CO3090/CO7090 Distributed Systems and Applications Coursework 1

Multi-threaded Word Frequency Counter

Important Dates:

Deadline: 24-Feb-2020 at 17:00 GMT

- This coursework counts as 14% of your final module mark
- This coursework is an individual piece of work. Please read guidelines on plagiarism on the University website and module documentation.
- This coursework requires knowledge about Java threads.

Tasks:

Word frequency analysis is the first step towards text mining and automatic web page classification. The aim of this coursework is to implement a multiple word frequency counter. The program should count the number of occurrences of a specific keyword on all web pages (from a specific domain http://…). Starting from a seed page ("base" URL), your program should be able to traverse deeper through every hyperlink found on the pages. You will need to take advantage of multi-threading parallelism to improve the performance of the program.

The following Java classes are provided:

`WebAnalyser.java`

Auxiliary classes:

```
WebStatistics.java
Helper.java
KeywordNoProvidedException.java
IllegalURLException.java
```

Note:

Some methods (e.g. extracting links, obtaining HTML, striping HTML tags etc.) are already implemented in `Helper.java`.. Note that it is not necessary to understand the implementation details in the auxiliary class. You do not need to edit the auxiliary class, though you are free to make any changes or add new auxiliary classes you deem necessary, if you wish.

You task is to implement methods in `WebAnalyser.java`, turning it into a multi- threaded program.

Question 1 [25 Marks]

(1.1) Explain why a multi-threaded letter frequency counter has better performance than the single-threaded version. [5 Marks]

(1.2) To fetch all the pages from a seed URL, the frequency counter needs to traverse through the site by following the links present on the pages. In general, there are two crawling strategies (Please refer to *Appendix 1.2* for more details). Explain the strategy you used for your implementation and justify your choice. [10 Marks]

(1.3) Given the diagram below, how many threads will be started according to your implementation? [5 Marks].

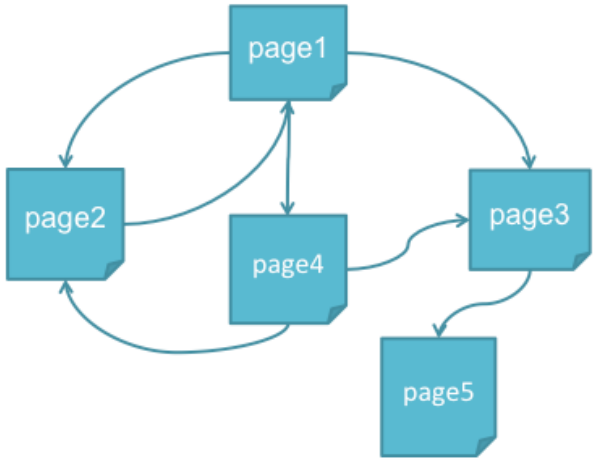


Figure (1)

Question 2 [45 Marks]

You will need to modify `WebAnalyser.java`

(2.1) Add appropriate **thread-safe** data structures to store the following information:

(a) The URLs visited.
(b) How many times a chosen keyword occurs on every page it has visited.
(Note: you are allowed to use built-in Java collection classes (e.g.Vector, ArrayBlockingQueue and ConcurrentHashMap etc.)

(Note: You are allowed to use Java thread-safe collection classes like Vector or ArrayBlockingQueue, - extra bonus will be given to those who implement their own version of thread-safe list, map or queue)

(2.2) Implement all abstract methods defined in `WebStatistics` interface.

[40 Marks]

Question 3 [30 Marks]

(3.1) Limit the maximum number of the threads running in parallel to `MAX_THREAD_NUM`

[10 Marks]

(3.2) The program prints the statistics when one of the follow events occurs:

- The number of the web pages visited by all counter threads exceeds `MAX_PAGES_NUM`.
- The total number of the words on all pages exceeds `MAX_WORDS_COUNT`.
- A specified time (`TIME_OUT` in milliseconds) has passed.
- All `WebAnalyser` threads (except main thread) have finished their executions.

[20 Marks]

Appendix

1.1 Auxiliary classes

`WebStatistics.java`

defines the abstract methods you will need to implement in `WebAnalyser.java`

`Helper.java`

contains necessary functions to fetch HTML code, extract URLs and calculate the number of occurrences of a specific word.

- `int countNumberOfOccurrences(String word, String text)`
  - Returns how many time a given word appears in a text (case-insensitive)

- `String getContentFromURL(String URLstring)`
  - Returns the HTML source code of a web page

- `ArrayList<String> getHyperlinksFromContent(URLstring, content)`
  -Extracts all internal hyperlinks from the source code of a html document.

Please refer to the Java documentation in `Helper.java` for more information.

`KeywordNotProvidedException.java`
`IllegalURLException.java`

Custom Exception classes, thrown if keyword has not been set or the program encounters an error when parsing a URL.

(Note: You do not need to modify these classes)

1.2 Breadth first search (BFS) and depth first search. (DFS)

There are two main approaches for crawling the web: Breadth first search (BFS) and depth first search. (DFS).

```
keyword: String   // one single keyword e.g. "Computer"
seedURL: String   // seed page e.g.
                  //"http://www…."
                   // where the search starts
Q:Queue         //a FIFO queue for storing URLs to be visited
visited:List    //contains a list of URLs already visited
results:Map<URL,count>   // used to store URLs and the number of word
               occurrences correspond to each URL.
```

Pseudocode for BFS search:

```
Main thread:

enqueue seedURL to Q

while Q is not empty
   start a new WebAnalyser thread to process URL
```

```
WebAnalyser threads:

procedure find(keyword, URL)

   dequeue a URL from Q
   add URL to visited
   if URL contains keyword then add URL and word occurrence to results
       for each hyperlinks l on  URL
             enqueue l onto Q
```

Note: All WebAnalyser threads (except the first thread) will obtain URLs from the shared queue rather than from the parameter list.

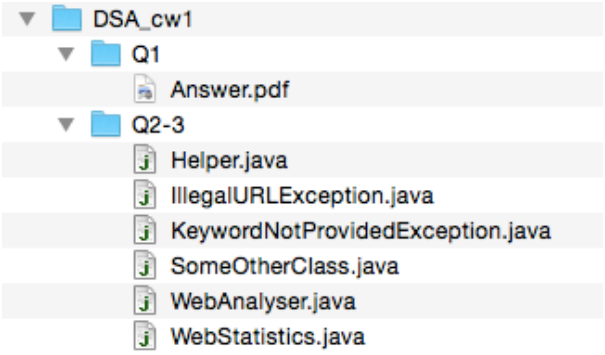
DFS: using a Stack instead of Queue would turn the BFS into a DFS search. Alternatively, you can use a recursive implementation of DFS:

```
Main thread:

start a new WebAnalyser thread;
```

```
WebAnalyser threads:

procedure find(keyword, URL)
     add URL to visited
     if URL contains keyword then add URL and word occurrence to results
         for each hyperlinks l on URL
             start a new WebAnalyser t and invoke t.find(keyword, l)
```

Submission

Please create a directory structure as the figure below.



- Subdirectory Q1 contains **Answers.pdf** (Answer to Q1)
- Subdirectory Q2-3 contains all Java files used in your solution (Answers to Q2, Q3)

Please compress the directory **DSA_cw1** using zip**.**

The archive should be named **DSA_(your_email_id)_cw1.zip** (e.g. DSA_yh37_cw1.zip). Your submission should also include a completed coursework coversheet (print and signed pdf or image). You need to submit the zip file via Blackboard and you are allowed to re-submit as many times as you like **before** the deadline.

Marking Criteria

<30%

- The submitted code does not compile.

30-40%

- No justification for the chosen crawling strategy.
- Appropriate thread-safe data types were not used.
- The program submitted is a single threaded application.
- No attempt on parallelism; no documentation.

40-50%

- The decision on the crawling strategy was made but poorly justified.
- Answers to Q1 are not consistent with the actual implementation.
- The submission is a multi-threaded but it fails to take advantage of parallel processing.
- Some thread-safe data structures are used, though there are some major issues with thread-safety (e.g. the crawler threads terminated unexpectedly, unhandled exceptions)
- No control over the number of threads running in parallel.

50-60%

- The decision on the crawling strategy was made and justified.
- Some answers to Q1 are not consistent with the actual implementation.
- The submission is a multi-threaded crawler
- Appropriate thread-safe data types are used, though there are some issues with thread-safety (e.g. deadlock or starvation occurred)
- Some mechanisms are used are used to control the number of threads running in parallel.

60-70%

- A good choice of crawling strategy with reasonable justification.
- Answers to Q1 are mostly consistent with the actual implementation.
- The submission is a multi-threaded crawler. Appropriate thread-safe data types are used.
- There might be still minor issues with the thread-safety. No deadlock or starvation.
- Mostly correct outputs from showTotalStatistics()

70-80%

- An excellent choice of crawling strategy, justification well explained.
- Well-documented source code.
- Answers to Q1 are consistent actual implementation.
- Guaranteed thread safety and correct character frequency
- >70% test cases passed with reasonably good performance.

80+%

- Apart from achieving all requirements above and passing automated test cases, the design should consider the needs for future extension.