”Intelligent” Internet Search Engine

(Report I)

A report submitted in partial fulfilment of the
requirement for the degree of Bachelor of Science

by Mr **Antony Richard Howat** (**960776751**)
Department of Information Systems and Computing,

Brunel University

January 2000

© 2000 Antony Richard Howat

xargle@eh.org - http://www.i-r-genius.com

Abstract

This paper will research the development of an “Intelligent” Internet search engine.

Research has focused on existing web search technology and the fields of Information Retrieval, document analysis, Data-basing methods and algorithms.

Acknowledgements

Thanks to: Dr. Simon Taylor at Brunel University for his guidance and encouragement as my project supervisor. Olly Betts at Muscat for his help in pointing me at some initial reading into search engine technology. Michael Williams at ARM Ltd for jogging my memory of vector mathematics. Ben Summers at Fluffy Clouds Ltd for further mathematical help. Pace Micro Technology for allowing me to research at work between take-overs and project cancellations. Dr. C. Lee Giles and Dr. Steve Lawrence at the NEC Research Institute for supplying me with reprints of his magazine articles as well as unpublished work. Finally thanks to my mother and father for paying for my continued existence at university. No thanks whatsoever to Microsoft for Word 2000, which caused me many hours of pain and distress during the production of this document.

All trademarks are acknowledged.

I certify that the work presented here, and all associated software is my own work unless referenced otherwise.

Signature………………………………………………… Date……………………………

TOTAL NUMBER OF WORDS: 11,991

CONTENTS

Abstract ii

CHAPTER 1- PROJECT DEFINITION 1

1.0 Introduction & Problem Definition 1
1.1 Keywords 2
1.2 Project Scope 2
1.3 Objectives 3
1.4 Areas Of Study 3

CHAPTER 2 – GENERAL SURVEY OF EXISTING SEARCH ENGINES 5

2.0 Introduction 5
2.1.0 Survey of Existing Search Engines 7
2.1.1 AltaVista 7
2.1.2 Yahoo! 7
2.1.3 Google! 8
2.1.4 Muscat EuroFerret 8
2.1.5 Excalibur RetrievalWare 8
2.1.6 Metasearch Engines 9

CHAPTER 3 - INFORMATION RETRIEVAL 10

3.0 Introduction 10
3.1 What is an information system? 10
3.2.0 Indexing and Document Analysis 12
3.2.1 Document Weightings 12
3.2.2 Zipf's Law 13
3.2.3 Document Surrogates 15
3.2.4 Stop Words 15
3.2.5 Controlled Vocabularies 16
3.2.6 Phrase Dictionaries 16
3.2.7 Compression 17
3.2.8 Stemming Algorithms 17
3.2.9 SoundEx 19
3.2.10 Thesauri 19
3.3.0 Query Techniques 20
3.3.1 Query Or Document? 20
3.3.2 Boolean Queries 20
3.3.3 Natural Language Processing 21

3.4.0 Matching Techniques 22
3.4.1 Exact and Range 22
3.4.2 Proximity 22
3.4.3 Vector Based Matching 23
3.4.4 Relevance Feedback 25
CHAPTER FOUR - DATABASE METHODS AND ALGORITHMS 26
4.0 Introduction 26
4.1 String Comparison 26
4.2 Hashing 27
4.3 Boyer-Moore-Horspool –Sunday Matching 30
4.4 Inverted Indexing 30
4.5 Binary Trees 31
4.6 "Chunked" Files 32
4.7 Database Management Systems vs. Custom Code 33
CHAPTER 5 – EVALUATION OF SEARCH ENGINES 34
5.0 Introduction 34
5.1 Measures of Information Retrieval Performance 34
5.2 Measures of Database Performance 35
CHAPTER 6 – CONCLUSIONS 36
6.0 Introduction 36
6.1 Methods and Techniques 36
6.2 Project Future 37
BIBLIOGRAPHICAL REFERENCES 40
GLOSSARY 43
Hash Collision 44
Homonyms 45
Knowledge Base 45
APPENDIX A – POSSIBLE FUTURE WORK IN THE AREA 47
A.1 Determination of "quality" of document by its rendered appearance 47
A.2 Determination of "quality" of a document by its use of language and structure 47
A.3 Combination of search engines and expert systems 47
A.4 Developing metasearch-type concepts to provide a truly distributed search engine 48
APPENDIX B – SOURCE CODE 49

Figures

Figure 1: A typical search engine display (Copyright AltaVista Company) 5
Figure 2 : A self contained search engine 6
Figure 3 : Abstraction from wisdom to index, and from desire to wisdom 11
Figure 4 : Plot of a vector query and document results 23
Figure 5 : Comparison of hashing spread for short ASCII 29
Figure 6: A simple binary tree structure 31
Figure 7 : An example of a chunked file 32

Tables

Table 1 : Demonstrating word frequency ranking 14
Table 2 : SoundEx encoding table 19
Table 3 : Checksum hashes of several sample ASCII strings 27
Table 4 : Comparison of number of hash collisions when generating keys for a dictionary of 45,373 English words in lower case ASCII 28

CHAPTER 1- PROJECT DEFINITION

1.0 Introduction & Problem Definition

The World Wide Web is a sprawling mass of disparate documents in no common format covering all imaginable topics. By its very nature the web is chaotic. Search engines try to make sense of this chaos by presenting a common interface to an indexed version of all these documents. Searching, usually by keywords, will present a list of pages which are possibly relevant to your intended search area.

This project aims to develop parts of an "intelligent" search engine. Most search engines, such as AltaVista, work by storing entire copies of web documents locally in a database and searching them. An "intelligent" search engine doesn't do this - it downloads documents and only stores the most important parts of a document for index.

Only a human can truly understand and summarise a document, and so have to compromise and use statistical methods developed in the field of information retrieval to get an approximate "jist" of the document. We can also eliminate "useless" search terms, words which have more to do the structure of English than having any meaning in their own right.

There are several justifications for the use of this sort of selective index. In February 1999 it was estimated the most comprehensive search engine on the web only covered 16% of the estimated 800 million publicly accessible web pages. (Lawrence, Lee Giles 1999) Not only is this figure small, but it is decreasing, as the web is growing faster than even the search engines coverage. In a 1997 study the most comprehensive engine covered 34% of the estimated 320 million indexable pages. (Lawrence, Lee Giles, 1999). From these figures it is evident that the existing engines are not keeping up. We may well reach a stage where full text engines cannot index an effective amount of data and still work quickly

Storing just significant parts of documents will reduce the amount of storage needed, and because of this reduction in data volume the search servers with an intelligent search engine can be much smaller and faster in operation - instead of a network of servers searching terabytes of complete documents, one or maybe two desktop machines can comfortably manage to search synopsises of several million documents. Consider the average web page size in an index of 100,000 documents to be 10k. A traditional engine would have to store a full gigabyte of straight pages. If the analysis techniques can reduce each page to a 1Kb index entry the index will only be 100Mb. Search engines typically index many millions of pages, and as more pages are indexed the cost/size benefits become more and more

evident.

The faster a search engine runs the more it can feasibly index, and so "intelligent" indexing would go some way to solving the coverage problems of existing full-text search engines, whilst also allowing smaller indexes to run on less sophisticated and less expensive hardware.

1.1 Keywords

Internet
World Wide Web
Database
Information Retrieval
Search Engine
Document Analysis
Query
Matching
Indexing

1.2 Project Scope

The study scope of this project is limited by time. A full featured search engine, which will index all formats, languages, character representations as well as supporting all web conventions would take many man-years of work.

For this reason we will only concentrate on indexing pages in English, written in HTML. No attempt will be made to analyse images for index, nor documents of any other formats, such as PDF or Microsoft Word. Support for HTML meta tags affecting search engines may not be complete, such things as document expiry dates may be ignored.

The major proposed parts of this project are :

- A **robot** to fetch documents from the web, follow links, and feed them to...
- The **analyser** to parse the HTML, pull out the best terms and feed them to...
- The **database** - a custom affair with data structures designed to search these lists of words.
- The **searcher** - parses search terms, ranks results in order of relevance.
- The **front-end** - providing a simple web interface for searches and submissions, to be fed to the search program or the robot.

1.3 Objectives

- Develop parts of a "intelligent" search engine, enough to demonstrate the techniques involved.
- Consider different architectures for such an engine.
- Investigate and evaluate document analysis techniques.

- Gain a deeper understanding of the concepts involved in information retrieval, and database structuring, and use a combination of these techniques to implement the engine.
- Keep implementation as lean as possible, in keeping with the intention that this engine should be fast and efficient.

1.4 Areas Of Study

To produce a search engine we will draw from several existing fields of study in Computer Science, Information Systems and Mathematics.

Information Retrieval is the process of retrieving content in a database that is relevant to a user’s information need. Information Retrieval is concerned with content, whatever form it takes, and the method for expressing the user’s “need” is not specified. Document retrieval can be seen as a subset of IR and has been the main focus of research in IR to date. Document retrieval consists of two main activities, indexing and searching. The indexing process is a matter of imposing a structure of some sort on to a collection of unstructured documents.

We shall also draw from the disciplines of Document Analysis and through that Statistics, in order to determine the relevance of potential query matches and rank them appropriately.

Knowledge Management as a term is open to interpretation. Malhorta(1998), as a Business Technologist writing for managers, defines it as : (next page)

> *"Knowledge Management caters to the critical issues of organizational adaption, survival and competence in face of increasingly discontinuous environmental change.... Essentially, it embodies organizational processes that seek synergistic combination of data and information processing capacity of information technologies, and the creative and innovative capacity of human beings".*
>
> Malhorta (1998)

In other words it is the application of Information Retrieval systems to a large knowledge base, in an attempt to form a coherent view of the relations within that knowledge. Whilst we may not be dealing with an organisation specifically, and there may be no issues of survival, we are undoubtedly attempting a knowledge management project. The combination of processing human written documents and the application of IR methods to them can be seen as the *"synergistic combination of data and information processing capacity"*.

CHAPTER 2 – GENERAL SURVEY OF EXISTING SEARCH ENGINES

2.0 Introduction

From the user's perspective an internet search engine typically provides a web interface, accepts a query string, and returns a list of pages which match that query ranked in order of apparent relevance to the query. New documents are added by being "submitted" to the search engine, which then downloads and analyses them, as well as following the links within them if appropriate and analysing those pages as well.

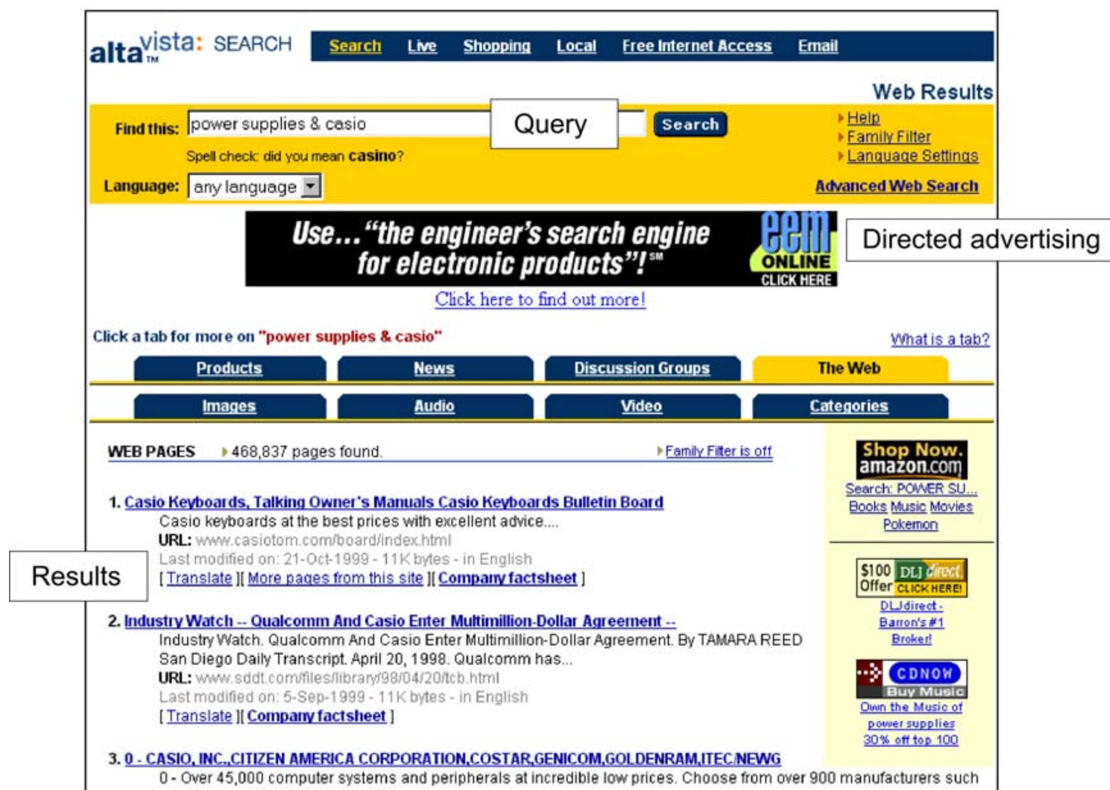

Figure 1: A typical search engine display (Copyright AltaVista Company)

Some search engines also provide a means of categorising or grouping pages in the index, this is typically done by the submitter and the categorisations are moderated by human editors.

To do any sort of search a computer needs to have data in some sort of similar, unified, regular format.

From these points we can conclude that a search engine performs two basic functions :

- **Indexing** – the process of turning dissimilar unformatted documents into indexes of terms ready to be searched.
- **Searching** – the process of interpreting queries, and ranking the returned documents in order of apparent relevance.

The indexer is fed by a "crawler" or "robot" which downloads web pages, and follows the links within them feeding each to the indexer as it goes.

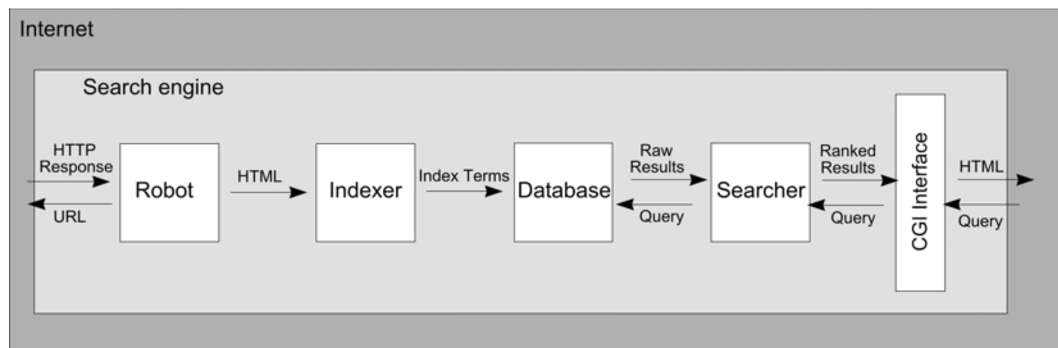

Figure 2 : A self contained search engine

Figure 1 is a simplified block diagram of how search engines, in general, work. A robot is "fed" web locations (Uniform Resource Locators) which it interprets to connect to and download the relevant document from the server using the protocol specified in the URL. In the diagram we assume that the documents are being served by HTTP (Hyper Text Transfer Protocol), although they can be served by many different protocols. The documents returned are passed to an indexer, which parses any mark-up in them, and extracts index terms. These index terms are then stored in a database ready to be queried.

The querying interface is provided using CGI (Common Gateway Interface) which is a standard method of extending web server functionality. A query from the web interface is passed to the searcher, which in turn queries the database. The database returns a raw list of document matches, the searcher ranks these in order of relevance and the CGI interface then turns these into a HTML document for the user.

2.1.0 Survey of Existing Search Engines

What follows is a survey of some of the existing search engines in order to establish their common qualities as well as their differences. All fit the basic model described above, although some distribute the processes in various ways in order to reduce search times. None of the existing search engines, understandably, divulge technical details on how they actually work, other than vague promotional literature. This is fully understandable, as search engines are very lucrative products. However, it does limit the usefulness of looking at others work, to merely establishing the “state of play” in the market.

In Europe companies have all the more reason to be cagey about the methods they use in information retrieval as it is very difficult to patent text processing systems as they compromise “a scheme, rule or method for performing a mental act, playing a game or doing business”. (Bott, Rowland, Coleman & Eaton 1996, p. 148) state that *“The European Patent Office have also denied applications for spell checkers and other text processing systems, such as methods and apparatus for abstracting a document, detecting linguistic expressions or homophone errors and generating synonyms and antonyms if these use the same steps as a mental process and employ conventional equipment ordinarily programmed”*. This covers much of the technology involved in a search engine.

In the examination of each engine I try to establish the techniques used by it, these techniques are explained later in the document.

2.1.1 AltaVista

AltaVista (meaning “view from above”) was the first web search engine, initially developed in late 1995. It is multi-lingual, and includes natural language processing capabilities. It is a fine example of the “brute-force” approach to a search engine. Every document is stored and indexed in full.

AltaVista is mostly funded by advertising, which it attempts to make relevant to the search terms given by the user. This can be seen in figure 1, where the advertising banner chosen is relevant to power supplies.

2.1.2 Yahoo!

“Yahoo!” itself is an edited index of pages, which is not automatically generated but compiled from submissions made by users. The index is split into many categories, and each entry has a brief abstract. The abstracts and document titles are searched as priority. “Yahoo!” will then drop back to a more

traditional search engine, known as "inktomi". It is difficult to establish how inktomi's engine works, and there website provides no solid detail.

2.1.3 Google!

"Google!" is one of the newer search engines, and uses a unique "PageRank" method of weighting documents. In analysing HTML, it treats a link from one page to another as a "vote" for that page. Pages with more "votes" are biased towards when producing search results for a user. PageRank is patented, but also, I feel, of little use anyway. Biasing towards the most popular is almost universally a bad idea, I would cite popular fiction and music as examples, where what is often superior is ignored on the grounds of its lack of common appeal in favour of the populist and simplistic. From its results "Google!" seems to be a non-full text engine, with no concept of word ordering.

2.1.4 Muscat EuroFerret

Euroferret is a search engine just indexing European web pages, primarily as a demonstration of Muscat's technology. The system indexes over 35 million pages on two Sun Ultra workstations, and is the only engine I have found that makes proper use of data reduction techniques rather than brute force methods. The system uses stemming (no surprise, as the developer of stemming Dr. Martin Porter works at muscat), as well as only indexing key terms, hence it has no concept of word ordering. Feedback is implemented to a degree in that the search engine suggests additional search terms which may be relevant. It does prove that such a system is feasible and useful.

2.1.5 Excalibur RetrievalWare

Excalibur RetrievalWare is a commercial system which is not just a search engine, but a knowledge management tool. It applied natural language processing through the use of multiple lexical sources, what they term a "semantic network", which is a database of word meanings and relationships. The "semantic network" appears to be a combination of a thesaurus and a dictionary. It features natural language queries. Rather than using stemming algorithms for indexing the software takes advantage of its knowledge of the English language to recognise words at their root level.

To extend its semantic network RetrievalWare uses Adaptive Pattern Recognition Processing, a proprietary technique which Excalibur's promotional literature describes as : (over page)

> *"Excalibur's APRP™ opens a new dimension in digital information retrieval. Modelled on the way biological systems use neural networks to process information, APRP™ acts as a self-organizing system that automatically indexes the binary patterns in digital information, creating a pattern-based memory that is self-optimised for the native content of the data".*
>
> (Excalibur Corporation 1999)

The software is specific to the English language. Whilst all the features sound interesting we can only judge it on the publicly available sales literature. The methods behind APRP are not documented, other than the use of neural networks. I would suspect such techniques lose effectiveness outside of a corporate intranet, where documents may well be nonsensical.

2.1.6 Metasearch Engines

Metasearch engines provide a common interface to several search engines, automatically forwarding the query to each in turn. Some meta engines attempt to integrate the results from the engines into a single rankings, others do not.

Dogpile is an example of a simple metasearch engine. It makes no effort judge the comparative relevance of results between search engines, and as such has limited usefulness.

Copernic is similar to Dogpile in that it searches many engines, but it actually consolidates the results of the engines to produce a single ranking of results. It is odd in that it runs locally on the user's machine rather than being a web based program running on a remote server. Copernic is a commercial program, with a restricted free variant available for download. The makers term it a "search-assistant". As it is only consolidating results from real engines it cannot be considered a true search engine.

MetaCrawler is a publically accessible web based metasearch engine which does attempt to combine results into a single ranking, it does this using the brief document summaries returned by the engines as well as normalising any percentage type ranking figure the engines may return. Its rankings are poorly constructed as they are based on very little data rather than complete documents.

Inquirus 2 is NEC Research's project to create a better metasearch engine. Rather than merely working on the titles and summaries returned by the other engines it downloads the documents and then analyses them so it can use a consistent ranking policy, as per a normal search engine. (Glover, Lawrence, Gordon, Birmingham, Lee Giles 1999)

CHAPTER 3 - INFORMATION RETRIEVAL

3.0 Introduction

In this chapter we will examine the techniques within the field of Information Retrieval, and their comparative relevance to the implementation of an internet search engine.

3.1 What is an information system?

Any information system works with data which is an abstraction of reality. A system will work from data, which is a partial view of a subject (in much the same way that this document is a partial abstraction of what I have learnt about the techniques involved in writing search systems).

Korfhage(1997) states the first abstraction principle :

> *“In any information system, the "real world" is represented by a collection of data abstracted from observations of the real world and made available to the system”*

The user has to compromise, in stating search requirements in a machine readable form, although work in the field of Natural Language Processing will allow some systems to accept limited queries in plain English (see section on NLP).

Korfhage(1997) states the second abstraction principle :

> *“A user's information need, whether for production, storage, or retrieval of information, is abstracted into a form that is commensurate with the information system to be used”*

Many traditional Information Retrieval systems centre around a particular subject field, in other words they are constructed to satisfy a particular "information need". This allows the system implementation to concentrate on that need, and the types of queries that the need is likely to generate.

The nature of the world wide web means we have no such subject area, pages may cover anything and everything, and possibly even nothing. Whereas in a system indexing technical documents we may be

able to rely on the presence of abstracts and other such features of a well-formed document, on the web we can rely on nothing.

The whole process of information retrieval, especially from the world wide web, is by its very nature imprecise.

Reducing the amount of data stored by a system, by whatever means, reduces the system's field of view of reality. *"There's no such thing as a free lunch"* -- Dr. Milton Friedman.

The whole process of using an information retrieval system involves several levels of abstraction, from query to search terms, and from document to index terms.

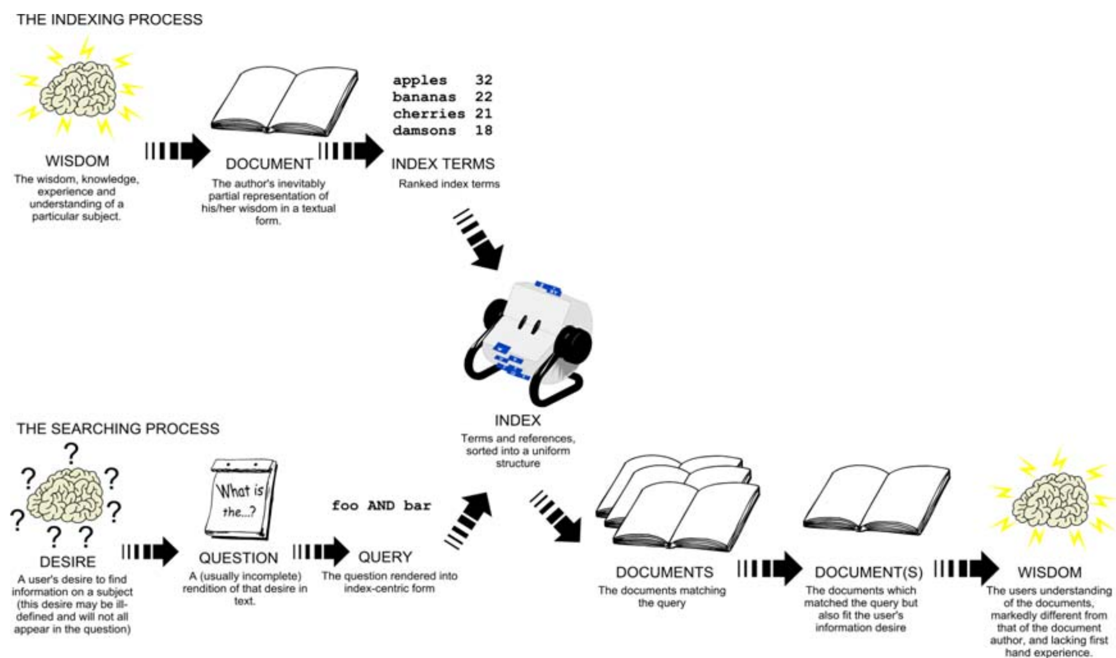

Figure 3 : Abstraction from wisdom to index, and from desire to wisdom

The results of a search engine, and indeed any information retrieval system are rated in terms of relevance and usefulness. Relevance being how close to the user’s information need a match is, and usefulness being how practically useful the document was to fulfilling that information need..

3.2.0 Indexing and Document Analysis

Producing an index is a method of turning our unstructured source data, into a data set with structure, a list of words and document references. The simplest method of indexing would be to store references to every word in every document. This would result in an index of roughly the same size as all the document indexed.

This is fine, as long as suitable storage is available to hold all the indexed pages, and there is enough processing power available to search the data without unreasonable delays.

With a large database indexing many documents it becomes increasingly important to reduce the data down to only useful index terms, and possibly group these index terms. This is done through data reduction methods. As long as the same data reduction is applied to the query as was applied during indexing we can match queries to results.

3.2.1 Document Weightings

As we generate an index we need to assign a “weighting”, a figure that indicates our estimation of relevance to that search term. The general method for determining good index terms relies on the frequency of each word in a document, and then using a data reduction method to pull out the less useful terms.

This weighting is normalised, so the weightings of documents can be compared and ranked accordingly. This normalisation occurs one of two ways :

$$w_{ik} = \frac{f_{ik}}{c_i}$$

Where f_{ik} is the absolute frequency of term i in document k, c_i is the total word count in document k, and w_{ik} is the resulting weighting for the document. This has a significant flaw, in that a short document containing only one mention of the word will ranked as more relevant document mentioning the word several times. In terms of logic this is perfectly reasonable, statistically a four word sentence mentioning “dogs” is more “about” dogs than an ten thousand word essay mentioning them twenty times, but in terms of usefulness a four word sentence won’t provide much information.

The second normalisation method addresses this by normalising across the entire index to give the *inverse document frequency weight*, the concept being that terms should be weighted towards if they

don’t appear many times in the index. The method also uses logarithms to moderate the effects of an increasing database size.

$$w_{ik} = f_{ik}\left[\log_2 N - \log_2 d_k + 1\right]$$

(assuming $d_k \neq 0$)

Where again f_{ik} is the absolute frequency of term i in document k, w_{ik} is the resulting weighting for the document. The additional terms N and d_k are the number of documents indexed and the number of indexed documents containing the term respectively.

Words may be assigned additional weightings, according to any other indicators of importance they may have (they may be part of a document title or abstract, for example).

Weightings may need to be compared which are using differing scales, for example a user may be specifying weights between zero and one, and a system may be using full thirty-two bit integers. In this case we can use a transformation formula, for the two scales s and u :

$$s = \frac{s_{\max}(u - u_{\min}) + s_{\min}(u_{\max} - u)}{u_{\max} - u_{\min}}$$

Weightings to search terms may also be added by a user profile which a search engine may maintain. A user profile is usually in the form of a set of weightings to indicate a user’s interest in a certain subject area, however in the internet search engine world user profiles are more commonly data collected for marketing purposes to allow directed advertising.

3.2.2 Zipf’s Law

Zipf’s law forms the basis for the statistical methods used to determine good index terms in documents. It is based around the fact that all the basic units of the English language, whether letters, groups of letters, words or phrases, occur with varying frequencies in documents. If the units of text, in this case words, are ranked in order of decreasing frequency then Zipf’s law is true :

$$\textit{rank x frequency} \approx \textit{constant}$$

If the law was strict the second most frequent word would occur half as many times as the first, the third one third as many as the first, and so on.

Through Zipf’s law we find that the most common terms are generally connective words with little meaning on their own, and so are bad index terms. Those words at the bottom of the frequency ranking are also of little use, as they occur so infrequently they give no indication of the document subject matter. The words which are relevant are those in the mid range, and by adjusting the upper and lower boundaries we can isolate what should be good index terms. However, this is only true of substantial quantities of text.

Consider the text :

> *“Mr. Lunch had been in the canoe for a long time. He decided it was time to explore on foot. He came upon a vendor selling packets of bird seed. What luck! He bought some for the elephant (who collected bird seeds). Could this mean there were birds nearby? When Mr. Lunch turned the corner he found his answer. He saw a large plaza full of pigeons. Mr. Lunch had never seen so many birds in one place before.”*

(Otto Seibold, Walsh 1994)

The frequency ranking (for the top 19 words) would be :

Word	Frequency	Frequency x Rank
Mr	3	3
Lunch	3	6
Time	2	6
Bird	2	8
Birds	2	10
Canoe	1	6
Long	1	7
Decided	1	8
Explore	1	9
Foot	1	10
Came	1	11
Vendor	1	12
Selling	1	13
Packets	1	14
Seed	1	15
Luck	1	16
Elephant	1	17
who	1	18

Table 1 : Demonstrating word frequency ranking

This hardly fits the law, and the most relevant words occur in the rankings which would normally be considered to be useless index terms. Unfortunately this problem also extends to longer documents. For example, the Labour party manifesto's 4th and 5th ranking words are “new” and “labour”, both terms which are of great significance to the document, but appear so high in the rankings according to even a light application of Zipf's law they would be ignored. As a point of interest “education” is 19th in the ranking, and “union” is 191st (“union” ranks 81st in the Conservative manifesto!). These word frequency statistics were gathered using an early version of my document analyser framework.

From this practical evidence we can only use Zipf's law as a guide. The logic is sound but in practicality it can cause serious omissions from indexes. We need a more reliable method for excluding bad search terms.

3.2.3 Document Surrogates

Some Information Retrieval systems rely on document surrogates, in other words the presence of lists of keywords, abstracts and marked up extracts which are likely to be of use when searching. This is a sensible approach where all documents are well-constructed, and the reliability of these surrogates can be relied upon. In an internet environment we cannot rely on such well formed documents, and although the HTML Specification (Raggett, Le Hors & Jacobs 1998) includes tag definitions for such surrogates they are rarely present and often abused. Lawrence and Lee Giles (1999) found only 34% of servers include even the most basic metadata on their home pages.

This data should not be discarded, but should be included in the word list with a higher weighting than the body text. Unlike in the body text only one occurrence of each word in these areas should be included in the weighting. This is to prevent abuse of the fields, many pages have keywords repeated several times in order to up their positioning in web search results. In connection with this it will also be worth considering ignoring sites with such repeated keywords all-together, as the content is likely to be of little use whatsoever if it needs to use such desperate means to attract visitors.

The Dublin Core initiative is an effort to define metadata elements to describe information resources, including web pages. (Kunze 1999). Although helpful it is yet another standard to be ignored, indeed only 0.6% of sites contain metadata using the Dublin Core according to Lawrence & Lee Giles (1999).

3.2.4 Stop Words

A stop-word dictionary attempts to exclude all the essentially meaningless conjunctive words in a language. Such words do not make good search terms, and so it makes good sense to ignore them. A large percentage of the words in a particular document will be able to be excluded with this technique, and so the size of the index is dramatically reduced (and hence search speeds increase, etc).

The removal of stop-words does introduce problems especially if a search system is intended to have a concept of word ordering within documents – how can you have a sensible idea of which order words are in the document when half of them are excluded by the stop-words dictionary? Also, consider a search for "The Smiths". A user would reasonably expect a list of matches pertaining to the band "The Smiths", but with one half of the query dropped due to the stop list the query would in fact return matches for any page concerning any "Smiths".

When implementing a stop word system it would also seem sensible to include code to exclude certain strings, such as all numeric etc, and also strip things like punctuation.

Any application of Zipf's Law in conjunction with a stop-word dictionary must be approached carefully, as they are pretty well mutually exclusive.

3.2.5 Controlled Vocabularies

A controlled vocabulary defines that certain specific terms be used to signify certain subjects, thus avoiding the problems introduced by the fact that many English words have multiple and/or similar meanings. Use of this technique will also reduce the number of terms that have to be indexed, as the search subjects are pre-defined. The controlled vocabulary can cover just abstracts and keywords, or the whole document. This is practical on a system where the data classification is manual, or the controlled vocabulary can be enforced on the authors. When indexing web documents we have no control over the authors, and typically a web index will involve so many documents that we cannot expect any human classification. Controlled vocabularies are also only suited to systems where specific subject areas can be identified, web documents are unlikely to have such well defined subjects, and many will in fact overlap each other.

3.2.6 Phrase Dictionaries

A phrase dictionary is not so much a data reduction technique, but a partial remedy for the loss of word-

ordering and context when documents are reduced to index terms. Treating single words as entities is a little naïve, as commonly they have little meaning on their own. For example, “kangaroo court” is a term which has nothing to do with antipodean wildlife, and so would be best indexed as one phrase so as to keep its context, perhaps as well as being indexed individually.

However, there is the requirement of a good dictionary of such terms, and such a database would naturally be language specific. Such a dictionary would also need maintenance, something which Excalibur RetrievalWare claims to do (Excalibur Corporation 1999) by “Adaptive Pattern Recognition Processing”. Unfortunately there is no data publicaly available on how this works. I can only assume that it works by analysing documents, looking for “word pairs”. Those which commonly occur in close proximity to each other can be considered pairs, and added to a phrase dictionary.

3.2.7 Compression

In order to reduce a system’s data storage need a compression algorithm may be used. Compression algorithms take two major forms, lossy and lossless. Lossy compression (such as JPEG) sacrifices quality over file size. Such techniques are typically used in audio and graphics compression. Lossless compression is a matter of finding patterns in a file, and optimising its storage by way of Huffman coding, a case in point is the popular Lempel-Ziv Welch algorithm. Typically, the sort of text data we intend to store will compress well, as only a hundred or so characters are regularly used out of the possible 256 for each 8-bit character. However, compression ratios are only significant on files of over a kilobyte, with small data areas the use of compression can increase the storage needed. There is also a processor overhead, which can be quite considerable if data is constantly being altered. Another problem is the legal status of LZW, which is covered by a patent held by Unisys (Welch 1985). UK law invalidates this patent, as you may not patent an algorithm (Bott, Coleman, Eaton, Rowland 1996), however internationally Unisys (the current holders of the patent) can and do demand licence fees. A simpler scheme would be to use LZ78, which is a predecessor of LZW. This is essentially a tokeniser, which replaces terms from a dictionary with indexes to that dictionary. Again, you need to have a dictionary in the first place, and benefits are only truly evident on long blocks of data.

Lossy compression, in the traditional sense of compression, cannot be sensibly applied to text. The other techniques described in this chapter, such as stemming, can be thought of as the textual equivalent of lossy compression.

3.2.8 Stemming Algorithms

Stemming algorithms are also known as *conflation algorithms,* and are defined in Sparck-Jones, Willet (1997) as *“a computational procedure that reduces variants of a word to a single form”.* Stemming works by stripping the word endings from potential index terms to leave the core, or stem of a word, which will be common to all forms of the word. To illustrate, “Connected”, “Connecting”, “Connection” and “Connections” all have the same stem, “Connect”, according to the Porter algorithm. The word suffixes are removed step by step, complex suffixes are treated as compounds of simple suffixes.

The Porter algorithm (Porter, 1980) is seen as the definitive stemming algorithm for the English language, and development of it has continued long after the papers publication.

Stemming algorithms do introduce problems, as with all data reduction techniques. Most significantly, they are heavily language dependant; an English stemming algorithm may not produce sensible results for German text, and may indeed seriously damage the indexing of foreign language documents. However, when the concepts have been applied to languages with a greater degree of morphological complexity similar benefits have been observed (Kraaj and Pohlmann, 1996; Popovic and Willet, 1992 – referenced in Sparck-Jones, Willet 1997).

There are also problems when stemming English text. Consider that “wander” may well stem to “wand”, “probate” to “probe”, and so on. The problem being that English obeys most of its rules some of the time, what appears to be a suffix may in fact be part of what should be the stem, and so seemingly unrelated matches can occur. Exceptions can be introduced, but as Porter (1980) notes *“There comes a stage in the development of a suffix stripping program where the addition of more rules to increase the performance in one area of the vocabulary causes an equal degradation of performance elsewhere”*.

As with all other data reduction methods, stemming will introduce inaccuracies, but the advantages in reducing the number of terms that are actually indexed, and the potential of grouping relevant documents with no user maintenance makes the technique worth considering seriously.

3.2.9 SoundEx

Soundex is an algorithm for encoding words so that similar sounding words have the same coding. The algorithm is very simple. The first letter is copied verbatim, and then the following letters are encoded as numbers, according to the following table.

1	bfpv
2	cgjkqsxzç
3	dt
4	l
5	mnñ
6	r

Table 2 : SoundEx encoding table

Characters which don't appear in the table are not encoded. Repeated letters are encoded as single characters. Encoding stops at four characters, any strings which are too short to fill four characters are padded at the end with zeroes. For example, the similar sounding words "moose" and mouse become "M200", whereas "apple" and "archer" would become "A140" and "A626" respectively.

Whilst ideal when matching customer details dictated to an operator to an existing database, this isn't a valid method of reducing data into a more compact yet meaningful form. Too much information is lost from the words, and accuracy would suffer severely.

3.2.10 Thesauri

The problem with thesauri is they group words together which have subtly different meanings. This may be acceptable to an extent, but the further away from the intended meaning the thesaurus entry goes the more irrelevant material will appear in the results.

Consider searching for a book title, "One Flew Over The Cuckoo's Nest" – automatic use of a thesaurus could result in many matches on the subject of flight, birds in general, eggs, homes etc. when we were searching for a very specific title.

Thesauri can be applied at different stages in the retrieval process. Some systems index according to thesauri; all words with equivalent meanings are indexed as one. The automatic application of thesauri is only practical in systems with a fixed field of knowledge, where equivalent words can be specifically entered into the system. Indexing of this manner in a more free-form system, such as an Internet search

engine, will result in many apparently irrelevant pages being associated in a permanent manner.

Similarly, when processing the query the thesaurus terms for each element of the query can be added automatically, preferably with lower weightings, so the initial search terms will always be preferred. This is far better than applying the thesaurus during analysis, as all the precise terms in the document are preserved instead of being distilled down to their root thesaurus entry.

A thesaurus can be useful in situations where few matches for the actual search terms are found, but I feel it should only be applied in these cases, and the results heavily weighted against.

Again, any thesaurus will be language specific.

3.3.0 Query Techniques

This subsection examines methods pertaining to queries, their formulation and processing.

3.3.1 Query Or Document?

It is a point of contention between researchers as to whether there should be a distinction between documents and queries. A query can, conceivably, be treated in exactly the same way as a document, and so the process of searching becomes one of document matching. Korfhage (1997) cites Harman's TREC research (1993, 1994, 1995) as a proponent of queries as documents. However, Bollman-Sdorra and Raghavan 1993 state that the query is likely to be short, and query terms will only appear once or twice. This makes word-frequency statistics meaningless. It is worth noting that the research centred around well formed documents, with easily identifiable titles, narratives and concepts.

For the purposes of this project I side with Bollman-Sdorra and Raghavan, whilst the concept of treating queries as documents may be desirable in terms of design (especially when thinking in terms of object orientation), the practicality of it would appear to get in the way when dealing with unstructured documents.

3.3.2 Boolean Queries

Boolean queries are the most common query method in Information Retrieval systems today. They are words/phrases strung together by simple logic operators taken from Boolean Algebra. For example, a query such as "fried & egg & sandwiches" will return documents containing the three words "fried",

“egg”, and “sandwiches”.

The common operators implemented are AND, OR and NOT, together with the use of parenthesises. More sophisticated systems implement options such as specifying how many terms need to be present from a list, ie. “3 OF (Cats, Dogs, Chickens, Pigs)”. Some also implement facilities for telling specifying allowed distances between words to be classed as a match.

The use of Boolean queries can confuse some users. For example it would be perfectly reasonable for a user to specify a search of “football & rugby & cricket” and expect a list of all the cricket pages, football pages and rugby pages, whereas he would only actually get a list of pages covering all three topics in one. Having said that, such confusion is easily resolved by providing a couple of lines of explanation to the user.

Extended Boolean systems will allow weightings to be included in queries, to rate the importance of each query term. Weightings are assigned between 0 and 1.0 (1.0 being equivalent to a term in a standard query) and the relevance of results are judged in accordance to these weightings. This is particularly useful when specifying queries such as a search for documents on Peter Fonda films, preferably “Easy Rider”, which could be expressed as :

$\text{“Film”}_{1.0}$ & $\text{“Peter Fonda”}_{1.0}$ & $\text{“Easy Rider”}_{0.5}$

Is the facility to add indicators of preference to a Boolean query worth the extra complexity in syntax?

3.3.3 Natural Language Processing

Search terms in some systems may be stated in plain English, for example "How does a rainbow form?" by use of Natural Language Processing. These queries work fine for simple questions where a specific topic to search for may be present, but any question that assumes knowledge or intelligence will fail. For example, you cannot reasonably expect sensible results from "Which songs did Tom Jones record between 1973 and 1976?", other than perhaps a match of sites mentioning "Tom Jones". To answer the question correctly, the search engine would not only have to parse the entire question, but realise it needs to find a discography, and then interpret that to find the answer.

3.4.0 Matching Techniques

This subsection is concerned with the various methods of matching queries to documents.

3.4.1 Exact and Range

Exact matching is the process of finding an exact match to a query within a database, a perfect match from query string to a string in a record. This is a perfectly valid technique when searching for such things as a customer name in a database, but in an Information Retrieval system working with free-format documents with no fixed vocabulary will result large amounts of potentially relevant material not being highlighted to the user. It is far better to use techniques which return a set of approximate or possible matches, and let the user filter what he or she actually needs.

A range match will match all terms within a defined set of boundaries, for example, you may query all employees in a database paid between £20,000 and £30,000 using a range match. Its use in database systems is clear, but how can this be sensibly transferred to an information retrieval system? My conclusion is that it can’t. Whilst you may be able to isolate numeric terms in an index and compare them, by the nature of most IR systems you will have lost the context of that numeric term, and therefore you may not be comparing like with like. A full text engine may be able to work back from the term and try and associate it with a context, but this is not a reliable method and matches may well be omitted.

3.4.2 Proximity

Proximity matching is used to match terms only when they occur within a certain distance of each other in a document, whereas the other methods will match according to occurrences of the words anywhere in the document. The method is useful in situations where a search term needs context to produce relevant matches. For example, a search for “Joe Cocker” with no proximity matching could return home pages of people called Joe, who in their page at some point mention their cocker spaniel, whereas what we need are matches specifically mentioning “Joe Cocker”. This can only be done on full text search engines, where the program has access to a copy of the original document text to check the proximity of the search terms.

3.4.3 Vector Based Matching

Vector based matching is used to determine how close a match is to a query. The query is put into a vector, and the result weightings for each search are also put into vectors into the same order. For convenience let us consider a search with two terms (note that the text is not intended to represent body text, but document titles to give a rough idea of content).

Query (Q) : $Cat_{0.5}$, $Dog_{0.5}$ -> As a vector (0.5,0.5)

Search result A: “All dogs go to heaven” (0, 0.25)
Search result B: “Cats and dogs” (0.33,0.33)
Search result C: “Siamese Cat” (0.5, 0)

To rank the search results we can visualise them plotted on a graph :

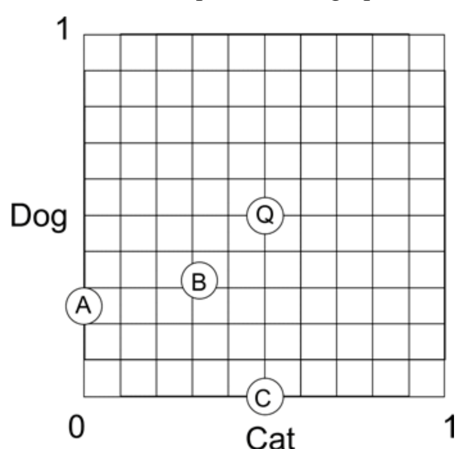

Figure 4 : Plot of a vector query and document results

To find out which is a closest match we simply look at the graph and see which point is actually closest in the literal sense, in this case it is obviously result B. Using a formula to calculate the distance between the two points is rather more convenient :

Let $r=(r_1, r_2)$ be a result vector, let $q=(q_1, q_2)$ be the query vector. Distance d is calculated as :

$$d = \sqrt{(r_1 - q_1)^2 + (r_2 - q_2)^2}$$

What if we have more search terms? Three dimensions seems viable, as distance in a three dimensional

space is measurable. It is when four dimensions or more are involved non-mathematical types like myself run screaming for the hills. Fortunately the distance between two points formula extends quite simply, so for n search terms :

Let $r=(r_1, r_2 \ldots r_n)$ be a result vector, let $q=(q_1, q_2 \ldots q_n)$ be the query vector. Distance d is calculated as :

$$d = \sqrt{(r_1 - q_1)^2 + (r_2 - q_2)^2 + \ldots\ldots + (r_n - q_n)^2}$$

We then rank the results according to absolute distance, closest being the best match.

However, there is a problem with vector based techniques in that we use a weighting of zero for terms which do not appear in a document's index terms. This value is really undefined as the term could conceivably be in the document, but been omitted by the data reduction techniques mentioned earlier. A solution to this problem is to have an "undefined" value, which, if encountered an element in the result vector, causes the element in both the query and result vector to be omitted from the distance calculated. All distances are then multiplied by a number derived from the number of terms present in the result, so as to increase the apparent "distance" of results which have terms missing.

Vector based matching also introduces the concept of cosine measures between a query and a document. This value is based on the angle between the query vector and a document. This angular measure has no concept of distance, and so theoretically a short paragraph on a subject would have a roughly equal cosine measure to a full length paper on a subject. Korfhage (1997) cites this formula for cosine measure :

$$\sigma(D,Q) = \frac{\sum_k (t_k \times q_k)}{\sqrt{\sum_k (t_k)^2} \times \sqrt{\sum_k (q_k)^2}},$$

Where t_k is the value of term k in the document and q_k is the value in the query, D and Q are the document and query respectively. In a small document collection this may be useful, but in a large document collection it makes more sense to treat long documents as "better matches" than shorter ones.

3.4.4 Relevance Feedback

Relevance feedback is the process of improving future search results by having the user return his/her own measures of relevance for matches to a query. This is then used by the system to alter its behaviour in favour of those matches that were deemed “good” by the user. The adjustments are usually done by finding common ground between good or bad matches and adding additional query terms transparently. By an iterative process of result and feedback stages the initial results will hopefully be refined rapidly into a set of relevant documents.

Feedback can alter behaviour of the system globally for those search terms, in order to improve the whole system’s performance, be added to a user’s profile, or used just for that particular query session. Any sort of storage of feedback beyond that individual query session is open to abuse, you cannot rely on a user’s feedback being sensible. However, we could be losing good, useful feedback. This is an issue in expert and corporate systems where we can assume people are acting professionally, but on a publicly accessibly web search engine the best use for relevance feedback would be to collect rough statistics for query performance.

CHAPTER FOUR - DATABASE METHODS AND ALGORITHMS

4.0 Introduction

In this chapter we will examine some of the major areas of database methods and algorithms and their relevance to a customised search engine index system.

4.1 String Comparison

The comparison of strings is a major part of a search engine. The fundamental process of a search engine is comparing textual search terms with the text of documents, hence a good string comparison technique is very important.

String comparison can either be Boolean (true, false), or can give a measure of similarity. The two main measures of similarity are the Hamming distance and the edit distance. Baeza-Yates (1992) defines the Hamming distance as “the number of symbols in the same position that are different (number of mismatches)”. The Hamming distance is only defined for two strings of the same length. The edit distance is defined as “the minimal number of symbols necessary to insert, delete, or substitute to transform string s1 to s2”. These measures of similarity are of limited use in our application. Do we care if, according to the Hamming distance, if “CARS” and “CART” are a pretty good match? The two words mean completely different things.

This kind of fuzzy string matching will allow minor variations in words to be classed as a match. This is useful in systems consisting of large numbers of optical character recognition sourced documents, and its use will also avoid omitting possibly relevant documents from query results because of typographical errors in them. However, in a large database such as a web search-engine we can possibly afford to lose such matches for the sake of overall search relevance.

What is more useful to us is a straight Boolean string comparison. If we need matches for similar strings we can apply a stemming algorithm or soundex to them before attempting a normal Boolean comparison.

The simplest method of doing a string comparison is to do a byte-by-byte comparison of the string, returning when a mismatch is encountered or the end of both strings is reached. More cunning

algorithms compare word-by-word (that is data word, rather than English word). Comparing groups of say, 32 bits, reduces the time needed to compare the strings considerably.

Either way, when a routine is being used intensively it is best to find an optimal way of doing it, or find a better way altogether…

4.2 Hashing

The process of string comparison is relatively slow, especially when done intensively, which is the kind of use we can expect when searching a sizeable database. Even if comparison is done word-by-word (typically 4 bytes, 32 bits) a string comparison will take considerably more processor time than a single integer comparison.

This is where hashing can become extremely useful. Strings can have their hash calculated when they are entered as a search parameter, and at time of words being added to an index. Then only these fixed length numbers need be compared during a search. A good hashing function with an even distribution will also result in efficient binary trees (described later in this chapter).

There are many hashing algorithms, the two most common being a checksum and a cyclic redundancy check. The simplest, the checksum is merely a sum of all the bytes in a data block, which wraps around to zero if the addition overflows. This is particularly unsuited to our application, as checksums from short strings will give a high incidence of "hash collision", where two differing data areas will give the same checksum. To illustrate :

String	ASCII	Checksum
TAB	84 65 66	215
BAT	66 65 84	215
DOG	68 79 71	218
CAT	67 65 84	216
TRAIN	84 82 65 73 78	382
PLANE	80 76 65 78 69	368

Table 3 : Checksum hashes of several sample ASCII strings

As you can tell the algorithm produces the same results for strings with swapped bytes ("BAT" == "TAB") – not massively useful as there are only a few letters in the English language, and so the potential combinations are rather limited.

A cyclic redundancy check (CRC) is far more sophisticated, and is calculated using division either using shifts and exclusive ORs or table lookup (modulo 256 or 65536) treating blocks of input bits as coefficient-sets for polynomials. (FOLDOC) These calculations are tedious to calculate manually, and difficult to illustrate.

Another method, as used by Java in its java.lang.string API uses exponents in the manner of :

$$s[0]*31^{(n-1)} + s[1]*31^{(n-2)} + \ldots + s[n-1]$$

using integer arithmetic, where $s[i]$ is the ith character of the string and n is the length of the string. I shall refer to this as the exponential method.

We should seek something close to what is known as a *minimal perfect hashing function* (Frakes, Baeza-Yates, 1992), one which will produce a unique result for all the data we feed to it. This is not possible unless we define what we aim to feed to it and how large a hash number we will allow. We can assume 32 bit words will be used for the hash, and that the data we will represent is a dictionary of words. It seems logical to feed a dictionary through potential hashing algorithms to determine their effectiveness. To this end I wrote a short program to process the standard UNIX dictionary, finding the number of hash-collisions using several hashing algorithms. The program itself is included in appendix B.

	Checksum	CRC	Exponential Method
Number of hash collisions	43,969	12,590	0

Table 4 : Comparison of number of hash collisions when generating keys for a dictionary of 45,373 English words in lower case ASCII

These results clearly show that in terms of minimising the occurrence of hashing collisions the exponential method is best.

A second function of the program was to find the "spread" across the range of the hash-space, by dividing the 32 bit range of the key size into a number of evenly sized blocks, and counting all the hash keys generated in each of these ranges. This is an important measure, as any bias in the distribution of keys will affect the efficiency of the generation of B-trees. The results of this test are graphed below :

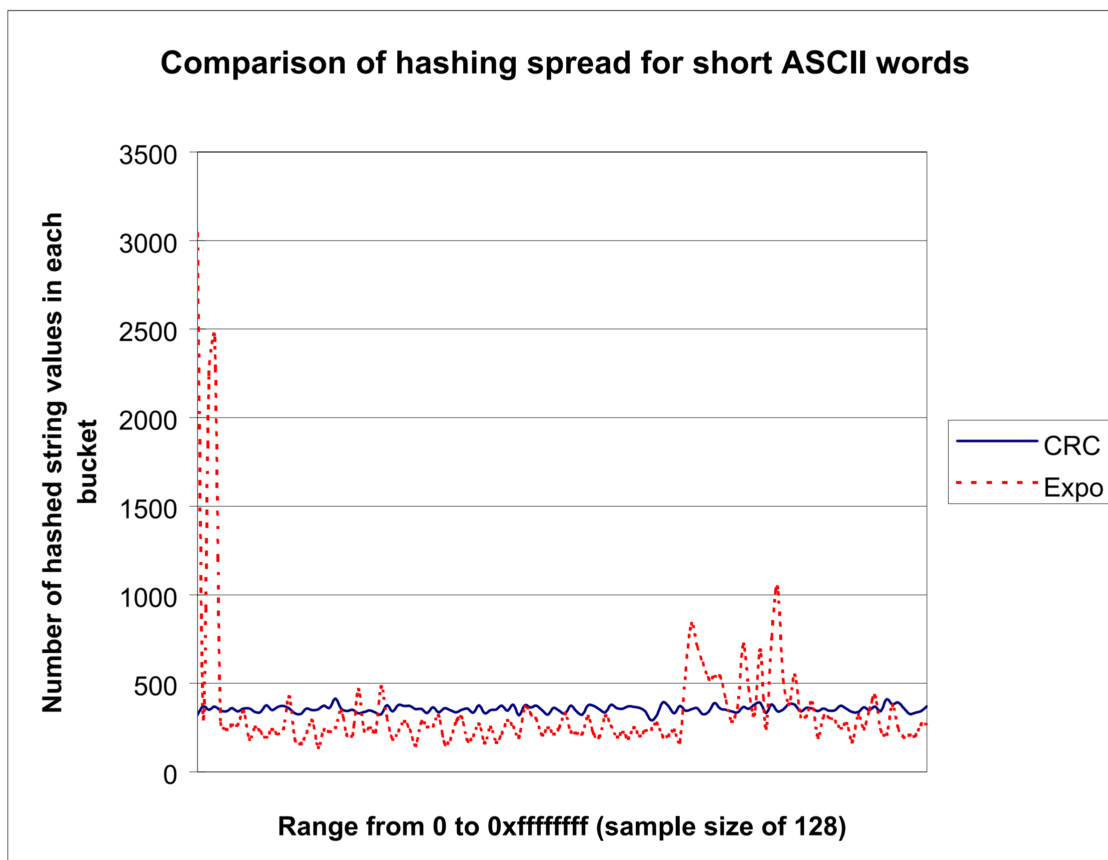

Figure 5 : Comparison of hashing spread for short ASCII

The checksum method is not featured on the graph, as all of its hash-keys fell in first sample, and so distorted the rest of the graph. Bearing this and the number of hash-collisions the method generated we can discount it from further consideration.

The graph shows an even distribution of CRCs across the range, but the exponential method is far less consistent. The large peaks early on in the graph could result in performance loss if the hash keys were used to form a B-tree.

A hash may also be used as a method of splitting a database up, either across files or across machines in a distributed system. Each file or machine can be allocated a range of hashes to store and operations on those records directed accordingly. Allocating ranges will fall foul of uneven hashes such as the exponential method, the load may well be spread unevenly. A better way is to use the

$$m = h \ MOD \ n$$

Where m is the number of the machine or file, h is the hash key of the record and n is the number of machines or files. This should produce an even distribution of load, no matter what the spread of the

hashing function is.

4.3 Boyer-Moore-Horspool –Sunday Matching

Boyer-Moore-Horspool –Sunday (BMHS) matching is a development of Boyer-Moore-Galil (BMG) matching, which in turn is a development of Knuth-Morris-Pratt (KMP) matching. I will only cover the basic concepts of the algorithms as Baeza-Yates (1992), in his analysis of matching techniques presents figures for performance of these matching techniques on the English language, and determines that BMHS is the best method. As such with limited time and space it would be foolish to cover all the less efficient algorithms in detail. It is the culmination of twenty years of work, and so I cannot possibly hope to explain or understand all the processes involved in the time available.

All these algorithms are designed for situations where it is necessary to find a string within a string. Consider searching the string "pilllspillar" for "pillar". It is not possible to use a simple hash comparison in such situations as the two strings will have different hashes, and the hashes will give no indication of the similarity. The brute-force approach would be to compare "pillar" with each substring of the same length within "caterpillar". That is six comparisons of complete strings, or thirty six character by character comparisons. There is clearly a lot of redundancy. KMP guarantees that that there are only *n* character by character comparisons for a string of length *n*. It does this by constructing a *finite state recogniser* which is effectively a shift table. By calculating shifts for mismatches at each of the substring character positions we avoid the duplicate comparisons in brute force methods.

The algorithm was improved on by Boyer-Moore-Galil (BMG), which again uses a finite state recogniser, but the substring comparisons start from the right. This means if the last character of a search term doesn't match the last character of the current section of the search data the search position can immediately be advanced by the entire length of the search string. Boyer-Moore-Horspool-Sunday (BMHS) is the most effective technique, the simplest to implement, and the most efficient. Its lookup table contains shifts to align the pattern with the text character appearing one character to the right of the current alignment. Its operation is complex, but the source implementation is simple.

4.4 Inverted Indexing

An inverted index contains all the index terms and a document reference for each document that term is contained in. The index is inverted in the sense that there is an entry for every mention of a term, whereas a traditional index would have a single entry for each term, followed by a list of pages/documents which reference that term. They may be searched and sorted by many methods, but

because of the potential size of our list many can be discounted at an early stage. For example, sorted arrays are not suitable as we cannot afford to sort or move data to allow alphabetical insertion.

4.5 Binary Trees

When dealing with a large database, sequential comparison of every record against a search term becomes too slow to be practical. In a worst case scenario, locating an individual record using a sequential search will take as many comparisons as there are records.

A binary tree, or B-tree is a method of data organisation which, surprisingly enough, resembles a tree (although in computer science they are drawn upside down by convention). The root is a data element, the first element which was added to the tree. Each element in a tree is known as a node. Each node has two pointers, a left pointer and a right pointer. When a new element is added its key value is compared with the key value of the root, and according to a rule it is added to either the left or right hand pointer if it is empty. If the branch isn't empty, the routine compares the existing node in the same way, traversing through the tree until an unattached pointer is found.

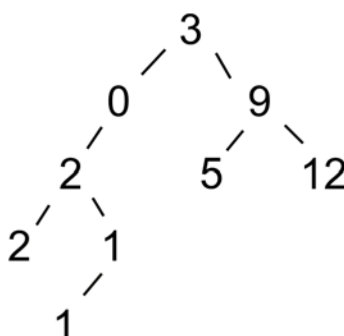

Figure 6: A simple binary tree structure

The tree above is a simple list of numbers, and is structured so keys greater than the parent node key go right, and keys less than or equal to the parent's go left.

Finding an element in a tree is simply a matter of traversing through it in the same way as you would when adding an element. The fewer hops a search has to make the faster it is, so it makes sense to keep the tree as evenly spread as possible.

Numeric keys are straightforward, but what about strings? The B-tree could be indexed by string, using a string comparison method that returns a measure of similarity, however it is far more efficient to index by a hash of the index string. By using a good hashing function we can be assured a good spread across the B-tree so performance will be consistent.

4.6 "Chunked" Files

A "chunked" file is a file full of uniform length records. These records are linked together into "chains". When an arbitrary block of data is added to a chunked file the program will create a new "chain" of chunks, linked together, with their data areas filled.

A free list pointer is kept at the start of the file, when a chunk is deleted it is linked into that list ready to be reused.

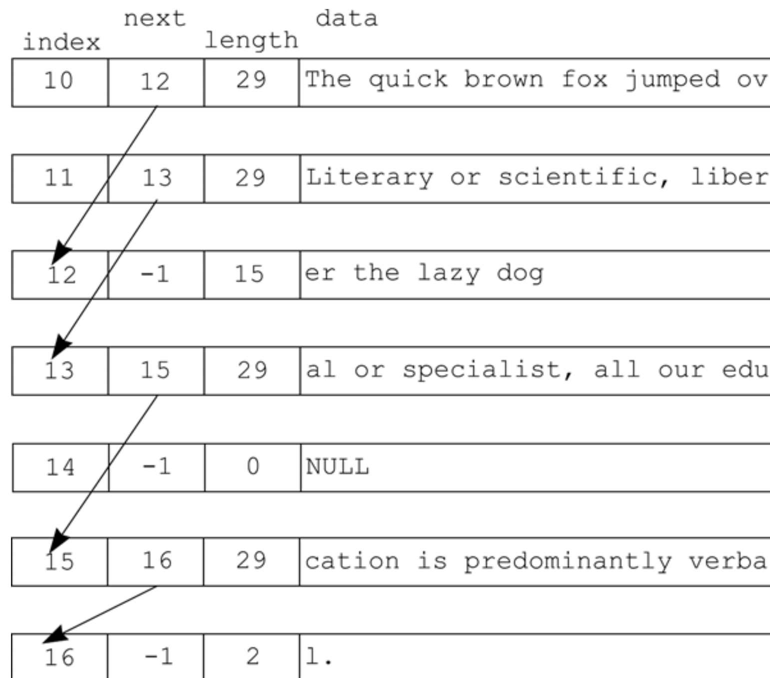

Figure 7 : An example of a chunked file

The diagram above has two distinct chains, one starting at index 10, the other starting at index 11. Notice that the use of –1 indicates "end of chain". There is a single empty chunk at 14, this would typically be in a chain of free chunks waiting to be re-used. The length measure is slightly wasteful, in that most of the chunks will typically be full. The use of a string terminator would be more efficient, but this way we are not restricted to storing strings, but can store any arbitrary block of data, no matter what it contains, with no fear of accidentally introducing a terminator.

The major advantage of this technique is we can have variable length records, which can be accessed in the same sort of time expected of a fixed length record database. Chains can also easily be extended. This makes it ideal for storing such things as lists of documents and weightings for each word. There is a file-size penalty, in that space is needed for each chunk header, and every non-full chunk will waste a certain amount of space, but by altering the length of the data stored in each chunk we can attempt to minimise this overhead.

If implemented properly we can ensure the start chunk of each chain always remains the same, and so its index (and therefore offset in the file) can be used as its key. This makes finding the start of any particular chunk chain as quick as one seek operation.

We can expect a certain amount of fragmentation as records are deleted, but in an ever-expanding database such as a search engine this is unlikely to be a serious problem, as any free chunks will soon be filled.

This type of structure is not specifically documented in any of the texts as far as I have found through my research, although it is no great leap of imagination, essentially just a linked list in a file. It is based on work I did on a 1996 project to develop a fast text indexed search system.

4.7 Database Management Systems vs. Custom Code

The use of an existing Database Management System as the actual data storage mechanism for a search engine has many advantages. Once developed in a DBMS environment all the usual thorny issues of a database, such as concurrent access, reliability, roll back, commit procedures, locking and such should be dealt with by the environment itself.

However, the needs of a search engine are unique and specific, and will not fit effectively into the traditional database paradigm. When dealing with such short entries and indexing so much data the extra flexibility introduced by coding the database from a high level language can become useful. Being able to specify exactly how records should be stored to disc and in memory allows the programmer to optimise the application, without being restricted by the facilities of a DBMS. Development time may be longer, but the end result should be a faster database.

To illustrate, in 1996 I implemented a custom database in C to word-index 400,000 records from a drawing office. The existing Microsoft Access system, searching the same data, took 5 minutes to complete a single term search. The custom code took an average of two seconds to complete a multiple term search. Access may be a particularly bad example, as its "Jet" DBMS is not widely recognised as being particularly fast, but it still shows there are benefits to be had.

Another issue is cost. In an organisation it may well be justifiable to spend several thousand pounds on a licence for a good DBMS, as money will presumably be saved by reducing the amount of time the programmers need to implement the database. As a student I have no such finances, and so the cheaper option is to just roll my own.

CHAPTER 5 – EVALUATION OF SEARCH ENGINES

5.0 Introduction

A search engine is a combination of information retrieval techniques, and a database system. So, it follows that we should apply the accepted measures of performance for both in our evaluation. In this section we evaluate established methods of information retrieval and database performance and their relevance to an internet search engine.

5.1 Measures of Information Retrieval Performance

Information retrieval systems are judged on a balance between their efficiency (speed, size) and effectiveness. Their effectiveness is the performance of the system from the user's point of view. Were they satisfied by the response they were given? Were the returned matches relevant to the query? This satisfaction measure is arbitrary.

There are two main measures of retrieval effectiveness, recall and precision. The method for calculating recall is :

$$R = \frac{r_r}{r_d}$$

Where R is the measure of recall, r_r is the number of relevant documents retrieved, and r_d is the number of documents in the database. In anything other than a small test collection the number of relevant documents in the database is unknown, and so a figure is estimated by sampling or some other method.

The method for calculating precision is :

$$P = \frac{r_r}{r_t}$$

Where P is the measure of precision, r_r is the number of relevant documents retrieved, and r_t is the

number of documents retrieved in total.

On their own these measures make little sense, a system could theoretically claim 100% recall by returning all the documents in it. A combined measure of recall and performance was developed by van Rijsbergen (1979), and is cited in Frakes Baeza-Yates (1992). The evaluation measure *E* is defined as:

$$E = 1 - \frac{(1+b^2)P}{b^2 P + R}$$

Where *P* = precision, *R* = recall, and *b* is a measure of the relative importance of recall and precision to the particular user (*b* of 2 would indicate that the user is twice as interested in recall than precision, 0.5 would indicate the opposite).

The measure above involves levels of guess-work and “fudging” to attempting to express unquantifiable as a figure. When we specify *b* in van Rijsbergen’s formula, how do you rate their preference (a qualitative measure) as a quantity other than wild estimation? Perhaps we are working with easily satisfied users? Personally I prefer more the more tangible measure such as speed and size, and a simple survey of satisfaction from users is just as useful as abstracting the evaluation into statistical methods. There is too much of an element of “pick a number, any number” in van Rijsbergen’s measure.

5.2 Measures of Database Performance

A database is judged on reliability, integrity, speed and size. These are all admirable qualities in an internet search engine too, but the nature of the data does mean certain allowances can be made. For example if index entries are lost through a crash, they can always be re-indexed. We are not indexing perishable data which will need to be re-entered by a user, so having to index a few pages again is not a significant loss. Speed is more of an issue, as web users will not wait for queries to process. To achieve such speed most current systems use a brute force approach, distributing the system over many machines. However, the other way of increasing speed is to optimise the use of resources, less data to search means the system will run quicker. Combined with scalability the total cost of the system is reduced in terms of “bang-per-buck”.

CHAPTER 6 – CONCLUSIONS

6.0 Introduction

The conclusions drawn for this part of the project will be a short analysis of the methods to determine which are suitable for the type of engine which will eventually be implemented. There will be no design documentation at this stage as this document is a literature review.

6.1 Methods and Techniques

Having spent six months researching information retrieval techniques I have found that the field is far wider than I had previously expected. With each area of research, more have presented themselves. Books are only just being published now with specific references to information retrieval and the internet, and so much of my work was reading about general information retrieval topics and considering their application to an web search engine.

An internet search engine may only draw from a small subset of these techniques as many of the methods depend on manual intervention in the indexing process, or the use of certain standards in the documents to make indexing more straightforward. HTML has such standards, such as the provision of keywords and abstracts, but there is no pressure to use them. The future use of XML and the Dublin Core will not improve this situation. The fact that we have no control over the documents we are indexing immediately invalidates such techniques as controlled vocabularies in the context of a web search engine.

Considering this we must concentrate on the automatic indexing of documents which do not provide abstracts and such, as they make up the greatest proportion of documents on the web, but also aim to use any meta-data which may be provided. For this I propose to form indexes by the use of word frequency ranking, a stop-list and Porter’s stemming algorithm. The inverse document frequency weight of terms will be stored, however there are problems associated with this which I will have to resolve first. Namely I am not confident that the simple use of logarithms compensates for a growing document collection. Zipf’s law will be ignored in terms of implementation, as in section 3.2.2 and other experiments I have found it to be so unreliable in practice as to be irrelevant.

Many of the techniques, such as the use of thesauri and knowledge bases introduce problems of their own. It could be argued that such methods, when not in the context of expert systems, actually hinder the retrieval process. Is it not better to admit the fact that automatic indexing is a dumb statistical process, and not have engines making half hearted attempts to be intelligent? Such systems remind me of the early "intelligent" computer psychiatrist ELIZA. Occasionally it gave sensible answers, but for the most part its impression of intelligent conversation was unconvincing ("Tell me about your fish cow?"). Although such systems have progressed the problem still exists. Computers are not intelligent, and can only make a vague impression of being so. When performing a presumably serious operation, looking for documents, do we really want such imprecision introduced?

Without having humans maintain the "knowledge" of such a system it has little hope of producing useful results. As soon as this need is introduced the benefits are reduced. Companies would rather have users put a little more thought into their search terms than have to employ any sort of maintainer for a search engine. It is only the large systems such as AltaVista whose business is searching which can afford to provide this sort of support for their engine.

A phrase dictionary will not be considered as no suitable dictionary is available. One could be added at a later date with relative ease. The use of phrase dictionaries is interesting and useful, and their automatic generation could provide yet another interesting project in its own right.

In terms of data formats an index will be tree based, and I intend to do further investigation to determine whether the polynomial string hash or a CRC would be better in practice for the tree keys. The tree will be formed in a chunked file. The combination of the two techniques will be an interesting challenge.

Matching will be done using a vector based system, modified in implementation to deal with missing terms in a sensible manner. Proximity matching cannot be sensibly implemented in a non-full-text engine.

The query interface will be Boolean, possibly extended. This is because it is the most straightforward, the implementation of other query techniques would warrant a project of their own.

Relevance feedback will be a possible future extension to the project, as our main concern is getting good results in the first place.

6.2 Project Future

A full featured search engine is a huge project, and so we need to narrow the scope of the work in order to achieve useful results, rather than attempting to implement everything badly in too little time. It is

also important to note that existing search engines have been developed over many thousands of man hours by teams of programmers, a single student cannot equal that effort in six months.

The focus of the implementation stage of the project will be indexing and document analysis.

The interface will be a simple Boolean search, possibly featuring weightings. This is because the use of other techniques, such as natural language parsing, would warrant a project of their own. However, the project will be modular in form, so at a future stage NLP or some other method could be implemented.

A project plan is not being submitted, as I feel the only limitation is time. I will investigate topics and methods as I progress. Any plan would restrict me, and as far as I am concerned the only plan with the flexibility I require is “no plan at all”. Instead I will define areas of work, and progress from there :

- Document Analysis – Investigate the use of as many as possibly of the document analysis methods which I have deemed to have potential for this project.

- Indexing – Attempt to produce the most efficient index database system possible in the time available.

- Matching – Investigate matching methods, and their performance with the rest of the system.

The demonstration of all such methods will involve parsing, analysing and adding web pages to an index database. Implementing this initial framework will be the first priority. From then on experimentation with different analysis and indexing techniques can commence.

It will be difficult to compare work with existing engines for several reasons :

- Lack of test data. I assume the speed of a search engine will not decrease in a linear fashion along with the increase in the number of documents indexed. I will only be able to index a small number of documents due to storage and network limitations here. Current JANET charging would make it very expensive for me to attempt any sort of large index, and it would be massively inconsiderate on a shared network.

- Lack of resources. I cannot distribute my system across many machines as I have not got the resources (hardware or time) to do so. I do hope to implement at least the groundwork for such a system, to allow for future expansion.

- Lack of comparable data. I cannot borrow another engines data to compare against, and I can only gather statistics on other engines from the most superficial of user perspectives. Few of

the engines I have looked at give any sort of detail on what hardware they are running or the size of their data set.

It would also be unfair to compare a prototype system with fully developed engines. This work should be considered a test-bed and proving ground for some of the areas I have researched.

BIBLIOGRAPHICAL REFERENCES

Raggett, D., Le Hors, A., & and Jacobs, I. (Editors) (24th April 1998) *HTML 4.0 Specification*
W3C Recommendation

Korfhage, R. R. (1997) *Information Storage and Retrieval*
John Wiley & Sons, Inc., New York

Stevens, W. Richard (1990) *UNIX Network Programming*
Prentice Hall PTR, New Jersey

Sparck Jones, K. and Willett, P. (editors) (1997) *Readings In Information Retrieval*
Morgan Kaufmann Publishers Inc., San Franscisco, California

Levine, J. R., Mason, T. and Brown, D. (1995) *lex & yacc*
O'Reilly Associates, Inc, Sebastopol, California

Frakes, W. B. and Baeza-Yates, R. (1992) *Information Retrieval*
Prentice Hall, Upper Saddle River, New Jersey

Kernighan, B. W. and Ritchie, D. M. (1998) *The C Programming Language (2nd Edition)*
PTR Prentice Hall, Englewood Cliffs, New Jersey

Owen, F. & Jones, R. (1994) *Statistics (4th Edition)*
Pitman Publishing, London

R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee (June 1999)
RFC 2616 Hypertext Transfer Protocol -- HTTP/1.1
http://www.faqs.org/rfcs/rfc2616.html

Fielding, F. (June 1995) *RFC 1808 Relative Uniform Resource Locators*
http://www.faqs.org/rfcs/rfc1808.html

Berners-Lee, T., Masinter, L. and McCahill, M. (December 1994) *RFC 1738 Uniform Resource Locators (URL)*
http://www.faqs.org/rfcs/rfc1738.html

Various/Anon (Viewed October 1999) *The Web Robots Pages*
http://info.webcrawler.com/mak/projects/robots/robots.html

Sanderson, M. (Viewed November 1999) *IDOMENEUS technology transfer server*
University of Glasgow
http://www.dcs.gla.ac.uk/idom/

Malhortra, Y. (1998) *TOOLS@WORK: Deciphering The Knowledge Management Hype*
Journal for Quality and Participation (July/August 1998) Vol. 21 No. 4, pp. 58 – 60.
Also published as *Knowledge Management for the New World Of Business*
Brint.com Institute, Fort Lauderdale
http://www.brint.com/km/whatis.htm

Peper, F. (Viewed November 1999) *What is Principal Component Analysis?*
Communication Research Laboratory of the Japanese Ministry of Post and Telecommunications
http://www-karc.crl.go.jp/avis/peper/pca.html

Porter, M. (Viewed November 1999) *The Porter Stemming Algorithm* (Official home page)
Muscat Limited
http://www.muscat.com/~martin/stem.html

Porter, M. (1980) *An Algorithm For Suffix Stripping*
In Sparck Jones, K. and Willett, P. (editors) (1997) *Readings In Information Retrieval*, pp. 313-316
Morgan Kaufmann Publishers Inc., San Francisco, California

Howe, D. (Editor) (Viewed November 1999) *Free On-Line Dictionary Of Computing (FOLDOC)*
Imperial College, London
http://foldoc.doc.ic.ac.uk/foldoc/index.html

J. Ziv and A. Lempel (May 1977) *A Universal Algorithm for Sequential Data Compression*
IEEE Transactions on Information Theory, Vol. IT-23, No. 3, May 1977, pp. 337-343

IBM Intellectual Property Network (viewed November 1999)
http://patent.womplex.ibm.com/

Welch, T. (December 1985) *US Patent 4558302: High Speed Compression and Decompression Apparatus and Method*
Sperry Corporation, New York
http://patent.womplex.ibm.com/details?&pn=US04558302__ (double underscore) (viewed November 1999)

Excalibur Corporation Homepage (viewed December 1999)
Excalibur Corporation
http://www.excalib.com/

Foster, J. (editor) (August 1994) *RFC 1689 A Status Report On Networked Information Retrieval: Tools & Groups*
University Of Newcastle-upon-Tyne
http://www.faqs.org/rfcs/rfc1689.html

Harel, D. (1992) *Algorithmics – The Spirit Of Computing (Second Edition)*
Addison-Wesley Publishing Company

Bott, F., Coleman, A., Eaton, J., Rowland, D. (1996) – *Professional Issues In Software Engineering (Second Edition)*
University Of Wales, Aberystwyth
University College London Press / Pitman Publishing

Neiderst, J. (1999) *Web Design In A Nutshell*
O'Reilly Associates, Inc, Sebastopol, California

Kirkpatrick, B (editor) (1998) *Roget’s Thesaurus of English Words & Phrases*
Penguin Books Ltd, London

Yahoo! Homepage (viewed January 2000)
Yahoo! Inc., Santa Clara, California
http://www.yahoo.com

Copernic Homepage (viewed January 2000)
Copernic Technologies Inc., Sainte-Foy, Canada
http://www.copernic.com

Lawrence, S., C. Lee Giles (April 1998) *Searching the World Wide Web*
NEC Research Institute, Princeton, NJ USA
Science Magazine, Vol. 280, 3 April 1998

Lawrence, S., C. Lee Giles (July 1999) *Accessibility of information on the web*
NEC Research Institute, Princeton, NJ USA
Nature Magazine, No. 400, 8th July 1999

Glover, E. J., Lawrence, S., Gordon, M. D., Birmingham, W. P., Lee Giles, C. (2000)
Web Search Your Way
NEC Research Institute, Princeton, NJ, USA
Artificial Intelligence Laboratory, University Of Michigan, USA
Business Administration, University Of Michigan, USA
Paper unpublished at time of writing, accepted for publication in Communications of the Association for Computing Machinery

OLCC (November 1997) *Dublin Core and Web MetaData Standards Coverage in Helsinki*
Online Computer Library Center, Inc. Dublin, Ohio, USA
http://www.oclc.org/oclc/press/971107a.htm

Kunze, J. (December 1999) *RFC 2731 Encoding Dublin Core Metadata in HTML*
http://www.ietf.org/rfc/rfc2731.txt

Inktomi Corporation Homepage (viewed January 2000)
Inktomi, San Francisco, USA
http://www.inktomi.com

Sample text for illustrating methods taken from :

Otto Seibold, J., Walsh, V. (1994) *Mr. Lunch Borrows A Canoe*
Viking, Penguin Books USA, New York

The Labour Party (1997) *Britain will be better with new Labour*
The Labour Party, London

The Conservative Party (1997) *The Conservative Manifesto 1997*
The Conservative Party, London

This bibliography will be expanded by literature surveys.

GLOSSARY

This Glossary covers terms which may be unfamiliar to the reader. A grounding in Computer Science is assumed, so standard computer science terms are omitted as they are assumed to be common knowledge – for example it would be ridiculous to expect anyone to understand this document without knowing what the World Wide Web or Internet is, so they are not defined here.

Antonym

A word having a meaning opposite to that of another word, "Wet" is the antonym of "Dry", for example.

Boolean Query

A query in which terms are connected by Boolean operators such as AND, OR and NOT. Extended Boolean queries may include weightings.

CGI

Common Gateway Interface, the standard method for interfacing programs with a web server.

Collision

Occurs in hashing where two different data areas give the result in the same hash code.

Compression

Use of algorithms to reduce the size of data by detecting patterns within it. Lossy compression sacrifices quality for increased compression, lossless compression loses no quality at the cost of limited compression.

Controlled Vocabulary

A restricted set of words allowed to describe documents in order to make indexing more straightforward.

Cosine Measure

Vector angle between two documents, used to measure similarity.

Crawler

See *Robot*.

Cyclic Redundancy Check (CRC)

A number (hash) derived from data, intended to allow the detection of errors within it.

Data Reduction

The process of reducing data to a smaller form whilst maintaining the necessary amount of meaning.

Directed Advertising

Advertising selected on the basis of information known about a user's interests.

Document

A stored data record in any form.

Document Surrogate

Data used to describe a document, such as a list of keywords, an abstract etc.

Document Analysis

The process of analysing a document to determine its parts, general subject area and so on.

Document Surrogate

A limited representation of a document, in the form of keywords, an abstract and so-on.

Dublin Core

A standard for metadata elements to describe documents and information resources.

Checksum

A simple hashing method that involves summing all the data elements in a block.

Cyclic Redundancy Check (CRC)

A hashing algorithm commonly used for error detection.

Extensible Markup Language (XML)

Successor to HTML, which is extensible allowing the definition of new elements.

Effectiveness

The quality of an information system's response to an information need.

Hashing

The process of turning a block of data into a number, either to identify that block of data or to provide error correction etc. Methods include *checksums* and *cyclic redundancy checks*.

Hash Collision

An incidence of two different blocks of data giving the same result when hashed.

Hashing Spread

The distribution of hashes of data for a given data set across the range of numbers available for a hash.

Homonyms

Words which are pronounced the same but differ in meaning, origin, or spelling.

HTML

Mark up language, text based with tags which is the main format for documents on the web.

HTTP

Hypertext Transfer Protocol, text based protocol for transferring documents. The protocol most commonly used for the transfer of HTML documents and other web material.

Information

Data – typically documents, that have been matched to a particular information need.

Information Need

The requirement to store data in anticipation of future use, or to find information in response to a current problem.

Information Retrieval (IR)

The location and presentation to the user of information relevant to a query.

Inverted Index

An index organised so each term directly identifies the documents containing that term.

Inverse Document Frequency

The logarithm of the reciprocal of the number of documents in a collection that contain that term.

Knowledge Base

Algorithms, facts, concepts and rules that form a representation of knowledge of a particular area.

Meta

A prefix taken from philosophy which means “one level higher”. In web orientated terms a metasearch engine is a search engine providing another level above several search engine to provide a single interface, and metadata are elements of a document at a different level to the document text, keywords and abstracts for example.

Morphological Complexity

The level of complexity of a language’s structure, word forms, inflections, derivations, compounds and such.

Natural Language Processing (NLP)

Processing of written text into a form useable by an information system.

Query

The formal expression of an information need.

PDF

Adobe’s Portable Document Format.

Recall

The proportion of relevant documents that are retrieved from a query.

Relevance Feedback

The process of using relevance ratings from users to improve the quality of future queries.

Robot

Program which downloads web pages, and follows the links within them to feed a search engine indexer.

Stop word

A word which should not be added to an index, typically because it has no meaning on its own.

Soundex

Method of translating a string into a code such that words which are pronounced the same but spelt differently (homonyms) produce a matching code

Stemming

The process of taking a word and reducing it to its morphological root, by removing tenses and suchlike. For example “stemmer”, “stemmed”, “stemming” would all be recognised as being variants of “stem”.
.

Synonym

A word or an expression that serves as a figurative or symbolic substitute for another.

Weighting

A numerical representation of a document’s relevance to a particular term or query.

XML

See *Extensible Markup Language*.

APPENDIX A – POSSIBLE FUTURE WORK IN THE AREA

A.1 Determination of "quality" of document by its rendered appearance

Develop a document analysis system, possibly by use of a neural network to identify well formed academic texts from homepages. In much the same way as tabloid style print can be distinguished from a broadsheet we should also be able to make general judgements by page formatting on the type of document we are analysing. Obviously an imprecise method, but then all the methods are imprecise! This would have implications on download and indexing speed, as all elements of a document would have to be downloaded, rather than just the body text.

A.2 Determination of “quality” of a document by its use of language and structure

Develop a learning system based on feedback from search results, to influence weightings in results from a search engine. Users rate the usefulness of documents, and it rates how good that “type” of document is. “Type” being a set of rough metrics gathered from the document at index-time, possibly along the lines of grammatical correctness, use of certain words, paragraphing style etc. These quality measurements could either be stored globally (in which case you would have to rely on all users telling the truth when they rated documents), or a profile of a user could be constructed, according to what type of document he/she rates as useful. This differs from traditional relevance feedback, as we are rating documents in groups of style and technical quality of writing, rather than individually. This method is again imprecise, as there are many syntactically well written documents which make copious use of extravagant verbiage at preposterous verbosity, which remain economic with any sort of factual justification. Any system of this nature will also be language specific.

A.3 Combination of search engines and expert systems

I stated in my description of natural language parsing that any query assuming knowledge will fail. However, by linking in a database of facts a natural language parser may well be able to produce reasonable results. Whilst expecting a system to parse a discography in an unknown format would be unreasonable, discographies could be stored in a more conventional knowledge base which is searched alongside the web index. In our example the system could recognise “Tom Jones”, look for “recorded” attributes on that object, and thus give details on all the songs.

A.4 Developing metasearch-type concepts to provide a truly distributed search engine

From the research of Lawrence and Lee Giles (1997, 1999) we can see that the idea of many independent search engines, whilst ideal for the companies that run them and their stockholders is not the best way of disseminating the most relevant and most comprehensive lists of results to users of search engines. As the web becomes a more and more important communications medium it becomes increasingly important that we can search it effectively. Metasearch engines go some way to improving the situation, but the engines they query overlap on many documents, whilst all constantly querying sites so as to update their independent indexes - so there is a lot of redundancy. I would propose the development of a truly distributed search engine system, with many crawlers responsible for indexing only those web sites on their immediate network geography. Data should be shared using open standards, in a form which can then be picked up by existing engines and added to their indexes. Network traffic would be reduced, index coverage would increase, etc. Alternatively queries could also be distributed, sent to search engines for each network-geographic area, and the results consolidated. One proposal for this could be that web servers themselves could be responsible for submitting their own indexing and weighting data, thus eliminating crawlers. A web server could recognise locally changed documents and automatically re-index and re-submit them.

APPENDIX B – SOURCE CODE

```
/* hashtest.c
 *
 * Author: Tony Howat, November 1999
 *
 * evaluates the number of hash-collisions which occur with short strings
 * and also outputs spread data
 *
 * (word data taken from /usr/dict/words - any newline separated dictionary
 * will do, the longer the better)
 *
 */

#include <string.h>
#include <stdio.h>
#include <stdlib.h>

/* #define CHECKSUM 1    for checksum test
 * #define CRC      1    for crc test
 * #define EXPO     1    for exponentiation test
 */

#define CHECKSUM 1

/* the following just sets up the string equivalent of the definition */

#ifdef CHECKSUM
#define HASH "checksum"
#endif

#ifdef CRC
#define HASH "crc"
#endif

#ifdef EXPO
#define HASH "expohash"
#endif

/* ------ utility functions --------------------------------------------*/

/* strip carriage returns from end of a string */
void stripcr(char *word)
{
  char *loc;

  loc=word+strlen(word)-1;

  while(loc!=word)
  {
    if(*loc=='\n')
      *loc='\0';
    else
      if(*loc=='\0')
        break;
    --loc;
  }
}

#ifdef CRC

/* calculate the crc of a data block of length count */
int hash(char *ptr, int count)
```

```
{
   int crc, i;
   crc = 0;

   while(--count >= 0) {
     crc = crc ^ (int)*ptr++ << 8;
     for(i = 0; i < 8; ++i)
       if(crc & 0x8000)
         crc = crc << 1 ^ 0x1021;
       else
         crc = crc << 1;
     }
   return (crc);
}

#endif

#ifdef CHECKSUM

/* calculate a checksum */
int hash(char *ptr, int count)
{
   int csum=0;

   while(--count >= 0)
   {
     csum+=*ptr;
     ++ptr;
   }

   return csum;
}

#endif

#ifdef EXPO

/* Generates a hash based on
 * s[0]*31^(n-1) + s[1]*31^(n-2) + ... + s[n-1]
 * using int arithmetic, where s[i] is the
 * ith character of the string, n is the length of
 * the string, and ^ indicates exponentiation.
 */

int hash(char *ptr, int count)
{
  int h=0, offset=0;

  while(offset!=count)
  {
    h=31*h+(*ptr++);
    offset++;
  }

  return h;
}

#endif

/* strlower - lower case a string */
void strlower(char *ptr)
{
  while (*ptr)
  {
    if(isupper(*ptr))
      *ptr=tolower(*ptr);
    ++ptr;
  }
}

/* ------ program definitions ------------------------------------------*/
```

```
/* structure for storing binary tree of words and hashes */
typedef struct
{
  char word[32];       /* word */
  int  hash;           /* hash of word */
  int  left;           /* binary tree left (less than) */
  int  right;          /* binary tree right (greater than or equal) */
} WordRecord;

/* structure for storing hash value/collision count pairs */
typedef struct
{
  int hash;            /* hashed value */
  int collisions;      /* number of collisions */
} CollisionCounter;

/* simple enum type used to store which direction a branch should be added
 * in
 */
typedef enum _branchdir
{
  BranchUnknown = 0,
  BranchRight,
  BranchLeft
} branchdir;

/* wordlist entries are allocated in blocks on WORDLIST_GRANULARITY, this
 * speeds up operation by reducing the number of mallocs/reallocs needed.
 */
#define WORDLIST_GRANULARITY 64

/* SPREADRES defines the number of "buckets" to divide the 32 bit field
 * into for the spread calculations.
 */
#define SPREADRES            128

WordRecord *wordlist=NULL;
int wordlistnext;
int wordsdone=0;

/* ------ main program functions ---------------------------------------*/

static void AddWord(char *word)
{
  int wordhash=0;
  int newbranchroot=0;
  branchdir newbranchdir=BranchUnknown;
  char *ptr;

  stripcr(word);                    /* dump any nl */
  strlower(word);                   /* lowercase the word */

  wordhash=hash(word,strlen(word)); /* calculate hash of word */

  /* first check any existing list entry, or find where our new entry
   * should link from */

  if(wordlistnext>0)
  {
    int index=0;

    while(index!=wordlistnext)
    {
      /* initially compare hashes */
      if(wordhash==wordlist[index].hash)
      {
        if(!strcmp(wordlist[index].word,word))
        {
          /* found it -- ignore (ie of course we'll get the same HASH for
           * the same string!  */
          return;
        } else {
```

```
          //printf("%s and %s have the same " HASH " (0x%x)!\n",
          //       wordlist[index].word,word,wordlist[index].hash);
        }
      }

      /* follow binary tree, if we reach an end (left/right == 0) then
       * set newbranchroot to the current index, and newbranchdir to the
       * direction which is appropriate.
       */

      if(wordhash<wordlist[index].hash)
      {
        if(wordlist[index].left)
          index=wordlist[index].left;
        else {
          newbranchroot=index;
          newbranchdir=BranchLeft;
          break;
        }
      } else {
        if (wordlist[index].right)
          index=wordlist[index].right;
        else {
          newbranchroot=index;
          newbranchdir=BranchRight;
          break;
        }
      }
    }
  }

  if((wordlistnext % WORDLIST_GRANULARITY) == 0)
  {
    /* we need a new block of memory */
    if(wordlist)
    {

//printf("realloc(%x,%x)\n",wordlist,(wordlistnext+WORDLIST_GRANULARITY)*size
of(WordRecord));
      wordlist=realloc(wordlist,

(wordlistnext+WORDLIST_GRANULARITY)*sizeof(WordRecord));
    } else {
      //printf("malloc(%x)\n",sizeof(WordRecord)*WORDLIST_GRANULARITY);
      wordlist=malloc(sizeof(WordRecord)*WORDLIST_GRANULARITY);
      wordlistnext=0;
    }
  }

  if(!wordlist)
  {
    fprintf(stderr,"AddWord: Failed to allocate/extend word list\n");
    return;
  }

  /* we have memory allocated */
  strcpy(wordlist[wordlistnext].word,word);
  wordlist[wordlistnext].hash=wordhash;
  wordlist[wordlistnext].left=0;
  wordlist[wordlistnext].right=0;

  /* add the new branch in to the tree */
  if(newbranchdir!=BranchUnknown) {
    if(newbranchdir==BranchLeft)
      wordlist[newbranchroot].left=wordlistnext;
    if(newbranchdir==BranchRight)
      wordlist[newbranchroot].right=wordlistnext;
  }
  ++wordlistnext;
  ++wordsdone;
}
```

```
/* calculate the spread of the hashes to determine if they occur evenly over
the
 * 32 bit hash-key space
 */
void CalculateSpread()
{
  int index=0;
  unsigned int spreadsamples[SPREADRES]={0};

  /* add them up */
  while(index!=wordlistnext)
  {
    ++spreadsamples[wordlist[index].hash / (0xffffffff/SPREADRES)];
    ++index;
  }

  /* throw out a table */
  index=0;

  printf("%8s %12s\n","Bucket","Frequency");

  while(index!=SPREADRES)
  {
    printf("%8i %12i\n",index,spreadsamples[index]);
    ++index;
  }
}

/* scan the binary tree counting the number of occurences of each unique hash
*/
void CalculateCollisions()
{
  CollisionCounter *clist;
  int index=0,cindex=0;
  int lastcollision=0, hashcollisions=0;

  clist=malloc(sizeof(CollisionCounter)*wordsdone);

  if(!clist)
  {
    fprintf(stderr,"CalculateCollisions: Failed to allocate collision
list\n");
    return;
  }

  /* build a list of hash keys, and count up how many strings had that hash
   * allocated to them
   */

  while(index!=wordlistnext)
  {
    cindex=0;

    /* try to find an existing entry for this hash key */
    while(cindex!=lastcollision)
    {
      /* if there is a matching entry increment collision count */
      if(wordlist[index].hash == clist[cindex].hash)
      {
        ++clist[cindex].collisions;
        break;
      }

      ++cindex;
    }

    /* we didn't find an existing entry for this hash, make a new one */
    if(wordlist[index].hash != clist[cindex].hash)
    {
      clist[lastcollision].hash=wordlist[index].hash;
      clist[lastcollision].collisions=0;
      ++lastcollision;
```

```
    }

    ++index;
  }

  index=0;

  /* total the collisions */
  while(index!=lastcollision)
  {
    hashcollisions+=clist[index].collisions;
    ++index;
  }

#ifdef DEBUG

  /* this code will list duplicates along with their index and left/right
   * pointers for debugging
   */

  cindex=0;

  while(cindex!=lastcollision)
  {
    if(clist[cindex].collisions>0)
    {
      index=0;

      printf("0x%x = ",clist[cindex].hash);

      while(index!=wordlistnext)
      {
        if(wordlist[index].hash == clist[cindex].hash)
        {
          printf(" %s (%i, %i, %i) ",
                 wordlist[index].word,index,wordlist[index].left,
                 wordlist[index].right);
        }
        ++index;
      }

      printf("\n");
    }
    ++cindex;
  }

#endif

  free(clist);

  printf("%i collisions out of %i unique
strings.\n",hashcollisions,wordsdone);
}

/* the main program */
int main(void)
{
  FILE *dict;
  char inbuffer[255];

  dict=fopen("/usr/dict/words","r");

  if(!dict)
  {
    fprintf(stderr,"Failed to open dictionary\n");
    exit(1);
  }

  while(!feof(dict))
  {
    fgets(inbuffer,255,dict);
    AddWord(inbuffer);
```

```
  }

  printf("Testing " HASH " method...\n");
  printf("Calculating spread...\n");
  CalculateSpread();
  printf("Counting hash-collisions...\n");
  CalculateCollisions();

  fclose(dict);
}
```