CpSc 2120: Algorithms and Data Structures

Instructor: Dr. Brian Dean — Fall 2019
Webpage: `http://www.cs.clemson.edu/~bcdean/` — MW 2:30-3:45
Handout 5: Homework #1 — Daniel 313

1 Writing a Miniature Web Search Engine

For this assignment, you will build a miniature search engine that uses the Google Pagerank algorithm. You will also get plenty more practice using hashing and linked lists.

To simplify this process (and to spare the Clemson network from being oversaturated), your instructor has written a simple web "spider" program that was used a few years ago to download nearly 70,000 web pages in the clemson.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, which is about 250 megabytes in size:

`/group/course/cpsc212/f19/hw01/webpages.txt`

As opposed to the labs, there is no extra code to start with for this homework assignment, although you are encouraged to build on your existing code from the labs. The Stringset class from lab 2 (in particular, modified in class to map strings to ints) may be especially useful, since hashing will be used on several occasions in this exercise. Using the hashing code you have already written in lab, if done carefully, can definitely help you economize on the amount of code you will need to write for this assignment.

2 Reading the Input Data

If you look at the file `webpages.txt`, you will see something like the following:

```
NEWPAGE http://www.clemson.edu
directions
http://clemson.edu/parents/events.html
funds
your
http://www.clemson.edu/visitors
emphasis
http://www.clemson.edu/admissions/undergraduate/requirements
tuition
institutes
visit
```

```
http://www.clemson.edu/giving
annual
academics
http://clemson.edu/academics/majors.html
tour
campus
NEWPAGE http://www.clemson.edu/parents/fund.html
diversity
reader
links
enrichment
call
http://www.clemson.edu/alumni
have
```

The contents of about 70,000 web pages are strung together in this file, one after the other. You will read this file word by word, for example like this:

```
ifstream fin;
string s;

fin.open("webpages.txt");
while (fin >> s) {
  // Process the string s here...
}
fin.close();
```

Whenever you encounter a string "NEWPAGE", the following string is the URL of a webpage, and all the strings after that (until the next "NEWPAGE") are the words and hyperlinks appearing on that page. To simplify the input for you, each word or hyperlink appears listed under a page at most once, even if there are multiple instances of the word or hyperlink on the page.

You can tell the hyperlinks from the words because the links start with "http://". However, some of these hyperlinks point to pages that have no corresponding "NEWPAGE" records for them in the `webpages.txt` file – for example links to sites outside the `clemson.edu` domain. These links should be completely ignored[1].

You should read the contents of `webpages.txt` into memory so that it is stored in a manner that allows you to quickly look up the record for a page in memory given its URL. As we mentioned in class, we can accomplish this by storing page records in a hash table (and this is fine to use for your implementation if you want). The approach we describe here is potentially slightly simpler, particularly if you use a pre-built hash table that maps strings to integers. We store an array of page structures, where an auxiliary hash table maps the URL of a page to its index in this array. Each page is represented in memory as a struct that contains key information about the page,

[1]While reading the file for the first time, it's not easy to know which links are "external" and to be ignored, since a link you encounter might be a reference to a "NEWPAGE" record you simply haven't reached yet. One nice way to address this is to build a hash table of webpage URLs during a preliminary scan through the input file (which you might be doing anyways, to count NEWPAGEs), which can used during a later input scan to filter out bad links.

including its URL, a linked list of words on the page, and a linked list of links on the page (possibly you can keep these in one big list, although keeping two separate lists may be easier). You will eventually need to associate a "weight" (a double) with each page, as part of the Google Pagerank algorithm. A schematic of the resulting structure appears in Figure 1 on the next page.

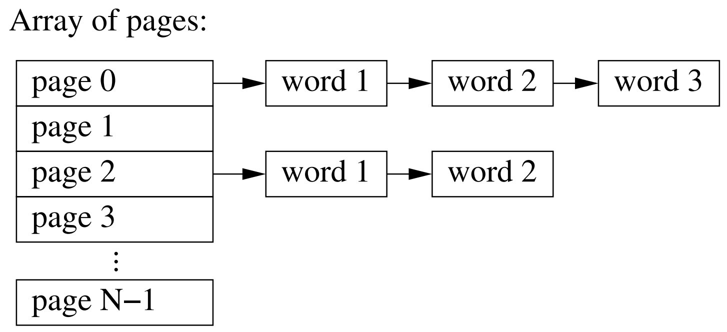

Figure 1: Storing web pages in an array, each page linking to a list of the words on that page (and possibly also a separate list of links on the page, not pictured).

For simplicity, you may want to read through the input file twice – once to count the total number of pages (so you can allocate an array of this size using `new`), and a second time to actually fill in the array[2]. Moving forward, you can now refer to any webpage conveniently by the single integer giving its index within this array. For example, page 0 might be "http://www.clemson.edu". Since it will be useful to have the ability to quickly determine the integer ID of a page given its URL (e.g., to learn that "http://www.clemson.edu" maps to page 0), you also want to build a hash table mapping page URL strings to integer IDs. Please feel encouraged to use your extended Stringset class from lab 2 so that it provides this functionality.

3 First Goal: Implementing the Google Pagerank Algorithm

Recall that 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[3]. 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 amongst 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

[2]You can also use the approach we discussed in class and used in lab 2, where an array is doubled in size every time it fills up. Later, we will get this functionality much more easily when we start using *vector* objects in the C++ standard template library (STL). For this assignment, however, please avoid use of any of the pre-built classes (e.g., vectors, sets, maps) from the STL; part of this exercise is understanding the underlying mechanics of these data structures, so we want to build them ourselves first before moving up a layer of abstraction and using them as black boxes.

[3]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%.

distribution. In pseudocode, this process looks like the following, where N denotes the total number of pages.

```
Give each page initial weight 1 / N
(remember from class 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.  We just need to make sure the initial weights
sum to 1.

Repeat 50 times:
  For each page i, set new_weight[i] = 0.1 / N.
  (new_weight[i] represents the weight of page i after this iteration
  of pagerank.  By starting with 0.1/N, this counts all the weight
  distribution due to teleportation).

  For each page i,
    (suppose page i has t total outgoing links)
    For each page j to which i links,
      Increase new_weight[j] by 0.9 * weight[i] / t.

  For each page i, set weight[i] = 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 1).
```

To explain the algorithm above, each page keeps track of a weight and a new_weight (both of these would be ideal fields to include in your `Page` struct). 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 $0.1/N$ at the beginning of each iteration, modeling the re-distribution of weight that happens due to teleportation: since we teleport with $10\% = 0.1$ probability, this means that the probability we arrive at any one specific page out of our N pages due to teleportation is $0.1/N$. 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).

Your task is to implement the algorithm above. This should be relatively straightforward by following the pseudocode, if you include weight and new_weight fields in your Page structure. The only non-trivial aspect is looping over all the pages j linked to by page i; however, this is nothing more than a walk down the linked list of links included in page i, doing a hash lookup on each one to locate the record of the page to which it links.

If your implementation is taking too long to run, see if you can avoid repeated hash lookups by storing along with each link URL the integer ID of the page it links to. By "caching" the hash outputs this way, we don't need to re-hash the same URL 50 times during the 50 iterations of Pagerank. You should be able to get the entire algorithm, including the code that reads the input file, to run in less than one minute (and it should definitely run in less than 5 minutes). You will be able to test your code after finishing the second part of this assignment.

4 Second Goal: Build an "Inverted Index"

The way we have stored our web pages, as an array of pages each pointing to a linked list of words, there is no efficient way to search for a specific word. Ideally, we would like to type in a word and instantly see a list of pages containing that word, as is the case with most web search engines. We can do this if we build what is sometimes called an "inverted" index, which is an array of words, each one pointing to a linked list of IDs of pages containing that word.

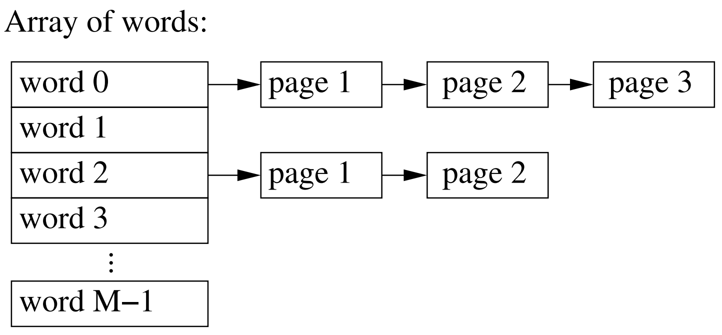

Figure 2: An inverted index consisting of an array of words, each one specifying the pages containing that word.

Just as with the web page array, we can now refer to a word using a numeric ID giving its index within this array. And just as before, we want to use a hash table to map the string representation of a word to its numeric ID, so if the user types in "algorithm", we can look it up in the hash table and see that this is actually the word at index 6 in the table above.

After you have built the inverted index, your program should enter a loop where it asks the user for a string, and then prints out a list of all the webpages containing that string. For example:

```
olympiad
44 http://www.clemson.edu/ces/computing/news-archives/index.html
14 http://features.clemson.edu/creative-services/faculty/2011/following-the-deans
10 http://www.clemson.edu/computing/news-stories/articles/usaco.html
10 http://www.clemson.edu/ces/departments/computing/news-stories/articles/usaco.html
13 http://www.clemson.edu/summer/summer-programs/youth/index.html
12 http://www.clemson.edu/ces/computing/news-archives
10 http://www.clemson.edu/ces/computing/speakers/archive.html
24 http://www.clemson.edu/computing/news-archives/index.html
43 http://www.cs.clemson.edu/~bcdean
10 http://chemistry.clemson.edu/people/lewis.html
21 http://www.clemson.edu/ces/departments/computing/news-archives/index.html
94 http://www.clemson.edu/summer/summer-programs/youth
12 http://www.clemson.edu/ces/computing/news-stories/articles/usaco.html
```

Print the pagerank score alongside each webpage URL. For convenience, please scale the pagerank scores up by a large factor and convert them to integers, so they are easier to interpret. In the example above, the scores are scaled up by a factor of $100N$, where N is the total number of pages.

If you scale up by the same factor, your output should match the output above (possibly in a different order). Ideally, this list would appear in reverse-sorted order, with the highest pagerank score on top. However, since we have not yet covered sorting, do not worry about this step quite yet (although feel welcome to try, if you feel so inclined).

5 Submission and Grading

Please submit your code using handin.cs.clemson.edu, just as with the lab assignments. *Please do not submit the* `webpages.txt` *file!* Your assignment will be graded based on correctness, and also on the clarity and organization of your code. Final submissions are due by 11:59pm on the evening of Saturday, September 28. No late submissions will be accepted.