Design and Implementation of a Search Engine Group Project Report

## XU Pengfei (12251054) Ting Ho Shing (14206692)

# Overall

The program has two parts, the gathering information part and the search engine part. We will briefly explain them one by one.

# Gathering Information

We design and write a program named WebSpider to gather the information from the given URL. The program will ask the user to input the URL, the value of X and Y and the program will run on its own. For demonstration purpose, the program will ask if you want to run the demo with the URL set as the main page of HKBU, X = 10, and Y = 100.

Every webspider will have these attributes:

1. **urlString** , a string that stores the url string of the current page;
2. **url**, URL object defined in Java API used to get the html file;
3. **domain**, a string that stores the domain of the current page;
4. **title**, a string containing the webpage’s title;
5. **x**, an integer that the spider should know how many links it should be extract from a page;
6. **y**, an integer that the spider should know how many links should be extract from all pages;
7. **inputOk**, a Boolean indicates that the user inputs are fine;
8. **URLPool**, a linked list that stores the urls extracted from current page;
9. **Keywords**, a linked list that stores the keywords of current page;
10. **KeywordNodes**, a linked list that stores the the keywords as well as the number of keywords in current page;

A spider will do two things basically, to find the hyperlinks and keywords in its page. The spider will request the HTML file from its stored url and read the HTML file line by line to extract useful information. Before we start to process the HTML file, we will check whether this page is a real one rather than error pages like 404 error ones. In case there are other unexpected errors on a page, the corresponding spider will kill itself and add the URL it contains to DeadLinkPool. Also, those error pages will not add to ProceeedURLPool. All the links in ProceedURLPool are openable pages. More details of how we process the links will be explained later.

## Extract keywords

In this program, we mainly focus on special tags that we think are the most possible ones that have meaningful contents to extract keywords in order to save time. So when the program read a new line, it will check if this line contains any of these tags we defined in string array KeywordTag. The spider will just ignore those lines which do not contain KeywordTag. Then a string array will store the candidate key words by splitting the line by space. Each candidate keyword will be sent to isAWord method of WordChecker class which looks up in an English word dictionary database called Wordnet. If the candidate keywords are the real English word and are not an element in StopList that is a string array stores meaningless English words, we will deal with it. If it's the first time we meet this word we will add the word into Keywords and create a new keywordNode for this keyword. If this keyword has been added before we will increment the number of its appearance by 1 in keywordNode. It's also possible that the keyword is an English phrase consists of two words that are not English words. For example, Hong Kong is an English word but Hong and Kong are not. So the spider will check the current word if it's not the first one and the one before it combining together to see if the phrase is an English phrase and will be dealt the same way as the single English word.

## Extract Hyperlinks

For the same line of HTML, the spider will also extract the hyperlinks from the tag contains href. Here we only consider the links in the main body i.e. the lines after reaching the <body> tag. Besides that, we filter out links that contain pdfs, pictures, Javascripts as well as css files. What we want is actual web pages other than nothing. After we extract a link we will check if this link is suitable for further processing. If it is, the spider will create a new spider with this link as its attributes and run this spider. In this way, the spider will automatically run recursively until the ProcessURLPool reaches the amount we want.

After the spider has gone through the whole html file, it will report the result and write out these results in spiderResult.txt file under the user directory (normally the file is located in the “glassfish/domains/<domain\_name>/config/” folder if the application is deployed on Glassfish) in the format of

*<domain>;<URL>;<title>;/ <keyword1>:<number of keyword1>/ <keyword2>:<number of keyword2> …*

Then that the whole life of a spider, then it will die peacefully. When the oldest ancient spider (Starting one) dies meaning that there are enough links in the ProceedURLPool. The program will then print out the results.

# Search Engine

To handle user search request, the search engine composed of two parts; a Servlet API application (src/SearchEngine/API/RequestHandler.java), and a Web interface on the Glassfish server.

## Flow of event

Upon receiving a user HTTP request on the Search Engine application, the Glassfish server will dispatch the index.html file along with its web resources to the user’s browser. After the user input searching keywords and press the send button for search result, the AjaxUIProcessing.js JavaScript will send the user input as a query to the Servlet API via XMLHttpRequest() GET request.

The request is as follows: /search?q=<keywords>

Then, the doGet() method in the Servlet API application will process the request and reply the result string which is in JSON format as follows



Finally, the AjaxUIProcessing.js will parse the JSON string and display the result to the user.

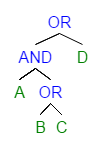
## Process user query

The RequestHandler class is responsible for processing query and return JSON data. The handler support different styles of query string.

The basic keyword search would be a query string containing one the keyword, “KEY”. The KEY notation represent an English word or a Java regular expression syntax. The KEY notation will be compare with the keywords and the title of the page by using Java Matcher class. If a substring of the keyword equal the KEY, then it is considered as a match. For example, “moodle” will match for any keyword or title containing “moodle”, which is equivalent to “.\*moodle.\*”. The query string “.\*” will return every page as it match anything.

To handle AND/OR operation, the two KEY should be include within a pair of bracket. For instance, “(KEY AND KEY)”. Each of the KEY notation can also be another AND/OR operation. For instance, “((A AND (B OR C)) OR D)”. Therefore, the user is allowed to request multiple conditions in one query. The handler will parse the query string into a binary tree and the binary tree of the above example is as below:

Each of the leaf in the tree will be handled as a basic keyword search, which also support regular expression.



## Result ranking

As the result generated by the WebSpider contains the page title and number of keywords in a page. The rank of a page will be represent by an attribute, weight. The higher the weight, the higher the page will be placed. Upon finding a match, RequestHandler will accumulate the weight of the page. If the match is found in the title, the weight will be added with 10. However, if the match is found within a keyword, the weight will be added with the number of occurrence of that keyword.