Final Project

COMP 250 Fall 2020

posted:	Thursday, Dec. 3, 2020
due:	Tuesday, Dec. 22, 2020 at 23:59

General Instructions

- **Submission instructions**
  - Please note that the submission deadline for the final project is very strict. **No submissions will be accepted after the deadline**.
  - As always you can submit your code multiple times but only the latest submission will be kept. We encourage you to submit a first version a few days before the deadline (computer crashes do happen and codePost may be overloaded during rush hours).
  - This is the file you should be submitting on codePost:
    - `Sorting.java`
    - `SearchEngine.java`

    Do not submit any other files, especially .class files. Any deviation from these requirements may lead to lost marks
- Please note that the classes you submit should be part of a package called `finalproject`.
- Starter code is provided for this project. Do not change any of the class names, file names, method headers. You can add helper methods if you wish. Note also that for this project, you are NOT allowed to import any other class (all import statements other than the one provided in the starter code will be removed). **Any failure to comply with these rules will give you an automatic 0.**
- The project shall be graded automatically. Requests to evaluate the project manually shall not be entertained, so please make sure that you follow the instruction closely or your code may fail to pass the automatic tests.
- Whenever you submit your files to codepost, you will see the results of some exposed tests. These tests are a mini version of the tests we will be using to grade your work. If your code fails those tests, it means that there is a mistake somewhere. Even if your code passes those tests, it may still contain some errors. We will test your code on a much more challenging set of examples. We highly encourage you to test your code thoroughly before submitting your final version.
- We included with these instructions a tester class, which is a mini version of the final tester used to evaluate your project. This class is equivalent to the exposed tests on codePost, with the addition of a stress test for `fastSort()`. The stress test is **NOT** included in the exposed

tests on codepost. Please note that these tests are only a subset of what we will be running on your submissions. We encourage you modify and expand these classes. You are welcome to share your tester code with other students on Piazza. Try to identify tricky cases. Do **not** hand in your tester code.

- You will automatically get 0 if your code does not compile.
- Failure to comply with any of these rules will be penalized. If anything is unclear, it is up to you to clarify it by asking either directly a TA during office hours, or on the discussion board on Piazza.

Learning Objectives

This project is meant for you to practice working with hashmaps and graphs. In particular, you will construct a graph of vertices and edges using a breadth-first or depth-first search. You will learn more in-depth about graph theory in MATH 240 and COMP 251. You will also implement one of the $O(n\ log(n))$ sorting algorithms that you've learned about in class, such as MergeSort or QuickSort.

Introduction

A search engine is software, usually accessed on the Internet, designed to carry out searches in a database of information according to the user's query. The results provided are meant to best match what the user is trying to find and they are usually displayed in order of importance. The first search engine ever developed is considered to be Archie. It was named to resemble the word "archive", and it was created in the 1990 by Alan Emtage, Bill Heelan and J. Peter Deutsch, computer science students at McGill. Today, there are many different search engines available on the Internet, each with their own abilities and features. The most popular of them all being Google.

A search engine performs the following tasks:

- Crawling and Indexing
- Ranking
- Searching

Crawling and Indexing Crawling is the process during which a search engine use programs, generally called *spiders* or *crawlers*, to collect information from web pages. By following links from a web page to another, the spiders find new content. Once a page is crawled, the data contained in the page is processed and indexed. Indexing is the process of storing and organizing the content found during the crawling. It generally involve associating words and other definable tokens found on web pages to a list of web pages that contain them. It also includes recording links to other pages. These associations are all stored in a database which is then used for web search queries. During this process the webgraph is created/updated. This is a directed graph where each node represents a page, and an edge represents an hyperlink from one page to another. This is also when what we refer to as the 'word index' is created. You can think of it as a mapping from single words to a list of urls containing them. Note that for actual search engine, this task (crawling and indexing) is never actually completed. Given the constantly changing nature of the Web, spiders are always crawling to keep the database up to date. The Google Search index contains hundreds of billions of webpages.

Ranking How useful a search engine is depends on how relevant are the results obtained when we query it. There are millions of pages containing a specific word, but some pages are generally more relevant, or authoritative than others. To determine which pages to show in the search results and in what order, search engines use the data collected to perform a ranking. Different search engines use different algorithms to rank pages. Developed by Larry Page and Sergey Brin in 1996, PageRank

is one of the algorithms used at Google to rank webpages based on their importance. Using the webgraph, each webpage is assigned a numerical value that measure its relative importance. In simple terms, the rank of a webpage depends on the webpages that link to it. The contribution to the rank is higher if these linking webpages have a high rank themselves, while it's lower if these linking webpage also link to a lot of other webpages.

Searching In this step, search engines receive a query from the user. Using the index created while crawling and the ranking based on the webgraph, the engine outputs a list of relevant web pages in order of importance. Do all search engines give the same results? Not necessarily. Search engines use different spiders to crawl and use different proprietary algorithms to index the data. Each index is therefore a search engine's representation of how they see the web. Also the algorithms to rank and search the data are different, so every search engine has its own approach to finding what you're trying to find. Finally, personalisation adapts the search to a specific computer/user. The results may be based on your geographical location, what else you've searched for, and what results were preferred by other users searching for the same thing, for example. Search engines might use and weigh all these factors in a unique way, which will lead to different search results.

The starter code contains four classes:

- `XmlParser`: This class has two methods: one to read the content of the xml database (which will serve as proxy for a web page for our project) given a url, and the other to extract information from it.
  - `getContent(String url)` returns an `ArrayList` of `String`s corresponding to the set of words in the web page located at the given `url`. You will need to use this method while crawling in order to build your word index.
  - `getLinks(String url)` returns an `ArrayList` of `String`s containing all the hyperlinks going out of the given `url`. You will need this method to build the graph representing the data found while crawling.

  You should NOT modify this class at all.
- `MyWebGraph`: This class implements the data structure needed to store the data collected during the crawling phase. It has an inner class called `WebVertex` which is used to store data related to a specific web page. You should NOT modify this class at all.
- `SearchEngine`: This is one of the classes in which you will have to add your code. You will be implementing methods that performs the three tasks described above.
- `Sorting`: This is a utility class containing methods that implement some sorting algorithm. You will be asked to implement one of them.

Your task

In this project you are going to write a search engine program that will:

1. explore a portion of the web (which we'll simulate through a database),

2. build an index of the words contained in each web page,
3. analyze the structure of the web graph to establish which web page should be ranked higher,
4. use this analysis to perform simple searches where a query word is entered by the user and a sorted list of relevant web pages available is returned.

Although we are going to apply these tools using a local database, what you will develop is a simplified version of the what is used at Google to answer actual web queries.

To be able to implement the search engine program described above, we need a graph data structure that will allow us to store all the information related to the web pages. Such class is provided to you.

[**Provided**] The class `MyWebGraph` is an implementation of a directed graph using adjacency lists. You will be using this type of data structure to store the information your program collects in the crawling phase. Each node in the graph will store a String corresponding to the url of a webpage. The class has the following field:

- A `HashMap` storing all the vertices in the graph. Note that we'll be labelling each vertex with the `String` corresponding to the url of the webpage represented by this vertex.

The class has also the following `public` methods:

- The constructor `MyWebGraph()` which does not take any input and initializes the field with an empty `HashMap`.
- An `addVertex()` method which takes as input a `String` representing a url. The method adds the corresponding vertex to the graph.
- An `addEdge()` method which takes as input two `String`s, i.e. two urls. The method adds to the graph an edge from the vertex labelled with the first input to the vertex labelled with the second input.
- A `getNeighbors()` method which takes as input a `String` representing a url and returns an `ArrayList` of all the hyperlinks contained in the specified url.
- A `getVertices()` method which returns the list of all the urls represented in the graph.
- A `getEdgesInto()` method that takes as input a `String` representing a url. The method returns an `ArrayList` of `String`s of all the pages that contains an hyperlink to the specified url.
- A `getOutDegree()` method which takes as input a `String` representing a url and returns the number of hyperlinks in the specified page.
- A `getPageRank()` method which takes as input a `String` representing a url and returns the rank of the specified page.
- A `setPageRank()` method which takes as input a `String` representing a url and a `double` representing its rank. The method assigns the input number as the rank of the specified page.

- A `getVisited()` method which takes as input a `String` representing a url and returns whether or not the specified page has been visited.
- A `setVisited()` method which takes as input a `String` representing a url and a `boolean`. The method assigns the input boolean to the visited field of the specified page.

[20 points] The `Sorting` class is a utility class that at the moment contains a `static` method called `slowSort()`. This method takes as input a `HashMap` where the values are `Comparable`. The method returns an `ArrayList` containing all the keys in the map, sorted in descending order based on the values they map to. The time complexity of `slowSort` is $O(n^2)$, where n is the number of pairs in the map. Your task is to implement a method called `fastSort` which performs the same task as `slowSort` but with a time complexity of $O(n \cdot \log(n))$.

[80 points] The `SearchEngine` class has the following fields:

- A `HashMap` called `wordIndex` used to represent the mapping from single words to a list of urls containing them.
- A `MyWebGraph` called `internet` which is used to store all relevant information regarding the structure of the webgraph.
- An `XmlParser` called `parser` which you should use to retrieve information for the database serving as a proxy for the web.

In this class, you should implement the following methods:

[35 points] `crawlAndIndex`

This method takes a `String` as input representing a url. It will perform a graph traversal of the web, starting at the given url. You can use either a depth-first search or a breadth-first search, and you can make it either recursive or non-recursive. For each url visited, the method should do the following:

- Update the web graph by adding the appropriate vertex and edges.
- Update the word index so that every word in the url just visited appear in the mapping.

Note that, it is possible for the graph to contain cycles. For instance, if it possible that two web pages link to each other. To prevent this method from getting stuck in an infinite loop, you should make sure to keep track of which web page has already been visited. This method should stop crawling once all web pages (in the database) have been visited.

[35 points] `assignPageRanks` and `computeRanks`

The method `computeRanks` is a helper method for `assignPageRanks`. We specifically ask you to implement it so that our grader will be able to assign partial marks to your implementation. The purpose of `assignPageRanks` is to assign a rank to each vertex in the web graph. It will only be called after `crawlAndIndex` has been executed. Here are the ideas we will be using to assign a rank to a web page:

- Good web pages are cited by many other pages. If we think about it in terms of the webgraph, this means that we should prefer web pages (i.e. vertices) with a large in-degree.
- Web pages that link to a large number of other pages are less valuable. In terms of webgraph, this means that we will value less web pages (i.e. vertices) with a large out-degree.
- A link from a web page is more valuable if the web page is itself a good one. In graph terms, higher the rank of a web page (i.e. vertex), more valuable an in-edge from it would be.

Note that the rank of a page depends solely on the structure of the graph we have created while crawling. To represent the ideas just described, let

- the $pr(w)$ be the page rank of a vertex w
- the $out(w)$ be the out-degree of a vertex w
- w_1, w_2, ..., w_k be all the vertices in the graph that have an out edge going into v.

Then, the following equation determines the page rank of a vertex v:

$$pr(v) = (1 - d) + d * \left(\frac{pr(w_1)}{out(w_1)} + \frac{pr(w_2)}{out(w_2)} + \ldots \frac{pr(w_k)}{out(w_k)} \right)$$

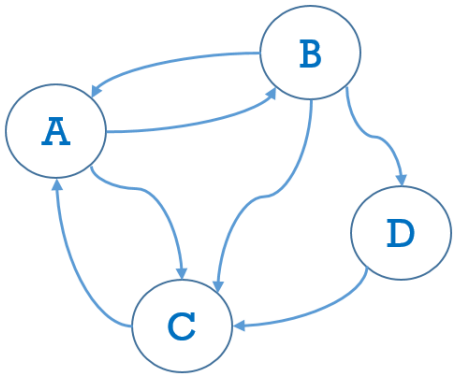
The constant d is called the *damping factor* and it is added for technical reasons to account for the probability that an imaginary surfer who is randomly clicking on links will eventually stop clicking. For this project, we will be using a damping factor of 0.5. As you may notice, $pr(v)$ is defined as a function of $pr(w_i)$. If N is the total number of vertices in the graph $(v_1, v_2, \ldots, v_N)$, then we have a system of N linear equations in N variables. The unknown values, i.e. our variables, are the $pr(v_i)$. To determine their values, we should solve the system of linear equations described above. To do so, we could use linear algebra, but the implementation of gaussian elimination runs in $O(n^3)$. To improve the efficiency, we can instead use an iterative algorithm that approximate the result. The idea is the following:

- Start by initializing $pr(v_i)$ to 1 for all $0 \leq i \leq N$
- Repeat the following until convergence:
  - compute $pr(v_i)$ for all i using the formula above.

    This single step is what the helper method `computeRanks` should be doing. The method takes as input an `ArrayList<String>` representing the urls in the web graph and returns an `ArrayList<double>` representing the newly computed ranks for those urls. Note that the double in the output list is matched to the url in the input list using their position in the list. That is, the page rank of the url stored in position i in the input list, can be found in position i in the output list.

Convergence is reached when $|pr^{k-1}(v_i) - pr^k(v_i)| < \epsilon$ for all i, where $pr^j(v_i)$ represents the computation of $pr(v_i)$ in the j-th iteration, and ϵ **is the input to the method**. Note that, you can assume that the web graph does not contain self loops and every vertex in the graph has at least one outgoing edge.

Consider the following web graph:



Suppose we'd like to compute the page rank assigned to each vertex with an ϵ equal to 0.01. We start by assigning value 1 to each of them:

- $pr(A) = 1$
- $pr(B) = 1$
- $pr(C) = 1$
- $pr(D) = 1$

Then we need to use the following formula (same as the formula above, but with damping factor of 0.5) to recompute the page ranks for each vertex, until convergence.

$$pr(v) = 1/2 + 1/2 * \left(\frac{pr(w_1)}{out(w_1)} + \frac{pr(w_2)}{out(w_2)} + \ldots \frac{pr(w_k)}{out(w_k)} \right)$$

After the first iteration we get:

- $pr(A) = 1/2 + 1/2 * (1/3 + 1) = 7/6 = 1.166$
- $pr(B) = 1/2 + 1/2 * (1/2) = 3/4 = 0.75$
- $pr(C) = 1/2 + 1/2 * (1/2 + 1/3 + 1) = 17/12 = 1.416$
- $pr(D) = 1/2 + 1/2 * (1/3) = 4/6 = 0.666$

After the second iteration we get:

- $pr(A) = 1/2 + 1/2 * (0.75/3 + 1.416) = 1.333$
- $pr(B) = 1/2 + 1/2 * (1.166/2) = 0.791$

- $pr(C) = 1/2 + 1/2 * (1.166/2 + 0.75/3 + 0.666) = 1.25$
- $pr(D) = 1/2 + 1/2 * (0.75/3) = 0.625$

After 5 iterations we reach convergence since the difference between the rank values in the 4th iteration and the rank values in the fifth iteration is smaller than ϵ. The ranks at the end are:

- $pr(A) = 1.270$
- $pr(B) = 0.819$
- $pr(C) = 1.274$
- $pr(D) = 0.636$

[10 points] `getResults`

This method takes as input a `String` representing a single-word query. It then returns an `ArrayList` of urls that contain such word, ordered based on their rank from most relevant to least relevant. This method will be called only after both `crawlAndIndex` and `assignPageRanks` have been executed. This method should use one of the sorting algorithms from the `Sorting` class. Ideally, it should use `fastSort()`, but if you did not implement it, you can use `slowSort()`.

Testing

To test your code we provided you with two XML files that act as a proxy for the internet. The XML files contains proxy web-pages indicated by the tag name webpage. Each web-page contains multiple hyperlinks to other web-pages indicated by the tag link. The contents of a web-page is marked using the tag content. To help you with testing, we also provide the **expected** ranks of the web-pages computed using the algorithm described in the previous section. Note that rank is not part of an actual web-page, we simply provide them to help you with debugging. Page rank is indicated by tag rank and can be accessed in your code using the function `XmlParser::getPageRank()` which takes as input the name of the page.

- testAcyclic.xml - The provided tester use this database to test your code. Note that this database does not have any circular chains - making it easy for you to test and debug.
- test.xml - We do not provide any test cases for this database but is more general than the one above. We encourage you to create your tests with this database that will help you find more intricate issues with your code.

Finally, you should create your databases that are much larger as we will test on a much bigger database in the final grading.