CECS 429 - Project Milestone 1

Milestone demo due October 1.

Overview

In this project milestone, you will piece together and extend many of the search engine features you have already programmed in this class. You may work in teams of up to 3 people, but as you will see, more people in a group will necessitate additional work.

You will program an application to index a directory of files using a positional inverted index. You will ask the user for the name of the directory to index, and then load all the documents in that directory and construct your index with their contents. You will then process the user's Boolean search queries on the index you constructed.

This first milestone will use a text-based interface as in your homework assignments. When processing a query, you must display the name of each document that matches the request, and then allow the user to (optionally) select a document from the result list. If the user selects a document, its **original** text (not processed, stemmed, etc.) should be displayed.

The requirements for this project are enumerated in full detail below. I have also included **TO DO** items that hint at some of the software you will need to program to complete the requirements.

Requirements

These **mandatory** requirements must be completed by all groups.

Corpus:

The official corpus for this project will be the full `all-nps-sites.json` corpus, split into individual document files. However, there will be a future need to support multiple corpora in the search engine, so your program will begin by asking the user to enter a folder/directory to index. You will then build your index with the documents in that folder.

The JSON documents for this corpus are very different than the Moby Dick text documents: their title and content are both pulled from specific keys in the JSON data, rather than the content being the entire byte content of the file. Read items 3(g) and 3(h) from Homework 3 that describe how to create a new `Document` class called `JsonFileDocument` and modify the `DirectoryCorpus` to know how to load our JSON document corpus.

TO DO: Incorporate directory-selection and JSON documents into your application.

Building the index:

Indexes: You will maintain one index for your corpus: the `PositionalInvertedIndex`, a positional index as discussed in class, where postings lists consist of (documentID, [position1, position2, ...]) pairs. Using `InvertedIndex` from Homework 4 as a reference point, create `PositionalInvertedIndex` as a new implementation of the `Index` interface. We will no longer have a need for positionless postings, so the `Posting` class will need to be updated to represent the list of positions of a posting. You will also need to modify `addTerm` to account for the position of the term within the document.

The index should consist of a hash map from string keys (the terms in the vocabulary) to **(array) lists of postings**. You must not use any other hash maps or any **"set"** data structures in your code, only **lists**.

TO DO: Program the `PositionalInvertedIndex` class and incorporate it into the indexing process.

Indexing and tokenization: Using the `DocumentCorpus` and `TokenStream` framework from the homework assignments, index the user's selected directory corpus. Tokenize the text of each document using the `EnglishTokenStream`. To normalize a token into a term, perform these steps in order:

1. Remove all non-alphanumeric characters from the **beginning and end** of the token, but not the middle.
   (a) Example: `Hello.` becomes `Hello` ; `192.168.1.1` remains unchanged.

2. Remove all apostropes or quotation marks (single or double quotes) from *anywhere in the string.*
3. For hyphens in words, **do both**:
   (a) Remove the hyphens from the token and then proceed with the modified token;
   (b) Split the original hyphenated token into multiple tokens without a hyphen, and proceed with all split tokens.
   (So the token "Hewlett-Packard-Computing" would turn into `HewlettPackardComputing`, `Hewlett`, `Packard`, and `Computing`.)
4. Convert the token to lowercase.
5. Stem the token using an implementation of the Porter2 stemmer. Please do not code this yourself; find an implementation with a permissible license and integrate it with your solution.

Do not modify the `BasicTokenProcessor`; write a new implementation that performs these steps. Note that step (3) above **might return more than one term for a single token**; this means that the `processToken` method signature must change, because that method only returns a single String. I leave this task to you.

TO DO: Write a new `TokenProcessor` to handle the above rules. Update the `TokenProcessor` interface to support multiple terms per token.

Query language:

We will support user queries in the form

$$Q_1 + Q_2 + \cdots + Q_k$$

where the + represents "OR", and we call each Q_i a **subquery**, defined as a sequence of **query literals** separated by white spaces. A **query literal** is one of the following:

1. a single token; or
2. a sequence of tokens that are within double quotes, representing a phrase literal

Examples:

- `shakes "Jamba Juice"`
  here, there is one subquery Q_1 with two positive literals (`shakes` and `"Jamba Juice"`). We want all documents with `shakes` and with the phrase `"Jamba Juice"`.
- `shakes + smoothies mango`
  here, we have two subqueries, Q_1 =`shakes` and Q_2 =`smoothies mango`. We want all documents with `shakes` or both `smoothies` and `mango`.

I will give you a sophisticated `BooleanQueryParser` class that is built around the following observation: in the language definition above, the query itself, its subqueries, and its literals all have something in common: they all represent a whole or part of a query, and each whole or part can return a list of postings for the documents that satisfy that part.

We thus create an abstract base class `Query` that represents any whole or part of a Boolean query, which has a single method `List<Posting> getPostings(Index i)`, that returns all postings that satisfy this part of the query. From this class we derive several implementations to represent the various parts of a query:

- `TermLiteral`: one individual term in a query. Since an `Index` can already give the postings for a single term with its own `getPostings` method, a `TermLiteral` simply calls that method on the given `Index`.
- `AndQuery`: a "subquery" from the definition above; to represent the "and"ing of two or more terms, an `AndQuery` holds a list of `Query` objects. To satisfy `getPostings`, the `AndQuery` calls `getPostings` on each of its composed `Query`s, then performs the "AND merge" routine from lecture.
- `OrQuery`: likewise, consists of a list of `Query`s, and answers `getPostings` by getting the postings of its components and performing the "OR merge" routine.

And then we program `BooleanQueryParser.parseQuery`, which takes a string query and returns a `Query` object that represents the entire query structure, by parsing each part of the query and wrapping those parts

inside appropriate `Query` types. The object returned by `parseQuery` can then be "executed" by calling its `getPostings` method, passing the `PositionalInvertedIndex` you constructed earlier in the program.

Like the other "starter" assignments, the code I'm giving you is incomplete. `BooleanQueryParser` currently recognizes **individual terms** as `TermLiterals`, as well as AND and OR queries... but it does not recognize **phrase literals**. If you give it a query like `shakes "Jamba Juice"`, the parser will *currently* interpret that as an AND query with three individual terms: `shakes`, `"Jamba`, and `Juice"`. You will need to fix this by upgrading `BooleanQueryParser.findNextLiteral` to detect the start of a phrase literal, constructing and returning a `PhraseLiteral` object instead of a `TermLiteral` when appropriate.

Once you can parse phrase literals, you will need to finish the `getPostings` methods of all the `Query` types (except `TermLiteral`). `AndQuery`/`OrQuery` need to program their "merge" routines; `PhraseLiteral` needs to perform the more difficult positional merge from lecture, with assuming a limit to the number of terms in the phrase.

Finally, `TermLiteral` assumes that the literal found in the query is the exact term to retrieve from the index... but your indexer will perform normalization on the tokens of the corpus. You must upgrade the query parsing system so that a `TokenProcessor` is made available when calling `getPostings`, so that `TermLiteral` (and `PhraseLiteral`) can normalize the query token before going reading from the index. **Note**: do **not** perform the "split on hyphens" step on query literals; use the whole literal, including the hyphen.

You may assume that the user always types in a properly-formatted query, and do not need to do error checking or reporting. Note that this query language does not support the `NOT` operator. It also does not support queries of the form `smoothies (mango + banana)`.

TO DO: Download `QueryFoundations.zip` and incorporate its code into your application. Finish `BooleanQueryParser` and its related classes:

- Update `findNextLiteral` to recognize and construct `PhraseLiteral` objects.
- Somehow incorporate a `TokenProcessor` into the `getPostings` call sequence.
- Write the `getPostings` methods of `AndQuery`, `OrQuery`, and `PhraseLiteral`.
- Integrate `BooleanQueryParser` into your application.

Special queries: Your engine must also handle a few "special" queries that do not represent information needs, but instead are used for administrative needs. If the user query starts with any of these strings, then perform the specified operation **instead** of doing a postings retrieval.

- `:q` – exit the program.
- `:stem` *`token`* – take the token string and stem it, then print the stemmed term.
- `:index` *`directoryname`* – index the folder specified by *`directoryname`* and then begin querying it, effectively restarting the program.
- `:vocab` – print the first 1000 terms in the vocabulary of the corpus, sorted alphabetically, one term per line. Then print the count of the total number of vocabulary terms.

You should special-case these special queries – don't try to work them into the `BooleanQueryParser` system, simply check the user's input to see if it matches one of these queries, and then dispatch to the parser if it does not.

TO DO: Implement an architecture for the special queries.

Main application:

Your search engine application must behave in the following way:

1. At startup, ask the user for the name of a directory that they would like to index, and construct a `DirectoryCorpus` from that directory.
2. Index all documents in the corpus to build a positional inverted index. **Print to the screen how long (in seconds) this process takes.**
3. Loop:

(a) Ask for a search query.
   i. If it is a special query, perform that action.
   ii. If it is not, then parse the query and retrieve its postings.
      A. Output the names of the documents returned from the query, one per line.
      B. Output the **number of documents** returned from the query, **after** the document names.
      C. Ask the user if they would like to select a document to view. If the user selects a document to view, print the **entire content** of the document to the screen.

TO DO: Using the `Indexer`-type applications from previous homeworks as an example, build the main application. Your application should be programmed to **abstract interfaces only** – although you certainly need to construct a `PositionalInvertedIndex` at some points, all the methods of your application should only refer to the `Index` interface after the point of construction. **Speak to me early if you don't know what this means.**

Additional Requirements

Each of the following additional requirements are worth a certain amount of points. Your group must select and implement additional features worth **at least** P **total points** from the following list, where P is equal to...

- **two times** the number of people in your group...
- **plus** 2, if *at least one* member of your group is enrolled in CECS 529.

Options:

Wildcard queries - 4 points: Amend the query parser to support wildcard literals with arbitrary numbers of * characters representing wildcards. To support this feature, edit the definition of **query literal** above to:

1. a single token
2. a sequence of tokens that are within double quotes, representing a phrase literal
3. a single token containing one or more * characters, representing a wildcard literal

Add a new `WildcardLiteral` query component type, and upgrade `findNextLiteral` to construct a wildcard object if the literal you locate contains at least one `*` character.

Implement 1-, 2-, and 3-grams for each vocabulary **type** in the vocabulary. Represent the k-gram index with a new class. The design of this class is up to you.

`WildcardLiteral`'s `getPostings` implementation is tricky: it must generate the largest k-grams it can for its literal, then retrieve and intersect the list of vocabulary types for each k-gram from the k-gram index. (You will have to architect a means for `WildcardLiteral` to have access to the k-gram index.). The intersected list contains all candidate type strings; implement a post-filtering step to ensure that each candidate matches the wildcard pattern. OR together the postings for the processed term from each final wildcard candidate and return them as the postings list for the `WildcardLiteral`.

You should be able to use wildcard literals inside of phrase queries, as part of a NOT query, or part of a NEAR query, depending on the additional requirements you select.

Your own corpus - 3 points: Choose a corpus of documents to use with your search engine. Download (or otherwise retrieve) your chosen corpus, which must contain at least 10000 documents totaling at least 10 MB of uncompressed disk space. Your corpus **must** require either a special token processor, or a special `Document`-derived class for determining the content of the documents.

You must submit a **corpus proposal** to me by September 18 in order to select this requirement. Your proposal must contain: a description of the corpus you would like to use; *why* you would like to use this corpus; how you will retrieve the corpus; any special processing rules needed when tokenizing the corpus.

Foreign language indexing - 3 points: Once again, I am an ignorant American, so "foreign" here means non-English. Expand your search engine to handle non-English language documents and queries. The language

you choose must have different tokenization/normalization rules than the simple rules we use for English language, and you must program your engine to handle those rules. Example: you may choose German if you design a software module to split compound nouns, so that *Computerprogrammierer* is indexed separately as *computer* and *programmierer*. (You will probably have to do your own research into how this is done.) Your engine must still operate on English documents so I can test it, and you may use two different "modes" for operating on English or non-English collections if desired.

You must submit a summary of your language of choice, the issues that make it harder than indexing English, and a summary of how you will solve those issues when you turn in your project proposal.

Note that you will need to also complete "Your own corpus" for this option to really make any sense.

Expanded query parser - 3 points: Our base Boolean query language only supports disjunctive normal form (DNF) queries: all queries are "ORs of ANDs", being one or more AND queries joined with ORs.

Tweak your query parser and execution engine to redefine the term **query literal** as:

1. a single token
2. a sequence of tokens that are within double quotes, representing a phrase literal
3. a pair of parentheses surrounding a DNF query.

Your engine now supports queries like `smoothies (mango + banana) + "(vanilla + chocolate) shakes"`, for someone wanting to know about smoothies that have either mango or banana, OR wants to find the phrases "vanilla shakes" or "chocolate shakes".

Upgrade `findNextLiteral` to detect an opening `(` signifying a DNF query. Locate the next `)` in the query and then parse the string between the parentheses as a **full query**. Return the Query you get back (represnting the DNF query inside the parentheses) so that it can be incorporated into the rest of the parsed query.

Soundex algorithm - 3 points: Read the textbook chapter 3.4 regarding phonetic correction using the "soundex algorithm," then implement the algorithm so your search engine's users can search Boolean queries on the **authors** of documents in your corpus, using tolerant retrieval for misspelled names via the soundex algorithm.

The National Park Service corpus does not have author information in its documents. To complete this option, you will need to download a a different corpus: `mlb-articles-4000.zip` from BeachBoard, containing articles about Major League Baseball. The JSON format for these articles contains an `author` field.

Create a class to represent a soundex index. A soundex index maps from soundex hash keys (four-letter strings) to document IDs that contained a string which produced the hash. When indexing a document, after you have processed all the tokens in the body of the document, process the `author` of the document as well, by hashing each term in the author field (the author's first and last names, presumably) using soundex, and inserting a posting into the soundex index.

When querying a corpus, allow the user to search the author soundex index by specifying a special query of `:author _______`. A single term may follow `:author`, and that is the term you should hash and retrieve postings for. Print all documents that matched the author query, as well as document's author him/herself.

NOT queries - 2 points: Amend your query language to support the boolean NOT operator as a minus sign ("`-`"). To support this feature, add a new query component class `NotQuery`, which wraps/owns a single other `Query` object, and returns that component's postings as its own.

The only "NOT" query that makes sense is "AND NOT", so the `AndQuery` class will be responsible for performing the actual "AND NOT merge" routine. You will need to expand the definition of `Query` to indicate whether a component is a "positive" or "negative" query. Only `NotQuery` objects are negative (all others are positive), and the `AndQuery` can then be programmed to do an "AND NOT" merge if one of its child components is a negative query.

You must upgrade `findNextLiteral` in `BooleanQueryParser` to detect NOT components by looking for a term that starts with "`-`". Use recursion to extract the next literal following the `-` and wrap it inside a

`NotQuery` component. If you do this correctly, you will support NOT over both term literals and phrase literals (and any other literal you support, like NEAR or wildcards).

Your implementation must not assume the order of the negative component relative to the positive, i.e., you must support a query like "`-shakes dairy`". (Hint: "AND" is commutative, and you can reorder the components of an AND query without affecting its output.)

Unit testing framework - 2 point: Implement a unit testing framework for portions of your search engine. Using a style and method appropriate to your language of choice, design, implement, and run unit tests to verify the correctness of the following portions of your search engine:

1. **Positional inverted index**: write a set of 5 simple text documents with a combined vocabulary of 10-15 words. Build a positional inverted index **by hand**. Write unit tests to constuct an index over your documents, then verify that the positional postings match the expected output based on your hand-built index. You do not need to verify each word in the vocabulary, but should be sure to test words that occur more than once in a particular document.
2. **Query processing**: using your example corpus from above, write unit tests to verify the correctness of your search engine results. Write tests for each kind of query you have to implement: phrase queries, AND queries, OR queries, and NOT queries (if you chose this option). Verify the results match your expected outputs by solving the queries by hand. Make sure you write queries for which there is no result set, and queries for which all documents are relevant.
3. **Other modules**: you must also write tests for the other optional modules you have selected for your project, appropriate to each module.

Graphical user interface - 1 point for simple desktop UI - 2 points for Web UI: Instead of a text interface, program a graphical user interface. When the program starts, show a dialog that lets the user select a directory to index. When the index is complete, show a text field for the user to type their queries. Upon submitting the query, show a list of document titles that match the query. When the user double-clicks a result in the list, open a new window showing the contents of the document.

You must provide GUI-appropriate alternatives to the "special queries" mentioned in the required features.

NEAR operator - 1 point: Implement the NEAR/K operator. To simplify the parsing of NEAR operators, we will require the entire operator to be in square brackets, as in `[baseball NEAR/2 angels]`. Given a value k, do a positional merge between the terms to the left and right of the NEAR operator, selecting documents where the second term appears at most k positions away from the first term. (We will say that the order of the terms matters: "angels love baseball" will match the query `[angels NEAR/2 baseball]` but not `[baseball NEAR/2 angels]`.)

Create a new query component type `NearLiteral`, representing the three components of a NEAR query (the first token, the value of k, and the second token). Upgrade `findNextLiteral` to recognize the `[` character and build this new literal type.

Biword index - 1 point: When indexing the corpus, additionally build a biword index. When the user enters a phrase query literal, use the biword index to satisfy the `PhraseLiteral`'s postings if and only if the phrase query contains only two terms; use the positional index otherwise. The biword index can be a nonpositional index, that is, you will record document IDs for each biword entry, but not the position of the biword in the document.

Your own interests - depends on the proposal: Are you interested in something else that we've talked about in lecture? Maybe you want to do some research and implementation on your own? Come chat with me and let me know.

Summary

I realize this is a long document and it may be hard to keep track of what's required. Here is a brief list of important things to keep in mind. (This is NOT exhaustive, you are ultimately responsible for everything in this document even if it isn't listed in this summary.)

You must:1

1. Build a positional inverted index over a user-specified directory of text files.
2. Process tokens from files with special rules; and use similar rules for user queries.
3. Finish the Boolean query processing module so you can handle queries from the user and merge postings lists for the query.
4. Support phrase queries with quotation marks via the positional index.

You must choose additional features to implement according to your group size.