections using the small function if_connect(URL_1, URL_2)
# Time complexity: O(n^2), as we need to go through all of the items in the N*N matrix
3 sort the rows into two lists: row that has no 1s / row that has 1s
# Time complexity: O(n^2), as we need to go through all of the items in the N*N matrix
4 if a row has no 1s, replace each element in that row with 1/N
# Time complexity: O(n^2), as we need to go through all of the items in the N*N matrix
5 in other rows, divide each 1 by the number of 1s in that row
# Time complexity: O(n^2), as we need to go through all of the items in the N*N matrix
6 multiply the resulting matrix by (1 - a)
7 add a/N to each entry of the resulting matrix
# Time complexity: O(n^2), as we need to go through all of the items in the N*N matrix
8 keep multiplying the matrix by a vector π (1, 0, 0, ...) until the difference in π between iterations is below a threshold
8.1 create a vector v0 = [[]]
# Time complexity: O(n), as the length of the vector is n
8.2 keep multiplying the matrix by a vector π (1, 0, 0, ...) until the difference in π between iterations is below a threshold
# Time complexity: O(n?), depending on the threshold (if it is regarded as an input as well), the keywords in different pages, the number of pages, etc.
9 return pageranks for all pages
# Time complexity: O(n), depending on the number of pages crawled

calculate_idfs(crawled_pages, crawled_words, all_words): calculate the idf for all words and save them in a JSON file; do this in the crawler.py once to save time for searchdata.py and search.py
# Time complexity: O(n*m*r), n = number of words crawled, m = number of pages crawled, r = number of words on the specific page
calculate_tfs(crawled_pages, crawled_words, all_words): calculate the tf for all words and pages; do this in the crawler.py once to save time for searchdata.py and search.py
# Time complexity: O(n*r*r), n = number of words crawled, r = number of words on the specific page

searchdata.py

Main functions:

get_title(URL)

get_outgoing_links(URL): input a URL string, output a list of other URLs that the page with the given URL links to

1. if the URL was not found during the crawling process then return None

   # Time complexity: O(1), since crawled_links_hash is a dictionary

2. read the HTML on the page using webdev module
3. find the links in the HTML using find_links(html, URL) function from crawler.py

   # Time complexity: O(n), depending on the length of the HTML and <a> tags

get_incoming_links(URL): input a URL string, output a list of URLs for pages that link to the page with the given URL

1. if the URL was not found during the crawling process then return None

   # Time complexity: O(1), since crawled_links_hash is a dictionary

2. if the URL is in the list of the outgoing links of another URL, then include that other URL in the list

   # Time complexity: O(m * n), m = number of crawled pages, n = number of links included in a page

get_page_rank(URL)

1. if the URL was not found during the crawling process then return -1

   # Time complexity: O(1), since the pageranks is a dictionary

2. match the pagerank from the pagerank.json

   # Time complexity: O(1), since the pageranks is a dictionary

get_idf(word)

1. if the word was not present in any crawled documents, this function must return 0

   # Time complexity: O(1), since the words_idfs is a dictionary

2. else, open the “word-idf.json” file and match the word with its idf

   # Time complexity: O(1), since the words_idfs is a dictionary

get_tf(URL, word)

1. if the URL was not present in any crawled documents, this function must return 0

   # Time complexity: O(1), since the crawled_links_hash is a dictionary

2. else, open the “word-tf.json” file and match the URL and the word with its tf

   # Time complexity: O(1), since the urls_words_tfs and words_tfs are dictionaries

get_tf_idf(URL, word): calcualate tf_idf for the URL and the word using get_idf(word) and get_tf(URL, word) functions

search.py

Main function: search(phrase, boost)

1. open crawled-pages.json, crawled-links.json, crawled-words.json, all-words.json, crawled-pages.json, and assign them to variables for future use
   # Space complexity: O(n), depending on the number of pages crawled/number of words on the crawled pages/etc.
2. turn the phrase user entered into a query using get_search_query(phrase)
3. get the query vector using get_que_vector(phrase, query)
4. use the query vector to measure the left_denom
5. for each document(page), measure the similarity between the document vector and the query vector
   a. get the document vector for each document using get_doc_vector(URL, query)
   b. measure the numerator using the document vector and the query vector
   c. measure the right_denom using the document vector
   d. calculate the cosine of the page
   e. save it to the result = [] list
6. if boost == True, boost the result by PageRank value; use get_page_rank(URL) to get page rank
7. sort the result from the toppest to the lowest, and pick top 10

```
result = sorted(result, key = lambda x: x["score"], reverse = True)
```

# Time complexity: O(n log n) , since O(n log n) for using the build-in sorted() function, O(n) for reversing the list, n = number of items in the list (the number of pages crawled), so the time complexity for this line is O(n log n + n) = O(n log n)

Helper functions:

get_search_query(phrase): turn the phrase (a string with a few words separated by blank spaces) into a query, excluding word that has idf <= 0
# Time complexity: O(n), n = number of words in the phrase

get_doc_vector(URL, query): input an URL of a page and the query, output the document vector of the page for the query
# Time complexity: O(n), n = number of words in the phrase

get_que_tf(phrase, query, word): for each word in the query, calculate the tf (occurrences of the word in original phrase/total num of words in original phrase)
# Time complexity: O(n), n = number of words in the phrase

get_que_vector(query): regard user’s query as a small document, calculate tf-idf for the doc with each word, add each tf-idf to the vector
# Time complexity: O(n), n = number of words in the phrase