We write a web crawler that crawls pages from `wikipedia` and computes page ranks for the crawled pages. The crawler is a *topic sensitive* crawler - attempts to collect pages about certain topics.

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 the weight of x. The operations allowed are: `add`, `extract`. The method `add` places a tuple into Q; it works as follows: if adding a tuple $\langle x, n\rangle$ to Q where there is no tuple of the form $\langle x, m\rangle$, then we place $\langle x, n\rangle$ into Q. Otherwise, we do not place $\langle x, n\rangle$ in Q. The method `extract` works as follows: it returns the tuple with the highest weight (among the tuples present in Q), and removes that tuple from Q. If there are multiple highest weight tuples, then it returns the *first such tuple* that was added to Q.

As 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 Q has the following elements $\langle 1, 5\rangle$, $\langle 2, 3\rangle$, $\langle 21, 5\rangle$, $\langle 36, 4\rangle$. Suppose we perform `extract` again; now there are two tuples with the highest weight, $\langle 1, 5\rangle$ and $\langle 21, 5\rangle$. Since $\langle 1, 5\rangle$ was added to Q before $\langle 21, 5\rangle$, $\langle 1, 5\rangle$ will be returns and removed from Q. Now the weighted Q has the following elements: $\langle 2, 3\rangle$, $\langle 21, 5\rangle$, $\langle 36, 4\rangle$, and suppose we attempt to add $\langle 21, 9\rangle$ to the queue. Since there is a tuple $\langle 21, 5\rangle$ in Q, $\langle 21, 9\rangle$ will not be added.

We implemented `Weighted Q` using the following data structure. As expecting the crawler to be dealing with URLs, we set up `Map<String, Float> Q` to store the weights (in `Double`) of links (in `String`); moreover, since the order of adding elements into `Q` is essential, we implemented it with `LinkedHashMap` to keep track the sequence in which elements were added. For the property of hashing, we are able to check whether a link has been included in `Q` in constant time, and put the link into the queue if such link is not present in the queue. When performing extracting operation, we return the tuple with the highest weight that was added to `Q` first. The tuple is returned in the form of `<String, Float>` that is defined in a class **`Pair`**, and removed from `Q`. Note that when all the tuples are of same weight, then weighted Q *exactly* behaves like a First-In-First-Out queue.

2: Crawling Algorithm

Weighted BFS Recall the Breadth-First-Search algorithm. Q is a list of vertices and it is maintained as a *First-In-First-Out queue*. We adopted it to the case where vertices of a graph have weights. The algorithm is exactly the same except that Q is implemented as a `Weigted Q`.

Topic Sensitive Crawling Suppose that we want to crawl web pages about "tennis". We could select a set of words that describe the topic; the intuition is that any webpage about tennis will have one of these words. For example, they could be `tennis`, `grand slam`, `french open`, `australian open`, `wimbledon`, `US open`, `masters`. An approach to crawl the web for pages with tennis as the topic can be like the following. We start at a root node, say a `wiki page about tennis`, denoted as p. Page p has quite a few links, say $q_1, q_2, \ldots, q_l$. To know whether q_1 is about tennis or not, we send a request to page q_1 and check if that page has words from our topic set. However, this approach is expensive since we will be sending requests to many pages that are not about tennis. If we take a look at links in the wiki tennis page, most of them are not about tennis. Therefore, we used a heuristic to determine whether a link q is about the topic or not (without sending request to page q). That is, we assign a weight to the link q. The weight will be higher if our heuristic thinks that q is about our topic. The heuristic works as follows. It looks at the link q; if the anchor text of q or the http link of q contains

ant of our topic words, then it must be the case that q is about out topic. If neither the anchor text nor the http link has our topic words, then it looks 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 close to the link in position, and would be lower if the topic words are farther away from the link.

To be formal, let p be a web page and T be a set of words describing a topic. Assume that the web page p has following distinct web links $q_1, q_2, \ldots q_l$. Our goal is to assign a weight to each of the links. Given a word w and a link q_i, we define $dist_b(q_i, w)$ as follows: consider the first occurrence of q_i in p, if w does not appear before the first occurrence of q_i (in p), then $dist_b(q_1, w)$ is ∞, else $dist_b(q_1, w)$ equals the number of words between w and q_i. Similarly, we define $dist_a(q_i, w)$ as follows: if w does not appear after the first occurrence of q_i, then $dist_a(q_i, w)$ is ∞, else $dist_a(q_i, w)$ equals the number of words between q_i and w. Finally, we define $dist(q_i, w)$ as the minimum of $dist_a(q_i, w)$ and $dist_b(q_i, w)$.

We assign a weight to each q_i as follows. Look ar the first 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 $d = \min\{dist(q_i, w) | w \in T\}$. If $d > 20$, then $weight(q_i) = 0$, else $weight(q_i) = \frac{1}{d+2}$. If the topic set T is empty, then the weight is 0.

In the following we show the pseudo code of the class **TopicSensitiveCrawling**.

for each line of page p after the very first tag <p>[1]
  Extract all the links and their anchor texts[2]
  for each pair of link and text
    if the link does not contain "#" nor ":"[3], and the link has not appeared previously and link starts with `/wiki/`
      if the topic set is empty or *isWeighted*, set weight of the link = 0
      else if a keyword appear in either the link or anchor text, set weight of the link = 1
      else
        Find minimum $dist_a$ and $dist_b$, and let $d = \min\{dist_a, dist_b\}$ over T[4]
        Set weight of the link = $\frac{1}{d+2}$ if $d \leq 20$, else 0

Wiki Crawler This class have methods for crawling Wiki. Instead of crawling the entire wikipedia site, we do a *focused crawling* - only crawl pages that are about a particular topic. The crawler performs a weighted BFS on the web graph and use the mechanism described above to compute a weight of a web page. The crawler write the discovered graph to a file. We focus only on crawling wiki pages. This class includes the following constructor and methods.

WikiCrawler has parameters, a string *seedUrl* - relative address of the seed url, array of strings *keywords* containing key words that describe a topic, an integer *max* representing a maximum number of pages to be crawled, a string *fileName* representing the name of a file to which the

[1] A typical wiki page has a panel on the left hand side that contains some navigational links. We want not to extract such links. Wiki pages have a nice structure that enables us to do this easily: the "actual text content" of the page starts immediately after the first occurrence of the html tag <p>.

[2] Generally, links follow `href=` and texts are surrounded by > and <. A subroutine is designed to retrieve correct text when special context occurs.

[3] We do not explore any wiki link that contain "#" or ":" since links that contain these characters are either links to images or links to sections of other pages.

[4] We mark the location of the link and remove all the tags in order to count the distance between a text and a keyword. We also compress the text into one word by removing all spaces in between.

graph will be written, and a boolean *isWeighted* - if false, the weight of every link/page will be set to 0; otherwise, the weight of a page/link is determined based on the heuristic described above.

crawl crawls *max* many pages. The crawling is done using weighted BFS. When *isWeighted* is false, then it will be doing crawling via normal BFS. This method constructs the web graph of all collected pages and write the graph to the file *fileName*. The number of vertices in the graph will be exactly equal to *max* unless there are less than *max* links that are relevant to the topic set. However, this is very unlikely to happen with wiki pages.

As an instance, `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);
```

This will start crawling with `/wiki/Tennis` as the the root page, and collect 100 wiki pages using weighted BFS algorithm, and determine the web graph over these hundred vertices. The graph will have exactly 100 vertices and be written to a file named `WikiTennisGraph.txt`, which will list all edges of the graph. Each line of this file have one directed edge (from the first listed vertex to the second) except the first line where the number of vertices will be indicated. There is no duplicate edges in the graph. See the named file in the same directory for the content.

In the following we show the pseudo code of the class **WikiCrawler**.

Download `robots.txt` and store sites that are disallowed to crawl[5]

// **weighted BFS**

Initialize `Weigted Q` `wq` $= \emptyset$, `HashSet<String> visited` $= \emptyset$, `vertexNum` $= 0$ and `iteration` $= 0$

Place *seedUrl* in `wq` and `visited`, and `vertexNum` $= 1$

while `wq` is not empty **do**

Let `u` be the element obtained by extracting from `wq`

// **crawl**[6]

Send a request to serve at `u` and download `u`[7]

Discover links and obtain weights from `u` using **TopicSensitiveCrawling** (described above)

Remove *seedUrl* from extracted links to avoid self loops

`iteration = iteration + 1`

Wait for at least 1 second every 10 requests[8]

for every link `v` that is discovered in `u`

if `v` $\in$ `visited`

Write `u` and `v` to *fileName*

[5] from the domain `https://en.wikipedia.org`

[6] Any network error occurs, a web page cannot be accessed, for example, an exception will be generated and caught, and the program will proceed to the next extracted `u`.

[7] A `static final` global variable named `BASEURL` with value `https://en.wikipedia.org` is in use in conjunction with links of the form `/wiki/XXXX` (*seedUrl*) when sending a request to fetch a page.

[8] To deliberately not continuously send requests to wiki, the program waits for a random number of seconds between 1 and 2.2.

else if v $\notin$ `visited` and `vertexNum`$<$*max* and v is allowed to crawl[9]
Add v to `wq` and `visited`
Write u and v to *fileName*
`vertexNum` = `vertexNum` + 1[10]

3: Page Rank

This class has methods to compute page rank of nodes/pages of a web graph, having the following methods and constructors.

PageRank constructor has parameters: name of a file that contains the edges of the graph and ϵ. We assume that the first line of this graph lists the number of vertices, and every line after lists one edge. We also assume that each vertex is represented as string, every edge of the graph appears exactly once, and the graph has no self loops. Given a vector v, let us define $Norm(v)$ as sum absolute values of all entries of v. We compute an approximation to the page rank vector iteratively till we find a t such that $Norm(v_{t+1} - v_t) \leq \epsilon$ for a small ϵ.

pageRankOf gets a name of a vertex in the graph as parameter and returns its page rank.

[9] according to `robots.txt`

[10] `vertexNum` denotes the number of vertices that have been added into the graph. v will be included in the web graph only when the discovered number of vertices have not reached *max*, i.e., `vertexNum`$<$*max*. That is, the graph is built based on the first *max* discovered vertices; no new vertices will be considered afterward.

Table 1: Top 10 vertices and number of iterations with different ϵ values

	$\epsilon = 0.01$	$\epsilon = 0.005$
Top 10 in-degree vertices	/wiki/Australia /wiki/Grand_Slam_(tennis) /wiki/United_Kingdom /wiki/France /wiki/Spain /wiki/Tenni /wiki/South_Africa /wiki/The_Championships,_Wimbledon /wiki/Switzerland /wiki/Australian_Open	
Top 10 out-degree vertices	/wiki/The_Championships,_Wimbledon /wiki/Grand_Slam_in_tennis /wiki/Career_Golden_Slam /wiki/Grand_Slam_(tennis) /wiki/List_of_Wimbledon_gentlemen%27s_singles_champions /wiki/French_Open /wiki/Rod_Laver /wiki/List_of_Grand_Slam_related_tennis_records /wiki/Roger_Federer /wiki/Rafael_Nadal	
Top 10 page rank vertices	/wiki/France /wiki/United_Kingdom **/wiki/Grand_Slam_(tennis)** **/wiki/Spain** /wiki/Australia /wiki/Switzerland /wiki/Tennis /wiki/Czech_Republic /wiki/Romania /wiki/Serbia	/wiki/France /wiki/United_Kingdom **/wiki/Spain** **/wiki/Grand_Slam_(tennis)** /wiki/Australia /wiki/Switzerland /wiki/Tennis /wiki/Czech_Republic /wiki/Romania /wiki/Serbia
Number of Iterations	6	8

Table 2: Similarities between sets(lists)

	$\epsilon = 0.01$	$\epsilon = 0.005$
A and B	0.389	0.389
A and C	0.724	0.724
B and C	0.307	0.307

Table 3: `MyWikiRanker` on other root URLs and keywords

seedUrl	`/wiki/Pop_music`	`/wiki/Physics`
Topic words	"album", "formula", "funk", "gospel music", "hit", "pop song", "rapper", "single", "soul music", "verse"	"system", "mass", "velocity", "collision", "speed", "inertia", "kinetic energy", "momentum"
Top 10 page rank vertices	/wiki/International_Standard_Book_Number /wiki/Internet_Archive /wiki/Jazz /wiki/Popular_music /wiki/Public_domain_music /wiki/Rock_music /wiki/The_Beatles /wiki/Pop_music /wiki/Blues /wiki/Country_music	/wiki/Isaac_Newton /wiki/Ancient_Greek /wiki/Galileo_Galilei /wiki/Physics /wiki/Universe /wiki/Scientific_revolution /wiki/Homer /wiki/Astronomy /wiki/Thales /wiki/Democritus
Number of iterations	7	6

outDegreeOf gets a name of a vertex in the graph as parameter and returns its out-degree.

inDegreeOf gets a name of a vertex in the graph as parameter and returns its in-degree.

numEdge returns the number of edges in the graph.

topKPageRank gets an integer k as parameter and returns an array (of strings) of pages with top k page ranks.

topKInDegree gets an integer k as parameter and returns an array (of strings) of pages with top k in-degree.

topKOutDegree gets an integer k as parameter and returns an array (of strings) of pages with top k out-degree.

We wrote a program named `WikiTennisRanker` to run the page rank algorithm on the graph from `wikiTennis.txt` (in the same directory). We compute the page rank with $\epsilon = 0.01$ and 0.005. The output is reported in Table 1. Note that in-degree and out-degree vertices are independent of ϵ so we have only 4 lists. ϵ serves as a stopping threshold in a way that when it is small, we tolerate less vector difference between two consecutive iterations. As a result, more iterations are required to converge when ϵ is smaller. From the page rank lists, there are no great differences between selecting the two ϵ values; we observe the third and forth ranked links are switched in the resulting lists. Potentially the two links have close distribution probabilities. It is worth noticing that among the in-degree, out-degree and page rank lists, the similarity between the in-degree and page rank lists seem to be higher than other combinations out of the three lists.

We next explored the more vertices for the same set of ϵ values and compute Jaccard similarities. Three sets were computed: set A is the set of top 100 in-degree vertices, set B is the set of top 100 out-degree vertices, and set C is the set of top 100 page rank vertices. Then we computed Jaccard similarities between A and B, A and B, B and C. The results are included in Table 2. One can see that there is a relatively substantial similarity between top 100 in-degree and page rank, which can be understood in the way that as more links go to these nodes, the likelihood is higher that they get visited. This matches the intuition and idea of random walk and ranking pages. We chose β to be 0.85 for all the experiments.

We wrote an additional routine named `MyWikiRanker` with sets of words presenting topics of our choices. Seed urls were chosen and graphs were formed. We compute the rank of each page in the graphs and output top rank vertices in Table 3. *max* and ϵ were set to 100 and 0.01, respectively.