CSE 4081: Introduction to the Analysis of Algorithms Spring 2024
Assignment 04: Joy with Centrality
Due Date: Thursday, March 21; 23:00 hrs
Total Points: 20

In data mining or network science, the three main problems of interest are that of *similarity* or measuring how close objects or nodes are related to each other, *clustering* or identifying a set of nodes or objects that are all related to each other, and *centrality* or finding the most important or influential nodes in a network. Centrality is the principle idea behind recommendation systems and search engines. In this assignment, we will build a miniature search engine that uses the Google Pagerank algorithm.

1 Miniature Web Search Engine

To simplify this process (and to spare the Appstate network from being oversaturated), we wrote a simple web "spider" program that was used to download roughly 10,000 web pages in the appstate.edu domain. To make these easy to read as input, they have been parsed to remove extra html formatting, leaving just the textual words on the pages as well as the URLs of the hyperlinks on each page. All of this data appears in one large input file `asu-domain.txt`, which is about 39.1 megabytes in size.

1.1 Input Data

The content of `asu-domain.txt` is organized as follows (some content is abbreviated with ...):

```
URL: https://appstate.edu
CONTENT: generations of black mountaineers  from innovators to motivators  continue
to inspire ...
LINKS: https://search.appstate.edu/ http://parents.appstate.edu/ ...

URL: https://titleix.appstate.edu/
CONTENT: the office of title ix compliance supports ...
LINKS:  http://appstate.edu/ https://titleix.appstate.edu/non-discrimination-statement
...
```

The contents of about 10,000 web pages are strung together in this file, one after the other. Each page is characterized by three consecutive lines. The first line contains the url of the page. The second line contains its contents as a space-separated list of words (in small-case letters and with common punctuation like periods and commas removed). The third line contains a space-separated list of hyper-links to other web pages. Some pages may have no content or links, in which case these lists are empty, but you will still see the words CONTENT and LINKS.

This assignment contains many moving parts, and therefore we advice you to implement them in sequence.

1.2 Reading the Input

Please read the input and store them in an array of `Webpage` objects that contains the following fields.

```
url:        // e.g., https://appstate.edu
num_links:  // number of outgoing links
num_words:  // number of words in the page
links:      // a list of links
words:      // a list of words
weight:     // importance of the page, using the Pagerank algorithm
```

Note that the same word might appear multiple times on a given page. If so, it should be stored multiple times. That is, the words array should contain all the words appearing in the same order as they do on the page. This allows us to print a small snippet of the content of the page where a search word is contained.

The data above can be stored globally, or if you prefer you can also put it into a class definition. The same holds later for storing word data.

The links and words are stored as arrays. You are welcome to use linked lists instead, which have the advantage of being able to grow on demand as elements are added, and this could make your code more concise as a result. If you use arrays, you may find yourself needing to make two passes over the input on occasion, the first pass being used to count the number of elements, after which you can allocate an array of the appropriate size, and the second pass used to fill in the array.

Here are some suggestions for creating this array of pages:

- Read the input file once, and store the entire file in an array of strings, where each string contains the data of that line. Then you can go over this array several times to build your data structures.
- You may need to read the input several times. In the first pass over the input, for example, you can simply count the number of pages. After this, you can allocate your pages array. Another pass over the input will let you count the number of links and words on each page, so you can then allocate the links and words arrays for each page. Using linked lists or vectors may alleviate the need to do multiple reading passes; this design decision is up to you.
- The "weight" field can be ignored for now; we'll use that later when we implement the Google Pagerank algorithm.
- Some links in the input file are "external" – pointing to urls that do not have a corresponding "URL" record in our input file. These should be ignored. They should not contribute to the num links count and they should not appear in the links array. How might you filter out these external links? A hash table of urls of all your webpages might be handy here (this might be another reason to read the input file in multiple passes, since an earlier pass lets

you build this hash table, which is then used in later passes to filter out external urls). Note that if you are removing external urls, then you also need to update the `num_links` field to reflect the number of proper links on each page.

- Note that because we remove external urls, the url of each link points to a webpage we are storing somewhere in our large array of pages. We may therefore find it convenient to represent links as integers, not strings. That is, a link is stored as the integer index (within our pages array) of the webpage to which it points. A link to `https://appstate.edu` would therefore be stored as 0, since `https://appstate.edu` will reside at index 0 in our array of webpages. To translate links from strings into these indices quickly, please use a hash map. Storing links in this form will make the Pagerank algorithm run much more quickly later on, and in general it will make it easier to find data on the page to which a link points.

Moving forward, you can now refer to any webpage conveniently by the integer giving its index within this array. For example, page 0 will be `https://appstate.edu`. We'll call the index of a page its ID.

1.3 Indexing to Build a Search Engine

Currently, there is no efficient way to search for all the pages containing a specific query word. The goal of this part is to build an "index" allowing fast searches of this form. To do this, we will build an array of words, using the page array we've already constructed. The words array should contain the following fields:

```
text:       // the word as a string
pages:      // list of integer IDs (or urls) of these pages
num_pages:  // number of pages containing this word
```

Please also build a hash map that quickly tells you the index of a word in this array. For example looking up "Appstate" might lead you to the word stored at index 7. Just as with pages, we say the ID of "Appstate" is 7 in this case.

You can now build the first iteration of your search engine. You should prompt the user for a word:

```
Enter search word:
```

When the user types a word, you can use your hash map to look up its ID, from which you can then retrieve the list of pages containing that word and print them out. To avoid copious output, please only print at most 5 pages that match your query word. However, you should print out the total number of pages that match. To make your output look nice, beneath each page URL you should print a short snippet of the sequence of words on the page containing the query word plus a few surrounding words for context. Color the output so the query word stands out. An example of how to change the color of your text is provided in the starter utility code. An example of what the output should look like appears at the end of this document.

Included in the utility starter code is a function called `process_keystrokes` that reads keyboard input in real time, without waiting for the user to press Enter. Every time the user changes the query string, this function calls a function `predict` with the new query string. Please put your code for doing a search query here, so it will update in real time as the user types. Since you are using hashing, it should run quickly enough to update as fast as the user can type.

This means that you need to include the utility starter code within your program, and have the `main` method call `process_keystrokes`, which then calls `predict`. This function should have access to the data structures so that it can respond to this query appropriately. Feel free to experiment with different software design principles in implementing this.

Before you proceed, make sure your output is formatted correctly. You are welcome to experiment with different colors, but at this point, as the user types in a word, we would like to retrieve 5 pages that contain this word. When we are done with this assignment, these 5 pages will be the most relevant pages based on its weight or Pagerank scores.

1.4 Implementing the Google Pagerank Algorithm

The Google Pagerank algorithm assigns a numeric weight to each web page indicating the relative importance of that page, as determined by the linking structure between the pages. Weights are computed using a simple iterative process that models the probability distribution of a random web surfer: in each step, there is a 10% probability that the surfer will teleport to a random page anywhere on the web, and a 90% probability that the surfer will visit a random outgoing link [1]. Accordingly at each step of the algorithm, the weight assigned to a web page gets redistributed so that 10% of this weight gets uniformly spread around the entire network, and 90% gets redistributed uniformly among the pages we link to. The entire process is continued for a small number of iterations (usually around 50), in order to let the weights converge to a stationary distribution. In pseudocode, this process looks like the following, where N denotes the total number of pages.

1. Give each page initial weight of 1. Note that this doesn't actually matter, since after sufficiently many steps of a random walk, we converge to the same stationary distribution no matter what distribution we started with. Note that the total weight in all nodes is N.
2. Repeat 50 times:
    (a) For each page i, set `new_weight[i] = 0.1`. (`new_weight[i]` represents the weight of page i after this iteration of pagerank. Since the total weight in all pages is N, this counts all the weight distribution due to teleportation).
    (b) For each page i, go through each page j to which i links, and increase `new_weight[j]` by $0.9 *$ `pages[i].weight` / `pages[i].num_links` (this spreads 90% of the weight of a page uniformly across its outgoing links. As a special case, if page i has no outgoing links, please keep that 90% on the page by increasing `new_weight[i]` by `pages[i].weight * 0.9`.
    (c) For each page i, set `pages[i].weight = new_weight[i]`.

    Note that these are three separate "for each page i" loops, one after the other. Also note that weights on pages are never created or destroyed in each iteration of Pagerank — they are only redistributed, so the total of all the weights should always be N.

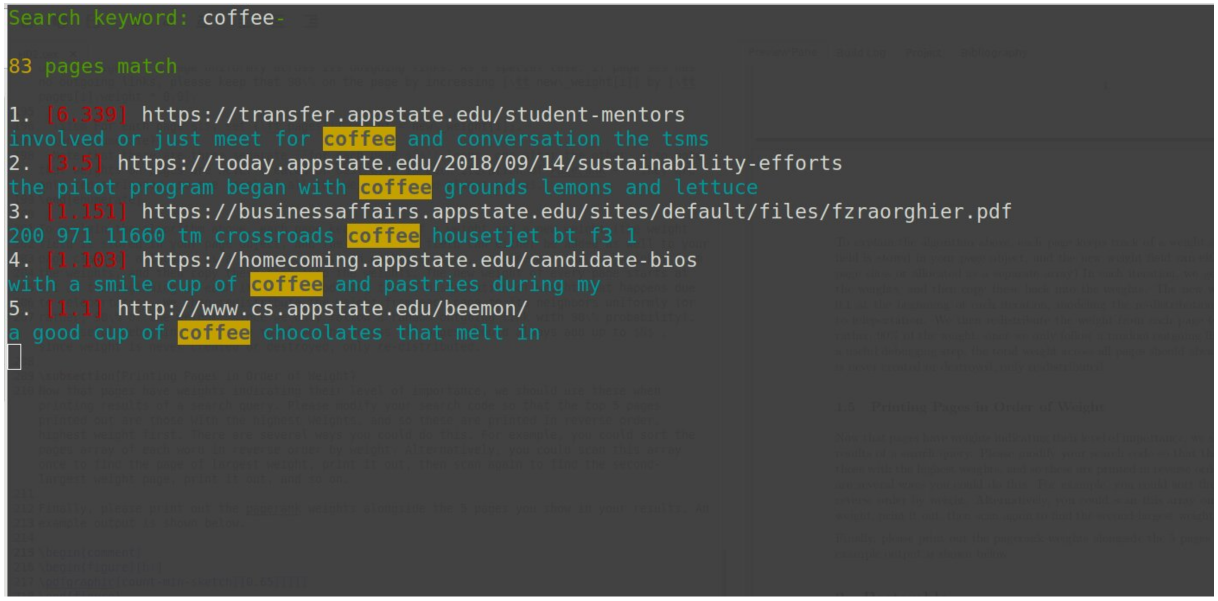
[1] The numbers 10% and 90% are chosen somewhat arbitrarily; we can use whichever percentages ultimately cause our algorithm to perform well. For this assignment, please stick with 10% and 90%.

To explain the algorithm above, each page keeps track of a weight and a new weight (the weight field is stored in your page object, and the new weight field can either be added as well to your page class or allocated as a separate array) In each iteration, we generate the new weights from the weights, and then copy these back into the weights. The new weight of every page starts at 0.1 at the beginning of each iteration, modeling the re-distribution of weight that happens due to teleportation. We then redistribute the weight from each page to its neighbors uniformly (or rather, 90% of the weight, since we only follow a random outgoing link with 90% probability). As a useful debugging step, the total weight across all pages should always add up to N , since weight is never created or destroyed, only re-distributed.

1.5 Printing Pages in Order of Weight

Now that pages have weights indicating their level of importance, we should use these when printing results of a search query. Please modify your search code so that the top 5 pages printed out are those with the highest weights, and so these are printed in reverse order, highest weight first. There are several ways you could do this. For example, you could sort the pages array of each word in reverse order by weight. Alternatively, you could scan this array once to find the page of largest weight, print it out, then scan again to find the second-largest weight page, print it out, and so on.

Finally, please print out the pagerank weights alongside the 5 pages you show in your results. An example output is shown below.



```
Search keyword: coffee

83 pages match

1. [6.339] https://transfer.appstate.edu/student-mentors
involved or just meet for coffee and conversation the tsms
2. [3.5] https://today.appstate.edu/2018/09/14/sustainability-efforts
the pilot program began with coffee grounds lemons and lettuce
3. [1.151] https://businessaffairs.appstate.edu/sites/default/files/fzraorghier.pdf
200 971 11660 tm crossroads coffee housetjet bt f3 1
4. [1.103] https://homecoming.appstate.edu/candidate-bios
with a smile cup of coffee and pastries during my
5. [1.1] http://www.cs.appstate.edu/beemon/
a good cup of coffee chocolates that melt in
```

2 Postamble

2.1 Data Credits

Many thanks to Daniel Jeffries in writing a web crawler to collect all the pages in the ASU domain, and also to format it in a way that can be used in this problem.

2.2 Utility File

The function `process_keystrokes` processes each character of the input, uses ANSI color codes to print the output in different colors, and uses `termios` functions to output the data raw onto the screen, as characters are being typed. This function calls your `predict` function.

2.3 Makefile

As part of the submission, please submit a makefile that includes the following targets with their actions as described.

`copyfiles`	Copy all the test files from the directory specified by the argument `filepath`.
`compile`	Compiles all the files of your solution.
`run`	Runs your solution. There are no arguments.
`list`	Lists the files of your solution.
`show`	Prints your solution, one file at a time.

A sample makefile for my solution is in `makefile`. Please use this to build your own makefile.

2.4 Submission

Please submit all the files as a single zip archive `h05.zip` through Canvas. The zip archive should only contain the individual files of the assignment, and these files should not be inside folders. You can use the `zip` command in Unix for this.

```
$ zip h04.zip <space separated list of files>
```

Please only include files that represent your solution. Please do not include other extraneous files, including pre-compiled class or object files. We will compile your solutions before executing them so these files are not necessary.

2.5 Proper Use of Libraries

For this assignment, you are allowed to use the following data structure utility libraries. In general, you are allowed to use libraries for input and output, math and random functions.

C++	STL libraries `pair`, `vector`, `list`, `deque`, `queue`, `priority_queue`, `stack`, `set`, `multiset`, `map`, `multimap`, `unordered_set`, `unordered_multiset`, `unordered_map`, `unordered_multimap`
Java	`Arrays`, `Stack`, `ArrayList`, `LinkedList`, `TreeSet`, `TreeMap`, `HashSet`, `HashMap`, `Queue`, `Deque`, `PriorityQueue`
Python	tuples, lists, sets and dicts as defined in the Python language. Also allowed are `deque` and `heapq` from `collections`.

You may use these `Geeks for Geeks` resources for more information, or please check out the official language documentations.

C++ STL: `https://www.geeksforgeeks.org/the-c-standard-template-library-stl/`

Java Collection: `https://www.geeksforgeeks.org/collections-in-java-2/`

Python: `https://www.geeksforgeeks.org/python-programming-language/`