# CPS842 (Web search and Information retrieval)

# Project 1 Report

Thusitharan Paramsothy, 500275858

Warren Pinto, 500396119

November 28, 2013

**Algorithm**

Creation of inverted index.

We used the Term and DocumentNode class to store information such as term word, term frequency, documentID, line position and line entry in which the term occurs. The program is supplied with the root directory for which to search. Using C#’s Directory class, the program was able to collect each file’s path (current directory as well as subdirectories).

The PDF text extracting

To extract text in a .pdf file, the iTextSharp tool was needed.

The HTML text extracting

To extract text in an .html, the text in the file had to be stripped of unnecessary html tags. Using C#’s Regular Expression library, it was easy to distinguish html tags and text data. Texts that contained html tags were ignored.

The .DOCX text extracting

To extract text in a .docx file, Microsoft Office tools were needed (Microsoft.Office.Interop.Word). The extraction of texts would only work if the user has Microsoft Word installed on their system as it is needed to extract the text files.

Text extracting in .txt files

Data in a .txt file are extracted using the FileReader class. This reads data from a file line by line.

Once the program has read the text from the specified file, it splits the words into an array according to spaces (using .split(‘ ‘)) and for each term, it stores all the information about that term (name, line, position, docID, author) into the Term and DocumentID class (like a package of information for each term). By doing this, we can easily find all the information of the term that we want.

To distinguish between title, author, abstract etc., the programs uses Regular Expressions to determine which is which (eg. @“\.A” = author, @“\.I\s[0-9]\*” = docID).To retrieve a term information, we search for the term string key in the Sorted Dictionary and retrieve the Term class as an object(which contains all the information).

Once the inverted index has been made, we output the sortedDictionary to a class object file (serialization) for easier retrieval. Before the serialization process, all term frequencies and its weights have been calculated. The program also reads the stop word file and stores each word in an ArrayList for easier and faster retrieval in the program.

After the first indexing program has been completed, the query processing program can be used. After starting the query processing program, the program first needs to de-serialize the data from the previous program.

The dictionary is ordered alphabetically by term. Each term will have a document name (path of the document). Calculations for the document tf, wtf, idf, w, and nw are done when initializing the inverted index. The query terms tf, wtf, idf, w, and nw are calculated upon retrieving relevant documents.

Data-structures

The program uses a SortedDictionary table with key (type: string) and value (type: Term). At first we considered using a hash-table. Hash-tables have an O(1) complexity, however, hash-tables are also unordered. We would have to waste time sorting the hash-table alphabetically, which could be spent on other performance. This is the reason for why we used a SortedDictionary. The sorted dictionary gives us an ordered list as well as lookup by key-value pairs. Although sorted-dictionary has a O(log n) complexity, its generic class is a binary search tree. The list of values for each term is stored in an Array-List. Because Array-Lists are implemented using arrays, we get a fast retrieval.

**How to run the program**

The program is coded in C# and tested on Visual Studio Express 2012.

Running the invert:

Goto .\CPS842\_InformationRetrievalAssignment2\bin\debug\ directory

Then run the CPS842\_InformationRetrievalAssignment2.exe file

* The program will first ask if you want to enable stop words. Type yes/no .
* The program will then ask if you want to enable stemming. Type yes/no .
* The program will then output the doc IDs that it traverses so you can visually see that the program is running. Once it is done reading the last doc, the program will create the postings and dictionary files along with the object data file.

Running the test:

* Goto .\ConsoleApplication1\bin\debug\ directory
* Then run the ConsoleApplication1.exe file
* The program will first de-serialize the data
* The program will then ask the user for a word to lookup.
* The program will then calculate the similarity scores and output to a file named "docSim.txt" in the "\ConsoleApplication1\bin\Debug" folder
* Based on the query, the program may return a list of documents in decreasing order of similarity scores.