Project 2: Search
Due: 5:00PM, March 23, 2018

Contents

1 Introduction 1

2 Indexer 2
   2.1 The Wikis 3
   2.2 XML Primer 4
   2.3 Parsing the XML 5
   2.4 Tokenizing 6
   2.5 Stop Words 7
   2.6 Stemming 8
   2.7 TF IDF 8

3 PageRank 10
   3.1 PageRank Example 14
   3.2 Testing PageRank 15

4 Representation 15

5 Querier: A REPL 16

6 Assignment 16

7 Testing 17
   7.1 Documentation 17

8 Helpful Hints 18

9 Handin 18
   9.1 Design Check 18
   9.2 Final Handin 19
   9.3 Grading 19

10 Extra Features (Optional) 20
1 Introduction

You are probably familiar with a few search engines: Google, Bing, Duck Duck Go. For this project, you're going to be writing your own search engine! Pretty cool, for an intro course, right?

Specifically, your search engine will allow users to query a corpus of Wikipedia pages. The complete collection of all Wikipedia pages is rather large, so you will be working with a small subset we created for you.

Your search engine will work a lot like a typical search engine. A user will type in what is called a free text query (i.e., some words), and your program will return what seems to it to be the most relevant pages for that query.

One naive way to build a search engine would be to ask the user to enter a query, and then scan through the corpus looking for documents that match that query. This approach would require many many traversals of the corpus—one per query. Imagine trying to do this for every single web query! It would not be fast.

Instead, most modern search engines consist of two programs, an indexer and a querier. The indexer is the preprocessor; it preprocesses the corpus before answering any queries. But how can it do this without knowing what the queries will be? Easy, it precomputes a measure of how relevant every term is to every document! Yes, this takes a long time; but remember, this is a preprocessing step. So, if it takes one hour, or even one day, it does not negatively impact the user experience, because it does not affect query response time.

Note: Once again, we see an example of a time vs. space trade-off. While preprocessing the data requires enough space to store a measure of how relevant every term is to every document, a naive search engine (i.e., one that does not preprocess) would take an unacceptable amount of time.

The indexer writes the information it gleans about the corpus to disk, in files called index files. Then, when the querier receives a query from a user, it uses the information contained in the index file(s) to respond to that query, quickly.

Your search engine will consist of exactly these two components: an indexer, which preprocesses the corpus and writes the indexes it builds to some files, and a querier, which reads the files written by the indexer, and then answers queries posed by users. Once built, an example run of your search engine might look like this:

```
$ scala search.sol.Index /course/cs018/src/search/MedWiki.xml titles.txt index.txt
$ scala search.sol.Query titles.txt index.txt
search> cats
  1 Kiritimati
  2 Kattegat
  3 Lynx
  4 Morphology (linguistics)
  5 Kylie Minogue
  6 Fable
  7 Northern Mariana Islands
```

---

1 A corpus is a large collection of texts.
2 It consists of over 5,000,000 pages, in English alone!
3 Or pretty!
8 Mercalli intensity scale
9 W. Heath Robinson
10 Nebula
search> ruler
1 Mohism
2 Michael
3 Manasseh
4 Monarch
5 Islamabad Capital Territory
6 Jadwiga of Poland
7 Isabella d'Este
8 Empress Suiko
9 Henry the Fowler
10 Government
search> :quit

Hint: It is possible (likely even) that you will encounter a Java OutOfMemoryError while running your indexer. If you do, try re-running your program with the argument -J-Xmx512m:

```
'scala -J-Xmx512m search.sol.Index <arguments>'
```

The -J means it’s an argument for your JVM, the -Xmx tells the JVM how much memory it can use, and 512m means 512 megabytes. The maximum amount of memory you are permitted to use on a department machine is -Xmx2g, meaning up to 2 gigabytes of memory. But be forewarned; allowing the JVM to use this much memory can drastically slow down your machine’s other processes.

2 Indexer

Once again, the indexer preprocesses the corpus to create index files, which store information about the relevance of terms to documents. There are a number of steps required in this preprocessing, including:

1. Parsing the corpus to extract the text.
2. Tokenizing the text into tokens, which are mostly words, but include other things like numbers (e.g., 171819) as well, with all punctuation and white space removed.
3. Filtering out stop words, the most common words in the language.
4. Stemming the tokens into terms, which means converting them to lower case, and then dropping any part of a word that is not at its root: e.g., the stem of “stemming,” “stemmer,” “stems,” etc. is “stem” itself.
5. Scoring the relevance of each relevant (more on that later) term that appears in a document to that document. For example, the term “cat” might be relevant in a document about Convention Against Torture.

We will explain each of these steps in turn, but first, let us remind you that the purpose of building an index is to expedite your search engine’s response time to queries. So do not worry if your indexer
is relatively slow at indexing a very large corpus. We’d rather see your indexer take 15 minutes to build an index, with querying taking a mere 0.5 seconds, than your indexer take 5 minutes, and querying 1.5 seconds. That said, efficiency is definitely a strong priority throughout the entire project!

2.1 The Wikis

The corpus of Wikipedia pages you’ll be searching over is located in /course/cs018/src/search/src. The three main Wikis are called SmallWiki.xml, MedWiki.xml, and BigWiki.xml. We have also provided a PageRankWiki.xml as a basic PageRank test; more explanation coming below. The .xml extension indicates that they are all XML files, which we will also explain momentarily.

But first, if you list the contents of /course/cs018/src/search/src using ‘ls -l’, you can see the sizes of these files:

$ ls -l
-rw-rw-r-- 1 amy cs018ta 216466779 Mar 17 2015 BigWiki.xml
-rw-rw-r-- 1 amy cs018ta 24770652 Jan 17 2015 MedWiki.xml
-rw-rw-r-- 1 amy cs018ta 1667422 Jan 17 2015 SmallWiki.xml

To avoid copying these large files into your own directories, you should instead use symbolic links to access them. To do this, navigate to whichever directory you would like put them in and use the following command to create a (symbolic) link to the files:

‘ln -s /course/cs018/src/search/src/<filename> <choose your link name>’

The third argument to the ‘ln’ command is optional. If you leave it off, the link name will be set to the filename by default. For example, the following command will create a (symbolic) link called BigWiki.xml in the current directory, and that link will link to the file /course/cs018/src/search/src/BigWiki.xml:

‘ln -s /course/cs018/src/search/src/BigWiki.xml’

After executing this command, a directory listing (‘ls -l’) should include something like this:

lrwxrwxrwx 1 amy amy 37 Mar 1 12:00 BigWiki.xml -> /course/cs0180/src/search/src/BigWiki.xml

2.2 XML Primer

Programs often need to pass information to one another, and the most convenient way to do so is to use a data-interchange format, meaning a standardized text-based format. XML, which stands for eXtensible Markup Language, is one such format.

Invented by the World Wide Web Consortium (W3C), XML uses an HTML-style angle bracket specification. Each XML element is encompassed by matching opening and closing tags. Opening tags are delimited by angle brackets, like this: <tag>. Closing tags include a slash, like this: </tag>. Text appears between the opening and closing tags of an element:
An XML element can have any number of attributes, which are specified within the opening tag, as follows:

```xml
<zoo name="Bronx">
    <animal type="gorilla"></animal>
    <animal type="giraffe"></animal>
</zoo>
```

The zoo element has the attribute "name" with value "Bronx". The animal tags have the attribute "type" with values "gorilla" and "giraffe". The value of an attribute is always enclosed in double quotes.

Note that the two animal elements are nested inside the zoo element. As this nesting reflects the tree structure of XML elements, the animal elements are called children of the parent element, zoo.

Pages  The layout of our Wiki files is simple. Each consists of a single XML element tagged xml, within which there are a number of pages (a.k.a documents), each of which is enclosed by opening and closing page tags. Each page then contains a title, a unique, non-negative integer id, and some text, all of which are delimited by tags with those names.

Links  Links appear in the Wiki files in one of three forms:

- `[[Hammer]]`
- `[[Presidents|Washington]]`
- `[[Category:Computer Science]]`

If a link consists of two parts separated by a pipe then the text before the pipe is the page to link to, while the text after the pipe is the text to appear on the page. So, for example, the page on which `[[Presidents|Washington]]` appears would include the text Washington with a link to the Presidents page. If a link appears without a pipe or colon, then its text is both the page to link to and the text that appears on the page. If a colon appears as part of the link, as in "Category:Computer Science", then the link is to the meta-page for that namespace. A meta-page is one that consists of links only (in this case to the pages in the Category Computer Science). A namespace is a collection of pages with a common purpose. Other common namespaces include Help, File, and User.

Note: When parsing text, you should consider the links on pages as words. In our examples, Hammer should be used to compute relevance; so should Washington, although Presidents should not be; and both Category and Computer Science should be used.

2.3 Parsing the XML

When a program receives data in a standard text-based format like XML, its first job is to convert it into a useable form. How do we convert a string of characters (i.e., text) into an in-memory data structure that is useful for further processing? The answer to this age-old question is parsing.
Indeed, you will need a way to extract the text from the Wikis. As already mentioned, XML elements nest, so that XML documents naturally form a tree-like structure. Moreover, Scala has an XML library, which allows you to treat the XML elements as nodes, and easily extract their data and children.

To use this library, import the XML Node class at the top of the file in which you plan to parse the Wikis:

```scala
import scala.xml.Node
```

Then, to access the main XML node, you use `loadFile` like this:

```scala
val mainNode: Node = xml.XML.loadFile("SmallWiki.xml")
```

Next, to access all the immediate children of the main node (or some other node), use:

```scala
val children: Seq[Node] = node.child
```

Alternatively, to extract specific children use the selector `""`. For example, to select all the children with the tag "page", use:

```scala
val pageSeq: NodeSeq = node "page"
```

You can also further select all the children with the tag "id" like this:

```scala
val idSeq: NodeSeq = node "page" "id"
```

Alternatively, you can do this same deep search (i.e., search through a node’s children, grandchildren, etc.), using the selector `\`:

```scala
val idSeq: NodeSeq = node \ "id"
```

Observe that these selectors return an object of type `NodeSeq` (not to be confused with `Seq[Node]`). If there is only one node in this sequence, then `pageSeq.text` or `idSeq.text` returns that node's contents. If there is more than one node in this sequence, then calling `.text` returns the concatenation of the text of all the nodes in the sequence.

You can find a short example of a Scala program that uses this library here: /course/cs018/src/search/src/NodeSample.scala. In this example, you will observe that XML syntax is valid Scala syntax, and that it is automatically converted into an XML node. Moreover, a pair of opening and closing curly brackets “escapes” the XML, so that you can write Scala code between the brackets (e.g., `{ otherNode }`).

### 2.4 Tokenizing

The next task after parsing the XML is tokenizing the corpus, which means converting the text of the documents into tokens. Tokens are sequences of characters, sans all punctuation—squiggles, brackets, what have you—and whitespace.

One utility you might find useful when tokenizing is `split`. In Java, we used the `split` method to break up a string into smaller parts based on a character or character sequence, such as , or .. This method also works in Scala. For example:

```scala
"the quick fox ran over the fence".split(" ")
=> Array("the", "quick", "fox", "ran", "over", "the", "fence")
```

4 the Escape character
It is also possible to express this same command using a regular expression, as follows:

```javascript
"the quick fox ran over the fence".split("\s") => Array("the", "quick", "fox", "ran", "over", "the", "fence")
```

More generally, if you wanted to split up a string based on one or more whitespace characters, you could do so as follows:

```javascript
"the quick fox ran\n over the fence".split("\s+") => Array("the", "quick", "fox", "ran", "over", "the", "fence")
```

For more information on this, check out lab07 and the 'Regex' document on the course homepage!

The regular expression that we suggest you start with when tokenizing is:

```
"[\\[\[^[^\[]+?\]\]\]|^[^W_]+'[^W_]+|^W_+"
```

Well, that looks kinda disgusting. What could it all mean?

Let’s break it down. There are three parts, "[\\[\[^[^\[]+?\]\]\]|^[^W_]+'[^W_]+|[^W_]+", separated by pipes (|), which mean “or.” So we are actually matching three possible alternatives:

1. "[\\[\[^[^\[]+?\]\]\]"
   - The meaning: Match two left brackets ("[" and two right brackets ("]")", making sure there’s something in the middle, but also making sure there is not a left bracket in the middle, which would mean that somehow another link was starting inside this link ("[^\[]+?").
   - Returns links (e.g., “[[Some Wiki Page]]”)

2. "^[^W_]+'[^W_]+"
   - The meaning: Match at least one alphanumeric character ("^[^W_]+"), then an apostrophe ("'"), and then at least one alphanumeric character ("[^W_]+").
   - Returns words with apostrophes (e.g., “don’t”)

3. "[^W_]+"
   - The meaning: Match at least one alphanumeric character.
   - Returns words (e.g., “dog”)

In summary, this pattern matches links, words with apostrophes, and words without apostrophes.

**Note:** This is by no means the only regular expression that you can use. Feel free to modify it or create your own from scratch if you want to tokenize in some other way! For example, if you don’t want to match numbers, replace each instance of [^W_] in the regular expression with [^W_d]. You can also split this regular expression up into its parts, so that you can search for only links (using only the first part), or only words (using the second two parts joined with a pipe).
Regex  Scala has a built-in regular expression class, `Regex`, that you can use to search through a string for constituent patterns. Here’s an example, showing how you might do this.

First, create an instance of a Scala `Regex`, like this:

```scala
val regex = new Regex("\[\[\[\^\[]\[?\]\]\]\\|[\^\W_]+'\[\^\W_]+|[\^\W_]+")
```

Note that the regular expression is wrapped in triple quotes! This syntax allows us to express it using fewer characters, as we avoid needing to escape certain special characters.

Next, use the method `findAllMatchIn`, which takes as input a `String`, to find all the matches within that `String`. This method actually returns an iterator over `Match` objects, which contain the substrings matched by the regular expression, as well as some metadata we can here discard (groups, start index, etc.). We then traverse these `Matches` to extract the matched substrings.

```scala
// Call findAllMatchIn to get an iterator of Matches
val matchesIterator = regex.findAllMatchIn(someString)

// Convert the Iterator to a List and extract the matched substrings
val matchesList = matchesIterator.toList.map { aMatch => aMatch.matched }
```

2.5 Stop Words

Some words appear quite often in a language, but contribute little to meaning: words like “the”, “in”, “an”, or “or”. Such words are known as stop words.

In building a search engine, it makes sense to discard stop words (in both indexing and querying). Let’s look at an example to see why, say “the cat in the hat.” Is there relevant meaning/content in the stop words “the” and “in” in this query? Not so much. In reality, the shorter query “cat hat” gets at the essence of the longer query.

To help you identify stop words, we’ve provided you with a `StopWords` object, which exposes a single method, `isStopWord` that takes as input a `String` (i.e., a single word), and returns `True` if the word is a stop word, and `False` otherwise.

2.6 Stemming

The next issue to overcome when building a search engine is that (in spite of what we have suggested thus far), we are not really interested in computing a score for each word in each document; rather, what we want is a score for the root of each word in each document. For example, if you were searching for a place to buy some cat toys, you would probably want the page Cat Emporium to be included in your search results when you enter the query `store cats`. The way to accomplish this is to equate "cat" and "cats" in your scoring function, so that there is a single entry for the root of each word, rather than multiple entries for words with the same root.

We have built a `PorterStemmer` class for you that stems words, meaning reduces them to their roots (as best we can).\footnote{You can find it in the usual place: /course/cs018/src/search/src.}
The \texttt{PorterStemmer} class extends the \texttt{Stemmer} trait, which exposes two methods. The first, \texttt{stem}, takes as input a string of words and returns that same string, but with the words stemmed. The second, \texttt{stemArray}, takes as input an array of strings, each consisting of a single word, and returns an array of those same words, again with the words stemmed.

The output of the stemming process are called \textit{terms}. It is terms that are scored in relation to documents when building measures of relevance for a search engine.

### 2.7 TF IDF

Processing documents as described (parsing, tokenizing, and stemming) converts documents from text into a sequence of terms. Likewise, a query (after parsing, tokenizing, and stemming) is a sequence of terms. Consequently, to measure the relevance of a document to a query—that is, to score the document—it suffices to compare these two sequences of terms. There are numerous \textit{similarity metrics}, based on multiple factors (e.g., the number of terms in common; how many times those terms appear in each; how common or unusual those terms are; etc.) that can be used to carry out this comparison. We describe a particularly simple one here, but you are encouraged to consider other more sophisticated metrics as time permits.

There are two key ideas that the similarity metrics used by most practical search engines capture:

- A term that appears many times within a document is likely to be more relevant to that document than a term that appears fewer times.
- A term that occurs in a fewer documents in the corpus is likely to be a better signal of relevance when it occurs than a term that appears in many documents.

To capture this first idea, we will conceptualize each document in our corpus as a \textit{vector}. You might be familiar with vectors as quantities with both magnitude and direction. Indeed, we will represent a document as an $m$-dimensional vector, where $m$ is the number of terms in the corpus.

Then, implementing the first key idea, we will populate this vector with counts: i.e., the number of times each term $i$ appears in each document $j$, denoted $c_{ij}$. These counts are our first attempt at measuring a term $i$'s relevance to a document $j$.

This vector may be a useful representation of one document, but it is difficult to compare documents using this representation, because length can vary dramatically across documents. One document that mentions the terms “one-eyed one-horned flying purple people eater” five times and is very short might in fact be very similar to another which mentions these terms 5,000 times but is very long. Consequently, it is necessary to normalize these counts somehow.

One simple way to normalize is to divide each count by the maximum number of times that any term appears in a document. Specifically, let $a_j = \max_i c_{ij}$. We then define the \textit{term frequency} as follows: $tf_{ij} = c_{ij}/a_j$. Term frequency is a better measure of a term’s relevance to a document than straight up counts.

To capture the second idea, we will scale up the term frequency of less frequent words in the corpus. One simple scale is $n/n_i$, where $n$ is the number of documents in the corpus, and $n_i$ is the number of documents in which term $i$ occurs. For example, if $n$ is 1000, and the term

\footnote{A popular one is cosine similarity.}
\footnote{This is an inefficient representation, and cannot be used in practice! But as always, correctness before efficiency.}
‘supercalifragilisticexpialidocious’ occurs in only 10 documents, its count would be scaled by 100. In contrast, a term like ‘apple’, assuming it occurs in far more documents, would be scaled by a smaller value (if at all).

In fact, a more useful scaling factor is this one, which is called inverse document frequency (idf):

$$\text{idf}_i = \log \frac{n}{n_i}$$

This scaling factor captures the same intuition as our the original, but is less sensitive to large values of $n/n_i$.

We can combine these two ideas to measure the similarity between a query and a document, which we interpret as the relevance of the latter to the former:

$$s_{Q,j} = \sum_{i \in Q} tf_{ij} \cdot \text{idf}_i$$

In words, this metric scores a document $j$ given a query $Q$ by summing up that document’s scaled, normalized counts for all the terms in that query.

As an (extremely simple) example, consider a corpus consisting of the following three documents:

- the dog bit the man
- the dog ate the cheese
- the cheese bit the cheese

Our corpus then contains the words \{“the”, “dog”, “bit”, “man”, “ate”, “cheese”\}. We then convert our documents to their vector representations:

- \{2, 1, 1, 0, 0\}
- \{2, 0, 0, 1, 1\}
- \{2, 0, 1, 0, 2\}

With this information, we can compute the term frequencies $tf_{ij} = c_{ij}/a_j$:

<table>
<thead>
<tr>
<th></th>
<th>$i_0$ (the)</th>
<th>$i_1$ (dog)</th>
<th>$i_2$ (bit)</th>
<th>$i_3$ (man)</th>
<th>$i_4$ (ate)</th>
<th>$i_5$ (cheese)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$j_0$</td>
<td>1</td>
<td>1/2</td>
<td>1/2</td>
<td>1/2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$j_1$</td>
<td>1</td>
<td>1/2</td>
<td>0</td>
<td>0</td>
<td>1/2</td>
<td>1/2</td>
</tr>
<tr>
<td>$j_2$</td>
<td>1</td>
<td>0</td>
<td>1/2</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Next, we compute the inverse document frequencies for each word $i$ using $\text{idf}_i = \log \frac{n}{n_i}$:

<table>
<thead>
<tr>
<th>idf</th>
<th>$i_0$</th>
<th>$i_1$</th>
<th>$i_2$</th>
<th>$i_3$</th>
<th>$i_4$</th>
<th>$i_5$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>log(3/3)</td>
<td>log(3/2)</td>
<td>log(3/2)</td>
<td>log(3/1)</td>
<td>log(3/1)</td>
<td>log(3/2)</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0.405</td>
<td>0.405</td>
<td>1.099</td>
<td>1.099</td>
<td>0.405</td>
</tr>
</tbody>
</table>

---

8In fact, stemming will reduce “cheese” to “chees” and “ate” to “at.” We’ve chosen to keep the words as is for readability.
So, which document will we get when we search for “chili cheese dog”? We can plug into our formula for the score of a query, \( s_{Q,j} = \sum_{i \in Q} \text{tf}_{ij} \text{idf}_i \).

But wait! What will \( \text{idf}_{\text{chili}} = \log \frac{n}{n_{\text{chili}}} \) be? Since “chili” doesn’t appear in our corpus, it looks like we’ll have to divide by zero - until we realize that \( \text{tf}_{\text{chili},j} = 0 \) for any \( j \). Then, we can conclude that “chili”’s contribution to \( s_{\text{chili cheese dog},j} \) will be 0 and short-circuit before we ever try to calculate \( \text{idf}_{\text{chili}} \).

For the remaining words in our query, we compute
\[
\begin{align*}
 s_{Q,j_0} &= 0 \cdot 0.405 + 1/2 \cdot 0.405 \approx 0.2 \\
 s_{Q,j_1} &= 1/2 \cdot 0.405 + 1/2 \cdot 0.405 \approx 0.4 \\
 s_{Q,j_2} &= 1 \cdot 0.405 + 0 \cdot 0.405 \approx 0.4
\end{align*}
\]
and conclude that the latter two sentences will both be viewed as equally relevant.

3 PageRank

So far we have scored documents based on their relevance to a given query only. This is how early search engines, like AltaVista, worked. But then, in the late 90s, Google came along with an idea for ranking pages on their overall importance, independent of any particular query.

PageRank is an algorithm for ranking all the documents (i.e., pages) in a corpus. The algorithm pays no attention whatsoever to the pages’ content. Rather, the rankings it computes are based entirely on the link structure among the pages: i.e., which pages link to which. Pages with high scores (i.e., page ranks) are then thought of as “authoritative.”

Here are the key principles underlying the design of PageRank:

1. The more pages that link to a page \( j \), the more authoritative \( j \) should be.
   For example, if “Blog Brown” is linked to by 5 web pages, and the Wikipedia article on “Providence, Rhode Island” is linked to by 500 pages, then it makes sense to consider the Wikipedia article more authoritative.

2. The more authoritative those pages are, the still more authoritative \( j \) should be.
   Now, if “Providence Place Mall” is linked to only by “Providence, Rhode Island”, and “Bluno” is linked to only by “Blog Brown” it might not be enough to measure a page’s authority simply by a count of the number of pages that link to that page. Indeed, it makes sense to consider “Providence Place Mall” more authoritative than “Bluno” since it is linked to by a more important page.

3. The fewer links those pages have to pages other than \( j \), the more authoritative \( j \) should be.
   Assume “Research at Brown” is linked to only by a “NYTimes” article which links to only 10 other pages, while “Why Brown?” is linked to only by “Blog Brown”, which links to 200 other pages. Because the “NYTimes” page has only a few links, and “Brown Blog” has many links, a link from the “NYTimes” page can be considered to be more important than a link

---

9 Alternatively, we can define \( \text{idf}_{ij} = \log \frac{n}{1+n_{ij}} \), but then words that appear in every document will have a negative \( \text{idf} \), unless we remove such words before stemming, which is commonly done.

10 The original paper introducing Google, by Brin and Page.
from the “Brown Blog”, leading us to attribute more authority to “Research at Brown” than “Why Brown?”

4. The closer \( j \) is to another page \( k \) (measured by the number of links that must be followed to get from \( j \) to \( k \)), the more \( k \)’s score should influence \( j \)’s.

For example, if the average path from “Brown University” to “Boeing” is 100 links, and the average path from “Brown University” to “Amy’s Personal Website” is 5 links, then all other things equal, Amy’s page should be considered more authoritative than Boeing’s.

The most naive page ranking algorithm simply sums up the number of links to a page, and calls that the page’s rank. This algorithm satisfies only the first principle in this list.

To satisfy the second principle is a bit trickier. Let’s think about what it says: a page’s rank should depend on the ranks of the pages that link to it, but the ranks of those pages should also depend on the ranks of the pages that link to them, and so on. The trouble is, this definition is circular, which begs the question of where to begin. Not to worry: we can simply begin with the assumption that all pages are of equal rank. Specifically, if there are \( n \) pages in our corpus, we can initialize all page ranks to \( 1/n \). (One invariant of the PageRank algorithm is that ranks across pages always sum to 1.)

Now, back to the second principle. We would like to model the intuition that the more authoritative the pages \( k \) are that link to a page \( j \), the still more authoritative \( j \) should be. Here’s how to say that, in math. Letting \( r_j \) denote the rank of page \( j \), \( r_j \) should equal the sum over all pages \( k \) of the value page \( k \) ascribes to \( j \) times the rank of page \( k \):

\[
    r_j = \sum_k w_{jk} r_k
\]

Here, \( w_{jk} \) denotes the value page \( k \) ascribes to page \( j \). But where do these values come from?

The PageRank algorithm sets these values based on the link structure of the corpus in a way that is consistent with the third principle. Specifically, each page is given some weight (hence, the use of the letter \( w \)) which it can distribute across all the pages it links to. For example, if each page \( k \) is given a total weight of 1, then \( k \) can assign a weight \( w_{jk} = 1/n_k \) to each page \( j \) that it links to, where \( n_k \) is the number of pages \( k \) links to: e.g., if there are 10 pages in the corpus, and \( k \) links to three of them, then those three pages are each assigned a weight of \( 1/3 \), while all other pages are assigned weight 0. This weighting scheme attributes more authority to a page \( j \) if it is one of only a few pages linked to by another page \( k \) than if it were one of many.

In practice, there is one small tweak that is usually applied to this scheme in order to satisfy the fourth principle. Rather than uniformly distribute a page \( k \)’s weight across only those pages \( k \) links to, the following formula is used instead: for some (small) \( \epsilon > 0 \) (e.g., \( \epsilon = 0.15 \)),

\[
    w_{jk} = \begin{cases} 
        \frac{\epsilon}{n} + (1-\epsilon) \frac{1}{n_k} & \text{if } k \text{ links to } j \\
        \frac{\epsilon}{n} & \text{otherwise}
    \end{cases}
\]

Let us explore how this tweak gives rise to rankings that satisfy the fourth principle. Starting from

\[\text{Read that sentence again if necessary!}\]
some initial ranking $\mathbf{r}^0$, Equations 1 and 2 begin unfolding like this:

\begin{align*}
    r^1_j &= \sum_k w_{jk} r^0_k \quad (3) \\
    r^2_j &= \sum_k w_{jk} r^1_k \\
        &= \sum_k w_{jk} \left( \sum_l w_{kl} r^0_l \right) \\
        &= \sum_{k,l} w_{jk} w_{kl} r^0_l
\end{align*}

Looking at the first equation, we can see that the first update to the ranking of page $j$ depends on the weights $w_{jk}$, which encode the values that each page $k$ ascribes to page $j$. Likewise, from the last equation, we can see that the second update to the ranking of page $j$ depends on the multiplicative term $w_{jk}w_{kl}$. This term encodes not just the value that each page $k$ ascribes to page $j$, but rather the value that each page $k$ ascribes to each page $l$ multiplied by the value that each page $l$ ascribes to each page $j$. Embedded in this multiplicative term is the factor $\epsilon^2$. Likewise, embedded in the calculations of $r^3_j$ is the factor $\epsilon^3$. The $\epsilon$ in these equations can be interpreted as a dampening factor, because pages that are only one hop (i.e., link) away are adjusted only by $\epsilon$, but pages that are two hops away are adjusted only by $\epsilon^2$, and so on. Indeed, the fewer hops there are from $j$ to $k$ the more $k$’s score is influenced by $j$’s.

If you continue the iterative calculations spelled out in Equations 3 and 4, you will find that the rankings eventually converge. Indeed the rankings to which this process converges are the unique rankings that satisfy Equation 1.\footnote{This claim requires proof. Take CS 53, or any basic linear algebra course, to learn why this claim is true.}

Note: We sort of swept things under the rug when we first wrote down Equation 1. What we should have written to be more precise was: at the $i$th iteration,

\begin{equation}
    r^{i+1}_j = \sum_k w_{jk} r^i_k \quad (5)
\end{equation}

But now that we know that this iterative process converges, Equation 1 makes perfect sense!

So, in sum, the PageRank algorithm is an iterative algorithm, (parameterized by some value $\epsilon$) that begins with some initial ranking (such as uniform), and then continually updates that ranking via Equation 5 until convergence.

The only thing left for us to tell you, then, is how to check for convergence. This is usually done by measuring the difference between the two rankings (which is not unlike measuring similarities, by the way!). For example, Euclidean distance is a common way of measuring the difference between two rankings, $\mathbf{r}^i$ and $\mathbf{r}^{i+1}$:

$$\sqrt{\sum_j (r^{i+1}_j - r^i_j)^2}$$

Once this distance is sufficiently small (say less than $\gamma = 0.001$), you can declare the process converged, and output $\mathbf{r}^{i+1}$ (or $\mathbf{r}^i$, since they are so close) as your final ranking.
Note: There are a few special cases to handle when defining the weights $w_{jk}$ (before adjusting those weights by $\epsilon$):

- Links from a page to itself are ignored.
- Links to pages outside the corpus are ignored.
- Pages that link to nothing can be considered to link (once) to everywhere.
- Multiple links from one page to another can be treated as a single link, or not.

If there are 10 pages in the corpus, and $k$ links to three of them, linking to the first one once, the second one twice, and the third one three times, then those three pages can be assigned weights of $1/6$, $1/3$, and $1/2$, respectively. Or, those three pages can each be assigned a weight of $1/3$. Both are valid approaches.

Here is some pseudocode to get you started with your implementation of PageRank:

**Algorithm 1 PageRank**

<table>
<thead>
<tr>
<th>Initialize each weight $w(j)(k)$ as described in the accompanying text</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialize $r$ to be an array of $n$ zeros</td>
</tr>
<tr>
<td>Initialize $r'$ to be an array of size $n$</td>
</tr>
<tr>
<td>Initialize each rank $r'(j)$ arbitrarily: e.g., to $1/n$ { $r'$ is the initial vector of ranks}</td>
</tr>
<tr>
<td>while distance($r, r'$) &gt; $\delta$ do</td>
</tr>
<tr>
<td>$r \leftarrow r'$</td>
</tr>
<tr>
<td>for $j \leftarrow 0$ to $n - 1$ do</td>
</tr>
<tr>
<td>Reset $r'(j)$ to zero</td>
</tr>
<tr>
<td>for $k \leftarrow 0$ to $n - 1$ do</td>
</tr>
<tr>
<td>$r'(j) \leftarrow r'(j) + w(j)(k) \times r(k)$</td>
</tr>
<tr>
<td>end for</td>
</tr>
<tr>
<td>end for</td>
</tr>
<tr>
<td>end while</td>
</tr>
</tbody>
</table>
3.1 PageRank Example

Let’s consider a concrete example of PageRank on three pages connected as follows:

Linking structure between A, B, and C.

Linking structure with links added from B to everywhere (see Note on PageRank special cases).

Computing weights according to equation (2), we get:

\[ w_{AB} = w_{BA} = w_{CA} = w_{CB} = \frac{0.15}{3} + \frac{0.85}{2} = 0.475 \]

\[ w_{AC} = \frac{0.15}{3} + 0.85 = 0.9 \]

\[ w_{AA} = w_{BC} = w_{BB} = w_{CC} = \frac{0.15}{3} = 0.05 \]

Then, we can use those weights to iterate over page ranks for A, B, and C until they converge.

<table>
<thead>
<tr>
<th>iteration</th>
<th>rank(A)</th>
<th>rank (B)</th>
<th>rank (C)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.3333</td>
<td>0.3333</td>
<td>0.3333</td>
</tr>
<tr>
<td>1</td>
<td>0.4750</td>
<td>0.1916</td>
<td>0.3333</td>
</tr>
<tr>
<td>2</td>
<td>0.4148</td>
<td>0.2519</td>
<td>0.3333</td>
</tr>
<tr>
<td>3</td>
<td>0.4404</td>
<td>0.2263</td>
<td>0.3333</td>
</tr>
<tr>
<td>4</td>
<td>0.4295</td>
<td>0.2372</td>
<td>0.3333</td>
</tr>
<tr>
<td>5</td>
<td>0.4341</td>
<td>0.2325</td>
<td>0.3333</td>
</tr>
<tr>
<td>6</td>
<td>0.4322</td>
<td>0.2345</td>
<td>0.3333</td>
</tr>
<tr>
<td>7</td>
<td>0.4330</td>
<td>0.2337</td>
<td>0.3333</td>
</tr>
<tr>
<td>8</td>
<td>0.4326</td>
<td>0.2340</td>
<td>0.3333</td>
</tr>
</tbody>
</table>

Table 1: Convergence of ranks for three pages A, B, and C.
Here’s another, less worked through example:

```
A ---- B
   |    |
   |    |
   v    v
 C ---- D
```

<table>
<thead>
<tr>
<th>rank(A)</th>
<th>rank(B)</th>
<th>rank(C)</th>
<th>rank(D)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.2018</td>
<td>0.0375</td>
<td>0.3740</td>
<td>0.3867</td>
</tr>
</tbody>
</table>

### 3.2 Testing PageRank

We have provided you with a sample wiki file, *PageRankWiki*, made to test the results of your PageRank implementation. The wiki includes several pages where one page (Page 100) has an overwhelmingly larger number of pages linking to it compared to the rest of the pages in the wiki. We recommend checking the values your PageRank implementation outputs for each of the documents and confirming that Page 100 has a much higher score than the others!

**Note:** When implementing PageRank, your goal should be to strive for accurate results based on PageRank’s algorithm. Keep in mind this wiki tests only one way for a page to achieve a high PageRank score (based on how many pages link to it). Your program should take into account all four scoring factors detailed above.

### 4 Representation

After going through the trouble of parsing, tokenizing, and stemming all the documents, and then scoring each word as it relates to each document, your next challenge is to figure out how to store this information in such a way that 1. it does not waste space (as a completely naive representation of the term-frequency vectors would), and 2. your querier can easily access the information it needs to quickly respond to queries.

First, you will need to design a file format for each of your index files. There will be at least two such files: `titles.txt`, which associates document IDs with document titles, `index.txt`, which stores the word-document scores. Be sure whatever format you come up with is unambiguous and efficient. That is, no two distinct wikis will ever have the same index file, and your aren’t wasting any space in your file (unnecessary white space, characters, etc.). It will be the job of your indexer to write these files, and the job of your querier to read them.

Second, you will need to create a data structure (or two, or three, etc.) where your querier can store the information it reads. Remember, these data structures should enable fast access to this
information, so that your querier is quick to respond to user queries.

5 Querier: A REPL

You have been using REPLs all year. Now you finally have a chance to write one! In particular, the interface for your querier should be a REPL. It should prompt the user to enter a free text query, after which it should return the titles of the top 10 Wikipedia pages, one title per line. This process should repeat until the user types ":quit" at the prompt.

**Hint:** REPLs are easy to write. They are more or less an infinite loop (a method just keeps on calling itself), until some special character is read. For the purposes of this project, there are two such special characters: 'Ctrl-d' and ":quit".

6 Assignment

Like most modern search engines, your search engine will consist of two programs, an indexer and a querier. This section summarizes exactly what your indexer and querier should be capable of doing.

**Indexer** Your index program should take as input multiple arguments: the name of the Wikipedia corpus file, and the names of the files to which it should write the indexes it produces (e.g., index.txt). More specifically, it should do the following (though not necessarily in this order):

- Parse the XML file containing the corpus. For each document, you should extract the document ID, the document title, and the text of the document (which includes its title).
- Split up the text of each document into individual tokens.
- Discard all tokens that are stop words.
- Stem each remaining token to produce a term using the provided Porter stemmer.
- Associate each document ID with a document title, and store this mapping in a title index file.
- Store a score for each term-document pair in an index file.
- Store each document’s PageRank score in an index file.

**Querier** Your query program should take as input the title index and all other index files produced by your index program. It should also take as input the optional arguments --smart and --pagerank, which instruct the program to apply any enhancements you might have implemented, and to incorporate PageRank scores, respectively. If either command is used (you need not support both simultaneously), it should be the first argument provided. So running your querier might look like any of the following:

13If the user types "quit" without the colon, your search engine should run the query!
$ scala search.sol.Query titles.txt index.txt
$ scala search.sol.Query --smart titles.txt index.txt
$ scala search.sol.Query --pagerank titles.txt index.txt

Your query program should do the following:

1. Read the given index files into memory.
2. Prompt the user to enter a free text query.
3. Answer the query, as follows:
   - Find the terms in the query by discarding stop words, and then stemming.
   - Determine the set of documents that contain at least one of the terms in the query.
   - Find the score $s_d$ for each document in the set, based on term frequency and PageRank if it applies. Combine these two scores into one.
   - Print out the titles of the top 10 documents based on their combined scores. The first document printed should be the one with the highest score, and the last document printed should be the one with the tenth-highest score.
     **Hint:** You need not reinvent the wheel. On the contrary, you should feel free to use a library method to sort your data.
     If the tokens in the query appear in fewer than 10 documents, return only the documents on which they appear. If there are no search results at all, print an informative message.

4. Repeat the Steps 2 and 3 until the user types ‘:quit’.

7 Testing

Thoroughly testing your project should involve a combination of unit testing and system testing. Use the type of testing that fits the situation.

You should test the calculations computed by your search engine and other important pieces of functionality. Once you have a working search engine verify your index file is in the correct format and search results are reasonable. You should test various queries, with different types of pages, keeping in mind your search engine should handle all input gracefully.

7.1 Documentation

You will only receive credit for testing which is clearly documented in your README file. Document the input and expected output of your tests, as well as any arguments given to your program. The TAs grading your project should be able to recreate all your testing from your documentation.

We encourage you to create your own Wikipedia files for testing. However, our handin script ignores XML files to prevent you from accidentally handing in BigWiki.xml. As a result, you must handin any Wikipedia corpus you create without a .xml extension. For example, if I created “MyWiki.xml” for testing I should rename it “MyWiki.txt” before I handin.
8 Helpful Hints

Make sure you are using efficient data structures throughout; this will be extremely important when processing very large sets of data— for instance, with BigWiki.

Furthermore, there are ways to optimize your program when writing to a file. In general, String concatenation is a bit of an expensive process— think about how you can avoid doing this! For instance, suppose you have two strings, "Hello" and "world". You could write "Hello world" to a file with your BufferedWriter the following way:

```java
writer.write("Hello" + " " + "world")
```

Or, you could avoid concatenation altogether with the following method:

```java
writer.write("Hello")
writer.write(" ")
writer.write("world")
```

Lastly, be sure to close any Readers and Writers as soon as you are done using them; and similarly, do not open them prematurely.

9 Handin

9.1 Design Check

Design checks will be held on the week of March 13-14. We will send out an email detailing how to sign up for design checks; it is in your best interest to sign up as soon as possible (before all the prime time slots have been filled).

Reminder: You are required to pair program at least the design check portion of all CS 18 projects. We recommend finding a partner as soon as possible, as you will not be able to sign up for a design check without one.

For the design check, you must do the following:

- Explain how you will parse, tokenize, and stem documents; that is, how you will split them up into individual, lower case, stemmed terms that are not stop words.
- Explain how you will score documents.
- Prepare a written specification of the format of your index file(s). Related, what data structure will your querier store this information in when it reads these files?
- Walk the TAs through the steps that your program will take to answer a free text query.
- Explain how you plan to implement PageRank. Feel free to ask your friendly TAs as many clarification questions as necessary about PageRank in order to ensure you understand it fully before your design check.
9.2 Final Handin

The final handin is due by Friday, March 23 at 5:00PM. For the final handin, your `scalaproject` should contain the packages `search.src` (in the `src` directory) and `search.sol` (in the `sol` directory). Your code should be part of the `search.sol` package. That package should also contain a `README.txt` file.

Your ‘README’ file should include:

- instructions for use, describing how a user would interact with your program
- a brief overview of your design, including how all of the pieces of your program fit together
- a description of any features you failed to implement, as well as any extra features you chose to implement
- a description of any known bugs in your program
- a description of how you tested your program
- a list of the people with whom you collaborated

To hand in your files, navigate to the `~/course/cs018/workspace/scalaproject` directory, and run the command ‘`cs018_handin search`’. This will automatically hand in the contents of your entire `scalaproject` directory. Once you have handed in your project, you should receive an email, more or less immediately, confirming that fact. If you don’t receive this email, try handing in again, or ask the TAs what went wrong.

**Note:** Only one of you or your partner must hand in the project.

9.3 Grading

As with all CS 17/18 projects, a good design will make coding this project significantly easier; so you should spend a fair amount of time working on your program’s design before you begin writing any code.

The design check counts for 15% of your grade, including:

- An explanation of your plan for tokenizing.
- An explanation of your plan for scoring documents.
- Details about your index files, and the data structures your querier will use to store the information they contain.
- A walkthrough of the design of your querier.
- A demonstration of your understanding of PageRank.

Functionality counts for 70% of your grade, including:
• Correct implementation of Indexer: 23 points
• Correct implementation of Querier: 23 points
• Correct implementation of PageRank: 10 points
• Efficiency: 14 points

As always, partial functionality merits partial credit.

The final 15% of your grade will be reserved for comments, testing, and style. You should include documentation comments and test cases for all non-trivial methods. You should also perform system testing, to test interactions among methods. Additionally, comment any code which would otherwise be unclear.

10 Extra Features (Optional)

Writing a search engine is still very much an art, not a science. After completing everything that is required of you for this project, you may want to play around with trying to improve your search engine by implementing additional tweaks. If you run your querier with the ‘--smart’ flag (as the first argument), it should use your tweaks to rank results.

Here are some ideas to get you started:

• Try out alternative normalization schemes (rather than straight-up Euclidean normalization)
• Weight the text in a document’s title and body differently
• (Challenge) Weight based on how close together query terms appear in a document—if “cat” and “picture” appear far away from each other in a document, then don’t weight them very highly when the query is “cat picture”

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS18 document by filling out the anonymous feedback form: http://cs.brown.edu/courses/cs018/feedback