I427 Final Report

Jeffery Gough, Shaowen Ren
http://cgi.soic.indiana.edu/~shaoren/gough-ren-search.html
[other URL here]

Introduction:

For our I427: Search Informatics final project, we were tasked with creating a fully-functional search engine. Accomplishing this required using every major project we have completed over the semester; however, most of the projects had been largely unfinished and insufficiently tested, so stringing all of them together required significant rebuilding, debugging, and more testing. In addition to using our previous projects, there were several more tasks we needed to complete. This report will cover the final project in its entirety.
The search engine features a total of 5000 indexed pages and over 100,000 unique words. Crawling the pages began with the seed url http://gawker.com and the idea was to focus on both news and entertainment.
The front end features bootstrap styling, a logo, a search bar, and two links at the top. One of them goes to an experimental network graph created from crawling the pages, and the other goes to the final report.

Initiating the Search Engine: *run_setup.sh*

(or MPI_run_setup.sh, see "Extra-credit: Multi-core Processing" section below)

Multiple Ruby files are used to set up the search engine and prepare it for use by an end-user. These files are used to crawl the web, download webpages, create hash maps detailing the locations and occurrences of words, create a network map hash showing the relationship between web pages, and calculate a PageRank algorithm for every page, based on the network map. In order to facilitate ease of use for a developer setting up the engine, we made a shell script, *run_setup.sh*, which runs each file in succession (except for hashtojson.rb, which is run after the shell script).
The only command needed to set up the search engine is

```
$ ./run_setup.sh
```

but listed below each section will be an example each file being individually run from the command line.

Web Crawler: *newcrawler.rb*

```
$ ruby newcrawler.rb
```

The first part of any search engine is a web crawler. A web crawler basically looks all over the internet and tries to find web pages that it has not seen or that have recently changed. A web crawler uses a specific algorithm in order to keep track of the order in which it crawls pages. We implemented a breadth-first search algorithm. Upon accessing a web page (the first one is supplied by a hard-coded seed url) the crawler downloads the page and finds all of the links on that page. It sends the links to the bottom of a list upon which is constantly being iterated. All of the links on that webpage will be visited, and each one's source saved, along with a list of that page's links being sent to the bottom of the list being iterated upon. As soon as all of a page's links have been visited, the crawler will move onto the next page, and the process begins again. Each page is saved with a filename showing the order in which it was downloaded. For 1000 crawled pages, the filenames range from 0.html to 999.html. The crawler creates a file called *index.dat* which is used later on to match filenames with respective URLs. Furthermore, at the end of the crawler's operation, a file called *dict.dat* is created in order to keep track of outgoing links. The file stores a hash, or dictionary, of every page's links. It uses a URL for each key and stores a list of that page's outgoing link URLs as the hash value. An empty list denotes no outgoing links.

Note: The pages that are saved in this hash have already been downloaded. There may be more links outgoing from the URL key than the ones recorded, but the hash only records outgoing links whose pages have been downloaded. Many more links will be seen than will be downloaded, especially when only crawling a few thousand pages. Recording *all* links seen is possible, but isn't necessary when calculating a page's PageRank score, when relationships between pages matter (see PageRank).

```
Inputs:
     Seed Webpage, Number of web pages to crawl.
Outputs:
     Web page downloads
     index.dat
          Sample:
          1.html http://cnn.com
          2.html http://huffingtonpost.com

     dict.dat
          Sample:
          {"http://www.cnn.com/2015/12/11/politics/donald-trump-
          ted-cruz-iowa-ethanol/index.html"=>["http://www.cnn.co
          m/2015/12/11/politics/ted-cruz-donald-trump-leaked-aud
          io/index.html",
```

```
"http://www.cnn.com/2015/12/08/politics/ted-cruz-donal
d-trump-disagreed/index.html",
"http://www.cnn.com/2015/12/10/politics/donald-trump-t
ed-cruz-iowa-caucus-ground-game/index.html"],
"http://www.cnn.com/2015/12/12/politics/ted-cruz-donal
d-trump-iowa-poll/index.html"=>[],}
```

Indexer: *indexer.rb*

```
$ ruby indexer.rb pagedata/ index.dat
```

After crawling and downloading the web pages, the next step in the process is to parse each page, storing the occurrence of every unique word by page. This step is arguably the most important task because it allows the pages to be searchable.

This program first finds all of the words an html document, parses them into stems (simple tokens, using the base form of words) using the Ruby fast-stemmer gem, then creates a 3 dimensional hash, or dictionary, sorted by word, then by page, to keep track of word appearance. This dictionary is vital to finding the right page after submitting a search query. To create it, the words in each document are inserted into a preliminary dictionary to rid duplicates. Each word key has a blank value. That dictionary is saved and stored for later. Then, the words in the dictionary are looped through, and for each word, the program searches through every HTML file (every word in the HTML files are saved into a separate dictionary called htmlDict, allowing for fast searching of words. {"1.html"=>["word", "word", "word"....]}. This dictionary's key is an html file, and the value is a list of all tokens, duplicates allowed) and runs a simple count method on the dictionary key's list to find the count for the word. A new dictionary, subHash, is constructed for every word, and the count of each word in htmlDict's value list goes toward that page's value.

This program yields 2 hash tables, each one saved to a file. The first one, invindex.dat, records occurrences of words by the pages in which they show up. It keeps track of every unique word, where one can find each word, and the number of times the word appears on each page. The second hash is called docs.dat. This hash simply keeps track of three things for every page: the number of tokens (words) on the page, the page's title, and the URL for the page. Docs.dat is necessary for use by *retrieve.rb* when the search results are displayed for the user (Title and URL).

Quick explanation of every hash:

```
htmlDict: {"1.html"=>["all", "the", "tokens"....],
"2.html"=>["more", "tokens"...]}
```

^Created during the initial HTML token parsing loop allowing fast searches

subHash: {"1.html" => 2, "2.html" => 3}
^New one created after every word. Appended to superHash. Keeps word count per page.

superHash: {"informatic" => {"1.html" => 2, "2.html" => 3"}, "hello" => {"2.html"=>7, "23.html"=>1} }
^Initially blank to kill duplicates. Populated later in the program. Sub hash appended to superHash.

inputs:

Saved webpages in directory

index.dat

outputs:

invindex.dat

Sample:

{"informatic"=>{"1.html"=>4, "23.html"=> 6, "24.html"=>3}, video=>{"10.html"=>2}}

docs.dat

{"22.html"=>[1546, "Advertise - CNN.com", "http://www.cnn.com/services/advertise/specs/specs_overview.html"], "16.html"=>[6427, "COP21: Is 2 degrees the wrong climate goal? (Opinion) - CNN.com", "http://www.cnn.com/2015/12/08/opinions/sutter-1-5-degrees-climate-cop21/index.html"]}

docs.dat

Sample:

{"22.html"=>[1080, "Inside the Paris Climate Deal - The New York Times", "http://www.nytimes.com/interactive/2015/12/12/world/paris-climate-change-deal-explainer.html"], "25.html"=>[14058, "Speaking with One Voice to Solve the Climate Crisis | Al Gore",

```
"http://www.huffingtonpost.com/al-gore/speaking-with-o
ne-voice-to-solve-the-climate-crisis_b_8794402.html?ut
m_hp_ref=yahoo&ir=Yahoo"]}
```

Link Network Map *linksnet.rb*

```
$ linksnet.rb
```

Creating a network map was vital in the calculation of a PageRank algorithm. This program simply converts the *dict.dat* file into *linksnet.dat*. The major difference between the two files is that *linksnet.dat* uses filenames, whereas *dict.dat* uses URLs.

```
Output:
     Linksnet.dat
          {"0.html"=>["1.html", "2.html", "3.html", "4.html",
          "5.html", "6.html", "7.html", "8.html", "9.html",
          "10.html", "11.html"], "1.html"=>["12.html",
          "13.html", "14.html"], "2.html"=>[]}
```

PageRank Scoring *pageranker.rb*

```
$ pageranker.rb <delta factor> <damping factor>
```

```
Input: pagedata/linksnet.dat
Output: each page's pagerank score in pagedata/pagerank.dat
{"0.html"=>0.0912342,"1.html"=>0.2342312,"2.html"=>0.001235,....
}
```

(default: delta = 0.001, damping factor = 0.85)

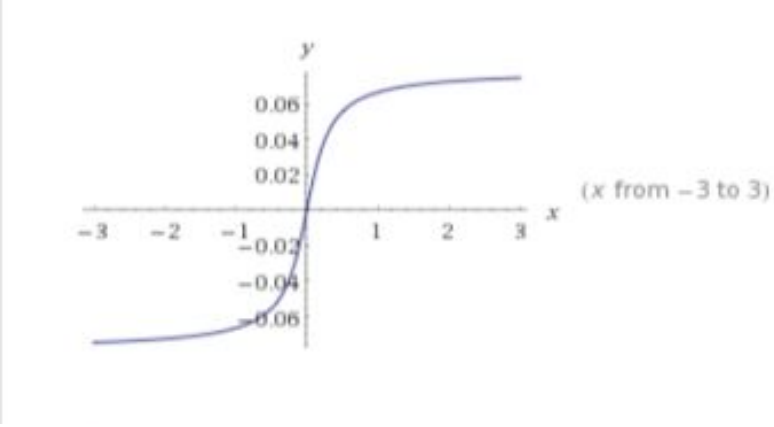
Besides the normal way to do the PageRank scoring, we added a factor, called delta factor. This factor aims to reduce the running time that might be considered "waste" in the calculation iteration of PageRank. Because along with the calculation in each iteration, the PR value of each page will converge to a stable score. In this case, conducting more iterations will be considered a waste of time. By setting the delta factor by the end of each iteration, we compare each difference between each old PR to the new PR, and if the difference is less than delta, we can quit the loop. This saves calculation time which would otherwise be wasted.

Retrieval/TF-IDF Scoring *retrieve.rb (class file, cannot execute)*

This file is used by the "*gough-ren-search-retrieve.cgi*" in the *cgi-pub/* . It is the final step in our back end.
How to combine TF-IDF score and pagerank score is a pay-off choice. By experimenting it with many users, we finally decided to use this way to combine them. That is:

TF-IDF * arctan(4*PageRank)/20

Because PageRank seems to have less importance in the purpose of finding the keyworks out. Most of the TF-IDF score are less than 0.001, no score are greater than 0.01.But PageRank will have very large score compare with it, and the score of PageRank are distributing very different.
graph for arctan(4x)/20:



This function makes little pagerank value becomes a little greater, but not increase a lot for larger PR value. Also ,the upper bound of this function is around 0.078, compare with the sample test PageRank largest result 0.2635129761849005, this is a lot less than it.

(Extra Credit) Multi-core Processing *MPIPageRank.java*

(requiring the result file from linksnet.rb)

```
$export MPJ_HOME=mpj-v0_38/
$export PATH=$PATH:$MPJ_HOME/bin
$ruby MPI_input_converter.rb

$javac -cp .:$MPJ_HOME/lib/mpj.jar MPIPageRank.java
$mpjrun.sh -np <number of processors> MPIPageRank <input> <output>
<delta> <damping factor> <iterations>
```

(default: pagedata/MPI_linksnet.dat pagedata/pagerank.dat delta = 0.001, damping = 0.85, iteration = 30)

In real world, the pages that need to be crawled is over billions. In this case, the speed of computing is significant important. Single processor will takes years to complete all this jobs. Thus, multi-process computing is strongly required. Under the thought of Distributed Computing, PageRank is a perfect example to implement Message-passing-interface. By dividing the work to each processors, let them compute part of the page's ranking score, and then sum them up to the main processor, will surely reduce a lot of running time it need.

Because of the silo permission issue, I was not able to installing ruby-mpi gem to the root. But I am allowed to using Java version of MPI(mpj) on our silo. Thus, this MPI version of PageRank is separately implemented in Java. And if you want to run the this version of PageRank, you can follow the instruction below:

In the jvgough-shaoren-final/ directory, run:

```
$ ./MPI_run_setup.sh
```

if it not working, type this first:

```
$ chmod 755 MPI_run_setup.sh
```

`MPI_run_setup.sh` will first setup the enviornment variable for the *"./mpj-v0_38/"*(which is included in our repository. It is the Java library of MPI).

Then, convert the *linksnet.dat*, which produced by *linksnet.rb*, to the format that *MPIPageRank.java* can read (*"MPI_linksnet.dat"*).
At the end, compile *MPIPageRank.java*.
Run it with default 8 processors, delta factor = 0.001, damping factor = 0.85, iteration = 30.
Everything else is the same with *run_setup.sh.*
(So `MPI_run_setup.sh` can be runned for initially setting up everything for our search engine. The only different is the way of calculating PageRank)

If you want to manually running the *MPIPageRank.java*, Here is the commands you need:

```
$export MPJ_HOME=mpj-v0_38/
$export PATH=$PATH:$MPJ_HOME/bin
$ruby MPI_input_converter.rb #linkesnet.dat needed
$javac -cp .:$MPJ_HOME/lib/mpj.jar MPIPageRank.java
$mpjrun.sh -np 8 MPIPageRank pagedata/MPI_linksnet.dat
pagedata/pagerank.dat 0.001 0.85 30
```

(all the commands above are included in *MPI_run_setup.sh*)

Do not change the location of any file or directory, that will make program doesn't work.

Network Visualization *hashtojson.rb*

```
$ ruby hashtojson.rb
```

What began as an afterthought turned into an actual experimental feature as we tried to figure out how we could turn our data into a graphic visualization for the user. After crawling the pages, a *linksnet.rb* file creates a *linksnet.dat* file detailing a directed network in the form of a dictionary, or hash. We created a file which uses this .dat hash to create a JSON file which gets exported to the cgi-pub directory and creates a link-node relationship (NOTE: You must edit the file to output to YOUR CGI-PUB directory!). This JSON file is used by network.html. There are currently too many nodes for the graph to handle, so we will still be working on the network.html Javascript until we get it right.

Extra Feature List:

- Delta factor in *pageranker.rb* & *MPIPageRank.java*
- Shell Script (*run_setup.sh* & *MPI_run_setup.sh*) that allow you to initialazing our Search Engine in one click.
- Experimental Network Visualization feature

Folder List:

- *pagedata/* :include every data file and html pages generated by initialization programs(*run_setup.sh / MPI_run_setup.sh*)
- *mpj-v0_38/* :include the MPI java library
- *cgi_backup/* : include the front-end files from ~/cgi-pub/, not useable in this directory

References:

- *http://sourceforge.net/projects/mpjexpress/files/releases/*
- *http://mpj-express.org/docs/guides/linuxguide.pdf*