CSCE 4013 - Information Retrieval
Howework-2 Report
Kanat Bektemirov

Objective

The objective of this project is to build an index for a collection of documents. I implemented the memory-based indexing algorithm (in Python).

Requirements

Upon typing:

```
$ tok_index.py [-h] -i INPUT_DIR -o OUTPUT_DIR [-n NUMBER_FILES]
```

the program generates (in the output directory):

1. *dict*: a file containing (term, num_docs, start_loc) tuples. None for empty slots.
2. *post*: a file containing (doc_id, tf) tuples that go with *dict*.
3. *map*: a file containing the doc_id => actual filename mapping. (doci_id == line number)

Flow

The structure of the program is as following using the memory-based indexing algorithm:

1. `tok_index.py` accepts input directory, output directory, and a number of files.
2. For each HTML file, I build a frequency counter for each token. (“local hash table”)
3. After processing each file, I update the global hash table.
    - Go through every token in the local hash table
    - Add a *(doc_id, frequency)* tuple to the list of tuples in my global hash table for that token.
4. Write the maps file.
5. Write the dict file. Each line is a entry from the array in the hash table. Write ‘None’ if the array slot is empty; otherwise, a *(token, num_docs, start_loc)* tuple.
6. Write the post file. For each non-empty slot in the array in the hash table, write a (doc_id, frequency) tuple.

Details

Since I was already doing the concept of “local hash tables” in my previous assignment, I did not have to do much for this assignment. I use Python’s built-in Counter module for this purpose. For the “global hash table”, I have a hash table that maps a token to the list of *(doc_id, frequency)* tuples. This list is what goes in the *post* file for this token, and the size of this list is *num_docs* for this token in the *dict* file. I don’t know what else to talk about here since my code itself is probably shorter than what I have described here.

Algorithm Analysis

Although the the memory-based indexing algorithm, is easy to implement and very fast compared to other algorithms, is a huge memory hog if we are working with millions of documents. Here are some comments about the program using this algorithm.

```
num docs = 1,600 files
avg_doc =  1,000 tokens
avg_len = 7 characters
uniques = 60,000 tokens
```

Memory

The program stores all unique tokens (N) in the entire document collections in memory. Below is a rough calculation of the memory the program uses:

```
collection size = num_docs * avg_doc * avg_len = 1,600 * 1,000 * 7
             = 11.2 MB

avg_doc_unique = avg_doc / 4 = 250
1 local hash table = avg_doc_unique * (7 + 4) = 0.00275 MB
# 4 == sizeof(int)
all local hash tables = num_docs * 0.00275 MB = 4.4 MB

# Average nodes per token.
avg_nodes_token = 50
global hash table = uniques * (avg_len + 50 * (4 + 4))
# The second part is the size of the (doc_id, frequency) tuple
# assuming the size of an int is 4 bytes.
global hash table = 60,000 * (7 + 50 * (4 + 4)) = 24.42 MB
```

(even though array size is 3 * uniques, it does not affect the memory much since the empty slots are set to None/null)

File Sizes

dict:
Non-empty slot: (token, num_docs, start_loc)\n
Empty-slot: 'None'\n

```
dict_file_size = (uniques * non_empty_size)
                 + (2 * uniques * empty_size)

                 = 60,000 * (1 + 7 + 1 + 1 + 8 + 1 + 1 + 12 + 1)
                    +  (2 * 60,000 * 6)
                 = 2.7 MB
```

post:
post_size: (doc_id, frequency)\n

```
post_file_size = uniques * avg_doc_unique * post_size
                 = 60,000 * 250 * (1 + 12 + 1 + 1 + 12 + 1 + 1)
                 = 435 MB
```

map:
map_size: filename\n

```
map_file_size = num_docs * map_size
                 = 1600 * (6 + 1)
                 = 0.011 MB
```

Big-O Analysis

Runtime: Inserting to the local hash table (Counter module) is O(1). Inserting to global hash table also O(1) since the chances of more than >2 collisions is very low because we allocate 3 * uniques slots. The overall runtime of the indexing part of the algorithm is O(N) where N is the number of total unique tokens in the document collection.

Statistics

1683 HTML files, the program yielded the following statistics:

```
unique_tokens: 63322, total_tokens: 1927610
global_hashtable non-empty: 63322, global_hashtable empty: 126644
all local hash table == total_tokens
$ wc -l out/map
    1682 out/map
$ wc -l out/dict
  189965 out/dict
$ wc -l out/post
  902946 out/post
```

```
$ head -3 out/map
e1.html
e10.html
e100.html
$ grep "^('the'," out/dict
('the', 1593, 286136)
$ grep "^('dogs'," out/dict
('dogs', 68, 747498)
$ grep "^('arkansas'," out/dict
('arkansas', 403, 671580)
```

8th token in dict: `('groza', 3, 77)`

```
$ sed -n 77,79p out/post
(1240, 1)
(1281, 1)
(1342, 1)
```

I checked the 3 files above and the token ‘groza’ occurs once in each one.

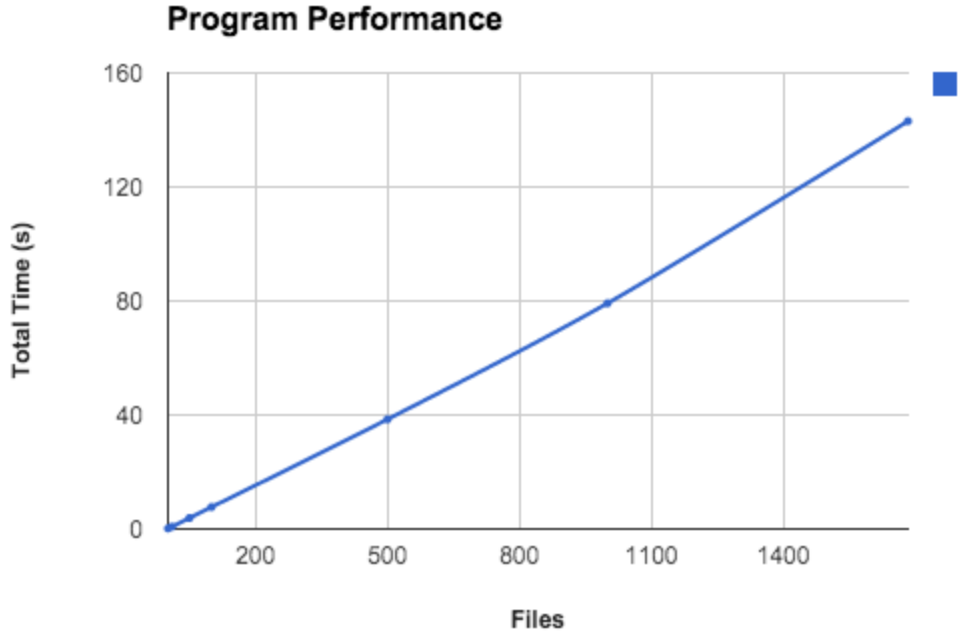
Performance

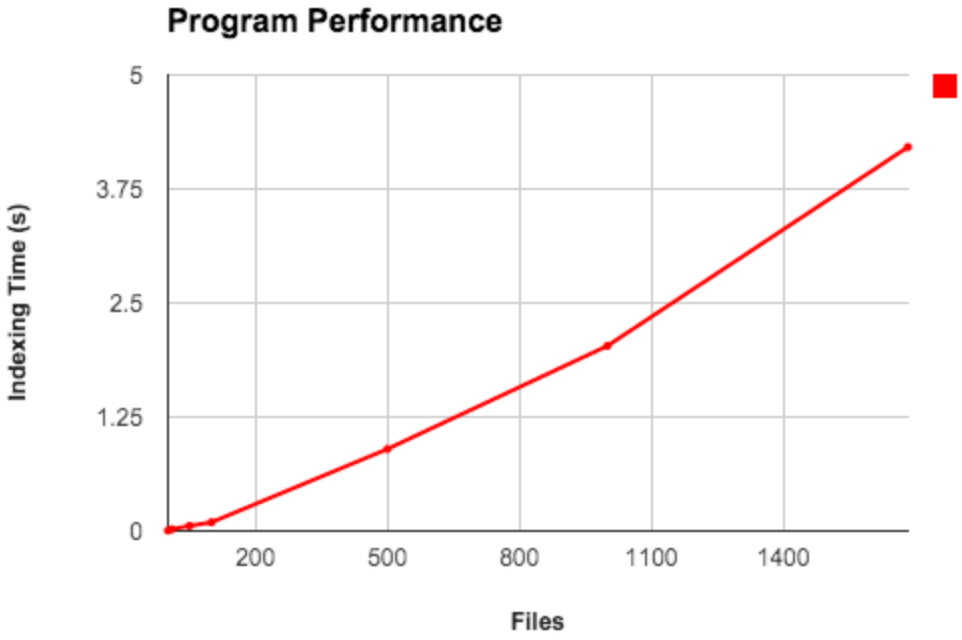
My program tokenizes the files first (same code from the previous assignment), and then does the indexing for each file. Here, I report both the total time indexing-only time.

On a machine with 8GB RAM, 2.6 GHz processor, and a solid-state drive, the program performed as following:

Files	Total Time (s)	Indexing Time (s)
1	0.18	0.005
10	0.79	0.019
50	3.84	0.056
100	7.7	0.095
500	38.5	0.899
1000	79.2	2.03
1683	143.1	3.92

Number of files vs. total runtime and indexing time in seconds.


Program Performance
160
120
80
40
0
Total Time (s)
200
500
800
1100
1400
Files


Program Performance
5
3.75
2.5
1.25
0
Indexing Time (s)
200
500
800
1100
1400
Files