CpSc 2120: Algorithms and Data Structures

Instructor: Dr. Brian Dean — Fall 2014
Webpage: `http://www.cs.clemson.edu/~bcdean/` — MWF 9:05-9:55
Handout 4: Homework #1 — Vickery 100

1 Writing a Miniature Web Search Engine

For this assignment, you will build a miniature search engine that uses the Google Pagerank algorithm, along with plenty of 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 has downloaded nearly 70,000 web pages in the clemson.edu domain. These 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 file, which is about 250 megabytes in size:

`/group/course/cpsc212/f14/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. In particular, the Stringset class from lab 2 may be quite useful, since hashing will be used on several occasions below.

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
character
deductible
global
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. You can tell the hyperlinks from the words because the links start with "http://".

You should read the contents of `webpages.txt` into memory so that it is stored as an array of pages (i.e., an array of `Page` structs, where you can choose what fields are appropriate to include in this structure). Each page should point to a linked list of all the words and links on the page (if you like, you can store two separate linked lists, one for the words and one for the links). A schematic of the resulting structure appears in Figure 1 on the next page.

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[1]. 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.

[1] In the future, this extra step won't be necessary, since we will be able to use an array that expands by re-allocating memory when needed, such as the vector class in the C++ standard template library.

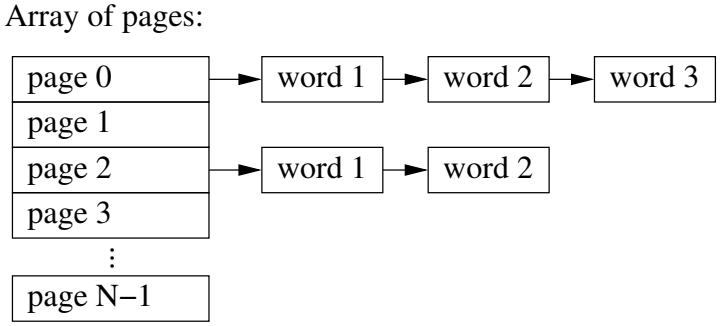

Figure 1: Storing web pages in an array, each page linking to a list of the words on that page.

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

```
Give each page initial weight 1 / N
Repeat 50 times:
  For each page i, set new_weight[i] = 0.1 / N.
  For each page i,
    For each page j (of t total) to which i links,
      Increase new_weight[j] by 0.9 * weight[i] / t.
  For each page i, set weight[i] = new_weight[i].
```

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).

[2]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 feel welcome to stick with 10% and 90%.

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 words included in page i, doing a hash lookup on each one to see if it is a link to a URL that appears in the page list.

If your implementation is taking too long to run, see if you can avoid repeated hash lookups by storing along with each word the integer ID of the page it links to, if any. You should be able to get the entire algorithm, including the code that reads the input file, to run in less than one minute. 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.

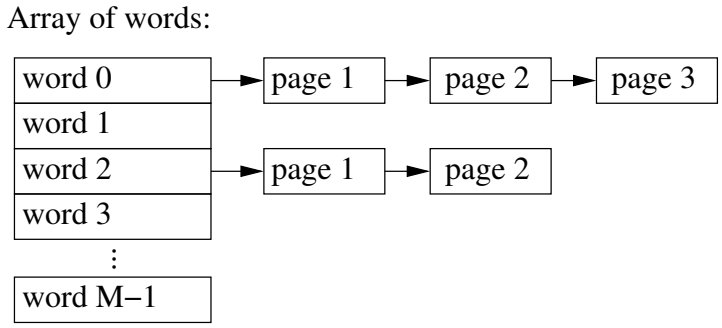

Figure 2: An inverted index consisting of an array of words, each one specifying the pages containing that word.

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
43 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
40 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
108 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, you may want to scale the pagerank scores up by a large factor and convert them to integers, so they are easier to interpret. 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. 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 Friday, September 19. No late submissions will be accepted.