Patrick Trinkle
CMSC676
Spring 2009
Project 3 Write-Up

Background

Once an Information Retrieval system has processed all the input files, it needs to store the term weights and in which documents each term appears. The records in both files should be of fixed length to provide for easy parsing as well as seeking into the file quickly to a known offset. The dictionary file is comprised of a list of records, each includes the token, the document frequency, and the starting index in the postings file. The postings files is also a list of records, each includes the document ID (in this case, a string representing the filename) and the term weight for that term in that document. The dictionary file is sorted by term alphabetically to provide for easy parsing and possible binary search. Depending on the size of the corpus this structure can be hierarchical or otherwise broken into pieces.

Implementation

Initially the program maintained a list of hash lists for each document, and when it generated the postings and dictionary files, it walked through the list checked each hash list for the term. Now, there is also a hash list keyed on term that points to a list of the documents (by document id string "x.html" => "x") in which the term appears. This term document matrix format is much faster. The program still maintains a list of hash lists for each document, because this contains the frequency of each term in that particular document. Whereas the term document matrix is for quick lookup.

The implementation is nearly the same as in the previous version, save for a few optimizations; including the in-memory term document matrix. The file preprocessing has also not changed for this version. However, if feedback from the query results in a future version indicate a deficiency in tokenizing certain terms, it will be necessary to reexamine the tokenizer for improvement. In this version of the implementation there is no temporary file creation, however it would be fairly straightforward to output information periodically to save on memory with a larger corpus. The previous version created temporary files as a stub for supporting larger data sets, but this was removed. Temporary files in future revisions will have a different form, therefore it keeping the old code was not useful.

To ease in the checking for correctness two things were done. Firstly, an initial smaller 4 document corpus was used with a matching spreadsheet. Secondly, each record in the postings file also included the term.

Interestingly of the approximately 313,000 tokens in the unprocessed documents, only around 41,000 remained. The initial count was 1 instance of a token per document, therefore if a token appeared seven times in a document it was only counted once, however if it appeared in seven documents it would be counted seven times. This list also includes all stop words and singletons. After all processing, including removal of singletons, the remaining value is only 13% of the total. This is a set, and therefore each term will only occur once in this list.

Document 014 includes the word "sorrows," which was a singleton and therefore removed although it may have had significance in identifying that specific document. Document 489 is not an English document, in point of fact it appears to be in Spanish. However, it must not be the only Spanish document in the corpus or most of the terms would have been removed as singletons. This informs us that the corpus is comprised of files which are not necessarily in English. If the user base for the

application does not typically search non-English words it may be beneficial to offload these terms onto a separate file to reduce the term space.

Conclusion

This implementation of the project is considerably faster than the previous version, cutting its total processing time by half. This time does include IO, and this version does much less IO, which is a probable factor in the timing difference. However, timings were run on an early Version 3.0 without the in-memory term document matrix, that indicates a running speed approximately as slow as Version 2.0.

Figure 1 illustrates the time increase in time required for processing and IO for an increasing number of input files. The input files in the corpus are non-uniform in content and size. This may account for the steeper increase on the higher end of the graph. Figure 2 demonstrates that the average file size for the files in the 301-400 file group is considerably larger than all previous and following input files. File size is calculated as the total instances of all tokens in the document, including stop words and singletons.

Figure 3 displays the slope between the points in Figure 1 and also indicates the largest increase in the increase in time (the acceleration) is caused by processing in the 301-400 file group. Still, the largest increase in overall time occurs when all 500 files must be processed. A program analyzer would help determine if the slowdown was caused by file processing or memory searches.

Another factor causing the exponential growth in the higher end of Figure 1 is the extra time to search hash lists for terms. This is factor is demonstrated by the slope of the graph between the points, see Figure 3. The time it takes to add a document ID to the end of the list in the term document matrix was also considered as a possible contributing factor, but has been ruled out via analysis and tests. The code, which adds the document ID was altered to insert the new document ID to the front of the list and this change offered no noticeable improvement. The container object List's implementation is managed by C# and therefore no internal analysis was reasonably feasible.

Examining the slope graphs and the timing graphs from each of the three phases so far indicates a decline in the processing time required overall as well as the impact more input has on the processing time, see Figure 4. Therefore not only is the program faster, but it handles the larger term space with less degradation. There are many factors contributing to this detail. The first incarnation was coded in Ruby, a scripting language, whereas the second and third are in C#, a compiled systems language. Also, in this current incarnation is only prints out two files in the end; whereas both previous versions printed a file per input file. This is a significantly smaller amount of disk IO. It would be interesting to examine the memory footprint as the input increases in three factors: content size; file count; and content diversity.

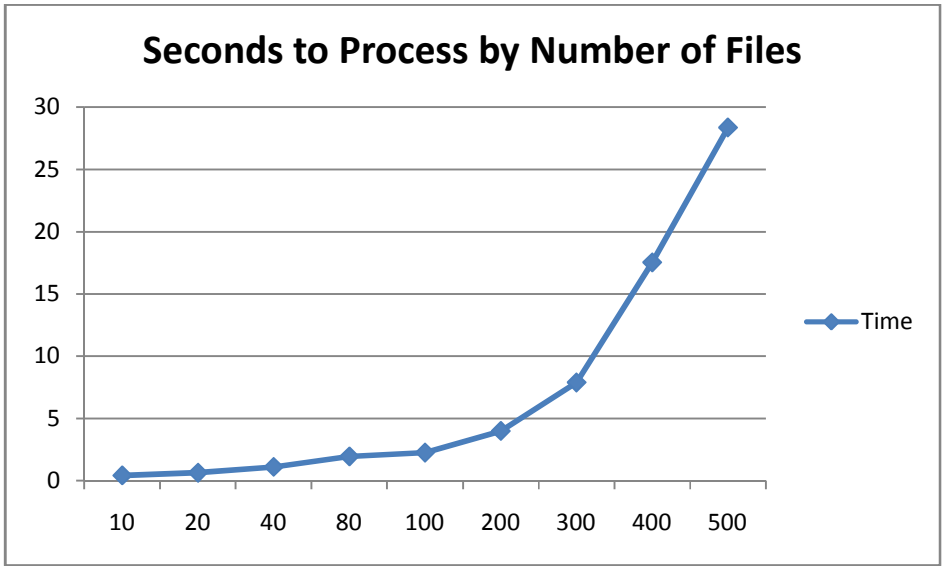

Figure 1 - Seconds to Process Input by Number of Files

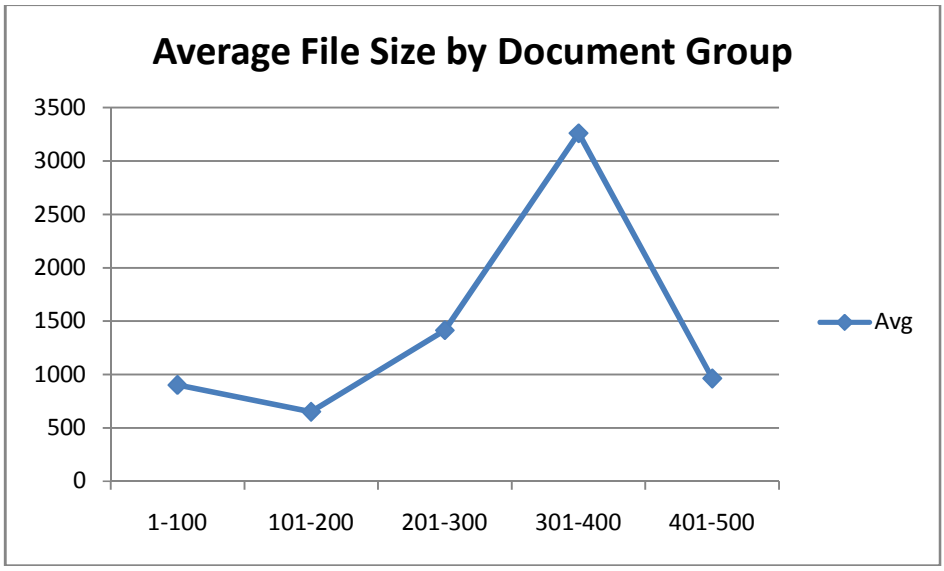

Figure 2 - Average File Size by Document ID Range

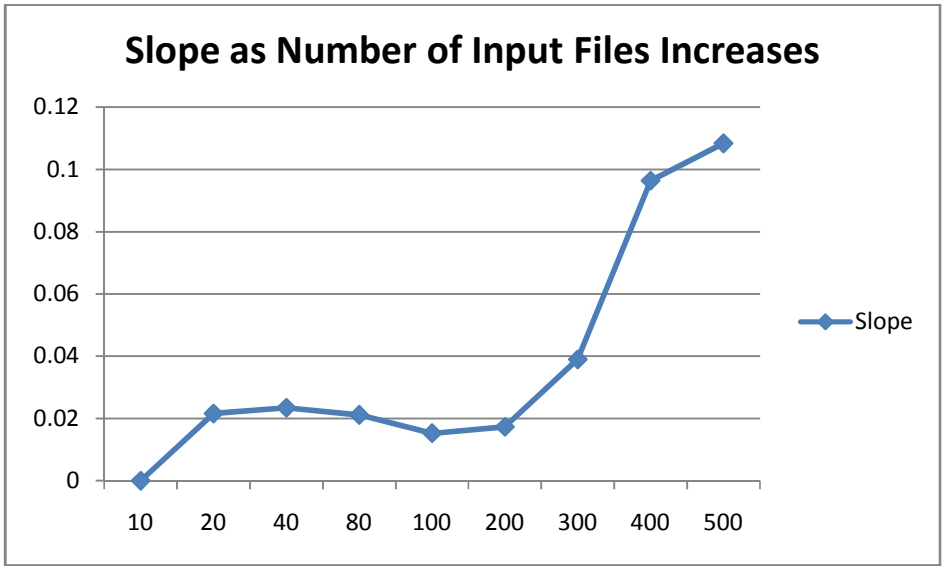

Figure 3 - The Slope between the Timing Points by Number of Input Files

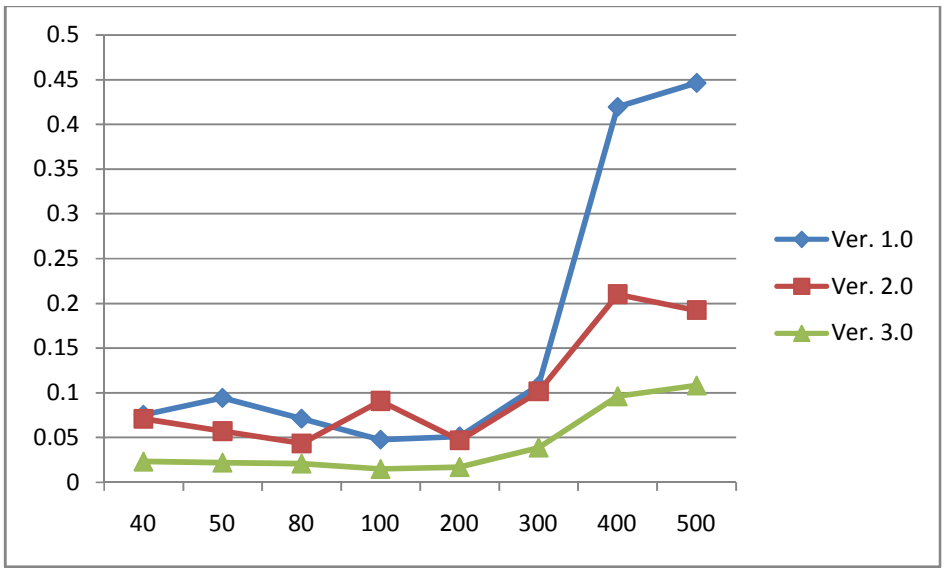

Figure 4 - The Slopes for the 3 Versions*

*Due to 2 misaligned data points for Ver. 1.0 the data was interpolated by surround points.