11/3/2015

Tianrui Yang (861246184)

Nikhil Kamthe (861245635)

Web Crawler

Information Retrieval

# Collaboration Details

**Tianrui Yang:**

* Conceptual Design
* Implemented Crawling Policy which involves Selection Policy and Politeness Policy
* Compression of Data before storage
* Testing
* Report

**Nikhil Kamthe:**

* Conceptual Design
* Basic program skeleton in JAVA
* Implementation of Multi-Threaded Crawling Agent
* Implementation of Duplicate Document Detection Algorithm (SimHash)
* Testing
* Report

# System Architecture

The crawler starts by reading set of URLs (seeds) from a seed file. A runnable object is created for each of these seed entries and is added to the entry queue. The crawler maintains a thread pool which is configured using the input given by user. This thread pool checks the request queue for pending tasks and executes them parallelly. The task may execute in a new thread or in an existing pooled thread. A crawling agent responsible for running the individual task maintains a map which stores the URLs and the corresponding file names. Every time the agent gets a new task to execute, it first checks if the corresponding URL has already been passed by checking its entry in the map. Once the agent determines that the URL hasn’t been parsed, it checks the crawling policy and downloads the data. It then extracts the text data and uses it for duplicate document detection (using SimHash). Once it establishes that the document is not a duplicate, it stores the document on the local file system. It also parses the HTML page to retrieve all the hyperlinks in the page. All these links are normalized and the corresponding runnable objects are pushed in the request queue. High level design of the crawler is as below [4]:

# 

# Crawling Policy

## Selection Policy

The selection policy determines the pages to download. It has been implemented by:

* **Restricting followed links**: The crawler follows only the HTTP links and simply ignores other protocols like HTTPS, FTP etc. Though it downloads a number of different file formats and stores them, only the HTML pages are parsed to fetch the hyperlinks.
* **URL normalization**: The crawler extracts all the hyperlinks in a given HTML page and crawls them recursively. The hyperlinks in HTML can be relative and hence should be normalized before storing them.

## Politeness policy

The crawler looks for a *robots.txt* file which is maintained in the root directory of the website. The file tells the crawler if and what is off limits. The crawler maintains a HashMap to store the allowed and disallowed links corresponding to each of the websites and checks the map before crawling any links.

# Data Structures

## ConcurrentLinkedQueue

The entry queue is used to store the runnable objects corresponding to the URLs and hence is updated by almost all the crawler threads. To do its task efficiently and effectively it needs to be fast, non-blocking and should work in a multi-threaded environment. Hence we use the ConcurrentLinkedQueue for serving this purpose. A ConcurrentLinkedQueue is an unbounded thread-safe queue based on linked nodes. This queue orders elements FIFO (first-in-first-out). A ConcurrentLinkedQueue is an appropriate choice when many threads will share access to a common collection. Like most other concurrent collection implementations, this class does not permit the use of null elements. This implementation employs an efficient non-blocking algorithm based on one described in Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms by Maged M. Michael and Michael L. Scott.

## ConcurrentHashMap

The crawler needs to store all the URLs which are already parsed along with the corresponding filenames (which stores the data on the URL) on the local file system. This data is accessed and updated by all the crawler threads to check if the current URL has already been parsed before. To serve this purpose the corresponding data structure should be fast in doing look ups and work well in multi-threaded environment. Hence we use ConcurrentHashMap for storing this information. A ConcurrentHashMap is a hash table supporting full concurrency of retrievals and high expected concurrency for updates.

## HashMap

A HashMap is used to store all the tokens (post stemming and stop-word removal) and their term frequency in a particular document which are then used for generating the SimHash for the document.

# Limitations

* The map of containing already parsed URLs is maintained in the memory which will adversely affect the scalability of the crawler.
* The timeout for the connection has been set to 5000 milliseconds after which the crawler skips the URL. A more sophisticated approach might add these URLs back in the queue and maintain a count for each try. The URL will be skipped only after it has been tried for a particular number of times.
* Only the HTML files are currently being parsed by the crawler (though it downloads and stores other formats like .pdf, .jpeg, .png, .gif etc. as well). In future, parsers could be written and integrated with the crawler to parse other file formats and extract useful information from these documents.
* Only HTTP protocol is being handled by the crawler. It simply ignores other protocols e.g. HTTPS, FTP etc.
* The maximum number of pages downloaded is restricted by the upper limit of Long in JAVA. The page limit does not guarantee the exact number of pages, it just gives an upper bound. However, the error will be less than or equal to the number of threads.   
  **(This has been done to keep the performance of the crawler intact in multithreaded environment.)**
* Currently the crawler only removes English stop words.

# Extra Credit

* The crawler is multi-threaded and maintains a fixed thread pool. The size of this thread pool is set by the user. The crawler uses the threads inside this thread pool to simultaneously access URLs and download the data.
* The crawler also provides an option to compress the HTML pages before storing them. This can reduce the space required to store these documents by 80%.
* The crawler uses SimHash to detect the duplicate documents before storing them. SimHash was an algorithm created by M. Charikar, from Google [1]. It combines the advantages of word based comparisons and the efficiency of finger printing. The algorithm has been implemented as follows:
  + The HTML pages are parsed to extract the text data. This text data is then stemmed after removing all the stop-words [2].
  + All the tokens are then stored in a HashMap along with their term frequencies.
  + A 32 bit hash is generated for each of these tokens using MurmurHash3 algorithm [3].
  + A 32 bit vector is then created such that the component of the vector is updated by adding the weight (term frequency) for a word to every component for which the corresponding bit in the word's hash value is 1 and subtracting the weight if value is 0.
  + A 32 bit fingerprint for the document is then generated by ith bit in the vector to 1 if the ith component in the vector is positive, or 0 otherwise.

# Instructions to deploy

* Make sure you have JDK 1.8 installed and added to the system path.
* The parent folder contains following folders:
  + **WebCrawler\_lib**: It contains all the third party libs used in the crawler
  + **config**: The folder contains files which are part of the project configuration
    - **root\_en.properties**: It contains the input parameter used by the crawler. The user needs to set these parameters before running the crawler or it will assume the default ones. Here are the list of input parameters:
      * Number of Threads
      * Levels
      * Page Limit
      * Input Directory
      * Output Directory
      * Seed File Name
      * Compression Enabled
      * Duplicate Detection Enabled
    - **stopWords\_en**: This file contains the list of English stop words. These words are removed before stemming and generating the SimHash for the document.
  + **input**: This directory is the default location for the seed file. In cases where user forgets to set the input directory path or provides incorrect argument in root\_en.properties, the crawler uses seed file located in this folder to start crawling.
  + **output**: This directory is the default output directory. In cases where user forgets to set the output directory path or provides incorrect argument in root\_en.properties, the crawler uses stores all the downloaded files in this folder.
  + **crawler.sh**: This file should be used to execute the crawler.
  + **README.txt**: This file contains instructions to execute the crawler.

# References

1. M. Charikar, *Similarity estimation techniques from rounding algorithms. In Proc. 34th Annual Symposium on Theory of Computing (STOC 2002)*
2. List of English stop words: <http://www.ranks.nl/stopwords>
3. MurmurHash3 implementation: <https://github.com/yonik/java_util/blob/master/src/util/hash/MurmurHash3.java>
4. <https://en.wikipedia.org/wiki/Web_crawler>