Programming Assignment 1—Web Search Engine

Points: 500

Due: Nov 10, 11:59PM
Early Submission: Nov 9, 11:59PM (10% Bonus)
Late Submission: Nov 11, 11:59PM (15% penalty)

Description of a programming assignment is not a linear narrative and may require multiple readings before things start to make sense. You are encouraged to start working on the assignment as soon as possible. If you start a day or two before the deadline, it is highly unlikely that you will be able to come with correct and efficient programs.

You are encouraged to consult your instructor, the TAs, or Piazza for any questions/clarifications regarding the assignment. Your programs must be in Java and must be able to compile and execute under Java 8. You should not use any external libraries, except `jsoup`. For this assignment, you **may work in teams of 2**. It is your responsibility to find a team member. If you can not find a team member, then you must work on your own. **Only one submission per team**.

In this programming assignment you will implement a rudimentary web search engine. Very broadly, a web search engine has two components: i) offline computation, and ii) online computation, where the search engine receives a search query from the user and outputs *ranked list* of web pages that are relevant to the search query. The offline computation has two parts

- Crawl the web and download contents of web pages and construct the web graph. Using the web graph structure, determine the *importance* for each web page.
- Based on the contents of the webpages, build a data structure known as *inverted index*. The inverted index is designed so that the online computation—retrieving a ranked list of pages relevant to a search query—is done very efficiently.

In online phase, the user inputs a query and the search engine, upon receiving a search query from the user, outputs a *ranked list* of web pages that are relevant to the search query. This phase uses the *importance* (computed in part 1 of the offline computation) and the inverted index (constructed in the second part of the offline computation) to determine and output a ranked list of webpages.

You will implement the classes `Crawler` and `Index` in addition to any other classes you may need in your implementation; in particular, you will need to provide some class implementing the `Graph` interface for your crawler to return from its `crawl()` method All your code must be in the package `pa1`. The sample code includes skeletons for `Crawler` and `Index`, along with some other code in the `api` package that you will need. **Do not modify any of the code in the `api` package**.

1 BFS and Web Graph

We can model the web as a directed graph. Every web page is a vertex of the graph. We put a directed edge from a page p to a page q, if page p contains a link to page q.

Recall the BFS algorithm from the lectures.

1. Input: Directed Graph $G = (V, E)$, and $root \in V$.
2. Initialize a FIFO (first-in, first-out) queue Q and a list *discovered*.
3. Place *root* in Q and add to *discovered*.
4. **while** Q is not empty **Do**
   (a) Remove the first element v of Q.
   (b) For every edge $\langle v, u \rangle \in E$,
      - If $u \notin$ *discovered* add u to the end of Q, and add u to *discovered*.

If you output the vertices in *discovered*, that will be BFS traversal of the input graph.

We can use BFS algorithm on web graph also. Here, the input to the algorithm is a *seed url* (instead of entire web graph). View that page as the the root for the BFS traversal. Here is the BFS algorithm on Web graph.

1. Input: *seed url*
2. Initialize a FIFO queue Q and a list *discovered*.
3. Place *seed url* in Q and in *discovered*.
4. **while** Q is not empty **Do**
   (a) Remove the first element *currentPage* of Q.
   (b) Send a request to server at *currentPage* and download *currentPage*.
   (c) Extract all links from *currentPage*.
   (d) For every link u that appears in *currentPage*
      - If $u \notin$ *discovered* add u to the **end of** Q, and add u to *discovered*.

This will visit all the pages that are reachable from the `seed url`.

2 jsoup

To avoid a great deal of tedious text parsing, and to ensure some uniformity in the results (to help us test your code), you are **required** to use the external library `jsoup` for extracting links and extracting the body text from a web page. Download the core library `jsoup-1.12.1.jar` from `https://jsoup.org/download`. In the sample code you can find some usage examples that illustrate everything you need to know.

You will need to have the jsoup jar file in your Java classpath. If you use Eclipse, just do the following:

- find your project in the Eclipse Package Explorer (pane at the left, in the default workspace)
- drag the jsoup jar file into your project
- right click on it and select Build Path, then Add to Build Path

If you use some other IDE or a command shell then presumably you know how to perform similar operations with it.

3 The Util class

The sample code includes a class `Util` with a small number of static methods that you will need, referred to in the discussion below. See the `Util` source code for details.

4 Crawling and Constructing Web Graph

One of the first tasks of a web search engine is to *crawl* the web to collect material from all web pages. While doing this, the the search engine will also construct the *web graph*. The structure of this graph is critical in determining the *importance* of a page. Different search engines use different (advanced) methods to determine the *importance* of a web page. In this assignment, the importance of a page is its *indegree* in the constructed web graph.

In this part of the assignment you will write a program to *crawl* the web and construct web graph. However, web graph is too large (more than a billion pages) and you perhaps do not have computational resources to construct the entire web graph. Thus your program will crawl and construct a partial web graph. You will start with a *seed url* and construct a partial web graph over the pages that are reachable from the seed url. You will use BFS algorithm for crawling, but the search will be limited both by *depth* (distance from seed url) and *number* (total number of pages visited).

Note that although we are using the BFS algorithm, the end result will not be to construct the BFS tree, but rather the graph including all links among the visited pages.

4.1 Crawler

Implement a class named `Crawler`. This class will have methods that can be used to crawl portion of the web. This class should have following constructor and methods.

`Crawler.` parameters to the constructor are

1. A string which is referred as `seedUrl`. This is a web address.
2. An integer `maxPages` representing maximum number pages that can be crawled.
3. An integer `maxDepth` representing the maximum depth of the BFS search.

`Graph<String> crawl()` This methods uses a BFS algorithm to construct a web graph over all the webpages that can be reached from the `seedurl`, subject to the limitations:

- Number of pages in the graph is at most `maxPages`

- Any page in the web graph must have distance at most `maxDepth` from the `seedurl`. Note that distance between two nodes of a graph (unweighted graph) is the length of the shortest path between them. By definition distance from a node to itself is zero.
- The graph must be a *simple graph*. I.e, there are no edges from a vertex to itself. There is at most one edge from a a vertex to another vertex. (Note that a webpage might contain a link to itself or can contain multiple links to another webpage.

This method returns a `Graph<String>` object representing the web graph.

Here is an example. Consider the following pages. $A, B, C, D, E, F, G, H, I, J$. Page A has links to B, C and D appearing in that order. Page C has links to E, F, B and D, in that order. Page D has links to G, H and A. Page B has links to C, I and J. Page E has a link to page A. None of the other pages have any links. If the `seedUrl` is A, `max` is 6, and `depth is 4` the the constructed graph will have the following edges: A to B, A to C, A to D, B to C, B to I, B to J, D to A, C to B, and C to D. However, if `max` is 6 and `depth` is 1, then the constructed will have following edges: A to B, A to C, A to D, B to C, C tp B, and D to A.

Extraction of the links from a page is performed by the statements,

```
Document doc = Jsoup.connect(url).get();
Elements links = doc.select("a[href]");
for (Element link : links) {... }
```

Where we say "in that order" in the discussion above, it means in the order returned by iterating over the collection returned by `select("a[href]")` as above. You are required to completely ignore certain links that refer to anchors within the same page or that are qualified by a ":" character in the path, using the method `Util.ignoreLink`. (You don't necessarily have to worry about duplicate links at this point, since the process of constructing the graph should eliminate duplicate links.)

Note that a web page may contain links to resources (such as images) that are not web pages, or that are inaccessible (nonexistent or forbidden). In the course of the BFS algorithm, these will be added to your graph, but they will end up as leaf nodes (no outgoing edges). Basically, your code that calls `Jsoup.connect(url).get()` to extract the links has to be surrounded in a try/catch block for exceptions of type `org.jsoup.UnsupportedMimeTypeException` or `org.jsoup.HttpStatusException`. See the sample code `JSoupTest.java` for a concrete example.

4.2 Implementing a graph

The `crawl` method returns an object of type `Graph<String>`, where `Graph` is specified as a generic interface that you can find in the sample code. See the Javadoc for an explanation of the behavior of each method. It is completely up to you how you want to implement the interface. All that is specified for the `crawl` method is to return some object satisfying the interface.

5 Inverted Index

Inverted index is a data structure that enables to retrieve webpages that are relevant to a search query. Let $S = \{p_1, p_2, \cdots p_n\}$ be a collection of urls. Let $W = \{w_1, w_2, \cdots w_m\}$ be all the words

appearing in the body text of these pages, *excluding* the so-called *STOP words*. A word is called a STOP word if it is unlikely to be relevant in a web query; for example, "the", "in", and "of" are typical examples of STOP words. For this assignment, the STOP words are defined by the method `Util.isStopWord`.

Given a word $w \in W$ and webpage p, let $f(w, p)$ denote the number of times the word w appears in p.

An inverted index over W and S is a set I where each element of this set is a word w from W and a pointer to a set denoted $list(w)$. For a given word w, $list(w)$ is a collection of tuples $\langle p, f(w, p)\rangle$ for which $f(w, p) > 0$.

Note that it is up to the designer for the choice of data structures to represent the sets I and $list(w)$.

Given a search query q, the goal is to obtain a *ranked list* of web pages that are relevant to the query q. For this we need to define *rank* of a webpage with respect to a query. While there are several (sophisticated) ways to define this notion, in this assignment we use the following. Let G denote a web graph as constructed by a `Crawler`, and let $S = \{p_1, p_2, \cdots, p_n\}$ be the collection of urls represented by nodes in this graph, and let W be the non-stop words in these pages as described above.

Given a word w and a url $p \in S$, the rank of p with respect to w, denoted $r_w(p)$, is defined as

$$r_w(p) = Indegree(p) \times f(w, p)$$

, where $r_w(p) = 0$ if $w \notin W$.

We define three additional notions of rank for queries involving two words. $r_q^{AND}(p)$, $r_q^{OR}(p)$ and $r_q^{NOT}(p)$. Let q be a query with two words w_1 and w_2.

- If at least one of $r_{w_1}(p)$ or $r_{w_2}(p)$ is zero, then $r_q^{AND}(p)$ is zero, else

$$r_q^{AND}(p) = r_{w_1}(p) + r_{w_2}(p)$$

- If both $r_{w_1}(p)$ and $r_{w_2}(p)$ are zero, then $r_q^{OR}(p)$ is zero, else

$$r_q^{OR}(p) = r_{w_1}(p) + r_{w_2}(p)$$

- If $r_{w_1}(p)$ is zero or $r_{w_2}(p)$ is not zero, then $r_q^{NOT}(p)$ is zero, else

$$r_q^{NOT}(p) = r_{w_1}(p)$$

Note that your construction of the index should include just the body text of the document, excluding the head and links. To extract the body text use `Jsoup.connect(url).get().body().text()`. You can parse individual words from the document using a default Java `Scanner`. You are then required to use the method `Util.stripPunctuation` to normalize each parsed word, and then to use `Util.isStopWord` to eliminate STOP words from the index.

5.1 Index

Implement a class named `Index`. This class will have the following methods and constructor.

`Index`. The parameter to this method is the information from the web graph that is relevant for constructing the index, namely the list of `urls` along with their indegrees. The type of this parameter is `List<TaggedVertex<String>>`, where a `TaggedVertex` is just a tuple containing a url along with an integer which is its indegree in the graph.

`void makeIndex()` Creates an inverted index based on the graph information provided to the constructor.

The four search methods below all return a list of urls ordered by rank from highest to lowest. For the return value, again use the type `List<TaggedVertex<String>>`, that is, you're returning a list of tuples $\langle url, rank \rangle$. The list does not include any url whose rank is zero.

`search(String w)` It is assumed that w is a single word. Returns a ranked list of web pages.

`searchWithAnd(String w1, String w2).` $w1$ and $w2$ are words. Returns a ranked list of web pages ordered by $r^{AND}_{w_1,w_2}$ (largest to smallest). Any page whose rank is 0, does not appear in this list.

`searchWithOr(String w1, String w2).` $w1$ and $w2$ are words. Returns a ranked list of web pages ordered by $r^{OR}_{w_1,w_2}$ (largest to smallest). Any page whose rank is 0, does not appear in this list.

`searchAndNot(String w1, String w2).` $w1$ and $w2$ are words. Returns a ranked list of web pages ordered by $r^{NOT}_{w_1,w_2}$ (largest to smallest). Any page whose rank is 0, does not appear in this list.

6 Report

Your submission must include a brief written report describing

- Pseudocode for the methods `crawl`, `makeIndex`, `search(·)`, `searchWithAnd(·,·)`, `searchWithOr(·,·)`, `searchAndNot(·,·)`.
- Describe the choice of data structures used and report the run-times of your algorithms.

Convert your report to pdf form and name it **report.pdf**. You must include the names of the team members at the top of the document.

7 Sample code

The types `Graph`, `TaggedVertex`, and the `Util` class are provided for you in the `api` package, and you must use them without modification. There is some example code `JSoupTest` in the `example` package that you can use to experiment with jsoup. There are basic skeletons for `Crawler` and `Index` in the `pa1` package. If you use Eclipse, you can just create an empty project and then drag the three packages into the project directory. Be sure to **check** the box "Copy projects into workspace". See the section above on jsoup for instructions on importing the jsoup library.

8 Guidelines

- You must use the `jsoup` Java library to parse html pages to extract links and contents of web pages. This library is at `https://jsoup.org/`. This is the only external library that you are allowed to use.
- Your program must adhere to the following politeness policy: Wait for at least 3 seconds after every 50 requests. No exceptions to this policy. **If you do not adhere to this policy you will receive ZERO credit**. You must use the following code for waiting.

```
try
{
  Thread.sleep(3000);
}
catch (InterruptedException ignore)
{
}
```

  Crawlers place considerable load on the servers (from which they are requesting a page). If your program continuously sends request to a server, you may be denied access.
- If most of you work in the last hour and start sending requests to pages in a domain it is possible that you get a deny service to any request from that domain . This can not be an excuse for late submissions or extending the deadline. Thus, you are advised to start working on the assignment as soon as possible.
- Your work will be evaluated for both correctness and efficiency. Note that the choice of your data structures and algorithms affects the efficiency of your programs.
- Your program must strictly adhere to the specifications. Class, method names, type and order of parameters to the methods, and the return types of methods must be exactly as specified. Any deviation from the specification will lead to a significant portion of points (even if your programs correct and efficient).

9 What to Submit

Your submission must be in the form of a zip archive containing

- Your report.pdf (please include names of team members).
- The package directory `pa1`, under which you have the following files:
  - Crawler.java
  - Index.java
  - Additional classes that you created, such as your `Graph` implementation

You must include any additional helper classes that you created. Please include only source .java files, do not include any .class files. **All the code you have written or modified must be in the package** `pa1`. Everything else will be ignored.

Please include all team members names as `@author` tags in the class Javadoc for each of the Java files that you modify or create. **Only one submission per team please. It does not matter which team member does the submission.**