Kevin Nguyen
EID: kdn433
Project 3 - CS371r

Project 3 Implementation

The purpose of project 3 is to implement page ranking algorithm with a set pages that need to be spidered. Spidering works by starting at a root node (root directory) and traveling along outgoing links until all paths have been followed. The page ranks show how relevant or popular a particular page is and the pages may not all have the exact same ranking. In fact, the pages could be dynamic; as a result, page rankings don't remain consistent. The additional features that were made to the project have not changed the original functionality of spidering. There are three extended files and they're PageRankSpider.java, PageRankingSiteSpider.java, and PageRankInvertedIndex.java. These three files will assist the original program by overriding and adding new components for page ranking and spidering. We will discuss the three files in more detail and then discuss some materials after experimenting. The overall approach of the algorithm was to break up the problem into parts and then solving each sub problem one by one. When each of the problems have been solved, the components are put together (with adjustments) to make the project complete.

The PageRankSpider.java class extends to Spider.java and it will implement the page ranking algorithm. The page ranking variables are named after the CS371r lecture slides on page ranking. There are also Hashmaps to serve as storage for the ranking and to be references that could be needed later. Some of the methods have been overridden such as go, processArgs, doCrawl, and indexPage. The go function directs the program to processArgs and then doCrawl respectively. ProcessArgs will parse the command line arguments and set the variables as needed. However, the directory name has been saved to store the .html files and page_rank.txt. DoCrawl function utilizes a queue to travel through each outgoing link; as a result, the travel is a breadth-first search method. When the outgoing links have been retrieved, we add to the graph and add the edge as we see them; pages and edges are only added to the graph if those links can be indexed. The getNewLinkEdges function apply the edges and pages to the graph as needed. And indexPage function (if indexing is allowed) will index any leftover pages and write the indexed pages to a file. When the spidering has been completed, the next step is to print out the graph and initiate the page ranking algorithm. The setInitialRanking function will populate the hashmaps to initial rankings. GetPageRankings functions will complete the page ranking algorithm by iterating through the set of pages multiple times for convergence. This part of the algorithm tries to obtain the summation of R(q) divided by Nq from every incoming link to our current page. And those values are multiplied by "(1-alpha)" and added by an E(p) value to get the page rank; page ranks are normalized with a c constant as well. Those results are stored temporarily for later use. When the entire iteration run has been completed, the temporary values are then written to the other hashmap, which is the real list of page rankings. After that function is complete, the next function is printRanksToFile function which will print the page and its page rank to a file called page_rank.txt. Finally, the page ranks are also

printed to the screen in printRanking function. Overall, this class implements the bulk of the algorithm for project 3.

PageRankSiteSpider.java extends PageRankSpider and it will restrict the spidering to a certain site. However, this class will also serve as the main entry point of execution. The overridden function is called getNewLinks. This function will mainly take in a list of links and filter out links that stray away from the site. The list is then returned as filtered list. This class is the most simple to implement since most of the functionality already exist and has been re-used.

PageRankInvertedIndex.java class will extend the InvertedIndex to get the relevant documents for querying. It should be noted that the global variables are copied so that the super constructor can be invoked. However, we will also save the weight value indicated by the user and multiply the corresponding page rank with the weight in addition to the original cosine similarity score. The main function will act as the main entry point for this class and it will parse the command arguments as normal; with additional weight logic added. The ranks are then read from the file, page_ranks.txt, and stored into a hashmap; this function is called readRanksFromFile. Retrieve function (overridden) will recompute with page ranking and weighting to the original score after indexing has been completed by the InvertedIndex class. ComputeFinalScore function will take and return the score added by the weight multiplied by the corresponding page rank. The new score is then outputted on the screen instead. Querying and indexing will still work as normal as the original InvertedIndex was not changed.

The implementation described above is primarily for page ranking. It should be noted that the retrieval process has not been changed significantly because the InvertedIndex remains the same and the incoming links are added to the graph as they are seen.

"Q1.* Does PageRank seem to have an effect on the quality of your results, as compared to the original retrieval code? Why or why not?*..?"

The page ranks appear to have a slight improvement to the quality of the document retrievals. Page ranks represent the popularity or authority of a page (document); as a result, the weighted page rank in addition to the similarity score improves the score. The pages are displayed with more weight to show that popular pages get popular because those pages are more likely to be referenced. The more relevant documents are displayed on top as usual and it may change based on the increasing quality of documents due to page ranking.

"Q2. How does varying weight change your results? Why do you think it changes in this way?"

The different amount of weights have changed the score slightly by increasing the original scores. The factor between weights doesn't appear too significant, but the addition of the page rank multiplied by the weight increases the score value. As a result, the higher weight values show more importance to a document. And more weight means (possibly) higher relevant

documents. However, no weight could mean small or no increases to the similarity score of each retrieved document.