Overview

In this part of the Course Project you will enhance the simplified search engine that you built in Part 1. First, you will add the support for wildcard queries. Then, given a query, you will output a set of pages that answer this query, sorted according to a relevance estimate based on a vector space ranking model.

Please read carefully the whole document before starting coding. If you are in doubt, ask the TAs.

1 Specification

You will extend the two programs (one for building the inverted index and the second for answering queries using this index) that you coded in Part 1.

1.1 Building the Inverted Index

As in Part 1, your first program createIndex will build a positional inverted index from a collection of documents.

1.1.1 Parsing the collection

You will adopt the same parsing rules that were described in Section 1.1.1 of the specification for Part 1.

1.1.2 Inverted index format

You can decide the format on disk of your inverted index and it should be contained in a single file, as in Part 1. After reading only the inverted index, you must be able to obtain the postings list associated with a term, with its position(s) within each document. Moreover, in view of the query procedure detailed in Section 1.2.4, you might want to clearly separate the dictionary of terms from the postings lists (so you can read only the former) and store the term weights in the index (instead of computing them at execution time).

1.1.3 Title index format

You should produce a title index too. You can decide its format on disk, provided that you store the id and the original title of each page. Your title index must be contained in a single file.
1.1.4 Input/Output

You will submit a bash script `createIndex.sh`, suitably invoking your `createIndex` program. Your bash script should accept four parameters, representing names of files:

- the list of stop words;
- the collection of pages;
- the inverted index to be built;
- the title index to be built.

To test your program, we will invoke something like this:

```
$ ./createIndex.sh myStopWords.dat myCollection.dat myIndex.dat myTitles.dat
```

1.2 Ranking of Search Results

Your program `queryIndex` will accept queries and output an ordered list of documents that answer each query, sorted using the procedure described below.

1.2.1 Query types and query parsing

Your program will handle the four query types from Project Part 1 and two new types of queries:

- One word query (OWQ): a single word (e.g., `Spartacus`);
- Free text query (FTQ): a sequence of at least two words, separated by a space (e.g., `Orange Clockwork` or `Eyes Wide Shut`);
- Phrase query (PQ): a sequence of at least two words, separated by spaces and in double quotes (e.g., `"Paths of Glory"` or `"Barry Lindon"`);
- Boolean query (BQ): any expression generated by the following grammar:

  ```
  boolquery → word | (boolquery) | boolquery AND boolquery | boolquery OR boolquery
  (e.g., `Full OR Metal`, `2001 AND Odyssey AND Space`, `(Fear AND Desire) OR Shining`, `((Lolita OR Strangelove) AND (Start AND Stop))`)  
  ```

- Wildcard query (WQ): a single word, containing at least an asterisk (e.g., `Oran*` or `Spar*c*`);
- Wildcard phrase query (WPQ): similar to PQs, but allowing wildcard words (e.g., `"Path* Glory"` or `"Bar* Lind*"`).
Please note that “word” here means any string of alphanumeric or punctuation characters (excluding double quotes, reserved for PQs; parentheses, reserved for BQs; and asterisk, reserved for WQs) terminated by a blank space: this definition allows the user to input queries like "2001: A Space Odyssey" or Killer’s Kiss.

Only capitalized AND and OR act as operands for BQs, that is, you can assume that if a query contains AND or OR, they represent the logic operators. The precedence order is the standard one, from highest to lowest: first ( and ), then AND, then OR. (Note that $x \text{ OR } y \text{ AND } z$ must be parsed as $x \text{ OR } (y \text{ AND } z)$.) You can assume that the PQs, BQs and WPQs are well-formed. In case a query is composed by a single word, you should consider it as a OWQ (or FTQ, as they are equivalent in this case) rather than a BQ.

PQs differ from FTQs because they require each matching document to contain the terms corresponding to the words specified by the query, one next to the other and in the same order. BQs require the returned documents to contain terms corresponding to the boolean condition.

The wildcard character (*) is expected to match any string of alphanumeric characters (hence terminated by a non-alphanumeric character, like a space or a punctuation character). For example, [Bomb*] matches Bomb, Bombs, Bombardier or Bomb2011. To simplify your work, you can assume that each wildcard word contains at most two wildcard characters, but they will not be contiguous (i.e., you might deal with Lol*t* but not with Strangel**ve).

1.2.2 Ranking documents

After parsing a query $Q$, you will have its type $T$ and a stream of $k$ terms $(t_1, t_2, \ldots, t_k)$, which are the stemmed version of the words contained in the query. You should first determine the set of documents $D_Q$ matching the given query $Q$. This should be done according to the type of the query:

- if $T$ is a OWQ, then $D_Q$ is the set of documents containing at least one word whose stemmed version is $t_1$;
- if $T$ is a FTQ, then $D_Q$ is the set of documents containing at least one word whose stemmed version is one of the $t_i$’s;
- if $T$ is a PQ, then $D_Q$ is the set of documents containing words which, after the removal of stop words and after being stemmed, produce the subsequence $(t_1, t_2, \ldots, t_k)$ in this order and with adjacent terms;
- if $T$ is a BQ, then $D_Q$ is the set of documents containing words which, after being stemmed, satisfy the boolean condition;
- if $T$ is a WQ, then $D_Q$ is the set of documents containing at least one match for the wildcard word;
- if $T$ is a WPQ, then $D_Q$ is as for a PQ, but allowing wildcard matches.

The last step is ranking the documents of $D_Q$ by their relevance to the query $Q$: you will write a procedure relying on the vector space model. Specifically, you must use the Algorithm CosineScore of Figure 6.14 of the textbook, where the parameter $q$ is the stream of terms $(t_1, t_2, \ldots, t_k)$ mentioned above.
You will weight terms within documents and queries as illustrated in Example 6.4 of the textbook. The weight of a term $t$ in a document (page) $d$ is its $tf$, with the Euclidean normalization over term frequencies:

$$wf_{t,d} = \frac{tf_{t,d}}{\sqrt{\sum_{t' \in T_d} tf_{t',d}^2}},$$

where $T_d$ is the set of all terms appearing in document $d$.

The weight of a term $t$ appearing in a query $q$ is its $idf$ in the collection:

$$w_{t,q} = idf_t = \log \frac{N}{df_t}.$$

### 1.2.3 Weights for Wildcard Terms

If the query contains wildcard words, you have to determine the weights as follows. Let $T$ be the set of terms that match a wildcard word $w^*$. You need to consider the set $DT = \cup_{t \in T} D_t$, that is, the documents which contain at least one term from $T$. The weight of $w^*$ for a document $d \in DT$ is defined as:

$$wf_{w^*,d} = \max_t wf_{t,d} \quad \forall t \in T.$$

For a query, you will use as weight of $w^*$ the highest $idf$ among all the terms in $T$, i.e.

$$w_{w^*,q} = \max_t idf_t \quad \forall t \in T.$$

**Example:** Let our dictionary contain the terms [Manet] and [Monet] with the following term $idf$’s:

$idf_{\text{Manet}} = 3, \quad idf_{\text{Monet}} = 2,$

and term-document $wf$’s:

$$wf_{\text{Manet},0} = 0, \quad wf_{\text{Monet},0} = 0.5, \quad wf_{\text{Manet},1} = 1.5, \quad wf_{\text{Monet},1} = 0.2.$$

Given a query $q = \text{M*net}$, we will adopt the following weights:

$$wf_{\text{M*net},0} = 0.5, \quad wf_{\text{M*net},1} = 1.5, \quad w_{\text{M*net},q} = 3.$$

### 1.2.4 Input/Output

You will submit a bash script `queryIndex.sh`, suitably invoking your `queryIndex` program. Your bash script should accept three parameters, representing names of files:

- the list of stop words;
- the inverted index;
- the title index.
Your program will read queries from the standard input, one at a time, and output to the standard output the list of the id's of the 10 most relevant documents. After retrieving and outputting the relevant documents for a query, your program will wait for another query, until the user closes the standard input with a CTRL+D. Please note that your program might be run in an interactive way. Reading all the queries first and then producing all the results is not acceptable.

More precisely, you should output the id's of the 10 documents with highest non-zero score, in decreasing order of score (highest first), separated by a blank space. If there are less than 10 relevant documents, output them all. If there is no relevant document, output a blank line. For example, the following might be a typical execution:

```
$ ./queryIndex.sh myStopWords.dat myIndex.dat myTitles.dat
Space Odyssey
10 42 12 34 56 14 89 0 13 1
Space AND Odyssey
10 12 14 0 1
Odyssey AND Space
10 12 14 0 1
"2001: A Space Odyssey"
10 12 14
"2001: A Space Od*"
10 12 14 0
Titan*

Full Metal Jacket
42 45 54 12 88 89 67 81 9 6
```

Note that the I/O interface specified above allows us to use redirection, as in the following example:

```
$ ./queryIndex.sh myStopWords.dat myIndex.dat myTitles.dat < myQueries.dat > myDocs.dat
```

The number of lines of myQueries.dat and myDocs.dat should be the same.

For this assignment (and the subsequent ones), your query queryIndex program is not allowed to read all the inverted index in memory. Instead, you can only load the dictionary before starting answering the queries, and then you must fetch the relevant postings lists directly from disk. Loading the dictionary should take no more than 3 minutes (for the full collection) and you should use a tree structure (e.g., a B-tree) to manipulate the dictionary once it is in memory. In particular, you should be able to handle wildcards efficiently. If you code in Python, you can use the /course/cs158/src/lib/btrees library. Your program should answer each query in less than 10 seconds, including fetching the relevant postings lists from disk. (The 10 seconds rule applies to “reasonable” queries, like the sample ones that we will provide. Of course, we will not expect your code to answer queries like `a*` in such a short time!)

## 2 Data

For this assignment you will need the following files located in `/course/cs158/data/part2/`:

1. `myStopWords.dat`  
2. `myIndex.dat`  
3. `myTitles.dat`
• testCollection.dat: the test collection, containing roughly 10% of the full collection (33 MB);
• fullCollection.dat: the full collection (warning: 340MB!);
• queries: directory with some sample queries on the full collection;
• stopWords.dat: the list of stop words;
• createIndex.sh: sample Bash script to invoke your createIndex program;
• queryIndex.sh: sample Bash script to invoke your queryIndex program;
• readme.txt: a plain text file you should fill in with your information.

You can find an implementation of the Porter stemming algorithm at http://tartarus.org/~martin/PorterStemmer/. Moreover, if you are coding in Python, you can use our parser of boolean expressions which you can find in /course/cs158/src/lib/bool_parser. To assist you in checking that your programs run correctly, we will provide you with the expected results for some of the queries on the collection. We will release this expected output a few days past the publication of this specification, announcing the details via email.

3 Report

Please include a report in PDF format answering the questions below. When analyzing your data structures and algorithms, please adopt the big-O notation instead of generic “very long” or “very big”, parameterizing your analysis according to the relevant quantities (number of documents/terms/etc.).

Please make sure that your answer is within the sentence limit, we do not expect more than that, less is fine as long as it is complete. Be concrete, precise and concise. Please do not answer questions that we do not ask! If you are not sure, ask the TAs.

1. Describe the format of your inverted index justifying your choice. What is its total size on disk for the given collection of documents, in absolute terms (number of bytes) and as % of the size of the original collection file. (5 sentences)

2. What is the time and space complexity of your createIndex program? Explain your answer. (5 sentences)

3. At execution time, when and how does your queryIndex program access the inverted index? (3 sentences)

4. Describe the data structures that your queryIndex program uses for manipulating the dictionary (once it is in memory), specifying how you use them to answer wildcard queries. What is the time complexity of answering a wildcard query (without taking the ranking into account)? (7 sentences)

5. Try your search engine on the given collection with your own queries, a few of each type. Which type of queries does it perform best/worst on, in terms of result relevance? How does this depend on the query words (common/uncommon/specific/etc.)? Provide at least 2 examples for each query type, one positive and one negative.
4 Collaboration Policy

You can work on the Course Project in pairs, with the same team-mate with whom you worked on Part 1 of the Course Project. Only one submission per team is required. Please email cs158tas at cs dot brown dot edu if you have any questions.

5 Submission

Please copy all the files mentioned below into a separate directory, cd there, and run the handin script from that directory. Be aware that since the handin script copies recursively all the files from the directory it is executed from, if you run the script from, say, your home directory, it will handin all your files and directories! The directory from where you run the handin script should contain only:

- Any source code you wrote and any external module (e.g., .py/.pyc) you need to run your code.
- The two Bash scripts createIndex.sh and queryIndex.sh, suitably modified to call your programs.
- The plain text file readme.txt, filled in with the required information. (Please do not edit the structure of this file (i.e. FULL_NAME, etc.), just substitute our data with yours!)
- Your report, in PDF format, named report.pdf.

Please submit only the above files (do not submit your index, the collection, or your output!) using the following command:

$ /course/cs158/bin/cs158-handin part2

We suggest you to create an empty directory, copy all the above files there, and then create a copy of the entire directory. Move to the latter and try to execute your bash scripts, in order to check that you do not forget to include files necessary to the execution of your programs. If your programs run smoothly, then you can submit your files from the “clean” directory.

6 Evaluation

We will run your programs on a CS machine with the full collection of documents, issuing queries that might or might not be in the sample set. We will verify that your programs produce reasonable results, looking at the retrieved documents. We will compare them with an external reference search engine, and we will judge both the relevance of the retrieved documents as well as the quality of the ranking (i.e., their relative order).

It is mandatory that your programs adhere to the I/O interface described in this specification. In particular, check for unwanted prompts, debugging messages, newlines that you might have forgotten in your code. It is your responsibility to make sure that your code runs on the department machines and satisfies naming and usage criteria described in this document. We will only evaluate submissions that respect the I/O specification. Moreover, we expect that your programs will run without any syntax or runtime errors.

Your grade will depend on:
• The algorithms/data structure you adopted.
• The size of the inverted index generated by your `createIndex` program.
• The quality of the rankings produced by your `queryIndex` program and its speed.
• The quality of both your code and report.

Have fun!