CpSc 2120: Algorithms and Data Structures

Instructor: Dr. Brian Dean — Fall 2020
Webpage: `http://www.cs.clemson.edu/~bcdean/` — MWF 9:05-9:55
Handout 5: Homework #1 — Flour 132

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 roughly 27,000 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 140 megabytes in size:

`/group/course/cpsc212/f20/hw01/webpages.txt`

The same directory also includes some starter utility code that can help you read this file quickly. If you were to read it token by token using an ifstream (our usual method of reading textual input files), this would likely take a long time. To speed things up, the starter code reads the entire file into a single string in memory using the C fread function, which is quite fast at reading large chunks of data from the disk at once. From this string, it then creates an istringstream (input string stream), which you can then read from just like you were reading a file, only now things are much faster since you are parsing data coming from memory and not from disk. See the starter code for specifics.

As another recommendation, you are highly encouraged to use the `StringIntMap` class we developed in lecture 12 to handle your hashing needs. If you use this class in the right ways, you can avoid re-writing hash table code yourself. Moreover, this code gives you a map from strings to ints that you can conveniently access using array notation — e.g., `A[‘‘Brian’’] = 100`.

2 Input Data

If you look at the file `webpages.txt`, you will see something like the following (some content is abbreviated with ...):

```
PAGE https://www.clemson.edu
LINK https://www.clemson.edu/apply-now/index.html
LINK https://www.clemson.edu/giving
```

```
LINK https://newsstand.clemson.edu
...
clemson
university
south
carolina
skip
to
content
clemson
university
let's
begin
...
PAGE https://www.clemson.edu/apply-now/index.html
...
```

The contents of about 27,000 web pages are strung together in this file, one after the other. You will read this file word by word, as shown in the starter code. Whenever you encounter the string "PAGE", the following string is the URL of a webpage, and all the strings after that (until the next "PAGE") are the words and hyperlinks appearing on that page. Hyperlinks are indicated by the string "LINK" followed by the URL of the hyperlink.

This assignment consists of five main parts, each worth up to 5 points. They are roughly sequential in the sense that the kth part. Each part is described in a section below.

3 First Goal: Reading the Input

You should store all webpage data in a large array of webpage structures, defined as follows.

```
\begin{verbatim}
struct Webpage {
  string url;     // e.g., https://www.clemson.edu
  int num_links;  // number of outgoing links on page
  int num_words;  // number of words on the page
  int *links;     // array of page IDs to which this page links
  string *words;  // array of all words on the page
  double weight;  // importance of the page, using the Pagerank algorithm
};

Webpage *pages;   // array of all pages, to be allocated with 'new'
```

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.

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 indicated above as arrays. You are welcome to use linked lists instead, or vectors[1], in which case you should modify the Webpage struct accordingly. Linked lists or vectors 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:

- 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 with the "new" command. 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 "PAGE" 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 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 will 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://www.clemson.edu` would therefore be stored as 0, since `https://www.clemson.edu` will reside at index 0 in our array of webpages. To translate links from strings into these indices quickly, please use a string-to-int 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://www.clemson.edu`. We'll call the index of a page its ID.

4 Second Goal: 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

[1]For this assignment, however, please avoid use of any of the pre-built classes (e.g., sets, maps) *aside from vectors* 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.

build an array of words, using the page array we've already constructed. The words array should be defined like this:

```
struct Word {
  string text;    // the word as a string
  int num_pages;  // number of pages containing this word
  int *pages;     // array of integer IDs of these pages
};

Word *words;
```

You should also build a string-to-int map that quickly tells you the index of a word in this array. For example looking up "clemson" might lead you to the word stored at index 7. Just as with pages, we say the ID of "clemson" 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 string-to-int 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.

Also 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.

5 Third Part: 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[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 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.

[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 stick with 10% and 90%.

```
Give each page initial weight 1
(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.

Repeat 50 times:
  For each page i, set new_weight[i] = 0.1.
  (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,
    For each page j to which i links,
      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)

  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 1).
```

To explain the algorithm above, each page keeps track of a weight and a new_weight (the weight field is stored in your page struct, and the new_weight field can either be added as well to your page struct 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.

6 Fourth Part: 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, for which you could use the built-in `sort` function. 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 is shown at the end of the next section.

7 Fifth Part: Predictive Completion

In this final part of the assignment, you'll print out the most likely next word that should follow the query word, just like how many search engines predict the next word you are likely to type.

Augment your word structure so that each word keeps track of its own string-to-int map storing frequencies of all the words that *immediately* follow it. For example, in the record for the word "clemson", you would expect "tigers" to appear with high frequency in this hash table, since "tigers" is often found right after the word "clemson". The word you should predict next is the one having the highest frequency count.

Final output, including all of the parts above, should look roughly like Figure 1.

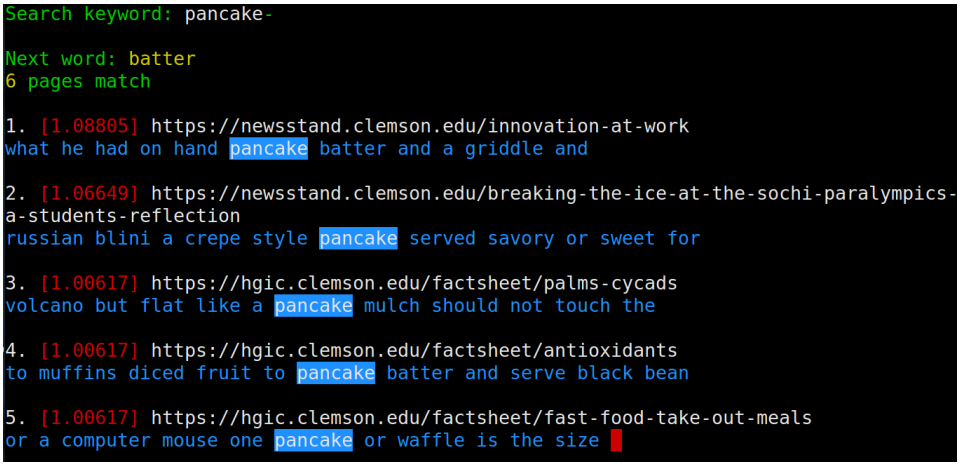

Figure 1: Output of a web search query.

8 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 Wednesday, September 30. No late submissions will be accepted.