CO3090/CO7090 Distributed Systems and Applications Coursework 1

Concurrent Letter Frequency Counter

Important Dates:

Handed out: 3-Feb-2019
Deadline: 24-Feb-2019 at 23:59 GMT

- This coursework counts as 13.33% 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.

Introduction

The frequency of characters in natural language texts is important for cryptography and data compression techniques. The aim of this coursework is to implement a multi-threaded character frequency counter to obtain the frequency of occurrences of letters on a website.

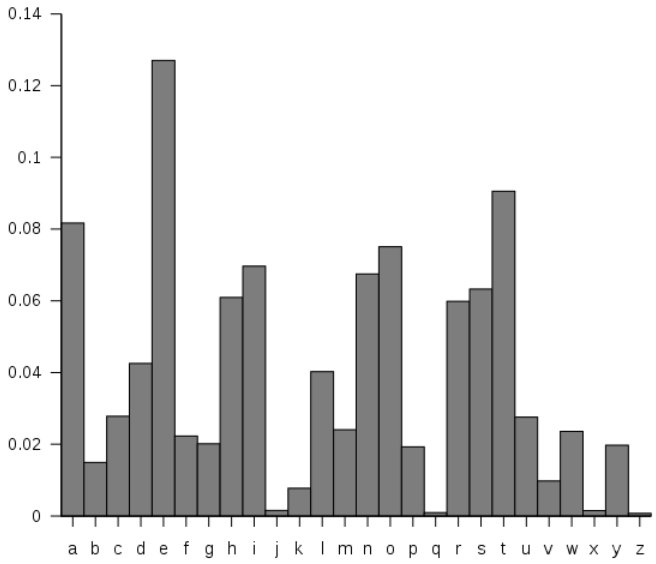


Fig 1 Relative frequencies of letters in English [1]

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.

[1] You results may vary depending on the content of the website.

The following three classes are provided:

- `CounterInterface.java`
- `WebCrawler.java`

You task is to implement all abstract methods in `WebCrawler.java`, turning it into a multi-threaded crawler.

Auxiliary class:

- `Utility.java`

Note:

Some methods (e.g. extracting links, obtaining HTML, striping HTML tags etc.) are already implemented in `Utility.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.

Instructions for setup in Eclipse

- Create a Java project; create a package named `uk.ac.le.cs.CO3090.cw1`
  Import all Java files (right-click on the package and select Import -> File System, select all java files)

- Download **jsoup-1.8.3.jar** and add it to the Java Build Path in Eclipse (right-clicking on the Project -> Build Path -> Configure Build Path. Under Libraries tab, click Add External JARs.)

Task 1 [20 Marks]

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

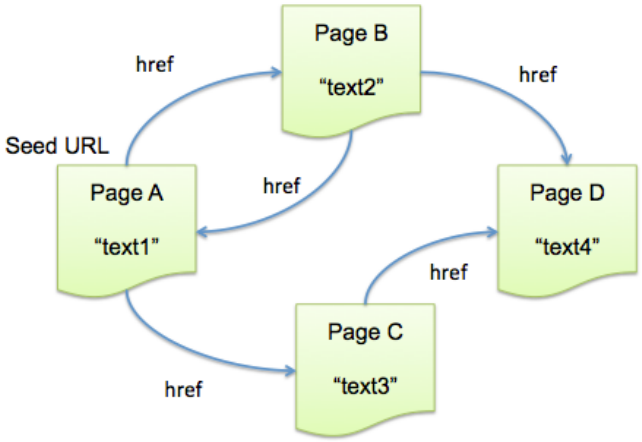
[5 Marks]

(1.2) The letter frequency counter will start from the "seed URL" and traverse through the website by following the links. Generally there are two crawling strategies, explain the strategy you used for your implementation and justify your choice. Please refer to *Appendix 1.2* for more information.

[10 Marks]

(1.3) Given the diagram below, how many threads will be started according to your implementation?

[5 Marks]



Task 2 [50 Marks]

You will need to complete `WebCrawler.java`

(2.1) The program should record these information using appropriate data structures:

(a) The URLs that have been visited.
(b) The letter frequency for every single URL.
(c) The letter frequency for all page on the entire website it has visited
(d) The total number of characters (on all pages it visited)
(Note: you are allowed to use built-in Java collection classes (e.g.Vector, ArrayBlockingQueue and ConcurrentHashMap etc.)

[10 Marks]

(2.2) Implement all abstract methods defined in `CounterInterface.java` and complete `WebCrawler.java`

[40 Marks]

Task 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 characters on all pages exceeds `MAX_CHAR_COUNT`.
- A specified time (`TIME_OUT` in milliseconds) has passed.
- All `WebCrawler` threads have finished their executions.

[20 Marks]

Appendix

1.1 Auxiliary classes

`CounterInterface.java`

This class defines the abstract methods you will need to implement in `WebCrawler.java`

`Utility.java`

This class provides necessary functions to get the html code, extract URLs from a web page and calculate the number of occurrences of each letter in the text (case-insensitive). For more information, refer to the documentation that comes with source code.

- `getTextFromAddress`
  - Return the HTML source code of a web page
- `extractHyperlinks`
  - Extract all internal hyperlinks from the source code of a html document.
- `getPlainText`
  - Strips all HTML tags from a string.
- `Calculate`
  - Return a HashMap storing the occurrences of each letter in the text.

(Note that you do not need to modify any auxiliary 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).

```
baseURL: String   // seed page URL
                  // where the crawling starts
Q:Queue           //a FIFO queue for storing URLs to be visited
visited:List      //contains a list of URLs already visited
results:Map<Character, Integer>
                  // used for storing occurrence-count of each character
```

Pseudocode for BFS search:

Main thread:

```
enqueue baseURL to Q

while Q is not empty then
   start a new WebBot thread t and invoke t.count(URL)
```

WebCrawler threads:

```
procedure count(URL)

   dequeue a URL from Q
   add URL to visited
   count characters, update results
       for each hyperlinks link on the page
             enqueue link onto Q
```

Note: All `WebCrawler` threads (except the first thread) should obtain URLs from the shared queue rather than from the parameter of the `count` method.

Pseudocode for DFS:

Main thread:

```
start a new WebCrawler thread;
```

WebCrawler threads:

```
procedure count(URL)
     add URL to visited
     count characters, update results
     for each hyperlinks link on URL
             start a new WebBot thread t and invoke t.count(link)
```

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

Submission

Please create a folder named **DSA_CW1**, the folder should include the files below:

- Answers.pdf (Answer to Q1)
- CounterInterface.java
- Utility.java
- WebCrawler.java

Please compress the directory **DSA_CW1** using zip**.**

The archive should be named **DSA_CW1_(your_email_id).zip** (e.g. DSA_CW1_yh37.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.