CSC326 Lab1 Report

Yefei Li 998920413

Hsuan Hsu 1000389288

Data Submitted: 10/08/2017

Frontend

The front end consists of a simple logo that we came up with along with a simple search bar. The user will be able to search for any search term in the search bar. Once the user clicks search, our code in “server.py” processes the GET request. Essentially, the code inserts every word in the search phrase into a dictionary along with its count. So if there’s a duplicate in the search phrase, the dictionary simply increments the count of that word instead of inserting a duplicate entry. After the dictionary is complete, we construct an html table to show the frequency of each word’s appearance in that phrase in the order that they are first seen. Then, we have a global minimum heap and a global word count dictionary that keep tracks of the top 20 most searched words for as long as the server has been running. We only update this data structure if a word’s count in global word count is greater than the minimum count of the word in the min heap. Once the min heap is updated, we resort it and generate the top 20 search table in the same fashion we generated the first table with. At the end, we return the two tables together.

Backend

In this lab, we implemented two methods **get\_inverted\_index()** and **get\_resolved\_inverted\_index()**. These two methods are getters for the inverted index we created after the crawler runs. The data structure we used to represent the inverted index and resolved inverted index are python dictionary. When the crawler sees a new document to process, it creates a unique doc\_id for it if it has not been seen before, and process the words in that document. Each unique word is assigned to a word\_id, which is used as the key for our inverted index data structure. Each word\_id in the inverted index dictionary is mapped with a list of doc\_ids, meaning it appears in those documents. We also maintained a doc\_id to url dictionary and a word id to word dictionary that are used in the get\_resolved\_inverted\_index() method. All words are represented in lower case.

Tests

We generated several tests to verify the integrity of the output of our crawler data structure. The first three tests test to see if we populate our data structures properly. So if the data structures are expected to have data in them but they don’t, our test cases will fail. Getting into the more interesting test cases, we test whether the we have the correct number of URLs for each word. So for example, if we have a word that appears in two URLs, the URL count for that word should be 2. If it is not, then the test case will fail. After running these test cases, we verified that our crawler algorithm works as intended.

To setup the test cases the user has to:

1. Navigate the ~/ss/crawler\_test\_setup
2. Run “server1.py”, “server2.py”, “server3.py”, and “server4.py” separately.
3. Now navigate to “~/ss” and run “python crawler\_test.py”
4. The test should be run