Information Retrieval Project One
By Ben Sikora

Complete

Around 45 Hours Total

To be honest, there isn't much that I wish I was told at the start of the assignment. Maybe a couple of clarification things about what is going into the memory constraint or the clarification about two and three phrase processing, but there is not much besides that.

Design Document

General Structure

There are three main classes that define my Project 1 implementation: **DocumentNode**, **FileManager**, and **Lexicon**. Document Node contains most of the information found for a term in each document and acts as a "row" of the posting list. It stores the document id, the frequency of a specific term in the document, and if applicable the positions of those terms. FileManager is a class that deals with the writing/management of files to disk and ultimately merging them together. Most of my implementation, however, lies within the Lexicon class. The Lexicon class contains an ordered dictionary that acts as both my posting list and lexicon. An ordered dictionary was imperative because as terms are added/removed it is important to not change the order of the dictionary as it is acting as my lexicon. Inside the ordered dictionary, lies a key value pair of the term and a second ordered dictionary. The second ordered dictionary stores a document id and that document's node as a key value pair. Therefore, with a term and it's docID, I can efficiently find and count the appropriate Document Node. Dictionaries were imperative in the design of my project because they support search, insert, and delete in log(n) time. As I will be accessing these dictionaries for each token of the database, any delay in these processes could have exponential effects on my running time (See Figure 2 and 3 in analysis).

(Note you MUST run my code with Python 3.7 or higher because ordered dictionaries were not supported until then)

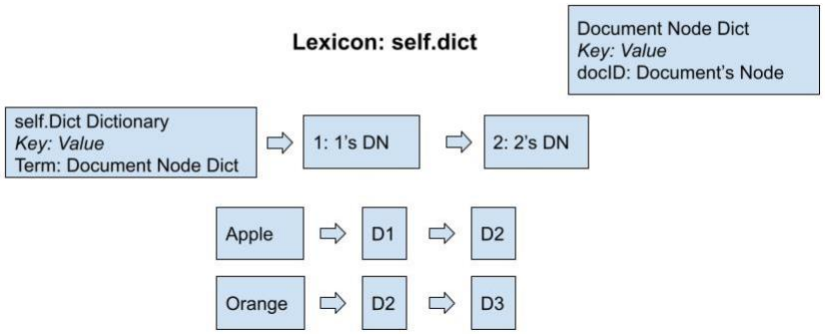

Figure 1: Outline of the nested dictionary structure.

Processing A Document: Start to Finish

The entire process for processing, tokenizing, and storing the documents lies within the "createLexiconDir" function of the Lexicon class. Much of the resetting and counting that happens between documents, also occurs in this function.

1) *Pre-Processing*

Lines are read from the files, one by one, using the "tagProcessor" function. "TagProcessor" will not output commented lines, xml tags, or empty lines. When tag-Processor comes to the end of a document it will output "</DOC>", which allows "createLexiconDir" to handle accordingly.

2) Tokenizing

Once a line has been read, it is passed to the "indexer" function to be counted. "Indexer" breaks the line by spaces and tokenizes the following words using "wordProcessing" and "countIndex." "WordProcessing" accounts for all special tokens and the case that words need to be separated by punctation. Once words have been processed, they are handed to "countIndex" -- the final part of the tokenizing process—and an index is added to the appropriate node in the nested dictionary and counted.

There are various switches across "indexer" that account for all the parsing methods. Since we are also mostly parsing by space, special handling was needed to account for line breaks in phrases and dates (the only special token with a space).

3) Creating and Merging Files

As documents are being read, "countIndex" is tallying what is being added to memory. If that tally reaches the memory constraint, at the end of the document the dictionary is written to a file using FileManager and the docNode dictionary is cleared. After reading all files, the "merge" function in FileManager will merge the temporary files together and output the final lexicon and posting list.

Special Token Design Decisions (All follow the requirements of the project. Just illustrating clarification for specific things.)

1) Monetary Value
    a. Monetary **symbols** are not stored but their values are. For example, $1,000 and 1,000 are both stored as 1000. This was clarified for me in office hours.
2) Dates
    a. If a date is given in two numbers from 00 -21 it is counted as 2000's. Otherwise, it is counted as 1900
    b. If a date is given an invalid number or symbol, it is parsed as individual terms. For example, if it was single term indexing and the text was September 45 2001, the lexicon would be September, 45, 2001. (This same method was applied to tokens, who did not meet the special token requirements).
    c. For date ranges (i.e 12-14), I stored the values individually. If a query was given it could still find the chosen document.
3) URL
    a. Since urls do not have a standard formatting, I only used common substrings to mark them (http, https, .com, .net, .edu, etc). If a url does not contain one of these common substrings, it cannot be detected in my implantation. In the future, I may implement an url "checker" to see if an url leads to a valid website.
4) Acronyms
    a. There was a question on how to distinguish acronyms with no periods in the discussion board (USA, DC, etc). I considered marking words in all caps as acronyms, but there was an abundance of normal text that was capitalized. I then made the decision to treat USA as a normal token in my implementation, as it will still be stored the same as U.S.A. I mention this because it has ramifications for my phrase indexing.

Indexing Design Decisions (All follow the requirements of the project. Just illustrating clarification for specific things.)

1) Single Positional
    a. Special tokens have been included in my single positional index. If there are multiple indexes for a special token, there positions are all the same. For example, in "part-of speech", part, speech, and partofspeech will all have the same position. It is important to mark them all as the same position because each of these indexes carry the same importance to the term before and after them.
    b. Positions are also marked by subsequent tokens. Therefore, if there were multiple line breaks or punctuation between tokens it would not change the position. This was chosen because I did not want the amount of space between tokens, to be differing across documents. (Benefiting some documents down the line but not others).
2) TwoPhrase and ThreePhrase
    a. My phrase parsings are separate and must be clarified in the command line.
    b. As I am clearing phrases that are not at a certain frequency, I wasn't able to clear them from the lexicon without ruining the order of tokens. This became a problem during merging. As such, I had to keep the lexicon with tokens that did not meet the specific phrase frequency. *Note when I give the number of phrases in lexicon, I only included those that appeared in the posting list.*
    c. While single positional contains special tokens, twoPhrase and ThreePhrase do not.
    d. Separate thresholds were chosen for my phrases. To be counted in my two-phrase indexer, a phrase in a document had to appear at least 6 times. To be counted in my three-phrase indexer, a phrase in a document had to appear at least 3 times. See Table 8 and 9 in my results section for my decision-making process.

Results And Analysis

Note in the timetables, I always rounded to one decimal place. The total time is inherently a sum of the top rows, but the rounding may change that slightly.

Table 1: Summary of Lexicons and Posting Lists for All Indexers

	Lexicon (# of terms)	Index Size Bytes	Max df	Min df	Mean df	Median df
Single Term	35671	5,329,000	1311	1	10.2	2
Single Term Positional	36121	13,126,000	1742	1	13.1	2
Stem	27679	4,581,000	1311	1	11.6	1
Two Phrase	1604	2,334,000	28	1	1.45	1
Three Phrase	1938	1,837,000	22	1	1.35	1

Analysis of Table 1

Overall, my results align strongly with what was expected. While there is no way to judge exactly how well my single indexer is working from the table alone, the rest of the indexers follow what was expected given the single term results.

Single Term Positional

The lexicon number of the single term positional, is only slightly higher than the single because the only difference was the addition of the stop list (There difference is 450 and the size of the stop list is 571). The index size of the single positional is also substantially larger than the single indexer, because it is storing both positions and doc frequencies (including the more frequent stop terms!). You can get an idea of how many more documents are being stored by looking at the increase in max df and mean df.

Stem

The lexicon number of the stem is much less than the single indexer and shows an increase in mean df. This aligns with the purpose of stemming, which is to decrease the lexicon size and combine multiple terms into one. I was slightly surprised that that the max df did not increase, but that must mean there is no difference in stemming for the index with the highest frequency.

Two and Three Phrase

The lexicon number, the index size, max df, and mean df of the two and three phrases are all substantially lower than the stem, single positional, and single indexers. The lower numbers are all reflecting that each phrase has only a couple of documents it could be in. This was chosen intentionally as I wanted phrases to be coherent, distinguishable, and incredibly specific for their document. For more information about this decision, go to Tables 8 and 9.

Table 2: Single Indexing Memory Constraints Time's

Triple List Size	1000	10000	100000	Unlimited Memory
Create Temp Files	35.1	30.9	29.9	29.2

Merge Files (Includes time to generate Lexicon)	9.4	4.7	1.8	0.1
Total Time	44.6	35.7	31.7	29.3

Table 4: Single Positional Indexing Memory Constraints Time's

Triple List Size	1000	10000	100000	Unlimited Memory
Create Temp Files	46.2	44.09	40.2	40.3
Merge Files	12.8	8.5	3.1	0.0
Total Time	59.0	52.6	43.2	40.3

Table 5: Stem Indexing Memory Constraints Time's

Triple List Size	1000	10000	100000	Unlimited Memory
Create Temp Files	57.7	52.3	50.4	47.8
Merge Files	7.7	3.7	1.6	0.1
Total Time	65.4	56.0	52.0	47.9

Table 6: Two Phrase Indexing Memory Constraints Time's

Triple List Size	1000	10000	100000	Unlimited Memory
Create Temp Files	32.9	29.6	28.1	27.2
Merge Files	0.2	0.2	0.1	0.1
Total Time	33.1	29.8	28.2	27.2

Table 7: Three Phrase Indexing Memory Constraints Time's

Triple List Size	1000	10000	100000	Unlimited Memory
Create Temp Files	28.6	28.6	28.4	29.2
Merge Files	0.1	0.1	0.1	0.1
Total Time	28.7	28.6	28.4	29.2

Analysis of Table 2-7

All the indexers follow a similar pattern of being fastest for unlimited memory and slower as the memory constraint becomes smaller. This aligns for what was expected because as the memory constraint decreases, there are more temporary documents to be written and more merging of such files. This is also reflected in the decreasing merge times as memory increases. These differences, however, are almost non-existent in the two and three phrase indexers. I believe this

is because, since the posting list is very small for these indexers, very few files are ever written to disk and merged. When I looked specifically for this at runtime, I was able to confirm this.

One thing that was also surprising was the substantial increase in running time for the stem indexing from the single indexer, despite it having a smaller index size and only a slightly larger lexicon length. The only thing that could contribute to this increase in running time, would be the stemming algorithm from the nltk library. Currently one drawback of stemming is the larger running time, but if a faster algorithm could be found that drawback could be eliminated.

Table 8: Lexicon and Posting List Summaries over various Phrase Frequency Thresholds Two Phrase

Phrase Threshold	3	4	5	6	7
Lexicon Number	6802	3726	2397	1604	1161
Max df	83	60	38	28	24
Min df	1	1	1	1	1
Mean df	1.68	1.6	1.5	1.45	1.37
Median df	1	1	1	1	1
Number of Docs in PL	1606	1301	1006	776	575

Table 9: Lexicon and Posting List Summaries over various Phrase Frequency Thresholds Three Phrase

Phrase Threshold	2	3	4	5
Lexicon Number	5893	1938	976	562
Max df	35	22	20	17
Min df	1	1	1	1
Mean df	1.35	1.35	1.28	1.22
Median df	1	1	1	1
Number of Docs in PL	1466	850	485	288

Analysis of Table 8-9: Picking a Phrase Threshold

The results in Table 8 and 9 were used to pick the phrase frequency thresholds for my two and three phrase indexers. My goal in picking a phrase frequency was to limit nonsense phrases and ensure that phrases are specific for their tailored document. As such, the three numbers I wanted to focus on were the Lexicon Number, mean df, and numbers of Docs. If lexicon number was dramatically decreasing across thresholds, I believe that nonsense phrases are mostly likely the ones being eliminated. And as my mean df is decreasing, my phrases are becoming more and more specific to the documents they were found in. At the same time, however, my number of docs illustrates the spread my posting list has over the database. There are a total of 1768

documents in the TREC database and if my posting list only covers a small fraction of the total documents, it won't offer much assistance.

For these reasons, in the two-phrase indexer, I picked a threshold of 6. At this point the lexicon number has dramatically slowed at the rate its decreasing, the mean df has decreased to 1.45, and the documents covered in the index is still at about 44% (776/1768). Any lower threshold substantially increases the lexicon length, adding in more nonsense phrases, and any higher threshold will heavily limit the amount of spread the phrases have over the documents.

For the three-phrase indexer, I picked a threshold of 3. Here the lexicon number has not slowed exactly to the point I would like (there is still almost a 1,000 drop off with the next threshold) but any higher threshold will only cover about 27% (485/1768), which I believe is far too low of spread for a posting list.

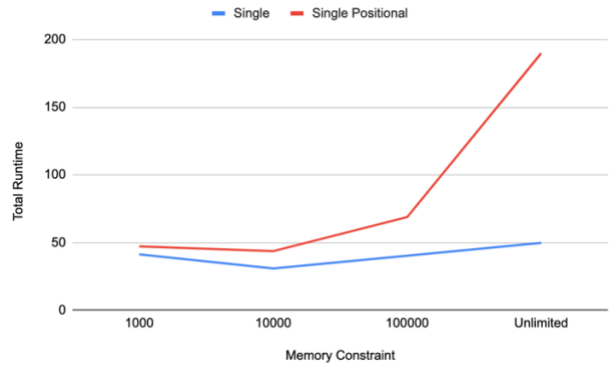

Figure 2(Left): Total runtime over various Memory Constraints and Indexers, while the Lexicon class is using a nested list in self.dict.

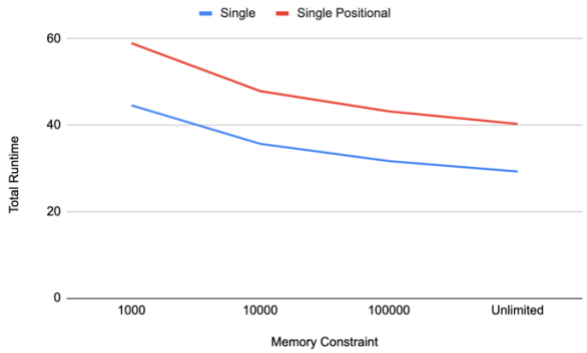

Figure 3(Right): Total runtime over various Memory Constraints and Indexers, while the Lexicon class is using a nested dict in self.dict. This was the default structure chosen for the project.

Analysis of Figure 2-3: Why a Nested Dictionary Was Used

My reason for choosing a nested dictionary structure in my Lexicon class is because of the results shown in Figure 2 and 3. When I first was creating my Lexicon class, I chose to use a nested list in my self.dict instead of a nested dictionary. I noticed, however, that my running times were exponentially increasing with more memory being added, especially with my single positional index. The reason for this was because counting a token in my dictionary always required iterating through the entire document node list. And as memory increased and the lists became longer, my runtimes began to shoot through the roof. It was from the results in Figure 2, that I decided to move to a nested dictionary. Instead of iterating through the list, I could use the dictionary lookup function and tally the nodes in a much faster lookup time. After my implementation, my runtimes became much more consistent and always decreased as memory became more and more abundant. It was because of these results that I felt justified in moving to a nested dictionary approach.

Ph.D., Ph.D, Phd, PhD U.S.A, USA B.S., BS $1,200 and 1200 F-16 CDC-50 15-D 59-DCD pre-processing post-test part-of-speech	for-black-tie-event 02/14/2000 02-14-2000 Febuary 14 2000 Feb-14-2000 02/14/00 02-14-00 Febuary 14 9999999 1,000.00 1,000 1000.00	1000.01 hello.pdf hello.html hello.txt hello.csv 192.0.2.1 192.0.2.1.5 bssikora@gmail.com bss72@georgetown.edu https://georgetown.instructure.com/courses/132765/files/folder/Projects? terms$broken^using*other/puncation	

Figure 4: A special case list that was used to test my single index. (The Lexicon was outputted in Figure 5).

phd, 0 usa, 1 bs, 2 1200, 3 f16, 4 cdc50, 5 cdc, 6 15d, 7 59dcd, 8 dcd, 9 processing, 10	test, 11 part, 12 speech, 13 partofspeech, 14 black, 15 tie, 16 event, 17 forblacktieevent, 18 2/14/2000, 19 febuary, 20 14, 21	9999999, 22 1000, 23 1000.01, 24 hellopdf, 25 hellohtml, 26 hellotxt, 27 hellocsv, 28 192.0.2.1, 29 192, 30 0, 31 2, 32	1, 33 5, 34 bssikora@gmail.com, 35 bss72@georgetown.edu, 36 https://georgetown.instructure.com/courses/132765/files/folder/projects?, 37 terms, 38 broken, 39 puncation, 40

Figure 4: The lexicon outputted in my implementation from the special case list made in Figure 4.

Analysis of Figure 3-4: Measuring of Single Indexer

While there were other pieces of evidence that confirmed my single positional, stem, and phrases were working properly, there was not much for the single indexer. Looking through my lexicon and posting lists outputs alone, I did not think was sufficient because it still could be misreading tokens. Since special phrases appear the most often in single indexing and could be a source of much error, I thought it would be best to ensure my special tokens were working properly. From the results of both figures, I was able to confirm that my special tokens were working as intended and could not be a major source of error for my single indexer. Although more testing is required to ensure I am capturing all 35671 tokens perfectly, it is reassuring to know that the special tokens are working in the ways I specifically wanted them for.

In Conclusion

Overall, this project has helped me to understand the importance and diversity of various indexers. I feel like I understand much better about the specific uses of indexers, and the type of posting lists they will generate. I also now have a better understanding of special tokens, and the importance of making sure they are stored correctly.

Although I feel confident in my results and my analysis of them, I still feel unsure about the correctness of my entire implementation. It is hard to know exactly what is working and what is not without comparing my lexicon and posting lists to other implementations of the same database. If I were to add another part of my analysis, I would possibly download an external single indexer algorithm that could provide some sort of comparison. Although I may ultimately prefer my own implementation, another perspective could surely help in debugging any errors.