Andrew Regan

Intelligent indexer for web presentations

MSc in Computer Science (Conversion)
1997/8

10,500 words

Contents

Chapter 1

Introduction

1.1 Motivation of the project

A common way for users to come to terms with a modern hypertext presentation, comprising a large and complex series of electronic documents, is for designers to produce an index. This would be supplied with links to the relevant sections within the original documents in order that the user can navigate around the presentation quickly in order to find the information they are looking for.

The index should therefore become a useful part of the hypertext presentation from which it was created. The index's contents are most important, as it is vital to make as accurate a representation of the original presentation as possible. The index must also be as useful as possible, because unless the user regards it as a helpful way of navigating the presentation it will not be used. The index should clearly be a different *kind* of document, but this will be covered later.

The objective of this project is to research the current literature on indexing in order to be able to assess the best way of developing a system that can produce an index for web presentations that handles the two major issues as well as possible. It should first pick out the most useful index terms, then output them as an HTML file in as useful a format as possible.

1.2 Overview of the project

I will firstly make a survey of the relevant literature in order to show how previous investigations have approached these two issues, and then, on the basis of this information, I will set out the approach I will take for my own program. After a brief account of the development of the program from a software engineering point of view, I will then cover in more detail the functionality of the program I have written. At each stage I will relate this to the approaches taken in the literature and justify the choices I myself made. Throughout this dissertation I will assess each stage of the development of the program, drawing conclusions as to their success or failure, and try to point out areas for further research where I have been unable to solve issues myself.

Chapter 2

Analysis

2.1 Literature survey

Salton refers to the compromise that must be made between recall and precision. High recall means a large number of words and phrases being indexed in a document, and high precision means a high proportion of these terms being relevant to the user's satisfaction. It seems that the only way to guarantee 100 per cent precision is to allow the user to specify a Keyword list - a list of words which are the only acceptable index entries. This is equivalent to the index-encoding schemes used by word-processors, where non-printing codes are inserted into the document in front of word that should be indexed, or surrounding the word.

The problem is that no such indexing scheme exists for the HTML page-description language, and there is no easy way of inserting indexing codes. Although HTML supports invisible characters with, for instance, "comments", there is no agreement as to how this would, in practice, be used for index encoding. It would also be cumbersome to add them, especially to large documents.

As Berk and Devlin report, the entries for the index of a large presentation would traditionally have been chosen by 'subject experts', but as the size and scope of presentations grow, this is becoming increasingly expensive, as it requires a considerable effort on the part of the document designers. As a result, "Automatic indexing has been studied as a way to create compact keyword descriptions of documents stored and accessed in large text retrieval systems" [1]. There would appear to be little disagreement over the aims of automatic indexing. According to Park, et al, "Automatic indexing leads to a form of document representation that visualises the content of the document more explicitly" [2] .

Automatic indexing produces an index for a document supplied by the user with no preparation or prior knowledge of the document's content. It must rely entirely on grammatical and semantic rules to try to determine the context and relevance of words within the document. It cannot guarantee 100 per cent precision, but I feel it can do extremely well if properly applied. This approach is far more interesting, and deserving of research than the encoded-index strategy.

There are many different ways of doing it; almost of which determine word importance on the basis of the frequency with which that word occurs. The program would record each different word found in the web presentation together with the number of times that word occurred. The most frequent words are considered to be of the least usefulness; since they

[1] Berk & Devlin p.119

[2] Park, et al., p.87

occur most often each instance presumably adds little to the document. The least frequent words are also considered less important since they are too rare to have a significant impact on the meaning of the presentation. Both the most frequent and least frequent words are removed from the index leaving the MFWs (Middle Frequency Words[3]). Only words from this category can appear in the index.

The problem I have with these methods is the assumptions that only words which are of middle frequency within a document collection are of value. If a word is used very extensively in one document it may find itself excluded for having a high frequency despite the fact that it appears in a highly specialised document which deals solely with the topic referred to by that word. On the other hand, one word might appear only once but be of great value - book indices are full of entries that appear only once. As Berk and Devlin put it, "There are simply too many useless words that occur with intermediate frequencies and there are too many useful words that occur infrequently". [4]

Surely what makes a word or phrase valuable in the index is its unexpectedness; if it would stand out for a typical individual it should appear irrespective of its frequency. Frequency therefore seems a very inadequate way of determining context.

An alternative strategy is as follows: according to Salton[5], a well-designed system would take all the words in the presentation, remove those that appear on a 'blacklist', award frequency-related weights to those that remain, then index those words with the highest weights.

2.1.1 A criticism of the literature

In the literature on automatic indexing, despite the wide variety of strategies on offer, the dominant idea is that what will appear in the index is individual words, albeit grouped into categories. However, as any index in a printed book will show, by no means all entries in the index are single words. After all, a reference to an individual would need to store all his names in order to be useful: his or her prefix, first name, initials, surname, and possible suffixes as well. Furthermore, it would be ridiculous to split up "World War Two" into three words and add a page reference for the word "World", yet the literature is dominated by this idea.

2.2 An alternative strategy

It is an important prerequisite of my program's indexing strategy that it is able to determine that the three individual words belong so closely together. The indexing strategy I will implement does *not* index terms on the basis of their frequency or the weights calculated for them, though it does employ weights at a later stage in order to provide more information for the user.

[3] Smith p.20

[4] Berk & Devlin, p.120

[5] Salton (1989) p.281

Instead the algorithm will look for runs of words that begin with a capital letter. What happens will be as follows. Words are determined first, by building a list of characters and stopping when a word break is found. When a word has been built in this way, if its first letter is a capital letter, then that word is joined to the end of the current phrase. This process continues until a word is found that doesn't start with a capital letter. At this point, the current phrase is finished, and is then considered for indexing, and the current word is ignored. The next time a word is found starting with a capital, a new phrase is started, and the word placed at the start of it. The program does not distinguish between words at the beginning of sentences and those within sentences, as I do not believe there is any grounds for this. The process is slightly more complicated than this, but the result is that phrases are built up, which may consist of one word, though often more.

My feeling is that when a word that begins with a capital letter follows another word that begins with a capital letter, it is highly likely that they are related, and that to split them into individual words would be a mistake. In the vast majority of cases this should work very effectively. For example, given these inputs:

"World War Two" would produce one entry for **"World War Two"**
"World war Two" would produce entries for both **"World"** and **"Two"**
"World<newline>**War**<newline>**two"** would produce one entry for **"World War"**

This is the basis on which we decide whether terms are indexable, and it is how I intend to offer to the user an index that is representative of the original presentation.

In some cases we will find documents with words and phrases that are deserving of inclusion but which, because they do not start with a capital letter, will not be indexed according to the algorithm. But without the capital letter what information does the program have to determine that the word might be important?

One possible rule we can use is that words surrounded by single or double quotes may be important since they may show that the author is 'coining a phrase', for example:

He described his agenda as one of "consociationalism" ...

The term "consociationalism" may not be deemed by the document author to merit a capital C, grammatically, but the presence of the quotes suggests that its information content might be sufficiently high for it to merit an appearance in the index. Many other things can occur within quotes but speech should already begin with a capital letter so our rule doesn't change the situation in such a case. So our rule will be that if a phrase is found in quotes, then even if the first word doesn't start with a capital, it will be considered for indexing. If the user doesn't agree that this is sufficiently worthwhile the rule can be deactivated.

One situation where our main algorithm may fail to produce the best results is as follows. Consider the following example:

"During the Falklands Smith was a soldier"

The algorithm would suggest "Falklands Smith" as a possible index entry. Unfortunately, it would be in error as the phrase is not a genuine one, but since the program cannot currently understand the meaning of the words within the phrase, this problem cannot be avoided.

A problem would arise if a document in the presentation contained only uppercase or only lowercase characters. In the former case, the entire document would be indexed as one entry, and in the latter case, no entries would be found at all. These are potentially serious problems, but they are extremely unlikely, especially so as the program was designed to work on a web 'presentation', suggesting something of publishable quality. In such a situation it is unlikely that a web page designer would have chosen to work solely in uppercase or lowercase.

2.3 A critique of automatic indexing

However, the American Society of Indexers have attacked the concept of automated indexing approach or, at least, in its current state of development. The section above has mainly concentrated on determining which words and phrases are suitable for indexing, (builds a concordance, or a word list) but according to the ASI this list of words is not sufficient to be called an index:

> "Although the manufacturers often claim these packages build indexes, the actual results are a list of words and phrases, sometimes useful in the beginning stages of building an index. Usability tests of these packages have shown that the word lists omit many key ideas and phrases, and cannot fine-tune terminology for easy retrieval, or build the needed hierarchies of ideas that professional indexing can." [6]

What this shows is how important it is to process the list of phrases we have obtained, and the importance of making it a useful guide to the user.

A major criticism of the available literature is that I have been able to find very little information about the connection between index creation and HTML/Hypertext, which is, after all, the major concern of this project. The literature has tended to fall into one of the two camps, concentrating either on complex text retrieval algorithms (e.g. Salton), or on the often-psychological theories behind hypertext and user interfaces (e.g. Nielsen). It is my hope that this project can go some way towards bridging this gap. The Forrester article[7] is the only one I have encountered that seems to have noticed the connection.

2.4 "Usefulness"

What determines utility is not immediately obvious. Firstly, an index is useful if it contains terms that are representative of the web presentation. However, not all terms in the index are of equal value to the user, and a useful index is one that indicates to the user which are the most important terms within the index. How is this to be done?

[6] In <http://www.asindexing.org/software.htm>

[7] Forrester, p.249

2.4.1 HTML Importance

In the User Requirement (see the Appendix) for this project it was stated that tags such as <EM> around an index entry should give the entry added prominence. The program maintains a list of important HTML tags that influence the importance that should be assigned to the index entries in the output file. At present, the tags are as follows: <STRONG>, <EM>, <B>, <U>, and <TITLE>.

When the parser iterates through the characters of the document, in addition to building up phrases for indexing, it also builds up HTML tags, and if one is found that appears in the list, then a flag is set for the relevant characteristic. For example, when <B> is found, the bold flag is set, and for so long as this flag is set, any phrase added to the index will have stored in its page reference the fact that it appeared in bold in the document, and should therefore be considered of greater importance as a result. The flag is reset when the tag is closed with a </B>. A very similar arrangement exists for the header tags <H1> to <H6>, and words that appear within HTML links are also treated with a higher priority.

Because of the huge number of combinations of the different flags I thought it better to devise some kind of formula to reflect what I perceived to be the importance web page designers would attach to each HTML code, and to arrive at a number between 0 and 7 that could be used as the overall importance value - the **HTML Importance** of each word and phrase in the index.

The calculation is as follows (in sequence):

* the importance is initialised to 0.
* if the **emphasis** flag is on, the importance is set to 1.
* if the **strong** flag is on, the importance is set to 2.
* if the **title** flag is on, the importance is set to 5.
* if the **isLink** flag is on, the importance increases by 4.
* if one of the **header** flags is on, eg. <H3>, the importance is set to 8 minus the header level.
* if the **bold** flag is on, the importance increases by 1.
* if the **underline** flag is on, the importance increases by 1.
* finally, the importance is adjusted so it is between 0 and 7.

In order to reflect these values in the output file I have allocated a combination of HTML styles for each of the importance levels. It makes sense to give the higher importance levels a more prominent style than the least important levels. For example: while level 0 should be displayed using a small font (size is one less than the standard size) and in plain, black text, level 7 is displayed in bold, red letters, using a larger font (size is one more than the standard size). It will also be marked by a picture reference (by default, a small GIF picture of a warning triangle).

As a brief aside, in addition to being able to output HTML files, the program can also show the same information in plain text format. The main advantage is that smaller files are produced, however, they are without the formatting information and are therefore of less use to an end-user.

2.4.2 Use of Colour

The program makes use of colour in its HTML output files in order to make them easier for the user to understand, more revealing, as well as more attractive.

According to Smyth[8] , one of the important principles for colour usage is "The principle of figure/background, where dark or dim colours are used for backgrounds, with bright colours used in the foreground." The program generates an HTML table, with two columns: one for index entries, the other for page references. To colourise these in such a way that they were both unobtrusive, in order not to be confused with foreground items, but distinct from one another, led me to choose two shades of grey, a medium one for the entries, and a light one for the references. Text would be formatted according to the styles given above. Black was used for importance levels below 3 because it implied "normalcy" and therefore references that were the least important. Red was reserved for values 6 and above because it implied "urgency". Meanwhile importance levels 3, 4 and 5 were coloured green, because legibility factors really left us only with blue and green, and blue page references could be confused with document links - on the same line - which are generally blue themselves. Blue was, however, safe to use for letter headings.

2.4.3 Index Weighting

Index weighting is a measure of importance more rooted in textual analysis than in HTML. The first weighting algorithm we use is that of Sparck Jones[9] , which gives weight w_{ij} for term j in document i as follows:

$$w_{ij} = tf_{ij} / \Sigma tf_{ij}$$

Where tf_{ij} is the frequency of term j in document i. Thus, the weights for the documents that contain any one term are always from 0-1, and must add up to 1 for each term, since $\Sigma(x/\Sigma x)$ must equal unity. If the term "Chirac" appeared in two documents, and the first was given a weight of 0.7 this would mean that 70% of the references to the term appeared in that document, therefore it is more important.

The second algorithm, by Salton[10] , gives the weight w_{ij} for term j in document i as follows:

$$w_{ij} = tf_{ij} \cdot \log (N / df_j)$$

Again tf_{ij} is the frequency of term j in document i, df_j is the number of documents in which term j occurs, and N is the total number of documents in the collection. The weight is the

[8] <http://helix.infm.ulst.ac.uk/~smyth/colour.htm>
[9] In Smith, p.20
[10] Salton (1989) p.280

product of the term frequency and the "inverse document frequency". Here, weights for the documents that contain the term increase as number of matches increase - unlike with the previous formula - though any term that appears in all docs in the presentation is weighted as zero for each occurrence. The higher the weight the greater the importance of the term in that document.

Weights are *only* calculated when the presentation contains more than one document. Since both formulae try to reflect differences in the frequencies of each term between the documents the term appears in, they have no value with only one document.

2.5 Indexing concepts

2.5.1 Keywords

There are two stages these words and phrases go through before we know they are suitable for indexing, both of which are voluntary - depending on how strict or lenient an index is desired. Stage One involves the Keyword list mentioned earlier.

If the user specifies a valid Keyword list when the program starts, then whenever the program identifies a word or phrase, it will only be indexed if it also appears within the Keyword list. One advantage of this approach is that it guarantees precision, since the user chooses the index terms he wants as Keywords, the results must be Keywords too - 100 per cent precision. It also allows the user to, say, find all references to "Britain" in the web presentation by passing the program a list with a single Keyword. The disadvantage is that the indexing process becomes rather less 'automatic' and the user must go to the trouble of creating the list of Keywords.

If we call P the set of possible index terms, and W the set of terms in the word list file, then those terms that pass the first stage is given by the intersection of the two sets: **P union W**. If the user does not specify a list, then all the "possible" are deemed eligible for indexing, and the result is the set **P**.

2.5.2 Suffix stripping

Suffix-stripping is a method I have tried to implement in order to clean up possible index entries, as recommended earlier by Salton. Also known as morpheme stripping, it is a mechanism whereby a family of words can be reduced to a common stem. For example, "producers", production", and "productivity" can be reduced to the stem they share, namely "produc". The advantage of this is that we may not want separate index entries for all the members of such a family - we may want them grouped under one heading. So, for example, entries for "Trade Union" and "Trade Unions" would be grouped together.

Salton says that index entries should all be stem words. Procedures do exist that can convert words into their stem, and I endeavoured to find such a procedure to incorporate it into the program. However, a number of problems arose. Having gone to the trouble of converting a complex series of routines from C into Java, I discovered that the procedure was incomplete to the point of unusability. Far more rules and suffixes would be required, and the program failed on all the examples of correct behaviour that were listed in Salton. There were 77 rules of the form:

```
new Rule( 218, "iveness", "ive", 6, 2, 0, 0);
new Rule( 420, "ive", "", 2, -1, 1, 0);
```

This one would replace an "iveness" with an "ive" one, reducing "Compulsiveness" to "Compulsive", and then to its stem "Compuls". Unfortunately, trying to complete the algorithm myself would have been far too large a task given my deadline. Salton provides 30 suffixes my routine lacks[11] , and that is but a subset of the whole.

Another problem is that suffix stripping is only appropriate for certain classes of word, and nouns are not such a class. For example: one common rule is that "ing" can be stripped from the end of words, so that "Looking" becomes "Look". However, the London Borough of "Havering" would become "Haver" which is nonsense. Without a system whereby the class of a word can be discerned - such as a thesaurus - the majority of nouns might be corrupted in this way. This issue will be covered in greater depth later.

The consequence is that only the most limited suffix stripping can be implemented, namely the removal of apostrophes.

2.5.3 Stop Words

Stage Two of the index improving process involves taking the list of possible phrases, and removing those with unsuitable word prefixes and, more significantly, removing unsuitable "noise" words. These "noise" words, also known as stop words, are words that will frequently be present in the possible phrase list but which are of no value in an index. Words such as "Because" or "Any" or "Thereafter" frequently start sentences and may therefore be classed as possibilities as they start with a capital letter. However, they have no value in an index, and should be removed from phrases. Consider the following sentence:

"Thereafter Smith was no more - Smith was dead"

Stop words are *extremely* important, because they avoid clogging up the index with entries for words and phrases of limited usefulness, and because they help us identify genuine index entries. Ordinarily the parsing algorithm would propose "Thereafter Smith" as a phrase, and would have it indexed, along with the other entry for "Smith". However, by stripping the stop words from the first phrase, we would have a more reasonable result, namely, two entries under "Smith".

[11] Salton (1989) Table 11.1, p.381

2.5.4 Bad Prefixes

Some words should not count as stop words but may still be unsuitable at the start of a phrase, such as, for instance, the word "Two". "World War Two" seems a reasonable phrase for indexing, so it follows that "Two" should not be in the Stop Word list, however, it makes far less sense as the start of a phrase. A small number of other prefixes are stripped too, including the numbers from one to ten (a reasonable upper limit). The word "Of" is another entry in the list of unsuitable prefixes. On its own it has no value in an index, but I decided that it shouldn't be in the Stop Word list because of its significance in separating other words.

2.6 Programming language and developing environment

An early decision was that the program would be written using the Java language. Firstly, Java offered a flexible range of input/output functions, and its networking classes promised to make the reading of web pages as easy as possible. The specification also seemed to require an object-based approach, so C++ was a possibility, but the useful String and Vector classes offered by Java made it the most obvious choice. I already had some experience with Java and extensive experience with the chosen development package - Metrowerks CodeWarrior Pro 3.1, using Java 1.1.5. All of the development work was carried out on my MacOS computer, but testing was also carried out on the University's public Windows NT PCs as well as on my project supervisor's UNIX system.

In accordance with good software engineering principles the program's source files have been extensively commented to improve legibility. Additions were made to the program with constant reference to the user's requirement, which was short and unambiguous. Meetings with the project supervisor and demonstrations of each stage of the program's developments ensured the project kept on track. As every feature was added the program was tested on a variety of different HTML documents to ensure the program worked, and some of the results appear later.

2.6.1 Program requirements

The only requirement is that the program operates on a system supporting Java 1.1 and later. The program should function identically on all such systems.

Few assumptions are made about the files within the presentation. Files must have .htm or .html suffixes to be indexed, so images and compressed files are excluded. It is assumed that all characters are 7-bit ones, that is, with ASCII values less than 128, and that foreign characters are referred to with valid HTML codes, such as &eacute. Provided the file satisfies the rules of HTML it is acceptable to the program.

2.7 Problems faced:

2.7.1 Spelling mistakes

Spelling mistakes in the web presentations pose a problem since the misspelling of index entries may result in entries for both correctly and incorrectly spelled versions, so we might face entries for “Goering”, “Goring”, and “Göring” in the same presentation. This is a situation where a link-up with a spell-checker might be valuable, but this issue will be covered later.

2.7.2 Special characters

In order to handle different character sets effectively, HTML defines many codes to represent non-English characters. So, for instance, the code **&uuml;** is used in place of an umlaut character in an HTML document. It is vital that this code is identified by the program for what it is and that the program does not attempt to index it. The program needs the single umlaut character to test for word breaks, and also for the plain text version of the output file, while the HTML code is used with the HTML output file.

To identify these special character codes, a comprehensive list is required. I discover what appears to be a full list[12], which has allowed my program to identify special characters flawlessly.

2.7.3 File names

When the program is given a file to open, before any such attempts are made, the file name is carefully adjusted and checked to ensure that it is a viable prospect.

Suppose the following HTML link was found:

<A HREF=""""The%20Communist%20Manifesto.html#Revolutionary_Praxis"">

Extracting the string inside the tag would give us:

“”The%20Communist%20Manifesto.html#Revolutionary_Praxis”

However, though the intended file may well exist, Java could not open it with this filename, so the following adjustments are made to any links that are located: any surrounding quotes are removed, instances of “%20” are replaced with space characters, and any anchors are removed because they are not really part of the filename. The filename we would then pass to Java routines for opening is thus ”The Communist Manifesto.html”.

Another type of HTML link is the anchor, or the internal link, where clicking on a link takes the user to a different part, usually within the same document. While the name of an external link would be treated as a filename to be opened, with this type of link there is no filename. Instead, the program will attempt to index the name of the link. For example:

[12] Etcode in *HTMLParser.java*

<A NAME=”Copyright Information”></A>
....
<A HREF=”#Copyright Information”></A>

The program would identify the “<A NAME=” prefix, and index the phrase “Copyright Information”.

2.7.4 Local and remote files

Documents within the presentation may reside on a local computer, or they may be on some other computer and only accessible through a network. As a consequence when encountering a filename the program must determine whether or not the file is remote because if so, it must be accessed differently - through a Uniform Resource Locator, or URL. If the program locates a link to a remote file and the “Use remote?” option is on, the program will attempt to connect to the relevant server and read the file before disconnecting. To prevent indexing the entire network, the user can specify the maximum depth of the hierarchy in order to save online time.

If the user has turned off remote mode, all filenames are interpreted as local filenames. So, even if a previous document contained a link to http://www.ukc.ac.uk/departments.html the program will look on the local computer for it (and undoubtedly fail).

When remote mode is on, the program will initially interpret any filename as a URL, and attempt to create a Java URL object for it. If this succeeds then the filename is a genuine absolute URL (like the last example) and the program will try to access it.

If the last test failed, there are two remaining possibilities: that the filename is a relative URL - such as /history_department/index.html - or it is a local file, but which? In order to determine this, the program attempts to determine the absolute URL by combining the valid host name of the document that originally referred to the current document (if any) with our filename, and create a Java URL object for it. So, for example:

If the parent document is www.apple.com/doc.html and the filename is “main.htm”, we will try to connect to: www.apple.com/main.htm. If this succeeds we have successfully determined the absolute URL from the filename and we can read from this file.

If this, too, failed, then we cannot derive any URL from the filename, and we must assume it is a local file. However, this is not the end of the story. For example: suppose we have two directories on a local computer, named “A” and “B”, with a web document called “cat” in each, plus one other one. If a document in directory “B” contains a link to a file named simply “cat”, although the file’s directory is not specified in the link, we should assume that it refers to the file in directory “B” because “B” is the directory of the file that called “cat”.

What the program will do is to try to locate a file called “cat” in the current directory first, assuming it is an absolute filename. If this should fail - which it will in our example - the program then assumes it is a ‘relative’ filename, as in our example, and then try to open the file in directory “B”. If this fails, the program is forced to give up on opening the filename.

2.7.5 Index data structure

Each document that is analysed by the program is associated with a SubIndexTable object which contains the list of entries and page references for that document. There exists another object, called a CompleteIndexTable - the main index for the program. As each document's SubIndexTable is completed, it is merged with the CompleteIndexTable. This is the index that is finally written out at the end of the indexing process.

Each index table contains a list of indexed phrases, stored in parallel with a list of page references for each.

2.7.6 Page numbers of index entries

Examination of the literature on HTML revealed an unexpected problem. This is that HTML documents have no concept of a "page". In browsers, each document simply scrolls by continuously. How much is visible to the user at any time is dependent upon the size of the window, and how much can be printed onto a piece of paper depends upon the size of the piece of paper, the page size chosen by the user, and the "Text Size" preference that allows browsers to scale up or down the size of the document in the user's window. Since HTML treats documents and document elements as a single "worksheet", page numbers are irrelevant. However, because the program must give the user a clue as to where the entry occurs, an alternative option was required.

2.7.7 Page references

Whenever a word or phrase is added to the index a new PageReference object is created to represent the instance of that word or phrase within the presentation. Each word in the index may have numerous references attached to it. The PageReference stores the name of the web document in which the word appears. It also stores the position in the web document of the entry. This cannot, however, the actual number of the "page" on which the entry appears, as has been covered. Instead we use the position of the first character of the word within the document. So, if the word is the first word in the document it will be position zero; if there is 1000 bytes of data before it, its position will be 1001.

Also stored are the HTML Importance, covered earlier, of the reference, as well as the level of the document in the presentation hierarchy. So, for instance, the first document in the presentation will be at the top level - level one. A document referred to within it will be level two, and documents referred to within that will be level three. Index terms found higher up should be differentiated from those found lower because, as suggested in the project specification, they are more likely to be associated with generality rather than detail.

The reference also remembers whether or not it was found in the title of a web page, and is therefore more important.

2.7.8 Sorting page references

Each page reference stores more than just a page number. So when it comes to sorting the page references for a given index term, it is important that it is done not only in way the user would expect, but also in a way that reflects their importance.

The sorting takes place in three stages. Firstly, those entries that are highest in the web hierarchy appear before those which are lower down. The next stage is that those entries that appear in documents whose name is early in the alphabet are sorted to be before those whose name appears later. The final sort stage places those entries that appear earliest in the document before those that appear later, so that, in effect, the page number is the last criteria used.

Consider the following example for the index entry "Gestation Period":

Order:	**Hierarchy:**	**Filename:**	**Position in file:**
	High up	"**Bear**"	Early in file
2	High up	"**Yak**"	Early in file
3	In the middle	"**Mouse**"	Early in file
4	In the middle	"**Mouse**"	Late in file
5	In the middle	"**Zebra**"	Early in file
6	Low down	"**Aardvark**"	Middle of file

To describe the process, (1) and (2) beat the rest as they are higher up the hierarchy, but (1) beats (2) to the top as its name is earlier in the alphabet. (2) is second. (3), (4), and (5) occur next as they are above (6) in the hierarchy; looking at the filename cannot distinguish (3) and (4) but (3) goes above because the entry appears earlier in the file 'Mouse'. (5) is next, followed by (6).

2.7.9 Displaying proper names

In general the program will find proper names in the following form: "Mr John Smith". However, duplicating this in the index would not be useful for the user, as all Professors would be sorted under the letter P rather than by their surname. Benson and Dunn outline[13] seven common problems for automatic indexers, and this is one of them. The way I plan to solve this in my program is by maintaining a list of acceptable name prefixes, such as "Doctor", "Captain", and "Miss" such that if we have a name with the following format: **Prefix Forename_List Surname** it will be replaced in the index by **Surname, Prefix Forename_List**. So, "Prime Minister Tony X Blair" would appear as "Blair, Prime Minister Tony X" which is what a book index would show.

[13] Benson and Dunn, p.93

According to the article, “Margaret Atwood” should appear as “Atwood, Margaret”, and I can extend the above rule by storing forenames too within the program - as many as possible, given constraints of time and space, such that **Forename_List Surname** is replaced with **Surname, Forename_List**. Only the most common names are stored at present but the list can be extended and most English names should be adjusted successfully.

Chapter 3

Implementation

3.1 Initialisation

The flowcharts in Figure 1 and 2 summarise the way the program works.

The Stop Words used by the program are stored in a Stop Word list file which the user can specify. This is a file with one word on each line which is read by the program when it is started, which stores all the words in a hash table. Lines which are blank or which start with C++/Java comment characters ("//") are ignored. The standard Stop Word list supplied with the program contains over 400 common or 'irrelevant' words. If the user has supplied a Key Word file, this is indexed in advance, as the program initialises as if it were part of the presentation.

3.2 Reading the presentation

The program starts by indexing the main document of the web presentation which is specified by the user when the program starts, and then indexes each of the documents referred to within the web presentation in turn.

A preliminary step is to adjust the filename as detailed in section 2.7.3, and the program then tries to access the file according to the rules set out in 2.7.4. If the program fails to access from it, the program will look through the current document for the next link, and for other documents to open. Otherwise, the file's data is read and stored by the program, with newline characters and any excess tab or space characters being removed - which would appear to be something all web browsers do. Once this is done, the program declares the reading and storage of the file a success.

3.3 Analysis of each document

Once a document has been read successfully, the process of analysis can begin. The program must now transform the array of characters it has read into indexable phrases. Starting from the first character of the data, each character is processed until none are left. The process is much more complex than this as characters can be part of an HTML tag, they can be the argument within such a tag, they can be part of a special character, or they can be an 'ordinary' character. Section 2.2 discusses how words and phrases are formed from these characters.

Once a valid HTML tag has been located by the program, the program must respond to it, depending on what type of flag it is. Perhaps the most important type is the link tag, which allows us to access the web documents that are linked together to make the web presentation.

3.4 Moving to the next document

When one web document contains a reference to another page, this next document starts to be processed straightaway (if it has not already been done) and when this is completed control returns to the web document that called it. As the presentation will represent a Tree structure, this a recursive process. Parsing should continue after the reference, not back at the start of the document. This is achieved by storing the current analysis position, passing control to the new document, then when this is complete, continuing at the saved position.

A list of all the pages that have been analysed is maintained by the program, as is the current hierarchy of documents so that control can be passed back to a parent document once the child has been dealt with. This hierarchy is built up as the program moves from one document to another, and only when all entries have been dealt with, and deleted from the list, has the analysis stage finished.

For example, if the HTML tag <A HREF="http://www.ndirect.co.uk/bio/bio.htm"> was found within a document, the program would identify the "<A HREF=" prefix, extracting the string "http://www.ndirect.co.uk/bio/bio.htm", which is stored. The program will try to open the file referred to by the link.

3.5 Processing possible index entries

Once a word of phrase has been found within a document, it is considered for indexing. If the user has specified a valid Keyword list then whenever the program identifies a word or phrase, it will only be indexed if it also appears in the Key Word list. How this works in practice is that phrases are only added to the index if there is already a (dummy) entry for that phrase (produced when the Key Word file was indexed). Any remaining dummy entries are removed once the indexing is completed.

A number of further rules are applied to remove discrepancies and stop unsuitable phrases being indexed. One rule is that numbers are not valid index entries, in fact, any phrase that starts or ends with a digit is not allowed in the index. Suffix-stripping takes place next; where a word ends with either 'apostrophe S' or 'apostrophe D', these two letters can be safely stripped off. Thus "Regan" and "Regan's" would share an index entry. Next, if the word or phrase appears in the Stop Word list, it will not be added to that document's index.

A further rule, which I felt would add some much-needed strictness to the index, allows the user to choose whether or not to exclude terms that end in either "...ing" or "...ally". This is not strictly suffix-stripping, it is actually the selective removal of entries based upon their suffix. However, I felt on the basis of sample results that "ing" suffixes tended to be Present tense verb forms, and "ally" words were almost always adverbs. In a large sample document with 2005 indexed terms, 58 fell into these categories. Removing them would allow the program to keep out terms such as "Approaching" and "Naturally". Unfortunately, due perhaps to the preponderance of proper names in the index, a number of these were lost too. In fact, of the 58 terms removed, 8 were the names of individuals, for example, "Lieutenant-General Sir Frederick Browning", and another two lost were "Viking" and "Anglo-Viking". Clearly, there are advantages from keeping out unnecessary words, but the risks from losing proper names may outweigh them. For this reason, the rule is optional.

We also assume that words less than two characters in length are always irrelevant, and the only two-letter words that we will accept are ones in capitals, since these may be company initials, such as "BT". All words greater than two characters (and which pass the other tests!) are deemed to be at least possibilities.

Now all the tests have been carried out, many of the words and phrases the parser thought were possible index entries have been eliminated or modified. Those that remain can be added to an index.

3.6 Building the index

If a new word is added to the index, an empty list for the storage of page references is created, and the word's reference is added to it. If the word already appears, there is already a list of page references, so a new one is created for the entry and added to the end of the list. See section 2.7.5 for the format of the page reference data structure. The fact that it is added to the end of the list of references is not a limitation of the program, since the list will be carefully and usefully sorted before the word and its references are output.

3.7 Producing the output

This section corresponds to the **Indexing Completion** section of the flowchart. Once the index has been constructed it must be put into a form that is both readable and useful by the user, and this program is designed to produce a single web page containing a representation of the complete Index table.

The first stage in making the output useful is to adjust the entries to make them more useful. It is here that proper names are adjusted, as detailed in 2.7.8. Another enhancement, which can only take place after the index has been compiled and prior to output, is to merge entries where one ends in "an" and the other is identical but with an additional "s". So for instance, where there are entries for "Americ**an**" and "Americ**ans**" these will be merged under the single category "Americans". This rule should prove useful when the presentation contains nation-orientated words and I foresee few unwanted side-effects. As with the name formatting rules this can only be applied once the index has been built.

It is then necessary to sort first the entries and then each entry's list of page references, to make them easier to understand. Sorting by name can be carried out either from A-Z or from Z-A, and as with all the other sorts performed by the program, a generalised QuickSort routine of my own design is used. The list of page references is then sorted, using the method detailed in 2.7.7.

The next stage is to calculate weights for each term, for each document in which the term appears, thus providing an alternative measure of importance. These can also be used to decide which entries in the index are written to the output file. Ordinarily the user will choose to output the entire index, but selecting entries with the highest weights allows the users to see the most 'relevant' entries.

Only Salton weights can be used for this process since Sparck Jones weights total one for each entry and are therefore inappropriate here. What the program does is determine the minimum Salton weight awarded to each entry, store these in a list then sort them. If the user specifies that they only want the 50% highest weights, then only those words whose minimum weight occurs in the top 50% will appear in the output file.

Word 1 - 'Jospin'
Word 2 - 'Chirac'
Word 3 - 'Mitterrand'
Word 4 - 'Le Pen'

Min. Weight 100, Word 1
Min. Weight 50, Word 4
Min. Weight 3, Word 2
Min. Weight 1, Word 3

In the example, if the user wants the top 50% this corresponds to the first two entries: words 1 and 4, namely, 'Jospin' and 'Le Pen'. The only slight difficulty comes when entries are tied as to their weights.

...
Min. Weight 28, Word 14
Min. Weight 22, Word 9
Min. Weight 22, Word 2
<--------- **cutoff here** ----------
Min. Weight 22, Word 17
Min. Weight 19, Word 3
...

In this example, it has been determined that the cutoff point is where the arrow is. But by cutting off all entries below it we are excluding an entry (for Word 17) despite the fact that it has the same weight as ones that are allowed. What we do, therefore is work upwards from the original cutoff point until we encounter a higher weight, here the entry for Word 14, and cut off all entries below this entry. The consequence is that we may only have 48 entries in the output file even though the user has specified 50. We do not, however, want to mislead the user by ignoring valid entries (for Word 17), so I have decided it is best that we ignore that weight (here, 22) altogether and produce a smaller output rather than a misleading but larger one. The only exception is that when all weights are tied - which would result in no output - the only solution is to allow all the entries through.

Once the user has specified a name for the output file, the file is created and the saving of the index data can commence. Though the program can easily output results in plain text form, the default behaviour is for the index table to be output in HTML format, and I will briefly cover the format of the output. Firstly, the following standard header is written:

```
<HTML>
<HEAD> <TITLE> Index </TITLE> </HEAD>
<BODY>
    <TABLE BORDER=2>
```

The <HTML> tag shows the document is a web document. The header section of the page contains the <TITLE> tag which sets the title of the document in a browser window to "Index", and the <BODY> tag indicates the main body of the document is to follow. The last tag creates a table which will be used to store the entries and their page references.

The program then takes the list of index entries and loops through it. Provided the entry has been selected for output on the basis of its weights, the entry is itself written to the file, followed by its list of page references. Each reference will be formatted to reflect the HTML importance of that reference, as described in section 2.4.1.

An important addition to make the output file more useful to the user is the addition of HTML link for each document in which an index term appears. So, for example, the user could click on "PoliticsDepartment.html" to open the document in which the term "Allende" was found, allowing the user to see the term in context.

The weights are also written to the output file, with the Sparck Jones weight first, followed by the Salton weight. The next section shows the HTML code that is output for an example index entry:

```
<TR>
        <TD BGCOLOR="#cccccc"> <H1><B><FONT COLOR="#0000ff">
        <A NAME="L"></A> L
        </FONT></B></H1> </TD></TR>
<TR>
        <TD BGCOLOR="#999999"> Lützow </TD>
        <TD BGCOLOR="#cccccc"> (<A HREF=main.html>main.html</A>)
                <FONT SIZE=-1 COLOR="#000000">43.</FONT> </TD> </TR>
```

Though it looks complicated, this is due to the extra formatting codes needed to make the output look attractive. The first <TR> code creates a new row in the table, with one column, for the new letter L. <TD BGCOLOR="#cccccc"> means it will have a light grey background, and the subsequent codes will display the letter in the most prominent header style <H1>, and coloured blue ("#0000ff"). The line starting <A NAME=... creates an anchor named "L" so that we can create a link to the start of this letter's section, and the final part of the line contains the "L" that will be drawn within the cell. The next line merely turns off the characteristics we have turned on, now they have been used, and finishes with the column and the row.

The next section appears once for every entry under the letter "L". <TR> creates a new row for it, and a new cell is created with a mid-grey background ("#999999"). Then, the index entry appears, here "Lützow", and the cell is finished with. Another cell is created next to it, in the second column, with a light grey background. The name of the most important document in which the entry appears is displayed (see the "Gestation Period" example), with brackets around it. This is made into a link so the user can quickly go to the relevant document.

This is followed by the first (and only) reference to the entry within that document. The next codes show that the 'page number' ("*43*") is to be displayed in plain, black text, in a small size. This diminutive presence shows that the HTML importance of the reference is not very high.

Here is an example of sample output, including weights:

```
Breeding Patterns
        (Bear.html Weight = 0.33 / 1.59) title, 1595, ...
        (Zebra.html Weight = 0.17 / 0.85) 703, ...
        (Mouse.html Weight = 0.5 / 3.61) 14, 39, 69, ...
```

After each entry, either new line is started (plain text output) or a new row of a table is added (HTML output). Once all the page references for that entry and that document are done, and all the documents have been covered, and all the entries for each letter of the alphabet have been done, it remains to finish the output.

The table is completed with the </TABLE> tag, and the following section is added:

```
Index: <A HREF="#A"> A </A>
<A HREF="#L"> L </A>
<A HREF="#R"> R </A>
<A HREF="#T"> T </A>
<A HREF="#V"> V </A>
```

It adds links to the web document such that, by clicking on a letter, the user can be taken to the index entries that start with that letter, making navigation of the document easier. The final section adds a small copyright notice to the output, and the last two codes complete the index document: </BODY></HTML>

As the program finishes, it will inform the user as to how many entries were available in the index, as well as how many were actually written to the output file (if the numbers are different) on the basis of the options chosen.

Chapter 4

Evaluation and Conclusion

4.1 Program performance

Program performance is good in terms of speed and efficiency. Individual documents as large as 800K have been indexed without complaint, with 65,000 word documents being analysed by the program in as little as 48 seconds. The program is stable and resilient and no crashes were recorded in the past months of development.

4.2 Program results

Automated indexing was never intended to produce back-of-the-book indexes. As Indexicon demonstrates so well, back-of-the-book indexes cannot be automatically generated.

Nancy Mulvany and Jessica Milstead, review of **Indexicon**,
Key Words, Sep/Oct 1994 [14]

The results obtained by my program cannot be neatly summarised since they depend entirely upon the presentation under scrutiny, but the output files, in a number of different formats, and for one document, are included for inspection, as is the original document.

I tested the program using the text of a 65,000 word book and compared the resulting index with that in the printed version. The most obvious feature was that my index contained 1909 entries compared to just 174 in the printed version. Every word in the printed index appeared in the electronic version so no valuable entries were lost. Recall is clearly very high, and precision is also very high in the sense that very few phrases are indexed that are incorrect as a consequence of a failure of the indexing algorithm I use - such as, genuine phrases being incorrectly split into two or more separate entries, or phrases formed out of two or more unrelated words or phrases.

On many occasions, individuals appeared several times in my index because they were referred to with varying degrees of detail in the book and the program did not know enough to realise they were all one person. Similarly, some names had prefixes or forenames too rare for section 2.7.9 to deal with, and so they were sorted by forename rather than surname. In general, though, I believe the program is at its most precise when dealing with proper names.

In most cases my index contained words that weren't strictly irrelevant, just not very important. However, I cannot extend the stop word list indefinitely. With over 400 entries, it manages to mask out the most common words but each new word added to it has less and less impact on its own, and diminishing returns are much in evidence. Besides, highly specialised documents may use words that ought to be in a stoplist due to their limited value, but that are

[14] In <http://www.asindexing.org/indfaq.htm>

too rare in normal usage to warrant entry. For example, words such as “Accesses”, “Returns”, or “Initialises” that might appear in documentation for some source code.

The problem is that I do not have enough rules to determine that some words are unacceptable, and that in the absence of this grammatical intelligence, the program must fall back on the stoplist, even though it lacks, in the absence of a massive number of additions, the ability to be really stringent.

The indexing procedure is clearly far too lenient, even though it may only index between 2 and 4 per cent of the words it encounters. Salton’s argument[15] is that precision should be maximised even if some valid entries are missed, on the basis that the user is spared the burden of useless data, but I am not convinced this is the right strategy.

Any imprecision is almost always a consequence of unnecessary terms appearing rather than necessary terms being missed out. Since it is much easier for the user to eliminate those entries he doesn’t like than try to locate those he ‘thinks’ he wanted after all, I don’t think our leniency is a serious problem, nor one that can really be affected given my resources.

[15] Salton (1989) p.278

Chapter 5

Future Development

5.1 Towards a more useful index

One issue that has not yet been covered is how to make the index hierarchical. A proper index in a book might have the following[16] :

Mauroy, Pierre 107, 117, 119, 122
“ austerity 116
“ background 108, 121, 211
“ Communist Party 115
...

Such an index expresses what the American Society of Indexers referred to earlier as a “hierarchy of ideas”. While the first entry refers to *the indexable item itself* (here, the former Socialist Prime Minister), the second item, “austerity”, refers to a condition *associated with* the object. The third entry, “background”, refers to a *property of* the object, while the reference to “Communist Party” represents another object which was *affected by* our object, and which appears itself elsewhere in the index.

Such an index could only have been created by something that actually knew things about the individual, or at the very least knew how to get information about him. This would almost certainly require a human with the appropriate knowledge. According to the ASI, “Book indexing involves a little bit of manipulating words appearing in a text, which computers can do, and a lot of understanding and organising the ideas and information in the text, which computers cannot do and will not do for many years to come.” [17]

My program was primarily designed to create ‘automatic’ indexes. As such, it has no ‘knowledge’ or experience of its own, but must make decisions based upon a number of grammatical rules which don’t even remotely cover such examples due to constraints of time and experience.

Nonetheless I feel the ASI’s statement is overly pessimistic even from a professional body anxious to protect the interests of its human members. Firstly, the program does not merely create a concordance as the ASI claims. It does not try to exhaustively list every word used in the presentation as the use of the Stop Word list, the suffix-stripping, and all the other word manipulation rules detailed above show.

According to Chrochemore, “Quick access to lexicons is a necessary condition for producing good parsers” [18], and it seems quite plausible that this idea can be extended for our

[16] Stevens, Ann. The Government and Politics of France, 1996
[17] In <http://www.asindexing.org/software.htm>
[18] Chrochemore p 5

purposes, by the indexer consulting another computer program such as an encyclopaedia or dictionary when in doubt as to the properties of and concepts associated with any concept we choose.

A simplified strategy would be just to group together terms that *mean* the same thing, though this still requires knowledge. Instead of determining what constitute the properties of a list of medicines, it could merely be determined that all are "medicines" and that they be listed together under the category :

Medicines:
" Kaolin
" Milk of Magnesia
" Zantac

Though this seems easier, it is still surprisingly difficult for a computer program to determine the meaning of an object. Perhaps the most prominent word returned from that word's entry in an online encyclopaedia could be taken as the word's meaning.

On a more theoretical level, if there existed a tree structure in which all knowledge could be characterised as subcategories and branches, the word's meaning might be represented as the vector of all those nodes for which the node's category was contained in an encyclopaedia reference. On this basis, similar words are those words which have similar vectors. One problem would be that we would undoubtedly have to index the reference in the encyclopaedia in order to be able to compare its prominent words with those found on the nodes. So, for each word in our index, we would have to index an entire document, although a well-designed encyclopedia would be pre-indexed.

Regrettably, all this seems to require resources far beyond those available for this project, and it seems that this program will be unable to classify or group index terms. Understandably, this makes the indexes produced by the program of limited use when the user seeks a summary of the ideas expressed within the presentation, though as a summary of the key phrases I believe performance is good.

5.2 Improvements to the algorithm

All in all, the capital-letter strategy I have outlined seems so effective, and common-sense, that I don't believe it can be discounted, despite the occasions on which it may fail.

5.3 Program Development

As has been seen, whenever an index term is written to the output HTML file, an HTML link is added for each document in the presentation in which that term appears, so that the user can click on the link and open that document in their web browser. However, we would like to be able not just to open the document, but to be taken to the place in the document where the term appears, which may be halfway down the document. The obvious mechanism would be to add HTML anchors to all the documents in the presentation for each term in the index; however, this is very difficult. The program offers a lot of freedom for the user as to what he can index, and although reading someone else's web page may not be a problem, altering it by adding markers to their page certainly would be. The file may be locked or private, for instance, and altering their file might well breach ethical and copyright standards. The HTML specification does not offer a mechanism to search for text strings within a document, this is left up to individual browsers to implement, and these browsers may or may not be capable of being scripted. JavaScript may offer a solution, but this problem must be left for future students to solve.

There are some changes that can be made to the program that would improve performance. The two main stages, of constructing the index, and outputting it, should be split up. This is because, at present, if the user wants two versions of the index output, he must go through both stages each time, including the construction of the index within the program, when in fact the same index can easily be used to generate many different output files. This should only necessitate minor changes to the user interface. Another improvement might be to disguise the many options from the user, replacing them with a single choice for the user, namely, to choose the required compactness of the index from, say, 1 to 5. High values would use more of the rules to eliminate entries, and may only output the top 10%, at the expense of losing some valid entries, while the lowest values might perhaps dispense with the Stop Word list at the expense of containing useless words.

The role of Artificial Intelligence could be increased if the program was extended to monitor changes that the user subsequently made to program-generated indices, and to take these changes into consideration as new rules, next time the program is run.

A potentially feature to implement would be to employ multiple 'threads' so that where possible the program could simultaneously index several different documents rather than just one, and save time. However, the difficult issue is that although the web presentation is a tree structure, it is also largely sequential in that most documents will not be known until referred to by a previous document, which limits the extent to which things can work concurrently. Nonetheless the benefits could be well worth the efforts made.

Chapter 6

Bibliography

* <http://www.asindexing.org/software.htm> (American Society of Indexers)
* <http://www.asindexing.org/indfaq.htm> (American Society of Indexers)
* Benson, E. and Dunn, L., 'The 'Routledge Encyclopedia' Project: Indexing Tools and Management Techniques for Large Documents", *Literary and Linguistic Computing*, Vol. 8, No. 2, (1993) pp.91-93
* Berk, E., Devlin, J. *ed. Hypertext / Hypermedia Handbook*, McGraw-Hill (New York, 1991)
* Chrochemore, M., Rytter, W., *Text Algorithms*, Oxford University Press (Oxford, 1994)
* Etcode, Ned, *HTMLParser.java* (Copyright 1995, 1996, 1997 Netscape Communications Corp. All rights reserved)
* Forrester, M., Indexing in hypertext (hypermedia) environments: the role of user models (sub-structure indexing based on what readers do when accessing indexes, with application to financial services), *The Indexer*, 19(4) Oct.1995, pp.249-256
* Huizhong, Y., A New Technique for Identifying Scientific/Technical Terms and Describing Science Texts. *Literary and Linguistic Computing*, Vol. 1, No. 2, (1986) pp. 93-103
* Mulvany, N. and Milstead, J., Review of Indexicon, *Key Words*, (Sep/Oct 1994)
* Nielsen, J., Hypertext and Hypermedia, Academic Press (Cambridge MA, 1993)
* Park, H., Han, Y., Choi K-S., Compound Noun Indexing in Korean Using an Information Theoretic Metric. Literary and Linguistic Computing, Vol. 12, No. 2, (1997) pp. 87-93
* Salton, G. Automatic Text Processing, Addison Wesley (Massachusetts, 1989)
* Salton, G. and McGill, M. J. Introduction to Modern Information Retrieval. McGraw-Hill Inc. (1983)
* Salton, G. and Buckey, C., Term weighting approaches in automatic text retrieval. Information Processing and Management, 24 (1988): pp. 513-23
* Sampath, G. An Introduction To Text processing, River Valley Publishing (Jeffersontown, 1985)
* Smith, P. D. An Introduction to Text processing, MIT Press (Massachusetts, 1990)
* <http://helix.infm.ulst.ac.uk/~smyth/colour.htm> (Bryan Smyth, MSc Computing & Design, University of Ulster, September 1996)