## Overall design of the system

## The file structures used in the index database

For forward index, we only have 1 Htree in JDBM database. The key is the URL ID which is an integer and given a key it returns a HashMap<Integer, Vector<Integer> > where the key of HashMap is the Word ID and the value, whose type is Vector<Integer>, store the position of all the given word in the specified URL. The positions are all absolute positions, we don’t compress index and postings files by just record the differences between successive document Ids or word positions.

For inverted index, we have 2 Htree in JDBM database and both of them use Word ID as the key. The first one stores the position of the words. Given a Word ID, it returns a HashMap<Integer, Vector<Integer> > where the key of HashMap is the URL ID and the value, whose type is Vector<Integer>, store the position of all the given word in the specified URL. The positions are also all absolute positions. The second one stores the term frequency. Given a Word ID, it returns a HashMap<Integer, Integer> where the first integer which is the key of HashMap stores the URL ID and the second integer which is the value of the HashMap stores the term frequency.

The index of titles has its own JDBM databases which are similar to the forward and inverted index described before.

## Algorithms used (including the mechanism for favoring title matches)

For recursively fetch the required number of pages, we use a queue to allow the spider fetch pages using breadth-first strategy. To avoid fetch the pages that the spider already fetches in this run, we use a HashSet to store the URL string of pages that the spider already fetches. When spider finds a link, it will first check whether the HashSet contains this link, if not, the spider will add it to the end of the queue and then add it to the HashSet.

For stemming, we use the Porter’s algorithm provided on the course website.

For the mechanism for favoring title matches, we decide to use a title score which is calculate by using the tf \* idf / max(tf) in the special inverted file as the weight and using the cosine similarity with the query. After we have the title score, we multiple it by a constant, which is 10, then add it to the page body score the get the final score.

## Installation procedure (it could be as simple as “Type make in the project directory”)

1. The installation procedure is for windows users only
2. Install Tomcat 9.0 server in your computer first
3. Unzip the finalPhase.zip. place the result folder under webapps directory
4. Create a folder called db under disk D and copy stopwords.txt to the db directory, the stopwords.txt file can be found in finalPhase/src
5. Double click on spider.jar inside finalPhase to start our spider program
6. After finishing fetching index from the website, you should find a list of .db file and .lg files in D:/db
7. Restart your tomcat server and visit our index.html file from your browser. The address depends on you tomcat setting.

## Highlight of features beyond the required specification

We implement PageRank use the method told in class. We also use the damping factor and set it to 0.85. The PageRank is implemented as a separate module. After the spider finishes fetching pages, the updatePageRank() method is called to calculate the PageRank scores of all fetched pages and save the scores to JDBM database. When scoring page, if PageRank score is needed, getRank(int URLID) method can return the PageRank score from JDBM database.

We implement the relevance feedback feature “get similar pages”. If the user clicks the “get similar pages” button of a search result, it will extract the top 5 most frequent stemmed keywords (excluding stop words) from the specified page using the forward index database and use the top 5 most frequent keywords as a new query to launch a new search.

We allow the user to see a list of stemmed keywords indexed in our database, browse through them, and select the keywords he/she is interested in, and then submit them as a new query to our search engine. Since there are too many stemmed keywords in our database, we decide to show the stemmed keywords in the page with highest PageRank score first since PageRank score is independent from the query. We also limit the number of stemmed keywords showed in the webpage since if we generate a list with too many keywords, the webpage will take a long time to generate and occupy a large number of memory.

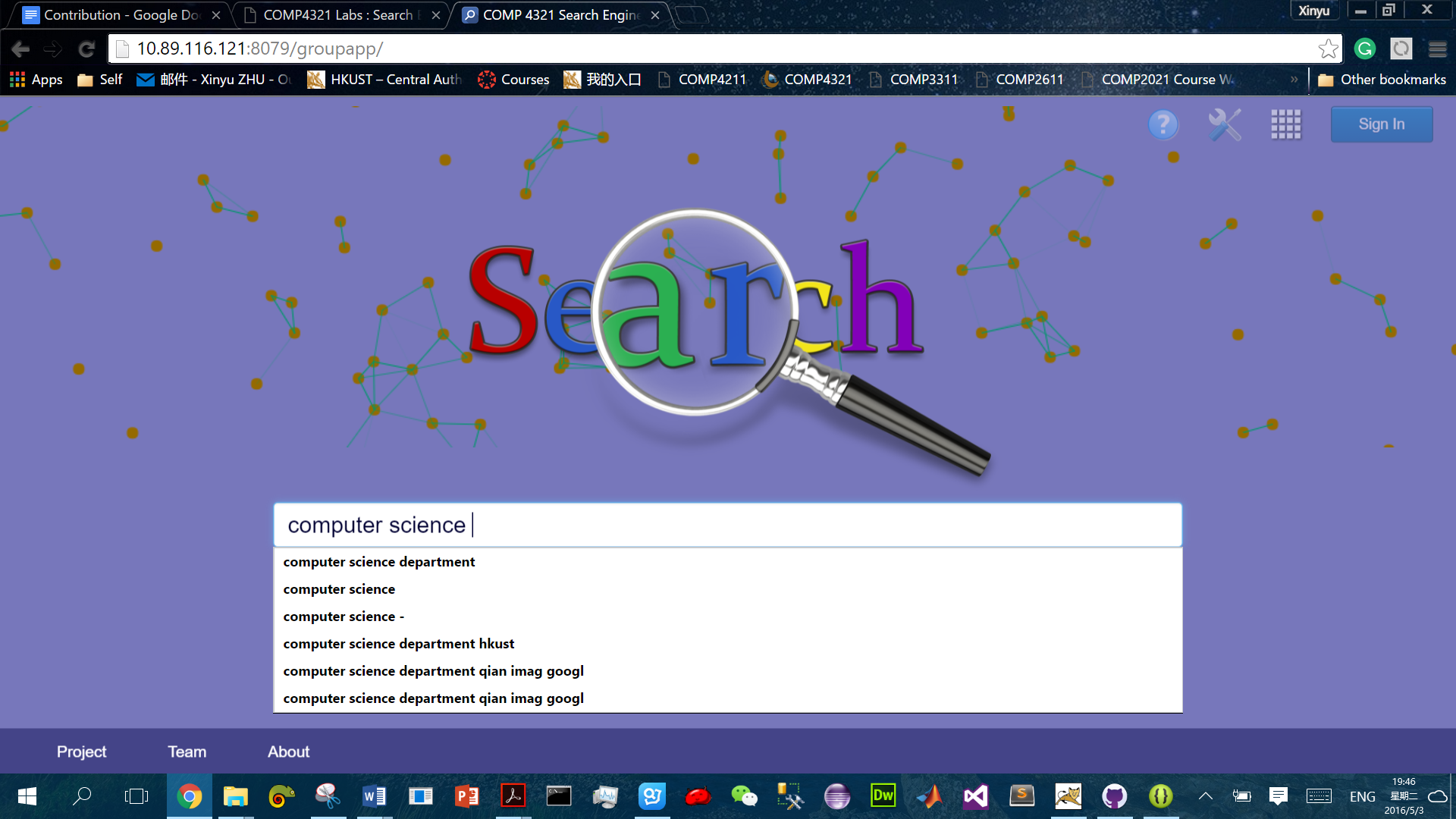
We spend a large number of time on designing the search engine webpage and the user-interface. We think our user-interface is easy to use and attractive. We also highlight the matched part of search results like Google.

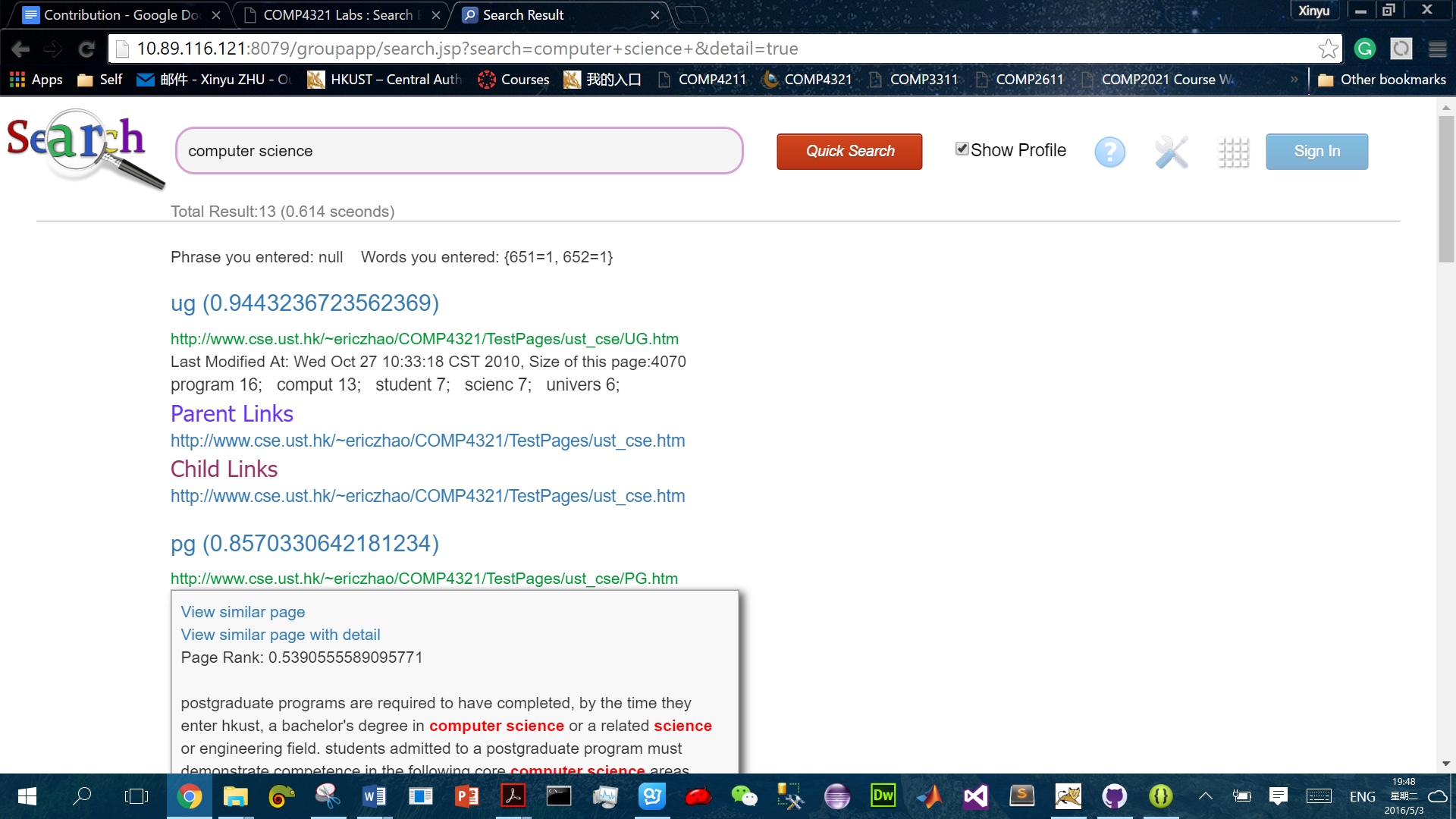
Also, we have a GUI to control the spider. In the GUI, it allows user to indicate the starting URL and number of pages to be fetched.

## Testing of the functions implemented; include screenshots if applicable in the report

There are many functions in our system, here we only show the testing for some of them:

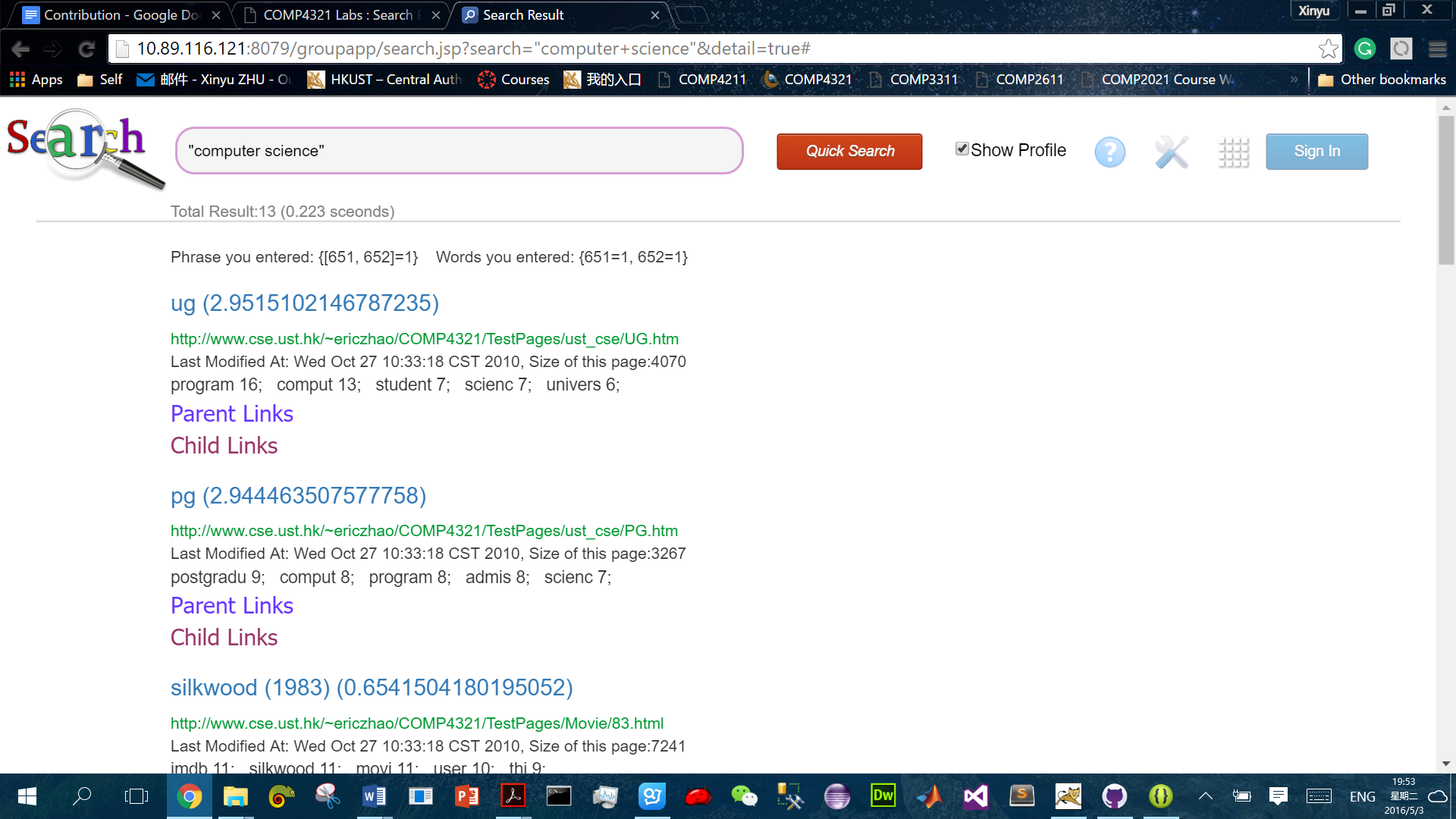
1 User inferface



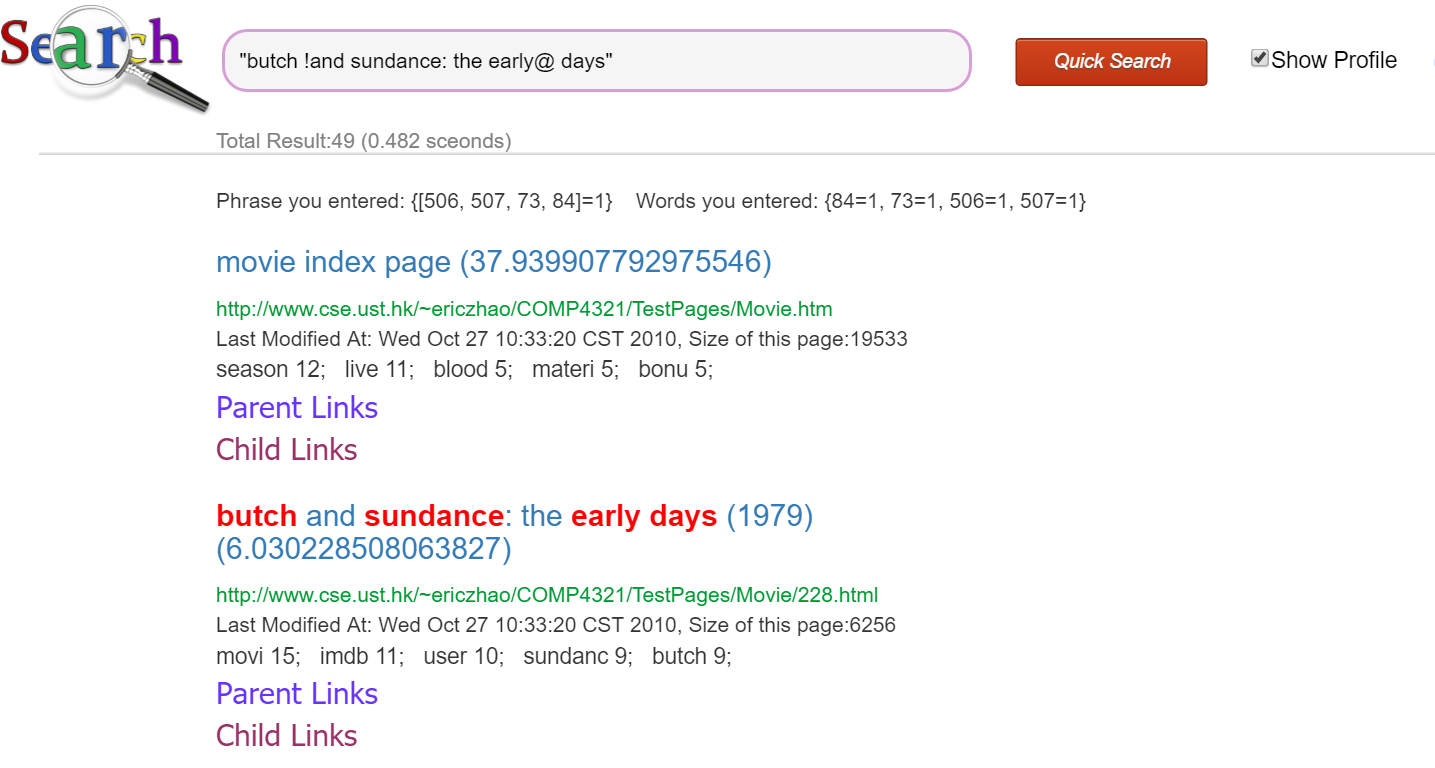


You need to Click on Parent Links and Child Links to unfold those detailed child and parent links. When your mouse is over the link, a little suspending window will appear to show you more information about that page. You need to check Show Profile before searching if you want to read the page profile. You can also see the similar page search function here.



Click on the Tools icon to open the stemmed word list. You can click on those button which will add the corresponding stemmed word to the search bar. 

You can add double quotation marks if you want to search some words as a group (phrase search) you can see the score of those pages with this phrase increases after adding double quotation marks.



Our search bar will ignore meaningless marks, you can also see title boosting in the above example. The first page get a very high score because its PageRank is very high.

## Conclusion: What are the strengths and weaknesses of your systems; what you would have done differently if you could re-implement the whole system; what would be the interesting features to add to your system, etc.

Strengths:

We think our search result is accurate. We use spider to fetch some pages on Wikipedia. Then we use a small set of keyword such as “History of computer science” to test our search engine and compare the result with the result of Google. The top 1 result is the same in most cases. But for top 10 results, since we only fetch a small number of pages, Google will always return some pages that we don’t fetch and it’s hard for us to compare the result since we don’t have enough computational resource to fetch a large number pages on Internet and store all the data that a search engine need about those webpages.

Also, we do many optimizations on the performance.

For URL objects, we only save the string representations and create new URL objects from the strings when necessary. This can avoid using the problematic equals() and hashcode() methods, which are often used implicitly when using hash tables and other hash containers, of URL class in Java. Those 2 methods will access the Internet, do a DNS name lookup to get resolved IP address, connect to the IP and get the page file since “Two URL objects are equal if they have the same protocol, reference equivalent hosts, have the same port number on the host, and the same file and fragment of the file.”, according to the Java docs. This is certainly a disaster. A larger amount of time can be wasted if URL objects are saved in Hash Tables such as the tables for the URL to URL ID conversion. First, since hosts comparison requires name resolution, this operation is a blocking operation and will slow down page fetching. Also, if the network is busy or the target URL has a long response time, the spider need to wait for it for a long time until it gets the response. Sometimes, the spider need to wait more than 10 seconds when the equals() and hashcode() is applied on some URLs. That is extremely expensive since in normal case it only takes 500~1000 ms to fetch a page. What’s more, the equals() method depends on the environment since it can be affected by the virtual hosting. For the above reasons, we think that URL objects should never be put into containers that use hash.

For getting the title of the page, we write a function instead of using the htmlparser to filter out the title node and read the context of title node. Our function reads the page line by line and returns the title immediately when tag of title is found. According to our observations, about 15% of time is used to get the page title when fetching and our method is about 20% faster than using the title node of htmlparser.

Also, when fetching links, we truncate the “#......”, which will guide you to a specified position of the target page, in the URL. This is helpful since it can help us avoid fetching pages that already fetch and ensure no duplicate parent/child link relation will be saved into database. It also prevents a page from pointing to another position of itself in link relation.

What’s more, we deal with the case 302 Found by changing the protocol to https when we meet 302 Found. This allow us fetch the correct pages into database instead of fetching the 302 Found page.

When we implement the whole system, we divide the functions of the system into different modules and implement the modules separately. Each module has its own java class and interface so that when we change the implementation of one modules, other modules will not be affected. Also, it will be easier for us to cooperate since the interfaces allow us to use the modules without understanding how they are implemented. This not only improve the productivity in team development, but also make our system more maintainable and reusable.

Weakness: When fetch a page, we notice that about half of the time is using on extract all the links in the page. We think this is somewhat slow. If we have enough time, we will try to find a faster way to extract all the links instead of using the Link Bean provided by the htmlparser.

We didn’t gather user feedback because it is too complex to realize (For example, when user click on a link, sent it to another server pregram). We have’t get many users, a little feedback can’t improve our system a lot. Maybe in the future, we will implement this.

We didn’t limit the number of returned result to 50 pages for the following reasons, first we print the result one by one so you can read some results even if we haven’t print all the results. Second, we just want to test the accuracy of our algorithm so we hope to see the exact number of pages containing the keywords we inputed.

If we could re-implement the whole system, we will try to find a way to decide when to write to disk during fetching such that the usage of memory and the I/O waiting time can be balanced. In our implementation, we will write to disk when finishing fetching a new page. This can save main memory space but will spend more time on I/O. This is also more reliable than only writing to disk when spider has fetched all the pages. If the spider is terminated by accident while running, we don’t need to fetch the pages that we already fetch before termination. Apart from that, we may also consider how to synchronize multiple spider program so that we can gather index from the Internet more efficiently.