Search Engines and Web Information Retrieval*

Alejandro López-Ortiz[1]

School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada
alopez-o@uwaterloo.ca

Abstract. This survey describes the main components of web information retrieval, with emphasis on the algorithmic aspects of web search engine research.

1 Introduction

The field of information retrieval itself has a long history predating web search engines and going back to the 1960s. From its very beginnings there was an algorithmic component to this field in the form of data structures and algorithms for text searching. For most of this time the state of the art in search algorithms stayed well ahead of the largest collection size. By the late 1980's with the computerization of the new edition of the Oxford English Dictionary the largest text collections could be searched in subsecond times. This situation prevailed until the second half of 1995 when the web reached a size that would tax even the best indexing algorithms. Initially the web was manually indexed via the "What's new?" NCSA web page, a role later taken over by Yahoo!. By early 1995 the growth of the web had reached such a ce that a comprehensive, manually maintained directory was no longer practicable. At around the same time researchers at Carnegie Mellon University launched the Lycos indexing service which created an index over selected words in web pages. In the Fall of 1995, OpenText started crawling the entire web and indexing "every word of every page". This last was the first comprehensive index of the Web. It utilized state of the art indexing algorithms for searching the Web. Shortly thereafter DEC launched the Altavista search engine which used what was then a massive computer to host their search engine. Since then there has been a steady stream of theoretical and algorithmic challenges in web information retrieval. Today Google indexes around eight billio pages using a cluster of an estimated size of 100,000 computers.

In this survey we consider the main challenges of web information retrieval. Typically a web search engine performs the following high level tasks in the process of indexing the web:

Typically a web search engine performs the following three high level tasks in the process of indexing the web:

* A shorter version of this survey appears as part of "Algorithmic Foundations of the Internet", A. López-Ortiz, SIGACT News, v. 36, no. 2, June 2005.

1. Crawling, which is the process of obtaining a copy of every page in the web.
2. Indexing, in which the logical equivalent of the index at the back of a book is created.
3. Ranking, in which a relevance ordering of the documents is created.

Over the next few sections we will we discuss in more detail how to implement each of these steps.

2 Crawling

Discovery process Search engines start by collecting a copy of the web through a process known as crawling. Starting from an arbitrary URL, a program called a "spider" (which "crawls the web") downloads the page, stores a copy of the HTML and identifies the links (anchor `<a href="...">` tags) within the page. The "spider" is sometimes also called a "crawler" or a "robot".

The URL of each link encountered is then inserted into a database (if not already present) and marked as uncrawled. Once the spider has downloaded and completed the analysis of the current page it proceeds to the next uncrawled page in the database. This process continues until no new links have been found. In the past search engines only crawled pages that were believed to be "static" such as `.html` file as opposed to pages which were the result of queries (e.g. CGI scripts). With the increase of dynamically served content search engines now crawl what before would have been considered dynamic content. Nowadays a large commercial web site is usually served from a database, which has control over who can access and update the content and how often it must be refreshed. So in a certain sense, all the content appears to be dynamic. Crawlers must distinguish then this dynamically served, but rather static, content from a true dynamic web page such as a registration page.

If we create a graph in which each web page is a node and each HTML link is a directed edge, the crawling process will only reach those pages that are linked to starting from the arbitrary URL. To be more precise:

Definition 1. *The* web-page graph*, denoted as P, has a node for every web page and the directed edge (u, v) if web page u has an html link linking to page v.*

In these terms, the crawling process described above will succeed if and only if there is a path in the web graph from the starting node to all other nodes in the graph. In practice it has been observed that this is not the case. Indeed the web-page graph is formed of thousands of disconnected components. To make things more complicated the web-page graph is a directed graph as links are not bidirectional. That is to say, if page a links to page b, then b is reachable from a, but the converse is not true, unless b explictly links back to page a. This process needs to be repeated a large number of times each with a different starting URL.

In fact Broder et al. [2] showed that the web-page graph has a bow-tie structure as shown in Figure 2. The center knot, or core is a strongly connected

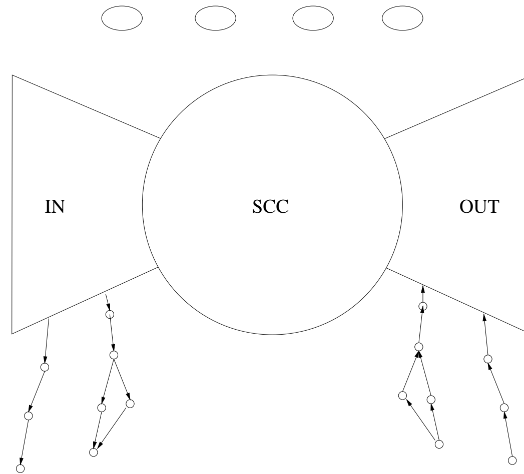

Fig. 1: Bow-tie structure.

component (SCC) on the web-page graph. In this core we can reach any one page starting from any other page in the core. The left part of the tie consists of pages that point to the SCC core but are not themselves reachable from the core by "clicking" on links from page to page. These pages are called *origination* pages (IN).

The right part of the tie are those pages that are pointed to by at least one page in the core, but they themselves do not point back to the core. Those pages are called "termination" pages (OUT). The finger-like structures are called tendrils, which are connected to the main bow-tie but in the "wrong" direction (dead-ends). Lastly there are the disconnected components. At the time of the study the number of them was estimated at 17 million, however since then there has been some indication that their prevalence might have been overestimated due to the practice by some commercial web sites to return non-standard "404 Not Found" pages.

Observe that for the crawler to collect all pages, it must visit all tendrils and pages in the origination section. This means that the crawler has to be manually seeded to start millions of searches in the origination and disconnected section.

Clearly, these seeds cannot be discovered by following links alone. Hence alternative methods to collect URLs are needed. In practice some commonly used ones are: submit URL, random IP/URL probing, passive listening, and previously discovered URLs.

Certain disconnected components can still be reached if their creators manually submitted the URL of their web site to the search engine. A second method is for the crawler to perform random samplings by generating a random IP address not already in the URL database and requesting a web page from that address. Passive listening consists of scanning open communication channels such

as newsgroups and IRCs extracting URLs contained in them, which are then added to the URL database to be crawled.

Once an URL has been discovered it remains in the database even if later on cannot be reached through an external link. This way even if a page eventually becomes part of a disconnected component it can still be reached by the crawler.

URL database The URL database has a relatively simple structure. The challenge comes from both the rate of access and the fact that most most web pages have more than one valid URL that refers to them. Observe that URLs can be added off-line to the database. That is, when the spider finds a link in the process of crawling, the spider can continue crawling while the URL database processes the insert/lookup operation for the link found. Later on the crawler can request from the database the next batch of links to be followed. This means that the URL database does not need to depend on the traditional on-line data structures such as B-trees which fully complete an operation before proceeding on to a second one.

Document duplication detection Another challenge is the duplicate document detection problem. Popular documents tend to be replicated numerous times over the network. If the document is in the public domain, users will make local copies available for their own benefit. Even if the popular content in question is proprietary, such as the web page of a news organization, it is likely to be duplicated (mirrored) by content distribution networks for caching purposes. Under certain circumstances a crawler can chance upon a mirror site and index this alternate copy.

In the case of public domain content, duplicate detection is complicated by the fact that users tend to change formatting and add extra headers and footers when creating a local copy. Hence we cannot use a straightforward tool such as checksum to check if two documents are identical. A second challenge in document duplication is the speed at which the documents must be processed. If time were not an issue, tools such as UNIX's `diff` whose performance is proven could be used to compare documents in a pairwise fashion. In the context of web page crawling this solution is not practicable as there are over four billions of pages on the web and hence a pairwise comparison would be too time consuming. To be effective and algorithm must make a determination of duplication within the time it takes to download a document.

Web site duplication detection Certain commercial web sites replicate their entire web sites in various geographic locations around the world. In such cases we would like to determine that the entire site is duplicated and hence it suffices to crawl only one of the copies. See [10, 3] which survey some of these problems in further detail.

3 Indexing

Once a copy of the web is available locally, the search engine can, in principle, receive queries from users searching for documents containing certain keywords or patterns. There are several different ways to accomplish this. For example, for small, rapidly changing files UNIX provides a facility called `grep` which finds patterns in a source text in time proportional to the length of the text. Clearly, in the case of the web which has an estimated size of several terabytes of text, `grep` would be too slow. Hence, we can benefit from the use of an index. A computerized index for a source text is not unlike the index of a book, where relevant words are listed together with the page number in which they appear. In the case of the web, we can generally index every word (relevant or not) appearing in the page or even every pattern together with the position in which they can be found in the corpus.

Definition 2. *The set of positions where a term can be located in the corpus is called the* postings set.

Typically, a user query is a collection of terms such as `(algorithm, index, internet)` which is interpreted as the set of documents containing all three terms. In other words, the answer to the query above is the intersection of the postings sets corresponding to each of the three terms.

Currently there are three main approaches to text indexing. The most extensively studied, from a theoretical perspective, is the *String Matching Problem* (SMP). In this problem the corpus is a single linear string of text. If there is more than one source document they are simply concatenated sequentially with some suitable markings to form one long string. For example, let the corpus consist of two documents as shown in Figure 2.

```
< d o c > N o t e s   I < / d o c > < d o c > N o t e s   I I < / d o c >
1 2 3 4 5 6 7 8 910 1 2 3 4 5 6 7 8 920 1 2 3 4 5 6 7 8 930 1 2 3 4 5 6 7
```

Fig. 2: Corpus with two documents. The numbers below denote the position of each letter in the text

The user queries for an arbitrary pattern, such as "otes". The corresponding output to the query is the position of all occurences of the pattern in the text. For example "otes" appears twice in the string above, with the first occurrence in position 7 and the second in position 25. The location is usually represented as a byte offset from the beginning of the corpus.

Another approach to search engines is the *Inverted Word Index*. In this case we only index words (or tokens) but not arbitrary patterns. Currently most, if not all, commercial web search engines use an inverted word index approach in their indices.

The third approach is the document based approach. As described above queries do not depend on the particular location of a word in a document, but only care about the presence or absence of the word in a given document. Indeed many search engines support this model only. Surprisingly, although widely used, this approach had not been formalized until recently.

3.1 The String Matching Problem

Suffix trees Traditionally algorithms supporting this type of queries use algorithms based on suffix trees. Suffix trees can be built in linear time using the algorithm of Esko Ukkonen [15]. In terms of space the straightforward implementation consumes $n \lg n$ bits of space. This is so as every tree edge is represented by a pointer of $\lg n$ bits long plus the position of the pattern in the text itself, stored in the leaves, also takes $\lg n$ bits. He et al. showed that the internal nodes can be implemented more succinctly using only a total of $O(n)$ space [9]. However the cost of the pointers to the text at the leaves remains the same for a total cost of $O(n \lg n)$.

For the case of the web n is in the order of trillions of bytes (2^{40}) and hence $\lg n \approx 40$. This means that a suffix based solution might require substantially more space than the original corpus size. Reducing this requirement is an active area of research and somewhat more efficient representations are known.

Burrows-Wheeler Transform The Burrows-Wheeler transform can be used for more space efficient indexing techniques. The algorithm for searching for a pattern P using the BWT takes time $O(|P| \log n + occ)$ and is as follows.

In terms of text indexing, to date the biggest challenge has been to devise a search structure that supports searches for a keyword or pattern P on a text of n bits long in time $|P|$ (i.e. proportional the length of the pattern P) while using an index of size n (bit probe model). To place this in context, we know that searching can be done using $O(1)$ space, aside from the text, at the cost of search time $O(n)$ where n is the length of the text. This is the case for `grep`-like algorithms such as Knuth-Morris-Pratt. Such an algorithm of course would be impracticable for the web, hence the preference for SMP-like solutions that have $O(|P|)$ search time, or a close approximation, at the expense of $\Theta(n \log n)$ space usage.

The state of the art in indexing techniques is very close to achieving the dual objective of linear time i.e. $O(|P|)$ and linear space index structures. The classic suffix tree structure supports searches in $|P|$ time but requires an index of size $n \log n$. In contrast the recently developed compressed suffix arrays support searches in time $O(|P|/\log n + \log^{\epsilon} n)$ using $O(n)$ bits in the unit cost RAM model [8, 12, 5].

3.2 Inverted word index

In this case we collect a set of words which are indexed using a standard index structure such as a sorted array, B-tree, or skip list. When the user queries for a

word we search in the B-tree using the word as a key. This leads to an external leaf of the B-tree which points to the posting lists of the word, if present in the text.

In practice, an English text results in an index of size around 20% of the original size using naive methods, and as little as 5-10% using some advanced compressing mechanisms.

3.3 Document Index

Another algorithmic challenge of interest is to devise data structures and algorithms tailored to an heterogeneous, document-based collection of documents such as the World Wide Web [13, 4]. Most of the classic indexing schemes presume a model in which the user is searching for the specific location of a pattern in a contiguous text string. In contrast, search engine users give a collection of query terms (around three or so on average) and are searching for the subset of documents that contain some or all of the terms, ranked by some relevance metric. Observe that in this setting the input is a set of strings or documents each with a unique ID and there is no inherent order between different documents.

The operations required in the abstract data type (ADT) are

- *list*(p) Report all documents containing the pattern p.
- $mine_k(p)$ Report all documents containing at least k occurrences of the pattern p, for a fixed, predetermined k.
- $repeats_k(p)$ Report all documents containing at least two occurrences of the pattern p at less than k positions away, for a fixed predetermined k.

S. Muthukrishnan gives algorithms for this problem [13]. A related problem consists of searching a large collection of documents that has been suitably tagged using XML. In such a setting the query language is extended to include predicates on the tags themselves, such as the XPath query language. There have been papers that study algorithms specifically tailored to XPath queries [6].

4 Ranking

Aside from the algorithmic challenges in indexing and query processing, there are ranking and classification problems that are, to a certain extent, unique to web content. Web publishing is not centrally managed; as a result, content is of varied quality and, as noted above, duplicates of popular documents are common. At the same time, and in contrast to other heterogeneous collections, content is crossed-linked by means of links (`<A HREF="...">`). In practice it has been observed that this structure can be exploited to derive information on the relevance, quality, and even content, of a document. To give a trivial example, the number of incoming links to a page is a reflection of the popularity of the topic as well as the quality of the document. In other words, all other things being equal, the higher the quality/usefulness of the document, the more links

it has. In practice this is not a very effective method to determine document relevance as popularity of the topic seems to occlude relevance. For example, a very popular page that mentions an obscure term in passing will have more links that an authorative page defining and discussing the term.

In 1998, Jon Kleinberg at IBM, and independently Larry Page et al. at Stanford, discovered a way to distinguish links due to popularity from those reflecting quality [11, 14]. Links are initially given equal weights and considered as votes of confidence on other web sites. After this is done the links from each web site are re-weighted to reflect the confidence ranking computed above and the process is repeated. In principle this process could go on forever, never converging as sites transfer weights between them. A key observation was to consider first the induced subgraph of the result set from the web graph, and then interpret the adjacency matrix A of that subgraph as a linear transformation. Then one can show that the re-weighting process in fact converges to the eigenvalues of the matrix $A^T A$. The eigenvalues rank the pages by relative relevance, as perceived by their peers. This presumes that when a web site links to another there is, to a large degree, an implicit endorsement of the quality of the content. If this is the case often enough, the eigenvalue computation will produce an efficient ranking of the web pages based purely on structural information.

It is difficult to overestimate the importance of this result. Traditionally, many, if not most, systems for ranking results were based on natural language processing. Unfortunately, natural language processing has turned out to be a very difficult problem, thus making such ranking algorithms impractical. In contrast, the eigenvalue methods used by Page's ranking and Kleinberg's HITS (hubs and authorities) do not require any understanding of the text and apply equally to pages written in any language, so long as they are HTML tagged and mutually hyperlinked.

A drawback of the hub and authorities method is that the amount of computation required at query time is impractically high. An open problem is to find an alternate method to compute the same or a similar ranking in a more efficient manner.

Interestingly the same eigenvalue computation can be used to categorize the pages by topic as well as perform other content-based analysis [7]. This subfield has come to be known as spectral analysis of the web graph. [1] show that under a reasonable probabilistic model for the web graph, spectral analysis is robust under random noise scenarios.

5 Conclusions

In this survey we presented an overview of the main aspects of web information retrieval, namely crawling, indexing and ranking.

References

1. Yossi Azar, Amos Fiat, Anna R. Karlin, Frank McSherry, and Jared Saia. Spectral Analysis of Data. In Proceedings of *ACM Symposium on Theory of Computing*

(STOC), 2001, pp. 6 19–626.
2. Andrei Broder, Ravi Kumar, Farzin Maghoul, Prabhakar Raghavan, Sridhar Rajagopalan, Raymie Stata, Andrew Tomkins and Janet Wiener. Graph structure in the Web, *Proceedings of the 9th international World Wide Web Conference*, 2000, pp. 309–320.
3. S. Chakrabarti, B. Dom, D. Gibson, J. Kleinberg, S.R. Kumar, P. Raghavan, S. Rajagopalan, and A. Tomkins. Mining the link structure of the World Wide Web. *IEEE Computer*, August 199 9.
4. Erik D. Demaine, Alejandro López-Ortiz, and J. Ian Munro. Adaptive set intersections, unions, and differences. In Proceedings of *ACM-SIAM Symposium on Discrete Algorithms* (SODA) 2000, pp. 743-752.
5. Erik D. Demaine, and Alejandro López-Ortiz. A linear lower bound on index size for text retrieval. *Journal of Algorithms*, Vol. 48, no. 1, pp. 2-15, 2003.
6. Richard F. Geary, Rajeev Raman, and Venkatesh Raman. Succinct ordinal trees with level-ancestor queries. In Proceedings of *ACM-SIAM Symposium on Discrete Algorithms*, (SODA) 2004, pp. 1–10.
7. D. Gibson, J. Kleinberg, and P. Raghavan. Inferring web communities from link topology. In Proceedings of the *ACM Conference on Hypertext and Hypermedia*, 1998, pp. 225–234.
8. Roberto Grossi, and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In Proceedings of *ACM Symposium on Theory of Computing* (STOC), 1999, pp. 3 97–406.
9. Meng He, J. Ian Munro and S. Srinivasa Rao. A Categorization Theorem on Suffix Arrays with Applications to Space-efficient Text Indexes. To appear in proceedings of *ACM-SIAM Symposium on the Discrete Algorithms*, (SODA), 2005.
10. Monika R. Henzinger. Algorithmic Challenges in Web Search Engines. *Internet Mathematics*, vol. 1, no. 1, 2004, pp. 115-126.
11. J. Kleinberg. Authoritative sources in a hyperlinked environment. In Proceedings of *ACM-SIAM Symposium on Discrete Algorithms* (SODA), 1998, pp. 668–677.
12. U. Manber, and G. Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, vol. 22, no. 5, 1993, pp. 935–948.
13. S. Muthukrishnan. Efficient algorithms for document retrieval problems. In Proceedings of *ACM-SIAM Symposium on Discrete Algorithms*, (SODA) 2002.
14. Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank Citation Ranking: Bringing Order to the Web. *Technical Report, Department of Computer Science, Stanford University*, 1999-66.
15. Esko Ukkonen. On-Line Construction of Suffix Trees. *Algorithmica* v.14 n.3, pp.249-260, 1995.