SEIS731 Project Report

Project Name: Shakespeare Search Engine
Reporter: Shuo Yang

Outline:

1. *Abstract & Feature list*
2. *Architecture --- The big picuture*
3. *Anatomy of System --- Implementation details*
4. *How to use*
5. *Query demo & Post Analysis*
6. *Summary & Future work*
7. *References*

Abstract & Feature list

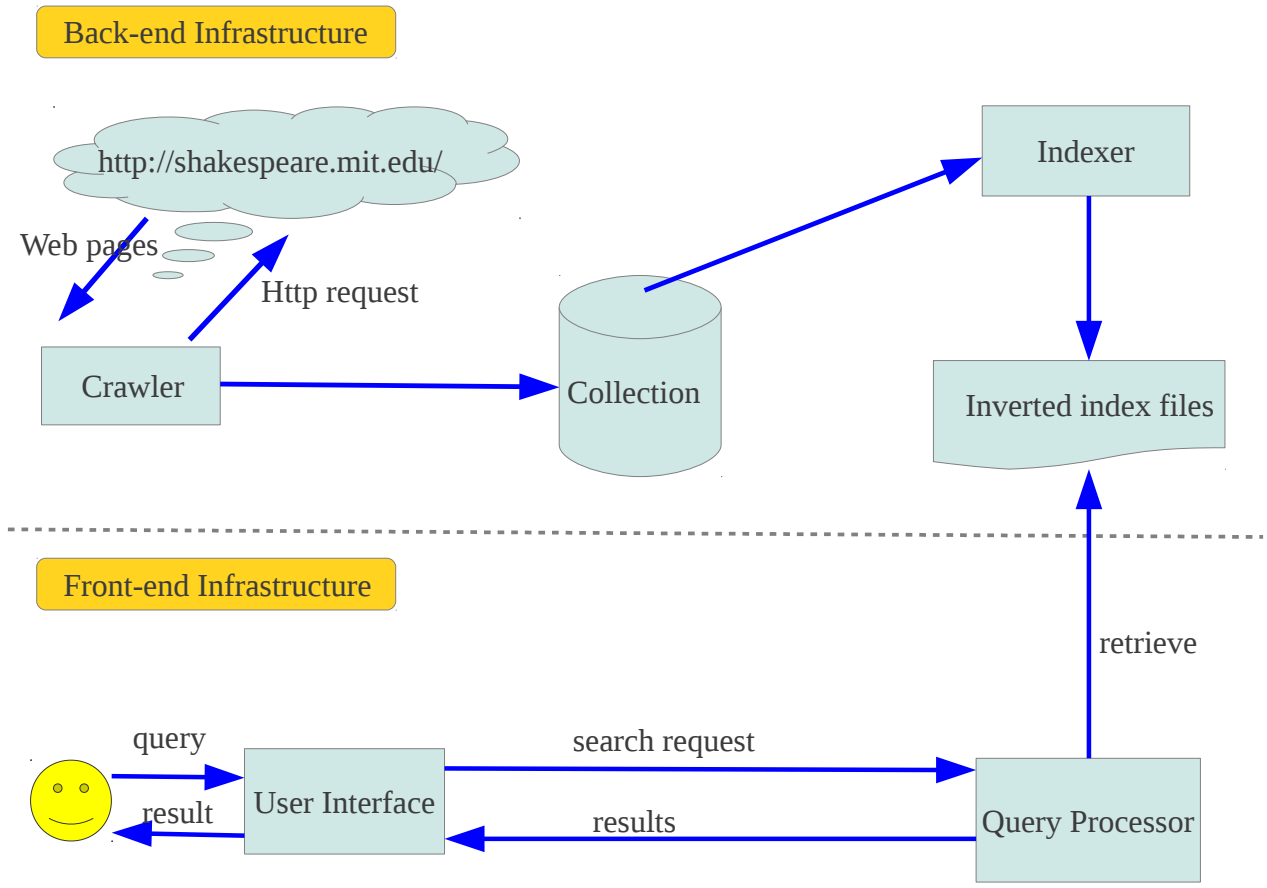
Shakespeare Search Engine is a small search engine that can let you perform both free text query and phrase query against Shakespeare's whole collection (except Poetry). The system contains a crawler, indexer and a query processor. The crawler can crawl Shakespeare's whole collection from http://shakespeare.mit.edu/. The indexer then builds inverted index based on the whole collection that have been crawled. The query processor provides a command line based user-interface that let user enter query and return the results. The goal of the project is to build a search engine from scratch and practice the techniques we discussed in class as much as possible in order to get a better insight of how a search engine works internally.

The report will first present the high-level architecture, then unfold the internal details by showing how each module and the core algorithms works. After that, we will do a basic performance analysis and show how to use the system. Finally, we will sum up a final checklist, lessons learned and future work.

Feature list

Query Type 1	Free text query/key words query
Query Type 2	Phrase query
User interface	Command line based
Data source	web
Collection size	Fixed, 7MB
Number of documents	750
Index	Inverted index, static, fixed, support positional index
Use of stemming	Yes (Got porter stemmer from [0])
Use of stop words	Yes (Got stop words list from [2])
Size of index	Total 23MB

Architecture – the big picuture



Above is the big picture which contains a back-end infrastructure and a front-end infrastructure.

Back-end part includes a crawler and a indexer. The crawler will retrieve links under http://shakespeare.mit.edu/ which hosts Shakespeare's whole collection, then get both meta-data of the collection and the real documents by analyzing these links. Crawler will also build a document hierarchy locally according to the meta-data. After that, indexer will parse the whole collection locally and create the inverted index. The search engine is powered by the inverted index. Indexer will store the index to files such that we only need to create index once.

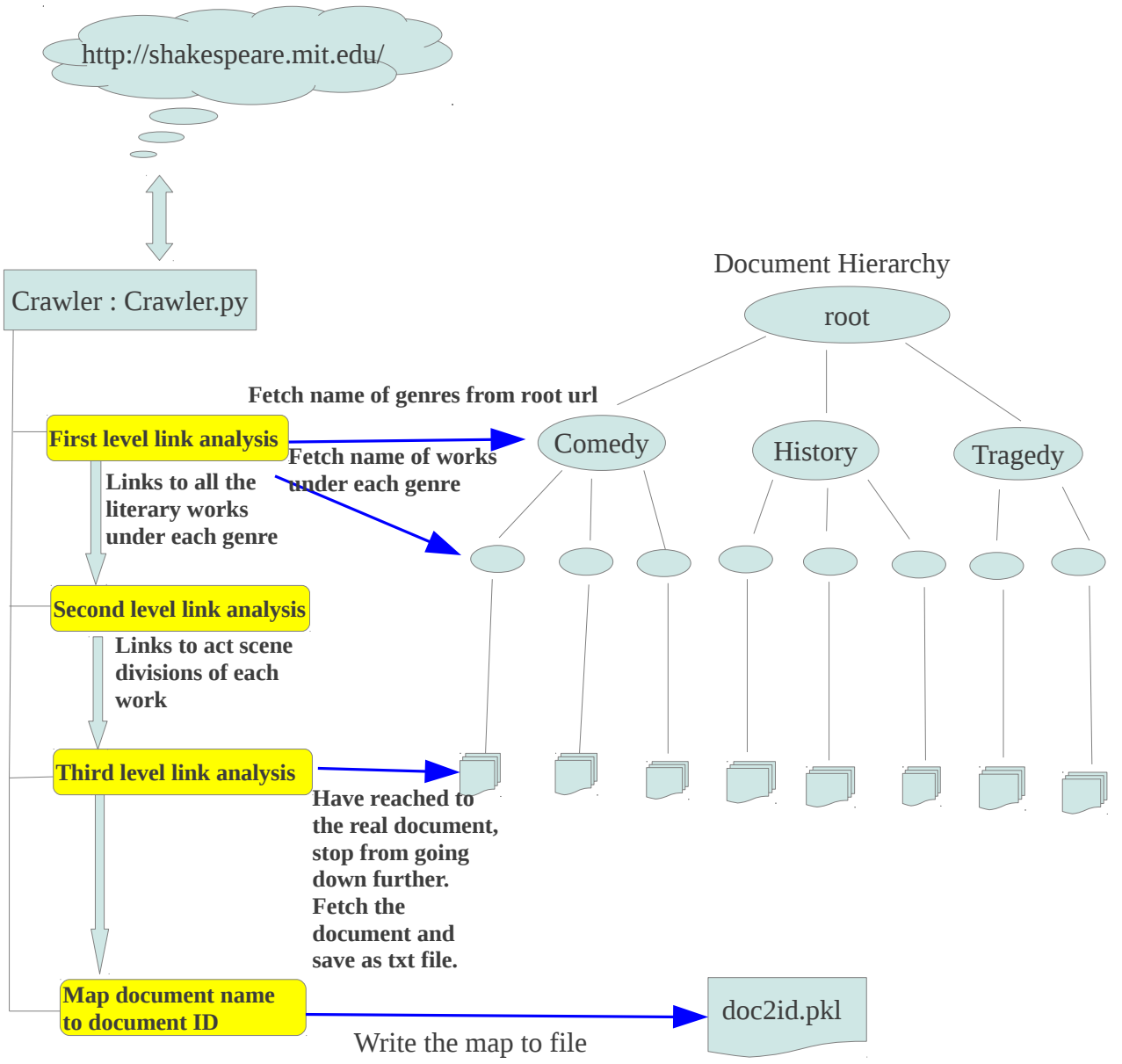
Front-end part contains a command line user-interface. User can enter queries and get results back from query processor who will retrieve the index files and run through a bunch of calculations to figure out the best matching documents.

Anatomy of the system – Implementation details

Crawler

Given the root url http://shakespeare.mit.edu/ as input, crawler will output a collection hierarchy on disk and a file stores data structure mapping document name to id for further usage.

Internally the crawler works as following:

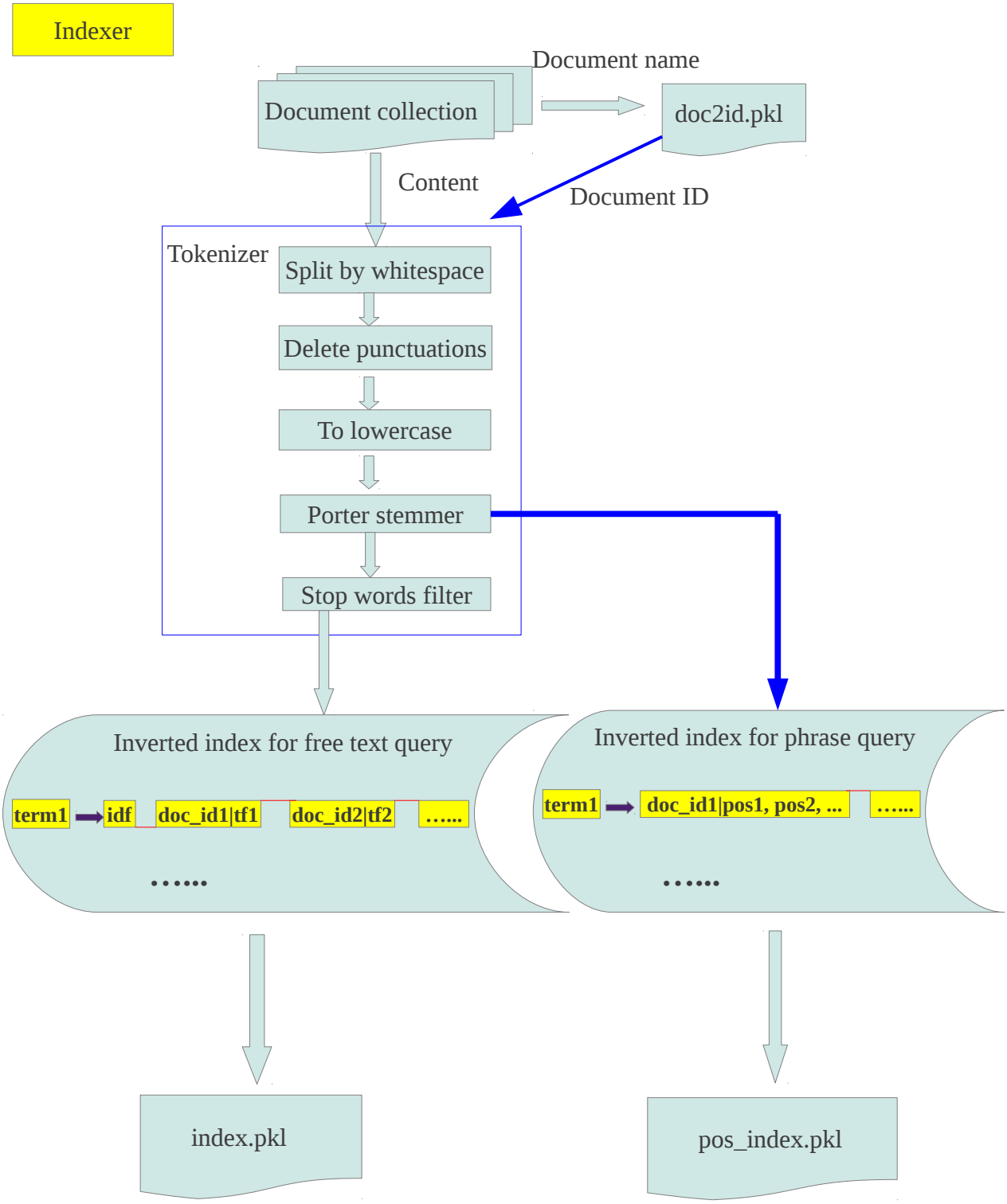


The crawler has been developed only for crawling Shakespeare's whole collection from http://shakespeare.mit.edu. It is not a generic crawler. It works in an iterative way by analyzing each level of links. If the link has nothing to do with the collection, throw it away, otherwise collect the link together and iterate through each link, fetch text related with the link as meta-data and get into it, repeat the same process until we finally reach to the real document. We then fetch the content from html page and save it as text file on disk. The final output of the crawler is a collection hierarchy created on disk and a file saves the map mapping document name to its id which need to used by indexer and query processor.

Indexer:

Taken collection hierarchy and doc2id.pkl as input, indexer will build two kind of index for the search engine. One index is for free text query, while the other one is for phrase query. I will illustrate the structure of both later.

Here is the anatomy of the indexer:



Indexer will iterate through each document of collection to build the index. We have two kinds of index. One is for free text query, the other one is for phrase query. For free text query, we need to

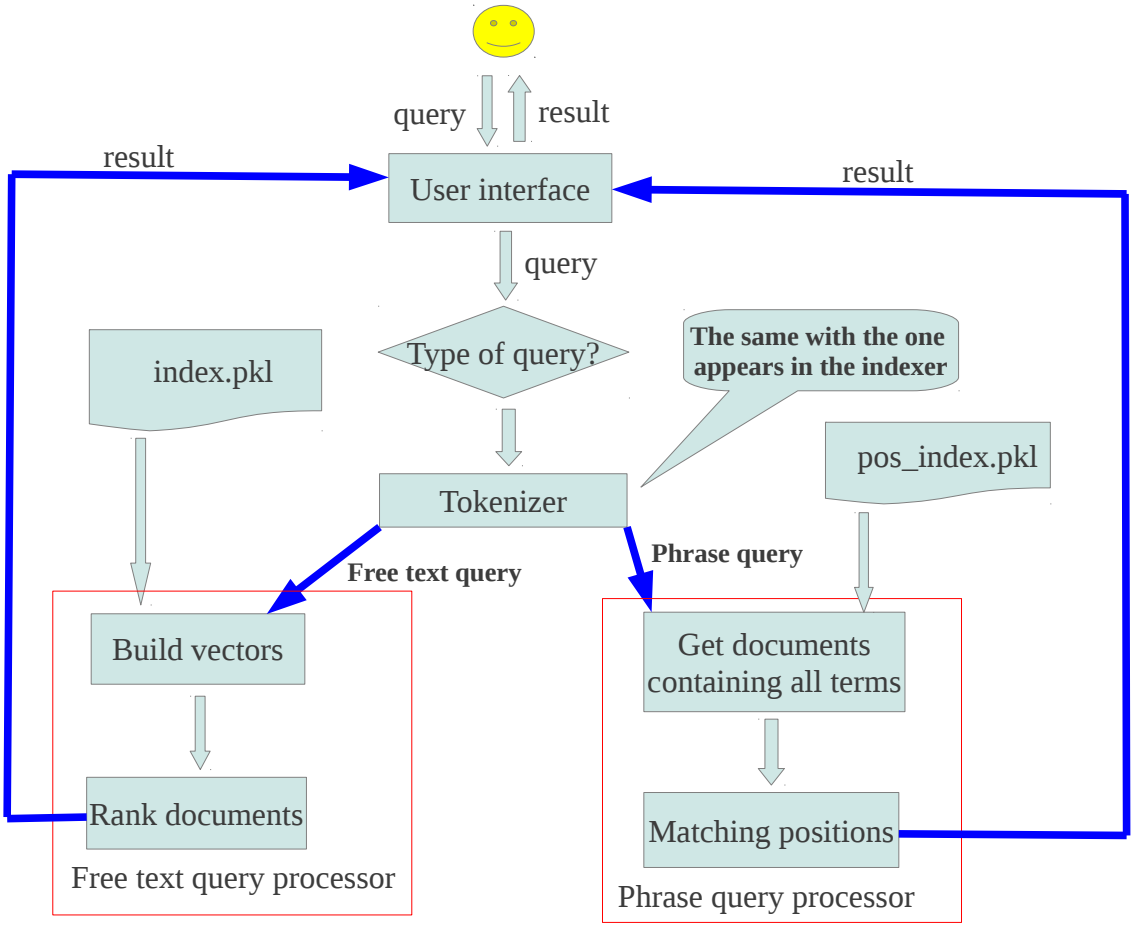
index each term to its posting list, which is a list of (document id, term-frequency) pairs, and in order to compute rank for documents, we also need to store inverse-document frequency before posting list. For phrase query, we need the positional information of each term in a specific document, thus the posting list for the term would be a list of (document id, a list of positions the term appears in) pairs.

The original content of documents need to be normalized by the tokenizer. Notice that for free text query index, the tokenizer will filter out all the stop words. But for the phrase query index, it won't because we want to do query like 'to be or not to be', since all the terms in it are stop words, we definitely do not want to filter them out. The disadvantages for doing so is big index file. For collection smaller like ours, it is not a big deal. But for larger collection, the performance would be dragged down.

The output of the indexer is two index files as shown above.
Another thing worth mentioning is that I do not compress these index files due to time limitation. But it is definitely worth doing it. I will put it in the future work.

Query processor

Here is the anatomy of the query processor.



With a simple command line based user interface, user can type in free text query or phrase query(by surrounding phrase with a pair of single quote, like 'to be or not to be'). The query would then be sent to the tokenizer, which is the same with the one appears in the indexer. If it is a free text query, free text query processor would build vectors for query and each document containing at least one query term by retrieving the index file built for free text qury, then compute cosine

similarities and return the best matching documents to the user. If it is a phrase query, phrase query processor would retrieve the positional index file and get documents containing all terms in the query, then match positions of terms in each document to find out which document contains the exactly the same phrase as user typed in.

Next I will show the algorithm for building the vectors based on the inverted index and the algorithm for matching the positions.

Algorithm for building the vectors:

```
build_vectors(query_terms)
    query_vector ← empty list // initialized to zero, length is the size of vector space
    doc_vectors ← a list of empty list // initialized to zero, length is the size of vector space
    term_index ← 0
    for each term in query_terms
    do
        if term appears in the index
        then
            query_vector[term_index] ← (idf of term)
            for doc in all docs in posting list for term
            do
                doc_vectors[doc][term_index] = (tf of term in doc) * idf of term
            done
        term_index ← term_index + 1
    done
```

Algorithm for matching the positions (inspired by [1]):

input: positional index, document, phrase
(We know document contains all the terms in the phrase, but we don't know if it matches the phrase exactly, so we need to verify it.)

algorithm:

1. For each term in the phrase, get a list of positions where it appears in the document, index these list from 1 to n, where n is the number of terms in the phrase.
2. Keep position list 1 unchanged, subtract each position in position list 2 by 1, subtract each position in position list 3 by 2, and finally subtract each position in position list n by n-1.
3. Sort these lists by length in ascending order, then intersect these list, if the result is not empty list, then document matches. Otherwise it doesn't.

How to use

There are two gzip files.
Shakespeare_search_engine_V3.tar.gz is a full version which means you can run it directly by typing [python query.py] in a unix like shell after uncompressing.

Shakespeare_search_engine_V3_source_only.tar.gz contains only source code. You can run crawler and indexer by typing [./run_crawler_indexer.sh]. You may need to given execution permit to the shell script by typing [chmod +x run_crawler_indexer.sh]. Upon finishing, You'll see some new files created. Under current directory, two log files (crawler_log and indexer_log) have been created in which you can see the log information and three index files have been created, they are doc2id.pkl, index.pkl and pos_index.pkl. If you go one directory back, you'll see a directory called whole_collection, under which you'll see the whole collection hierarchy. Then you can start running queries as I have mentioned before. Enjoy it!

Notice that before running crawler, you may need to install the python package BeautifulSoup first, because it is not a standard python package. Another thing is that it is better to run the program with python 2.7 or later version, because it seems that I have used several modules that are new in 2.7 version, like module 'argparse'.

Query demo & Post analysis

Query examples:

Free text query example:

```
Enter a query:(If you want to quit, type 'q')
quintessence apprehension angel paragon

RANK                                        DOCUMENT(SCORE)
------------------------------------------------------------------------------------------------------------
1     ../whole_collection/Tragedy/Hamlet/Act_2_Scene_2A_room_in_the_castle.txt (1.00)
2     ../whole_collection/Comedy/As_You_Like_It/Act_3_Scene_2The_forest.txt (0.70)
3     ../whole_collection/Comedy/The_Tempest/Act_2_Scene_1Another_part_of_the_island.txt (0.57)

Enter a query:(If you want to quit, type 'q')
```

Notice that I intentionally selected four terms that appear in tragedy Hamlet's Act-2Scene-2A, and the result also showed it as the best match. You can do you own queries to see what happens.

Phrase query example:
Let's do a classic one: 'to or not to be'.

```
Enter a query:(If you want to quit, type 'q')
'to be or not to be, that is the question'

Matching documents:
../whole_collection/Tragedy/Hamlet/Act_3_Scene_1A_room_in_the_castle.txt
```

Needless to say, the result is pretty much what we expected. You can go into the collection hierarchy to verify it.

Source code structure:
There are totally four python files:
crawler.py
indexer.py

PorterStemmer.py
query.py

Except PorterStemmer.py, which I got from [0], I wrote the other three which have a total 546 lines of code by running [cat crawler.py indexer.py query.py | wc -l]

There are totally three iteration I have gone through:
Iteration 1:
Implemented crawler, indexer and query processor. But indexer could not index position information. Query processor could only process free text query.
Iteration 2:
Improved indexer to index position information for each term in each document. Query process could process both free text query and phrase query.
Iteration 3:
Refined some code, add more comments to the code and wrote document for the project.

Time of crawling: 255s
Time of indexing: 37s

summary & future work

summary:

Overall, the search engine is small but complete and functions well. Especially phrase query is very accurate by using the right and efficient algorithm given by [3]. However, our document collection is quite small, but the size of index is more than 3 times bigger than the size of the collection. This is not reasonable. And in terms of the index construction, I used a very simple method, by just going through all the collection once and build a single dictionary to map term to its posting list. For the very large collection, this is not feasible because memory cannot hold huge amount of data at the same time. This means that the scalability of the search engine is not good. Further work needs to be done to build the index phase by phase, and merge the intermediate results into a final index structure. More work also needs to be done to build a web query interface instead of command line interface.

Future work:
combine two indexes as one;
compress index to limit its size and speed up performance;
build a web interface for the search engine;
run against a benchmark to test performance quantitatively.

References

[0] http://tartarus.org/~martin/PorterStemmer/
[1] http://www.ardendertat.com/2012/01/11/implementing-search-engines/
[2] http://www.lextek.com/manuals/onix/stopwords2.html
[3] http://www.rhymezone.com/shakespeare/
[4] http://nlp.stanford.edu/IR-book/