COMP 4321 Report

Jay Harris	20362789	joharris
Ethan Raymond	20363147	eoraymond

Overview

This project consists of three main parts, a crawler that crawls through pages on the internet and builds up an index of terms in those pages, a searcher that searches the index for documents matching a given query and ranks them based on their similarity and finally, a web portal, which allows users to submit queries to the search engine and see the results.

Crawler

The crawler traverses pages from a given starting point using a breadth first search. Stop words from the current page are removed, using the method produced during the third tutorial and the words are then stemmed using Porter's algorithm. The list of stop words (stopwords.txt in the application directory) is also from that tutorial. Visited pages are stored in a hash set, to ensure that we don't visit the same page multiple times.

The remaining words are stored in the data base along with various other pieces of information about the document from which they were retrieved. For more detail, see the 'Design' section.

Searcher

We used a vector space model with phase query support to perform searches. The query is divided into individual terms and phrases, where phrases are surrounded by quotes. We call these terms and phrases tokens. Each token is then used to retrieve posting lists from the inverted index. Tokens with individual words return one posting list. The entire posting list is added to the results because the vector space model considers any document that contains at least one keyword from the query.

Tokens containing phrases are filtered for documents that only match the phrase exactly. The search algorithm retrieves the posting lists for all keywords in the phrase. Then, the algorithm iterates through the smallest posting list and selects documents that contain all other keywords in same relative position as in the phrase. The algorithm reduces runtime by only considering documents in the smallest posting list.

The query is matched against the Title Inverted Index and the Document Body Inverted Index.

Once all relevant documents are found, document similarity to the query is calculated with the tf*idf method. Documents retrieved from the title inverted index are given a boost in similarity.

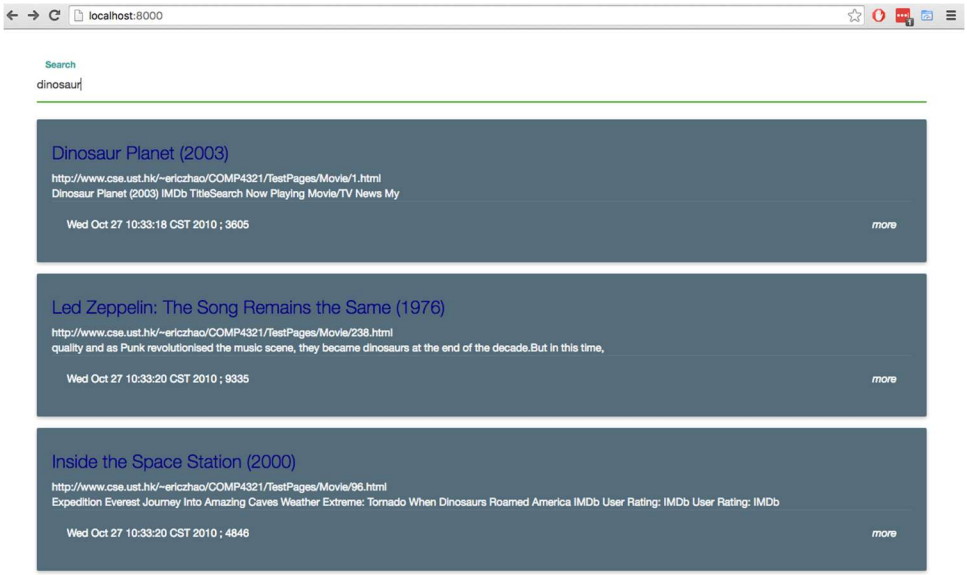
The results from the title inverted index and body inverted index are merged. The merge results are sorted in descending order according to similarity and the top 50 results are returned. These results are displayed on the search engine webpage in the required format.

Web Portal

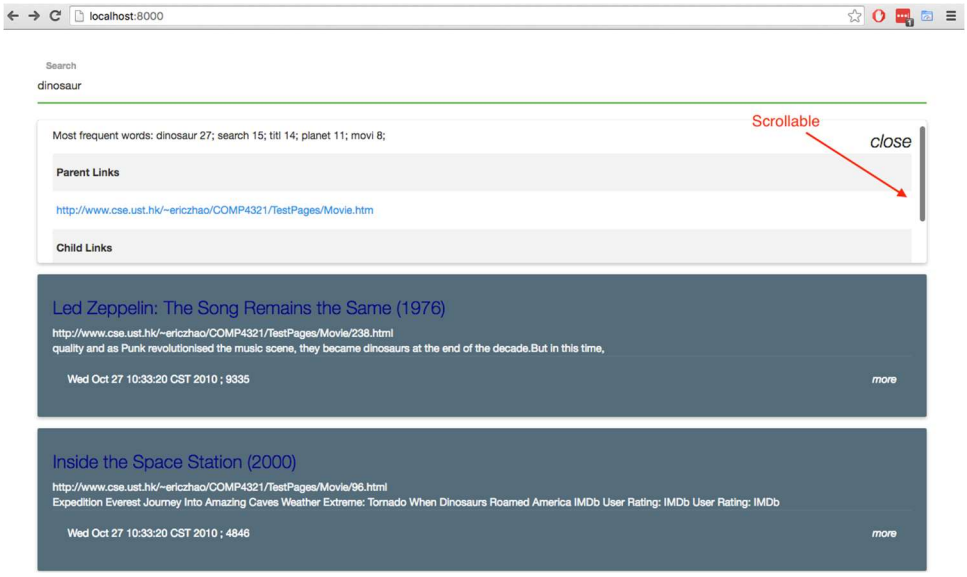
The web portal was not a major part of this projects, and was intended mainly as a way to verify that the searching and indexes aspects of the assignment were working as intended. For this reason, functionality is fairly limited, allowing search the user to input a query and see results which can then be opened.

We did not end up using java server pages for the web portal, as we wanted a more lightweight and reactive solution than can easily be created with JSP. For this reason, we wrote a simple web server

that can serve up our HTML page, its related style sheets, its related JavaScript and respond to ajax queries. The front end uses the open source Angular.js framework, created by Google to create a dynamic and responsive web app, so the user does not have to wait upon page reloads for their search results. The web app is styled through a combination of Bootstrap and MaterializeCSS to achieve an effect similar to that of Android's 'Lollipop' UI.



Clicking the 'more ' link on the bottom right hand side of the result will bring up a list of most frequent words and child/parent links for the result.



We are quite pleased with the final result, which updates as the user types.

Design

Crawler

We crawl the web pages using a combination of Java classes to represent each logical building block of the algorithm. We designed a Crawler to perform the breadth first search, a PageParser to extract information from each web page, a WebPage to represent a web page in our database, and a persistent database called Index.

The Crawler performs a breadth first search from the starting URL until no more pages can be found or the maximum number of pages have been crawled. We use a Set to efficiently check if a URL has been crawled and a linked list as the frontier of pages to be crawled. If the crawler encounters a page that has already been index, the web page is ignored unless the last modification date of the page is earlier than the last modification date we stored when first crawling the web page. If a webpage must be re-indexed, all old information is deleted before indexing the webpage again.

Page Parser

PageParser makes use of the Java HTML Parser library to extract the relevant information from the page, including all strings, all links, the title, last modification date, and page size. The date of the request is used when the last modification date is not included in the HTTP Response header.

Web Page

The WebPage class stores the URL, title, page size, last modification date, page rank, and a unique document ID for each web page. The object is hashed based on the document ID, allowing us to efficiently retrieve a web page by just the document ID.

Index

Index is a database containing many tables. The Index class also stores wordIndex, linkIndex, docIdIndex, wordCountIndex, bodyIndex, titleIndex, and docContentIndex.

Inverted Index

InvertedIndex maps a wordID to a posting list. Postings contain the document ID, term frequency, and the positions at which the word occurs. This information will be used when scoring document relevancy.

BodyIndex and titleIndex are the inverted indexes that store posting lists for each word we have encountered on a webpage. We separated words in the title and body into two different indexes so that we can give higher rankings to web pages that contain query terms in their titles.

Word Index

WordIndex is a hash-map that maps words to their ID. We store the word IDs in the posting list to conserve space and make faster comparisons. This consists of two maps, one from word to word id, and one from word id to word.

Link Index

LinkIndex maps links (urls) to their ID and stores parent and child relationships. It consists of four hash-maps; A link to document id map, a document id to link map, a map from document id to child documents and a map from document id to parent documents. We store map documents to integers to conserve space and we store the parent/child relationships so that we can easily reconstruct the web graph if need be.

DocIDIndex

DocIDIndex is a map from document ID to web page metadata. We store the data in the WebPage class, which contains docID, url, latest modification date, size, and title. We use the information to determine when a page needs to be updated in the database.

WordCountIndex

WordCountIndex maps a document id to a map of words and their occurrences. This makes it faster for us to print out the top five words for each document during the printing stage, as reconstructing word counts from an inverted index is a somewhat slow process.

DocContentIndex

DocContentIndex stores the content of all web pages crawled. It stores each word as a string. Each word is not stemmed and stop words are included. We use this index to retrieve a brief description for each document based on content matches to the query. Although including stop words and un-stemmed words reduces space efficiency, we gain the benefit of displaying easily readable web page descriptions. The DocContentIndex is implemented with a map.

Searcher

The search algorithm is implemented with the Search, Token, Tokenizer, SearchResult, and Document Vector classes.

Tokenizer

Tokenizer builds a list of tokens from the query. A token is defined as a single word not between quotes in the query, or a phrase wrapped in quotes in the query.

Token

Token stores an ArrayList of Strings for each word in the query it represents. It also stores the position of each word in the original query. We use this to calculate each term's offset from other terms in the phrase. The offset calculation is used to check if a phrase occurs in a web page. We use a separate array to keep track of term position in the query to correct for the position offset of stop word removal and prevent duplicate words in the query from offsetting our calculations.

Search Result

SearchResult is a class used to encapsulate all the necessary information to rank pages and display relevant information. We store the Document ID of the page, title, description, url (link), similarity to query (with pagerank taken into account), last modification date, page size, the top five most frequent words and their frequencies, a list of parent web page url's, and a list of child web page url's.

Document Vector

The DocumentVector class represents the document vector used in the web page similarity rankings. We encapsulate similarity measures such as CosineSimilarity within this class.

Searcher

The Searcher class contains the implementation of our search algorithm which is described in the "Overview" section under "Search". The main search algorithm is contained in the search function, with various helper methods to merge results, get web page descriptions, get web page information, and create document vectors.

Algorithms

We used a variety of different algorithms in the creation of this search engine. Some of the more notable are those specific to our solution, like our mechanism for calculating document similarities, our method for weighting title matches more heavily than body matches and our phrase matching system.

To calculate similarities, we query our index for all documents containing any of our search terms, and store the tfidf for those terms in a document vector. Similarity is currently computed using the inner product, though swapping in another similarity measure would be trivial due to the nature of the system. The matching documents are then sorted by their similarity to the query before being returned.

We make a distinction between two types of matches, matches occurring in the title of a document and matches occurring in the body. One of the requirements for this assignment was that matches within the title should be weighted more heavily than matches occurring in the body. To achieve this, we maintain two separate inverted indexes, one concerning itself with document titles and the other storing information about document bodies. We query both indexes and join the results on document Id's. Similarity scores for the merged documents are computed via the following formula

$$similarity = similarity_{body} + weight_{title} \times similarity_{title}$$

before sorting the merged results by the new similarities. This system has the advantage of being intuitive, fast and relatively robust. Currently, we have assigned the $weight_{title}$ parameter to five, that is, matches in the title are considered five times more important than those in the body.

Matching phrases gave us some trouble, as we were not entirely sure how the search engine should be dealing with them. In the end, we decided to treat double quoted words similarly to the AND operator in a boolean query, so only documents containing phrases will be returned. For example, searching "HKUST" university will only return documents containing "HKUST" whereas searching HKUST university might return documents only containing university. This was extended to whole phrases, such as "HKUST university" which would only return documents containing that exact phrase. Another point of discussion with phrase matching was how we should treat stopwords and stems. Should we expect the exact phrase without stopping/stemming in the target document or the phrase after stopping/stemming has been applied? Here, we opted to take the simpler approach and just search for the stopped/stemmed phrase, in order to save time and avoid maintaining another index.

We combined pagerank and similarity calculations with the following equation.

$$rank = (\alpha \times similarity + (1 - \alpha) \times PageRank), where\ 0 \leq \alpha \leq 1$$

Stopping and Stemming was achieved using the stop-words list and Porter's algorithm from the second lab which we simply included in the project. In our current implementation, $\alpha = 0.9$.

Installation Procedure

We recommend running the project from the executable jar file included with the project. To run the crawler run the following command

java –jar SearchEngine.jar –crawl <numpages>

Optionally you can specify a starting url with --url <starting_url>

To start the web server use this command: java --jar SearchEngine.jar --server

Optionally you can specify a port with --port <port_num>

You can view all commands with the --help parameter

A server will be started on localhost:8000. Navigate to localhost:8000 in your preferred webbrowser and you should be presented with the search page for our application.

If you wish to run the project from source, we recommend using Intellij IDEA by Jetbrains, as it should automatically handle importing the maven dependencies (and its free for students).

Import the source directory into Intellij (detailed instructions here https://www.jetbrains.com/help/idea/2016.1/importing-an-existing-android-project.html?origin=old_help).

Reimport the Maven project to ensure that all dependencies are available (detailed instructions here http://stackoverflow.com/questions/9980869/force-intellij-idea-to-reread-all-maven-dependencies).

To run the server, right click on Server.java in the searchengine.site package and select run. The server should start on port 8000.

To run the crawler, right click on Program.java and select run. Crawling should begin automatically from http://www.cse.ust.hk/~ericzhao/COMP4321/TestPages/testpage.htm for 300 pages.

Bonus Features

Page Rank

We implemented the page rank algorithm in the PageRank class. The algorithm visits each page with a breadth first search and calculates the page rank according to the algorithm

$$PR(A) = 1 - d + d\left(\frac{PR(B)}{L(B)} + \frac{PR(C)}{L(C)} + \frac{PR(D)}{L(D)} + \cdots\right).$$

We run the algorithm for 40 iterations so that the page rank values approach equilibrium. We then use the Page Rank to alter the relevancy scores of each web page to the query. This causes web page hubs to score higher. For example, the movie list web page, which has many parents, is moved up in ranking compared to other pages because of its high page rank score. The user benefits from this feature because the movie list is often a relevant web page for a user seeking a specific movie even though it may not score high in the vector space similarity calculation.

Inverted Index Batch Insertion

We use batch insertion to increase the speed efficiency of the inverted index. A temporary HTree is used to perform the batch insertion. Postings are inserted into the temporary HTree. After a constant number of webpages have been inserted into the temporary HTree, the temporary HTree is merged with the permanent HTree. The permanent HTree is saved to the database and all queries search the permanent HTree.

The batch insertion improves the speed of indexing because inserting a posting into a smaller hashtree requires less time. The posting lists in the permanent hashtree can grow very large, but the crawler only inserts postings into the small posting lists on the temporary hashtree. Merging the temporary and permanent hashtrees is efficient because the posting lists for each keyword are appended together. We can safely append the two posting lists because no document information can exist in both hashtrees simultaneously. The two hashtrees are merged between the indexing of individual documents, and after merging all postings are erased from the temporary hashtree. If the crawler encounters a webpage that has been updated since the last time it was crawled, we delete that document from the temporary and permanent hashtrees before re-indexing. We measured the performance of the crawler with and without batch insertion.

In the first test we indexed 300 pages starting at http://www.cse.ust.hk/~ericzhao/COMP4321/TestPages/testpage.htm. The batch insertion was performed every 50 document insertions. This number was chosen because it keeps the temporary hashtree small and results in a sufficient number of insertions to determine if the merging time reduced on increased crawling time. The batch insertion version performed marginally better. The batch insertion could be improved by reducing the merging time of the temporary and permanent hashtrees, which took about a second to complete.

	Trial 1	**Trial 2**	**Trial 3**
300 Pages With Batch Insertion	35.645 S	35.827 S	35.516 S
300 Pages Without Batch Insertion	40.548 S	41.699 S	42.592 S

In the second test, we indexed 600 pages starting at https://www.cse.ust.hk/. The batch insertion was performed every 75 documents. Batch insertion improved the speed of crawling by about 2 minutes.

	Trial 1 (MM:SS:MsMs)	**Trial 2**	**Trial 3**	**Trial 4**	**Average**
600 Pages With Batch Insertion	12:11:29	10:35:08	10:02:28	10:48:06	10:54:17
600 Pages Without Batch Insertion	11:23:57	14:22:51	13:13:07	12:32:42	12:52:45

It should be noted that these runtimes are dependent on the machine and environment on which they were ran.

User Friendly Interface

We display the search results in a user friendly way. Each result displays the page title, url, and a short description. The last modification date and page size are also displayed. Clicking on the "more" button reveals the top five most frequent words and parent and child links. We chose to initially hide the links and words because they could be very disruptive to the user experience, especially for pages with lots of parents or children. Curious users have the option to explore more information about the page through these statistics and the presentation of the additional information avoids overwhelming the user.

Descriptions

Short descriptions are printed out with every search result. For each document, we store a list of the words from the document in the same order that they appear when the crawler indexes the page. We use a HashMap with document ID's and the keys and a list of integer word ID's as the value. We include stop words in this list and store the full, un-stemmed version of each word. When the user submits a query, the first occurrence of a query term is recorded in the search result. The description is obtained by retrieving all words that occur within a certain range of that position from the document content index. We sacrifice memory by storing the document words in the document content index because we are storing document content twice, once in the posting list and once in the document content index. This sacrifice is offset by the speed advantage when retrieving the

descriptions because otherwise we would have to search the entire inverted index, display stemmed words, and exclude stop words. Including a short description is extremely useful for the user because they can quickly get an overview of the search result, to determine whether it is relevant without having to visit the web page.

Testing

For the most part, we did not create any automated tests. The one exception was the Tokenizer, for parsing queries into phrases and words, which we were forced to create tests for as we had no way to check it was working before the search functionality was written. However, these tests quickly went out of date as we rapidly modified the program to get it ready for submission, and those tests have not been included in the final version of the project. That said, we have manually tested as much of the program as much as we could, through a combination of printing output to the console and testing that output is as expected from the web portal.

Some of the more strenuous tests involved manually calculating what we expected page rank or vector similarities to be for a very limited subset of the documents, to verify that out implementation was working as intended.

Conclusion

Changes

If we were to do this project over, there are a few major things that we would like to do differently. For starters, the org.htmlparser library gave us no end of trouble, being vastly unintuitive and poorly documented. This was one of aspects of the project we poured the most time into.

It also felt like we could have planned out the index in a slightly more elegant manner, as by the end it felt we were hacking functionality onto it. For the most part though, we are pretty pleased with the way the project turned out.

Interesting Features

There are a number of interesting features to our project, not least of which being the way that phrases act similarly to an “AND” in a boolean query, which we imagine could be a highly desirable feature for power users.

The website, which is powered by Angular, turned out a lot better than we expected, and the live search results make the whole application into something a little different to other search engines, though it should be noted that Google does have some option for instant search.

We also make use of a version of Google’s PageRank algorithm when determining how to rank results, using an adjustable method of weighting page rank and similarity. In future it could be made possible for a user to modify how important to them PageRank and document similarity are to them when searching.

Potential Improvements

The search engine does not take into account identical pages with different URLs. We could incorporate tests to determine if a web page already exists under a different URL to avoid crawling the web page.

As instructed by the requirements, a page is not crawled if it has not been updated since it was indexed. If the starting URL fits that criteria, no web pages are crawled because we have not

extracted any other links to add to the frontier. The crawler could be improved by rejecting a starting url that has already been crawled and requesting another.

Insertions into the inverted index also slow down as the index gets large, due to the amount of information we are forced to read from the disk. An improvement here might be to use some other data structure, as we currently just store the entire list of words and retrieve it every time we wish to modify it, resulting in an unnecessarily large number of reads and writes.

Depending Projects

Throughout this assignment we made use of a number of awesome open source projects, which made saved us a vast amount of time. They are listed below:

AngularJS by Google https://angularjs.org/

We used this library to make our web front end. It's a JavaScript framework which, among other things, adds a really robust data binding framework to JavaScript, greatly reducing the amount of boilerplate code you are required to write.

Bootstrap by MIT http://getbootstrap.com/

A widely used set of styles and animations for web pages.

MaterializeCSS by Materialize http://materializecss.com/

We used this library as a way of giving our web app a clean and modern look, similar to Google's Android UI.

JQuery by the JQuery Foundation https://jquery.com/

This is a commonly used library for manipulating the HTML DOM through JavaScript. It was mainly included as a dependency of MaterializeCss and Bootstrap, as we make use of some similar functions provided by Angular.

Apache HttpCore by Apache https://hc.apache.org/httpcomponents-core-ga/

Apache HttpCore powers the web server that serves up our web portal. It was introduced to us during COMP 4111 so was already familiar to me and allowed us to create a very minimal web server.

JDBM by Jankotek https://github.com/jankotek/JDBM3

We used this library to store all the information we index. It provides a quick, simple, flexible and lightweight way of storing and retrieving information, which made it perfect for our purposes in creating a small inverted index of web pages.

GSON by Google https://github.com/google/gson

A library created by Google for serializing and deserializing JSON in Java. We use this in our restful search API so we do not have to worry about the details of how our search results are represented when we send them from our web server to the client.

Org.HtmlParser http://htmlparser.sourceforge.net/javadoc/org/htmlparser/package-summary.html

We use this library to extract information from web pages and store it in our inverted index. It is a free and open source library for parsing html pages.

Junit http://junit.org/

In our (admittedly limited) automated tests, we made use of the Junit framework.