Universidade de Aveiro

Departamento de Eletrónica, Telecomunicações e Informática

Assignment 2

Information Retrieval

November 12, 2018

Professor Sérgio Matos

Diogo Ferreira ***76504***

Luís Leira ***76514***

Context

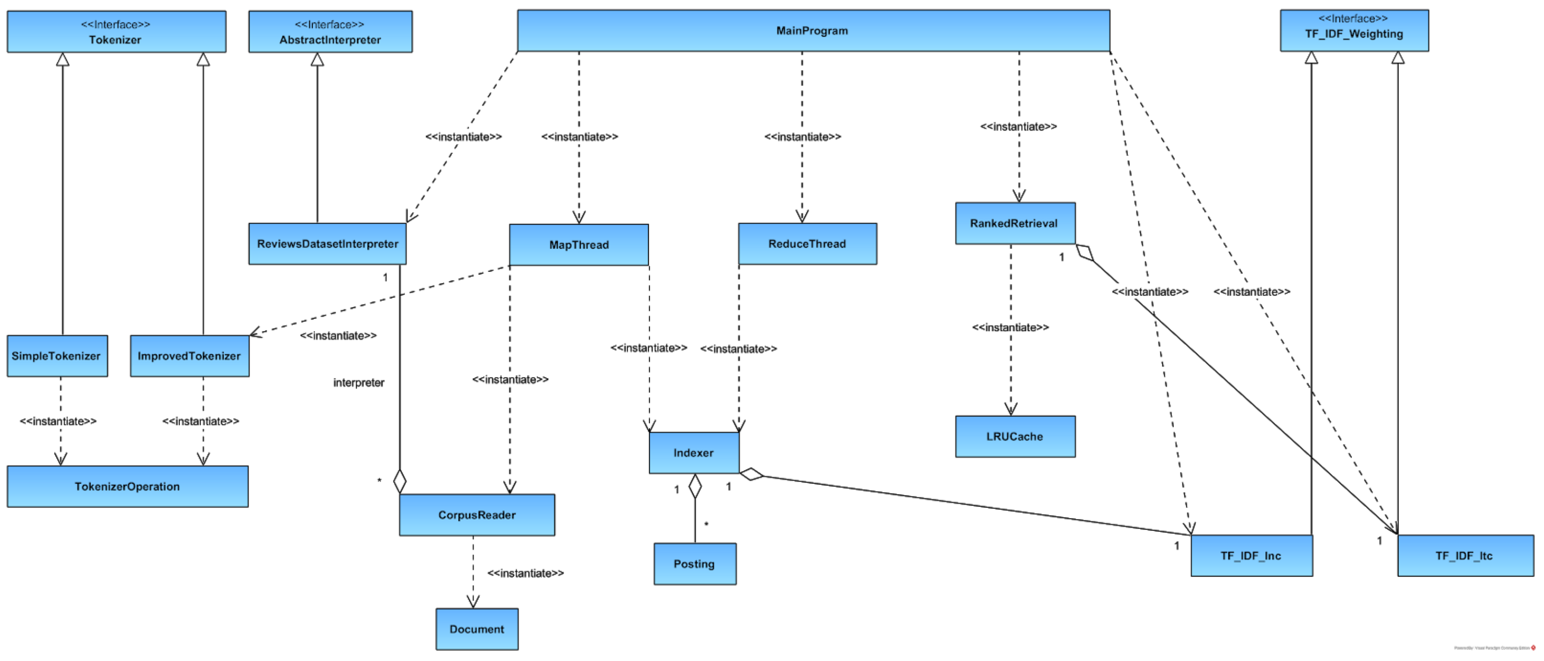
This report explains the work done in the second assignment of the Information Retrieval course. This assignment's aim is to create a weighted indexer using tf-idf and to search the indexed documents using a ranked retrieval system, with the help of a cache to speed up the results. It was also implemented a positional index and phrase and proximity search.

This work was made using the work previously done in assignment 1. The components unchanged from the previous version will also be mentioned, as well as the new or updated components, with the help of the architecture diagram and the data flow between them. It also contains details about execution instructions, external libraries used, efficiency measurements and final conclusions.

The corpus for this project is the Amazon Customer Reviews Dataset available at https://s3.amazonaws.com/amazon-reviews-pdf/readme.html.

Architecture

In this section, the software architecture is presented in the form of a class diagram. Each class function and its main methods are described below. The new classes created in this assignment in comparison with the first assignment are: TF_IDF_Weighting, TF_IDF_Inc, TF_IDF_ltc, RankedRetrieval and LRUCache. The Indexer, MainProgram, Document and Posting classes were also updated.



MainProgram

This class contains the main flow of execution of the project. It has the responsibility of creating the workers for the map and the reduce phase (explained in the next section, Data Flow). According to the maximum number of parallel threads and the maximum number of documents per indexer, the number of threads and documents per indexer will be adjusted. As such, before running on a new configuration, it may be needed to adjust the settings adequately to prevent memory errors or lower the execution time.

After launching workers to execute the map phase and the reduce phase, the MainProgram will call finalizeIndex(int numberDocuments, String pathPrefix) to finalize the writing of metadata on the index files. Then, the exported index will be read from the files and imported into the application.

Finally, three operations will be done with the indexer:

- The vocabulary size will be calculated;
- The first ten terms in only one document will be calculated with the method firstNTermsInMDocuments(Map<String, Integer> index, int n, int m);

- The ten terms with higher frequency in the indexer will be calculated with the method nTermsHigherDocumentFrequency(Map<String, Integer> index, int n).

After the processing of the index, a ranked retrieval instance is created, to search by queries in the indexed documents. The document ID's of the retrieved documents are then presented to the user.

MapThread

Each MapThread is responsible for reading part of the input file, specified by the start and end line. It starts by creating a corpus reader from a file and iterating through the allowed documents inside its range. For each document, it will apply a tokenizer and the resulting tokens added to the indexer. After indexing all the terms, the indexer will export the index to multiple files, each file containing the index with the terms that start with a different alphabetical letter. The index with all other terms that do not start with an alphabetical letter will also be saved into a different file.

ReduceThread

The ReduceThreads are launched after all MapThreads finish. Each thread is responsible for creating a unified index for the terms that start with a specific alphabetical letter (one thread is also responsible for creating the index with terms that do not start with an alphabetical letter). To do so, it is necessary to read all indexes with terms that start with an alphabetical letter and merge them all into one index. Because the indexes are sorted, it can be used a strategy to merge indexes without loading the entire indexes into memory. It can be specified a number of index entries to read from each file, and then sort them and append them into a temporary file. With this strategy, it is possible to construct large indexes without running out of memory. The long explanation of this algorithm is described in the Indexer subsection.

CorpusReader

The CorpusReader class deals with the opening, reading and pre-processing of the documents. It receives in the construction of a new object the path to the file where the corpus is, an interpreter to interpret each line as a document, a start line and an end line. The start line and the end line indicate a region from where to fetch documents from the file. The method *getNextDocument()* retrieves the next document object from the corpus in each call, and if there is no next document or the corpus reader has reached the end line, it returns null.

Document

The Document class represents a retrieved document from the corpus. It contains an ID (different for each document), the product title, the review headline and the review body. The Document constructor is private, and it is only possible to create a new document with the synchronized function createDocument(String productTitle, String reviewHeadline, String reviewBody). This Singleton construction causes the document ID increment to be an atomic instruction.

AbstractInterpreter

The AbstractInterpreter interface contains the method *interpret(string doc)* and it is used for other classes that want to implement a method for retrieving documents from strings.

ReviewsDatasetInterpreter

The ReviewDatasetIntepreter implements the AbstractInterpreter and it is the implementation of an interpreter for the reviews dataset used in this work.

Tokenizer

The Tokenizer interface specifies the method *tokenize(String words)* that returns a list of tokens given the input. The input is a list of words to be tokenized.

SimpleTokenizer

The SimpleTokenizer class implements the Tokenizer interface to create a tokenizer. The *tokenize(String words)* method does four operations: splits the words on whitespaces, converts the tokens to lower case, removes all non-alphabetic characters and removes terms with less than three characters.

ImprovedTokenizer

The ImprovedTokenizer class is another implementation of the Tokenizer interface. The operations available are the following ones:

- splits the words on whitespaces;
- converts the tokens to lower case;
- removes invisible characters;
- removes all tokens that are e-mails;
- removes all tokens that start by a number and have a punctuation next to the number (this dataset has many tokens with that format);
- removes meaningless characters;
- removes tokens that are hyperlinks;
- removes full stops at the beginning or end of token;

- removes all “^” characters without numbers on both sides of it (can indicate an exponential number);
- removes hyphenation at the beginning and end of tokens;
- removes the stop words according to a list of stop words;
- stems the tokens with the Porter Stemmer in English language.

  The operations are carefully described in the TokenizerOperations class subsection.

TokenizerOperation

The TokenizerOperation class has the methods to apply transformations to words and turn them into tokens. When the class is loaded, the stop words are loaded from a file, using the method loadStopWords(). When the class is instantiated, the constructor receives a string of words that transforms into a list of tokens, splitting the words on whitespace. The process results on a stream of tokens, that is stored inside the object. The stream of tokens can be retrieved using the method getTokens(). The operations can be piped into each other because the return type is the TokenizerOperation class. The methods that can be applied to tokenizer the words are:

- removeNonAlphabeticCharacters() – Removes all the non-alphabetic characters from the tokens;
- removeTermsWithLessThanNCharacters(int n) – Removes all tokens with less than n characters;
- toLowerCase() – Converts all token characters to lower case;
- stem() – Using the Porter Stemmer, applies the stem operation to all tokens;
- removeStopWords() – According to the loaded list of stop words, removes all the tokens that are present in that list;
- removesNumbersFollowedByPunctuation() – Removes all the tokens that start by a number and have punctuation next to that number (common problem in one analyzed corpus);
- removeMeaninglessCharacters() – Removes list of pre-defined characters from tokens. The list of characters is composed by special characters, like ‘$’, ‘!’, ‘[’ or ‘?’;
- removeSoloHyphenation() – Removes the hyphens from tokens where they are in the beginning or end of the token. Remove also all consecutive hyphens;
- removeHyperlinks() – Removes all tokens that are hyperlinks;
- removeFullStops() – Removes full stops at the beginning or end of token;
- removeExpoentSymbolWithoutNumbers() – Removes all ‘^’ characters that do not have a number by each side. When there is a number on both sides, the character is kept, because it may symbolize an exponential operation;
- removeEmails() – Removes all tokens that are e-mails;

- removeInvisibleCharacters() - Removes invisible characters that can be fetched when reading from a file.

Indexer

The Indexer class has the goal of creating an inverted index composed by the terms and its postings. The indexer will consist in a TreeMap, with the terms being the keys and a TreeSet of Postings as the values. The TreeMap and TreeSet both have complexity O(log2(n)) for insertion and search and keep terms and postings ordered.

The addToIndexer(List<String> tokens, int docID, TF_IDF_Weighting weighting) method receives as argument a list of tokens, the id of the document where the tokens were retrieved and the weighting scheme, and it will add the terms to the indexer along with its weights and its positions inside each document, using the method addTermToIndexer(String term, int docID, double termWeight, List<Integer> positionsInDocument) to each term.

The method addTermToIndexer(String term, int docID, double termWeight, List<Integer> positionsInDocument) starts by checking if the term is already on the indexer. If it is not found, it creates a new entry in the index and associates with it a new TreeSet with one posting. It the term is found, retrieves the list of postings of the term and the new posting is inserted into the list.

The function Map<String, TermValue> calculateTermsWeight(Map<String, List<Integer>> termsInDocument, TF_IDF_Weighting weighting) receives as argument a list of terms in the document and its positions in the document, as well as a weighting scheme to weight the documents. This function will calculate the term frequency weight, will apply normalization, and it will return a map with each token and a TermValue class, that will contain two records: the term weight and the positions in a document of each term.

The exportToFileByLetter(String prefix) method exports the created index to files. The index will be separated into multiple sub-indexes. 27 files can be generated, each one with the content of each sub-index. Each sub-index is identified by the first letter of its terms (26 indexes for a to z, and one index for other characters).

The mergeMultipleIndexes(String path, String postfix, String outputFilePath, int maxDocumentsInIndex) method merges the indexes created in the exportToFileByLetter(String prefix) by the first character of the terms in one index. To do so, receives as argument a postfix that all index files follow, to list them. Then, the reading of the indexes in those files is made in sequence. For all files, it will be read a limited number of lines, to not cause an out of memory error. Because the indexes are already sorted, the read index entries can be merged and sorted easily. It is kept track of the last term read in each file. When the number of lines is read from all files, it is searched for the last read term in all files with a lower value. It can be proved that from the beginning of the index until that term all entries are sorted, and can be appended into a file. The entries written into the file are removed from the index in memory and the files are read again. This process continues until all files are fully read. In the end, a complete index is outputted to a file, resulting in the merging of multiple indexes.

The finalizeIndex(int numberDocuments, String pathPrefix) method is to be called when the index files are all written, and it writes the index metadata in a separate file, containing the number of analyzed documents and the number of terms in the indexer.

Posting

The Posting class contains a document ID, the weight of the term in the document and a list of positions in the document where the term can be found. This class also implements the Comparable interface, to compare two postings by the document ID. This feature is useful to automatically sort the Posting instances in a sorted list, in the indexer.

TF_IDF_Weighting

This interface has the methods needed to implement tf-idf weighting: calculate the term frequency, calculate the inverse document frequency and normalization of terms.

TF_IDF_lnc

This class implements the IF_IDF_Weighting interface using lnc weighting scheme: log-weighted term frequency, no IDF and cosine normalization.

TF_IDF_ltc

This class implements the IF_IDF_Weighting interface using ltc weighted scheme: log-weighted term frequency, IDF weighting and cosine normalization.

RankedRetrieval

This class is used to search by a query in the indexed documents and to return the most relevant document IDs. The only function exposed for public access is the search(String query, int numberDocumentsTop, boolean proximitySearch) method, that given a query and the maximum number of documents to retrieve, returns a list with the ID's of those documents. The argument proximitySearch allows the caller to enable proximity search, only returning documents that have queries that exactly match the query made.

In the instantiation of an object of this class, it is required to pass as argument a tokenizer, to tokenize the queries made, a TF-IDF weighting scheme, the folder name where the index is, the maximum size of the posting lists in memory and a ratio to append the terms into cache. The maximum size of the posting lists will be used to limit the size of the cache to retrieved search documents. The maximum ratio to append terms in cache is a ratio that refers to the possibility of caching all found terms and posting lists, or cache only the searched ones. If the maximum number of posting lists in memory divided by the number of terms in the indexer in bigger than this ratio (the memory size is large relatively to the number of terms in the indexer), all terms found will be appended in the cache. Otherwise, only the searched terms will be appended. Internally, the Ranked Retrieval will get from the index metadata the number of documents in the index, to be used in the term weighting phase, and the number of terms in the indexer, to calculate the ratio to append all terms into cache.

Some relevant private functions are:

- Map<String, Double> calculateQueryTermsWeight(List<String> tokensList) - GIven a token list of a query, calculate the weight of the terms (term frequency, IDF and normalization);
- Map<Integer, Double> cosineScore(Map<String, Double> queryTermsWeight, int numberDocumentsTop) - Calculates the cosine between a query and all documents in the indexer, and returns a list with numberDocumentsTop documents ordered by its cosine score. The argument queryTermsWeight contains a map with the terms in a query and its respective weight.
- Map<Integer, Double> cosineScore(Map<String, Double> queryTermsWeight, int numberDocumentsTop, List<String> queryTokens) - Calculates the cosine between a query and all documents in the indexer that exactly match the query with the tokens in the same sequence, and returns a list with numberDocumentsTop documents ordered by its cosine score.

  Due to its complexity, this function deserves further explanation. For the analysis of the first token in the query, the cosine score will work just the same as the non-positional cosine score: for each document in the posting list of the token, it will add to the score of that document its term weight multiplied by the term weight in the query. It will also store the positions of the term in the document.

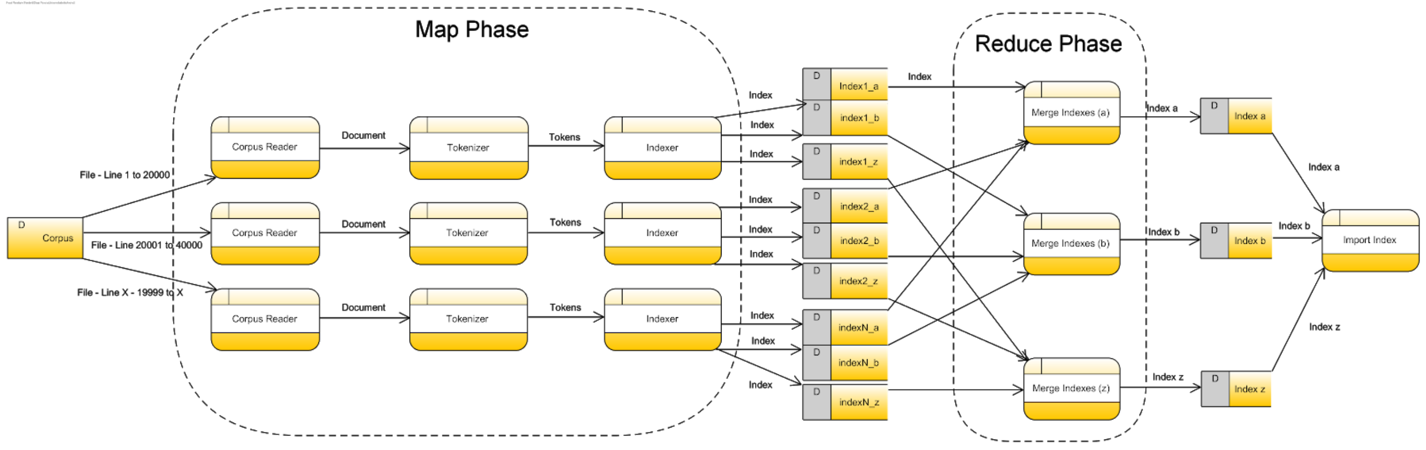
  For the next tokens in the query, the function will go through all the documents that have already been processed (i.e. documents where all the analyzed tokens are in the same sequence in the document as in the query) and it will check if they are in the posting list of the term. If they aren't, they are removed from the processed documents, and its score for the query is 0. If they are in the posting list, it will check if any position of the analyzed token in the document is next to any position of the last token in the same document. If it isn't, the document is removed from the processed document and its score in the query is 0. If it is, updates the last positions of the token in the document in an auxiliary structure and it will check if the token weight was already added to the document score. If not, updates the document score with its term weight multiplied by the term weight in the query.
- Map<String, Integer> calculateDocumentFrequency(Set<String> terms) - Given a set of terms, calculate the number of documents that each term has in its posting list.
- int getNumberOfDocumentsInIndexer(String pathPrefix) - Get the total number of documents in the indexer.
- List<Posting> getPostingList(String term) - Given a term, returns its list of postings, or returns null if the term does not exist in the index. To speed up the search phase, relies on an LRU cache that stores the recently accessed terms and its posting list. The LRU cache removes the eldest entry when maximum size of the cache is reached. If the number of indexed terms is not too big when compared to the size of the cache, all terms and its posting lists will be cached when found in the indexer. In this way, it is possible to cache most terms before even searching for them. Otherwise, if the number of terms is too large relatively to the available memory, only the searched terms will be added to the indexer. In this way, our goal is to optimize

the cache utilization, reducing the cache misses. The searched terms not found in the index are also cached with null value, to prevent from accessing the file when searching for them.

LRUCache

Implementation of a Cache with the replacement policy of Least Recently Used. It extends the implementation of the LinkedHashMap and overrides the removeEldestEntry to replace the eldest term when the maximum size of the cache is reached.

Data Flow



The data flow diagram of the project can be seen above. The map phase consists in a number of threads processing the documents in parallel. For simplicity, the number of threads depicted in the diagram is three. In each thread, the corpus is loaded by the Corpus Reader, and the Corpus Reader is responsible for reading part of the file. The corpus reader will produce documents, that are sent to the tokenizers. After the tokens for a document are produced, they will be sent to the indexer, to be indexed according to the document. Each index will be written on a file. Each thread will produce 27 indexes (one for each alphabetical letter, one for others characters).

The reduce phase will also be done with multi-threading. 27 threads will be launched: one for each alphabetical letter and one for other characters. Each thread will perform a merge of the indexes that start with a different alphabetical letter. To prevent from out of memory errors, each thread will read a number from lines from each file, merge the indexes, sort them and append them into a file. The process is repeated until all files have been fully read. In the end, each thread will produce a file with the index with terms that start with each alphabetical letter and one index for terms that start with other characters.

Finally, to answer the questions proposed in the assignment, the indexes will be read from memory and merge into one index. That index will not contain all document ID's from each term, only the number of documents of a term. In that way, the index can be merged into memory without the need for storage in intermediate steps.

Another variation of the data flow was also tested. In that variation, the threads that implement the map phase would not read directly from the file, instead they would call a method that would read sequentially the documents in the corpus. However, the execution time of the software was not as good as the data flow described above. A deeper explanation and discussion on the differences and the results is done in the conclusion section.

External libraries

The different tokenizer implementations have two methods that use an external library whose documentation can be seen in http://snowball.tartarus.org/. The stem method is an integration of the Porter stemmer according to the library, whose information is available at http://snowball.tartarus.org/download.html. The removeStopWords method is a stopword filter that uses a default list according to the same library, whose information is available at http://snowball.tartarus.org/algorithms/english/stop.txt.

Execution instructions

There are two main ways to compile and execute the code: by the command line or with an IDE. Both are explained below.

The parameters available to change (*MAX_PARALLEL_THREADS*, *MAX_DOCUMENTS_PER_INDEXER* and *MAX_POSTING_LISTS_IN_MEMORY*, explained in the efficiency measurements) are in the MainProgram. The file path of the corpus and the tokenizer are also specified in the main program.

Command line

To use this method, Maven must be installed.

Go to the project root directory.

The external library must be integrated by doing the download of the libstemmer (available at http://snowball.tartarus.org/download.html) and following the README instructions.

Then, build the project by executing the following command:

```
mvn package
```

After building the project, you may test the compiled and packaged JAR with the *MainProgram* from the solution by executing the following command:

```
java -cp target/IRProject-1.0-SNAPSHOT.JAR assignment2.MainProgram
```

IDE

This project can also be imported in any IDE that supports Maven like NetBeans or Eclipse.

After importing the project, do “Build” and then go to Run > Configurations > Customize > Run and select the *MainProgram* from the solution. No arguments are needed to run the application.

Finally, click on “Run project”.

Efficiency measurements

On this section it will be described the tests done to measure the execution time of the software and the total index size on disk.

These efficiency measurements were taken in a 3 year old laptop with an *Intel i7-4720HQ (4 cores, 8 threads) 2.60GHz*, *16GB RAM, 1TB HDD disk (5400rpm)* running *Windows 10*.

From the program, there are three variables to take into account:

- *MAX_PARALLEL_THREADS* - the maximum number of parallel threads that can be running simultaneously. This variable was tested with different values to find the best execution time.
- *MAX_DOCUMENTS_PER_INDEXER* - Maximum number of documents per index (to not exceed memory). This value was tested with different values, and it is expressed as 100000/MAX_PARALLEL_THREADS (depends on the previous variable). 100000 is a fixed value and it was chosen because it provided the best measurements among other tested values (10000, 100000 and 1000000).
- MAX_POSTING_LISTS_IN_MEMORY - Maximum number of posting lists that can be in cache memory to speed up the search time. Currently, the value is 10000.

There are two Tokenizer implementations (*SimpleTokenizer* and *ImprovedTokenizer)* but only the second one was used. The following datasets were tested:

- *amazon_reviews_us_Watches_v1_00.tsv* (~393Mb).
- *amazon_reviews_us_Wireless_v1_00.tsv* (~3900Mb).

The first dataset was tested with 4 threads and JVM memory restricted to 512Mb. It took 10m03s and the total index size on disk is ~289Mb.

The second dataset was tested with 4 threads and JVM memory restricted to 512Mb. It took 3h45m20s and the total index size on disk is ~2750Mb.

Results

On this section, we will exemplify the search with and without proximity, showing the results retrieved in the *sample_us* dataset. We will also analyze the performance of the search operation in bigger datasets.

The Ranked Retrieval system was tested in the same conditions as described in the Efficiency measurements.

It were tested the following queries:

- “game”;
- “perfect match”;
- “match perfect”.

For the first query, the fetched documents were the same with and without proximity search. Its content were:

- *“We play this **game** quite a bit with friends”*
- *“Even though both of my kids are at the top of this age recommendation level, they LOVE this **game**! I love how it caters to the kinesthetic learner by asking them to move their bodies into the shape of the letters. It even takes teamwork as sometimes two people are required to finish the letter. My kids know all of their letter sounds and shapes, but this didn't stop them from playing the **game** over and over.”*
- *“You need expansion packs 3-5 if you want access to the player aids for the Factions expansion. The base **game** of Alien Frontiers just plays so much smoother than adding Factions with the expansion packs. All this will do is pigeonhole you into a certain path to victory.”*
- *“Absolutely one of the best traps in the **game**. It is never a dead and always live since you can always pay half your lifepoints for its cost. It's main power is that it can stop any card. Hopefully this card comes off the Forbidden/Limited list soon.”*

As we can see, the four documents contain the word “game”. The second document contains the word two times, but the first document is considerably smaller, so its weight is bigger. The other two documents contain the word “game” once. By inspection of the corpus, we can check that the word “game” does not appear in any document besides the four presented above.

For the second query, the search without proximity returns the following documents:

- “*Fits my $20 bill **perfectly**.*”
- “G*reat quality wooden track (better than some others we have tried). **Perfect match** to the various vintages of Thomas track that we already have. There is enough track here to have fun and get creative incorporating your key pieces with track splits, loops and bends.*”
- “*I ordered these for my 3 year old son's birthday party as party favors. They were a huge hit & the **perfect** fit for a 3 year old!*”

As expected, the proximity search only returns the second document, where the words “perfect” and “match” are sequential.

For the third query, the search without proximity returns the same results as in the second query. The proximity search does not return any document, because there is no such document that contains the words “match” and “perfect” sequentially in the previous order.

On the bigger datasets the documents will not be shown. Instead, the time to search will be analyzed.

The queries done on the *amazon_reviews_us_Watches_v1_00* dataset were tested:

- “wonderful”;
- “some watches”;
- “water resistance weight stability”.

For the first query, the search without proximity took ~0.388 seconds and the proximity search took ~0.009 seconds.

For the second query, the search without proximity took ~0.757 seconds and the proximity search 0.089 seconds.

For the third query, the search without proximity took ~0.854 seconds and the proximity search 4.445 seconds.

The queries done on the *amazon_reviews_us_Wireless_v1_00* dataset were the following:

- “wonderful”
- “some iphones”
- “smooth treated rubber”

For the first query, the search without proximity took ~1.658 seconds and the proximity search took ~0.046 seconds.

For the second query, the search without proximity took ~1.280 seconds and the proximity search took ~0.152 seconds.

For the third query, the search without proximity took ~3.737 seconds and the proximity search it took ~2.850 seconds.

The caching system clearly reduces the time to search, as it can be seen by the two first search times in the bigger datasets. The time to search with proximity is much smaller in the two first searches because the terms searched are already cached due to the search without proximity done before. The third search in both datasets takes more time due to the processing of the documents that follow the sequential order. Because the number of tokens is bigger, there is a big overhead in storing the position of the terms in the documents that contain all the terms in the specified order. It is expected that the time to search with proximity increases at a faster rate when searching with a bigger query without proximity.

Due to the caching system, for small files and a small number of queries, the search will take more time than needed, because it will add to the cache all found terms in the index, even if not searched. However, for a big number of different queries or for large files, the performance gains in the long run will be higher. For a big number of queries, most terms will be in the indexer even if they are not searched. For large files, only the searched terms are appended to cache due to memory constraints.

Conclusions

In this assignment was made the conversion from an index where the posting list would consist in a document ID and its frequency, to an index where the posting list is the document ID, the term weight and a list with the positions of the term in the document. Besides that, it was also implemented search with and without proximity in a document, and an LRU cache to speed up the search phase.

The index size and the time to create the index are much bigger in the new implementation, due to the calculation of the TF-IDF weight and the inclusion in the posting list of the position list of a term in a document.

A big advantage of this implementation in the search phase is the LRU Cache and its adaptability to the number of terms and the available memory. If the number of terms is too big relatively to the available memory, only the searched terms will be added, reducing the cache misses. If the number of terms is not so big relatively to the available memory, all terms found while searching for other terms in the index will be cached, allowing for faster caching of all terms. While for a small number of queries and a small index the cache takes more time than needed because it will store all terms found in the index until it reaches the searched term, it is faster in the long run, with a higher number of queries.

The maximum ratio to append terms in cache is currently defined in 0.8. That value affination could improve the search time even more however the extensive testing with different parameters was out of scope of this report. An extensive evaluation of the queries time when searching was also out of scope.