Com S 435/535 Programming Assignment 3
600 Points
Due: Nov 16 , 11:59PM
Late Submission Nov 18, 11:59 (0.1% Penalty, Yes it is 0.1)

In this programming assignment, you will write a web crawler that crawls pages from `wikipedia` and will compute page ranks for the crawled pages. You crawler will be a *topic sensitive* crawler—attempts to collect pages about certain topic.

Note that the description of a programming assignment is not a linear narrative and may require multiple readings before things start to click. You are encouraged to consult instructor/Teaching Assistant for any questions/clarifications regarding the assignment.

For this assignment, you may work in groups of two.

1 Weighted Q

A weighted Q works as follows. Each element of the Q is a tuple: $\langle x, w(x)\rangle$, where x could be any data item and $w(x)$ is weight of x. The operations allowed are: `add`, `extract`. The method `add` places a tuple into the Q. The method `add` works as follows: If you are adding a tuple $\langle x, n\rangle$, if there is no tuple of the form $\langle x, m\rangle$ in the Q, then place $\langle x, n\rangle$ in the Q. Otherwise, replace $\langle x, m\rangle$ with $\langle x, n\rangle$ if $n > m$. The method `extract` works as follows: It returns the tuple with highest weight (among the tuples present in the Q), and removes that tuple from the Q. If there are multiple highest weight tuples, then it returns the *first such tuple* that was added to the Q.

Here an example. Suppose we start with empty weighted Q and added elements in the following sequence: $\langle 1, 5\rangle$, $\langle 2, 3\rangle$, $\langle 5, 7\rangle$, $\langle 21, 5\rangle$, $\langle 36, 4\rangle$. If we perform `extract`, then it will return $\langle 5, 7\rangle$ as 7 is the highest weight and removes $\langle 5, 7\rangle$. Now the Q has following elements $\langle 1, 5\rangle$, $\langle 2, 3\rangle$, $\langle 21, 5\rangle$, $\langle 36, 4\rangle$. Suppose if perform the method `extract` again. Now there are two tuples with heaviest weights: $\langle 1, 5\rangle$ and $\langle 21, 5\rangle$. Since $\langle 1, 5\rangle$ was added to the Q before $\langle 21, 5\rangle$, it will return $\langle 1, 5\rangle$ and removes it from Q. Now the weighted Q has following elements: $\langle 2, 3\rangle$, $\langle 21, 5\rangle$, $\langle 36, 4\rangle$. Now suppose we attempt to add $\langle 21, 9\rangle$ to the queue. It will replace $\langle 21, 5\rangle$with $\langle 21, 9\rangle$.

Note that when all the tuples are of same weight, then weighted Q *exactly* behaves like a First In First Out Queue.

2 Weighted BFS

Recall the BFS algorithm discussed in the lecture. In the algorithm Q is a list of vertices and it is maintained as a *First In First Out Queue*. We will adopt the it to the case when the vertices of the graph have weights. The algorithm is exactly the same except that Q is implemented as a `Weighted Q`.

3 Topic Sensitive Crawling.

Let p be a page and w_1 and w_2 be two words that appear in p. Suppose that w_1 appears at ℓ_1th, ℓ_2th $\cdots$, and ℓ_pth positions in p, and w_2 appears at r_1th , r_2th, $\cdots$, and r_kth position in p. Then

the distance between w_1 and w_2 is

$$dist(w_1, w_2) = \min_{1 \le i \le p, 1 \le j \le k} abs(\ell_i - r_j)$$

where *abs* denotes absolute value.

Suppose that we want to crawl the web pages about "tennis". Here is an approach: Select a set of strings that will describe the topic. The intuition is that any webpage about tennis will have one of these words. For example, for tennis, set of strings could be `tennis, grand slam, french open, australian open, wimbledon, US open, masters`. Imagine crawling the web for pages with tennis as topic. We start at a root node say `wiki page about tennis`. Lets call this page p. Page p has quite a few links, say $q_1, q_2, \cdots, q_\ell$. How do we know whether the page pointed by q_1 is about tennis or not. The best way is to send a request to page q_1 and check if that page has words from our topic set. However, this approach is expensive. We will be sending requests to many pages that are not about tennis. Instead, we will use a heuristic to determine whether the page pointed by link q is about tennis or not (without sending request to page q). We will assign a weight to the link q. The weight will be higher, if our heuristic thinks that page q is about our topic. The heuristic is simple: Look at the link q. If the anchor text of q or the http link of q contains any of our topic strings, then it must be the case that q is about our topic. If neither the anchor text not the http link has our topic words, then look at the text surrounding the link. If any of the topic words appear in the surrounding text, then we should be reasonably confident that page q is about our topic. Our confidence would be higher, if the topic words are closer to the link; our confidence will be lower if the topic words are away from the link. We will now formalize this.

Let q_i be a link in a page p and T be a set of words. We define the distance between q and T (with respect to page p) as

$$dist(q_i, T) = \min\{dist(q_i, w) \mid w \in T\}$$

We will assign a weight to each q_i as follows: Look at the all occurrence of q_i in the page p. If the `anchor text` of the link or the `http` address within the link contains a word from the set T, then $weight(q_i) = 1$. Otherwise, let compute $d = dist(q_i, T)$. If $d > 17$, then $weight(q_i) = 0$, else $weight(q_i) = \frac{1}{d+2}$. If the topic set T is empty, then weight of every link is 0.

4 Wiki Crawler

This class will have methods that can be used to crawl Wiki. Instead of crawling entire wikipedia, you will do a *focussed crawling*–Only crawl pages that are about a particular topic. Your crawler must perform a weighted BFS on the web graph and use the above described mechanism to compute weight of a web page. The crawler will write the discovered graph to a file. You will be crawling only wiki pages. This class should have following constructor and methods.

`WikiCrawler.` parameters to the constructor are in the following order.

1. A string *seedUrl*–relative address of the seed url
2. Array of Strings *keywords* –contains key words that describe a topic
3. An integer *max* representing Maximum number sites to be crawled

4. A string $fileName$ representing name of a file–The graph will be written to this file

5. A boolean $isTopicSensitive$.

`crawl()` Method to crawl max many pages. If $isTopicSensitive$ is false, then weight of every link/page to 0. If $isTopicSensitive$ is true, determine the weight of a page/link via the above described heuristic. Then the crawling must be done using weighted BFS. Note that when $isTopicSensitive$ is false, then you will be doing crawling via normal BFS and thus use *First in First Out Queue*. This method should construct the web graph over all pages visited by the BFS and write the graph to the file $fileName$. The number of vertices in the graph must be exactly equal to max. Note that this the graph over the **first** max vertices visited by the BFS algorithm.

For example, `WikiCrawler` can be used in a program as follows

```
String[] topics = {"tennis", "grand slam"};
WikiCrawler w = new WikiCrawler("/wiki/Tennis", topics, 100, "WikiTennisGraph.txt", true);
w.crawl();
```

This program will start crawling with `/wiki/Tennis` as the root page. Collects 100 wiki pages by using weighted BFS algorithm, and determines the web graph over these hundred vertices. The graph will have exactly 100 vertices. It writes the graph to a file name `WikiTennisGraph.txt` This file will list all edges of the graph. Each line of this file should have one directed edge, except the first line. The first line of the graph should indicate number of vertices. There should not be any duplicate edges in the graph. Below is sample contents of the file

```
100
/wiki/tennis /wiki/Tennis_ball
/wiki/tennis /wiki/Tennis_court
/wiki/tennis /wiki/Wheelchair_tennis
/wiki/tennis /wiki/England
/wiki/tennis /wiki/Real_tennis

... ...
.. ...
... ...
```

The first line tells that there is a link from page `/wiki/Tennis` to the page `/wiki/Tennis_ball`.

4.1 Clarifications and Suggestions

1. The seed url must be specified as *relative address*; for example `/wiki/Tennis` not as `https://en.wikipedia.org/wiki/Tennis`.

2. Extract only links from "actual text component". A typical wiki page will have a panel on the left hand side that contains some navigational links. We would like not to extract any such links. Wiki pages have a nice structure that enables us to do this. The "actual text content" of the page starts immediately after the first occurrence of the html tag `<p>`. Thus for the purpose of this PA, "actual text component" is the text that appears after the first `<p>`.

3. Your program must consider links in the order they appear in the page.

4. Your program should only form the graph of pages from the domain `https://en.wikipedia.org`

5. Your program should not explore any wiki link that contain the characters "#" or ":". Links that contain these characters are either links to images or links to sections of other pages.

6. The number of vertices of your graph must be exactly max.

7. The graph you constructed should not have self loops nor it should have multiple edges.

8. You **must** follow politeness policies. Download `robots.txt` file and do not crawl any site that is `disallowed`. Your program should not continuously send requests to wiki. Your program must wait for at least 2 seconds after every 10 requests. You will receive ZERO credit for not following politeness policies. No exceptions. Use `Thread.sleep` to wait.

9. Class `WikiCrawler` must declare a `static, final` global variable named `BASE_URL` with value `https://en.wikipedia.org` and use in conjunction with links of the form `/wiki/XXXX` when sending a request fetch a page. Otherwise you will receive ZERO credit (no exceptions). For example, your code to fetch page at `https://en.wikipedia.org/wiki/Physics` would be

```
URL url = new URL(BASE_URL+"/wiki/Physics");
InputStream is = url.openStream();
BufferedReader br = new BufferedReader(new InputStreamReader(is));
```

10. Your program should not use any external packages to parse html pages, to extract links from html pages and to extract text from html pages. You can only use inbuilt packages of Java that are of the form `java.*`.

11. If most of you work in the last day/hour and start sending requests to wiki, then it is quite possible that wiki may slow down or deny requests. That can not be an excuse for extending the due date.

5 PageRank

This class will have methods to compute page rank of nodes/pages of a web graph. This class should have following methods and constructors.

The constructor `PageRank` will have following parameters

1. Name of a file that contains the edges of the graph. You may assume that the first line of this graph lists the number of vertices n, and every line (except first) lists one edge. Each vertex is represented as an integer in the range $[1, \cdots, n]$, and every edge of the graph appears exactly once and the graph has no self loops. Below is input format

```
4
1 3
```

```
3 1
2 3
2 4
4 1
```

2. ϵ; approximation parameter for pagerank.

3. β: Teleportation Parameter.

A method named `pageRankOf`: gets an `int` representing a vertex of the graph as parameter and returns its page rank which is a `double`.

A method named `outDegreeOf`: gets an `int` representing a vertex of the graph as parameter and returns its out degree.

A method named `inDegreeOf`: gets an `int` representing a vertex of the graph as parameter and returns its in degree.

A method named `numEdges` that returns number of edges of the graph.

A method named `topKPageRank`: gets an integer k as parameter and returns an array (of ints) of pages with top k page ranks.

A method named `topKInDegree`: gets an integer k as parameter and returns an array (of ints) of pages with top k in degree.

A method named `topKOutDegree` : gets an integer k as parameter and returns an array (of ints) of pages with top k out degree.

5.1 Crawling and Page Rank

Write a program named `MyWikiRanker`. For this, pick a set of words representing a topic of your choice. Choose an appropriate seed url and form a graph over 500 vertices. Compute the page rank each page in your graph. Consider the following sets:

- A: Top 20 vertices (web links) based on outdegree.
- B: Top 20 vertices (web links) based on indegree.
- C: Top 20 vertices (web links) based on page rank when $\epsilon = 0.01$ with $\beta = 0.85$.
- D: Top 20 vertices (web links) based on page rank when $\epsilon = 0.005$, with $\beta = 0.85$.
- E: Top 20 vertices (web links) based on page rank when $\epsilon = 0.001$, with $\beta = 0.85$.

Program should output exact Jaccard Similarities among all pairs from $\{A, B, C, D, E\}$.

5.2 Spam Farm

The goal of this class is to add additional nodes and edge to a graph so that the page rank of a given vertex increases. Write a class named `SpamFarm`. This class will have following Constructor and methods.

Constructor will have following parameters

1. Name of a file that contains the edges of the graph. You may assume that the first line of this graph lists the number of vertices n, and every line (except first) lists one edge. You may assume that each vertex is represented as an integer in the range $[1, \cdots, n]$, and every edge of the graph appears exactly once and the graph has no self loops.

2. A target vertex called *target* which is an integer between 1 and n. The goal is to increase the page rank of this vertex.

This class will have a method named `CreateSpam`, that gets a string `fileName`. This class will add edges to the given graph and writes the new created graph to `fileName`. The new graph must be created as follows:

- Total number of new vertices must be at most $n/10$.
- All the new vertices must be named $n+1, n+2, \cdots$
- The new edges added must be among the vertices $\{target, n+1, n+2, \cdots, \}$.
- You can place a new edge from $\{target, n+1, n+2, \cdots\}$ to any other vertex of the graph
- Output file format should be similar to the input file format.

6 Report

Include the following in your report

- Data structure used to implement `Weighted Q`.
- Pseudo code of your crawling algorithm. Please describe how your procedure will ensure that the graph formed will have exactly `max` many vertices.
- Output of the program `MyWikiRanker` (top 10 page rank, in degree and outdegree pages and Jaccard Similarities)
- Number of iterations for your page rank algorithm to converge (within ϵ) on the graph `wikiTennis.txt`, for both choices of epsilon, when $\beta = 0.85$.
- Number of iterations for your page rank algorithm to converge (within ϵ) on the graph `wikiTennis.txt`, for both choices of epsilon, when $\beta = 0.25$.
- Consider the graph created by your crawler. Pick a vertex with the smallest page rank. Create a spam farm to increase the rank of the vertex. How much the rank is increased?

7 What to Submit

Create a zip file with

- WikiCrawler.java
- PageRank.java

- SpamFarm.java
- report.pdf

and submit the zip file.

Group that creates the best spam farm will receive 100 points as extra credit. As before, you may work in groups of 2. Only one submission per group please.