RETRIEVING CITATIONS ON THE WEB

Dalibor Fiala Karel Jezek
Dept. of Computer Science and Engineering
University of West Bohemia
Univerzitni 22, 306 14 Plzen, Czech Republic
{dalfia, jezek_ka}@kiv.zcu.cz

Abstract – A fundamental feature of research papers is how many times they are cited in other articles, i.e. how many later references to them there are. That is the only objective way of evaluation how important or novel a paper's ideas are. With an increasing number of articles available online, it has become possible to find these citations in a more or less automated way. This paper first describes existing possibilities of citations retrieval and indexing and then introduces CiteSeeker – a tool for a fully automated citations retrieval. CiteSeeker starts crawling the World Wide Web from given start points and searches for specified authors and publications in a fuzzy manner. That means that certain inaccuracies in the inputs are taken into account. CiteSeeker treats all common Internet file formats, including PostScript and PDF documents and archives. The project is based on the .NET technology.

Keywords: Citations, Retrieval, Web, Fuzzy Search, .NET, C#

I. INTRODUCTION

Research papers and reports often cite other articles and, in turn, they are cited elsewhere. The number of citations (or references to a particular paper) is the only objective way of evaluation how important or novel the paper's ideas are. In other words, the number of citations expresses the paper quality. It may also, under certain circumstances, express the quality of a scientist or researcher.

With an increasing number of papers and articles (publications) available online on the Web it has become possible to retrieve citations in a more or less automated way. The task is to develop a sophisticated system that would enable searching the Internet for references to specific research reports and papers or to their authors. All common Internet file formats such as HTML, XML, PDF, PS should be considered including compressed files (ZIP, GZ) as these are frequently used with papers.

Also certain inaccuracies in the inputs have to be taken into account. Errors may occur on either side – in the query as well as in the data to be searched. Thus, some approximate (or fuzzy) comparison must be employed. Unlike other citations retrieval systems, which are based on formulating SQL queries on a vast database of publications or parts of them, our way consists in a systematic Internet searching. From given start points the search expands to all directions determined by the links in the documents being searched. The resulting application is called CiteSeeker. The start points for Web crawling are specified by the user or they can be obtained from a conventional search engine. The program uses existing tools for extracting text from non-textual files and returns results as a list of URLs where the references were found.

With regard to decision support the system is applicable to personnel policy – acceptance of employees, search for experts in a particular domain. Especially for universities or research institutions it is often useful to know whether a candidate is respected in the research community, how much he/she is cited and by whom.

In the rest of the paper we introduce common Web documents in Section II and briefly describe a few search engines in Section III. Section IV is devoted to the problem of a fuzzy search (search with errors) and Section V presents CiteSeeker design issues. The reader becomes familiar with some implementation details in Section VI. Section VII deals with inputs and outputs of the system, some experimental results are shown in Section VIII and, finally, we come to conclusions in Section IX and suggest a couple of possible improvements.

II. WEB DOCUMENTS

The actual navigable Web pages that enable browsing the World Wide Web forward and backward via the system of links are HTML files and their derivatives. In addition to static HTML files, which remain the same when transferred from servers to clients, dynamic pages (PHP, ASP, etc.) generate their content when accessed either on the server side or on the client side. Especially the latter may cause severe problems when processing their source. Rich text format (RTF) files are text files amended by text formatting information. On a similar basis Microsoft Word (DOC) and PowerPoint (PPT) files are made up with that difference that the formatting properties are binary. Although there are tools for extracting text from these files [23] or those tools might simply be created, the irrelevant information can easily be ignored in either case.

The vast majority of online research papers are in Portable Document Format (PDF) [7] or in PostScript (PS) [25]. Files on the Web are often compressed to reduce their size and thus to make their accessibility easier. It is essential for a search engine to be able to unpack compressed files so as to access the information in them. The frequent archive types are Zip, Gzip and

Tar. The corresponding unpacking utilities are pkunzip [26], UnZip [27], gzip [28] and tar [29]. The first program is shareware, the last three are open source.

The actual text in PS and PDF files that would be printed is, in general, not readable from the source. Therefore, external tools that allow extracting text from them must be used. There are a few free utilities which enable extracting plain text from PS and PDF files via command line – Pstotext [24], PreScript [9, 4] and Pdftotext [14]. All of them require the support of Ghostscript, an open source PostScript interpreter [8]. The reliability of freely available software is by far not 100 % as will be shown in Section VIII.

III. Search Engines

The number of Internet hosts, i.e. machines connected to the Internet directly or via dial-up, was about 180 million in January 2003 [17]. Of course, not all of the Internet hosts provide Web services. The total number of Web servers estimated by Netcraft was 40 million approx. in April 2003 [31]. In February 1999, Lawrence and Giles estimated the number of publicly accessible Web servers to be 2.8 million and the number of Web pages about 800 million [1]. According to their research none of the search engines examined by them covered more than 16 % of the Web. With the information above we can make an estimate of the current Web size. Provided the relation between Web servers and pages is the same as in [1] there would be about 11.4 billion Web pages at present (800 / 2.8 = 11 400 / 40). If we assume that Google's 3 billion Web pages (see Section III.A) cover 16 % of the Web, there would be 18.75 billion documents on the Web. Thus, we can guess that there are 10 – 20 billion Web documents ($10^{10} – 2 \times 10^{10}$) at present.

A. General Search Engines

Some information about search engines may be found in [10], [11], [16]. Due to Google's superiority to other search engines in almost all features we are going to take a look at it as a representative of this category. There are well over 3 billion (3×10^9) documents in the main Google database now. [18] claims that, as of December 2001, some 73 % of them were indexed Web pages and 1.75 % were not HTML-like Web pages. PDF and PS files were by far the most numerous - about 90 % (March 2002). Except daily reindexed pages, the most are refreshed every 4 – 8 weeks [19, 20, 21]. CiteSeeker makes use of free Google Web APIs services [15] for automatized querying.

B. Specialized Search Engines

The most representative search engines in context of citations retrieval are ISI Web of Science [22] and ResearchIndex (formerly CiteSeer) [12]. ISI Web of Science is a commercial product. Its database consists primarily of papers from about 8 500 research journals and some Web sites. Services as well as the full source code of ResearchIndex are freely available. Furthermore, unlike ISI Web of Science, the citation index is constructed in a fully automated way – no manual effort is needed. Some internals of ResearchIndex may be found in [2].

IV. Fuzzy Searching

The problem of fuzzy search, which can be reduced into the problem of approximate string match, is essential for finding strings in a text that differ to some extent from those in input. For instance, a possible difference between the names of one publication stated in two various places may consist in the order of letters, in missing, redundant or completely distinct letters (slips), or in missing or redundant word separators (usually spaces).

All of the strings should be in lower case so that the comparison is case insensitive. Also all word separators are supposed to have been converted into a single space beforehand. Apparently, this approach enables comparing strings with diacritics as well and it is often useful when one of the strings includes diacritics while the other one does not.

The method of comparing two strings used in CiteSeeker is discussed in [3] and [5] and the algorithm is, among others, applied in GNU diff 2.7 as stated in [30]. The first step in determination of two strings' similarity is finding the shortest edit script (SES). We use a modification of the basic greedy algorithm which requires O((M + N) D) time, where M and N are strings' lengths and D is the length of SES. See [3] for details. The dissimilarity of two strings can be calculated rather intuitively as SES length divided by the total length of both strings. In general, the similarity of strings A and B can be computed by this formula:

$$sim(A, B) = 1 - D / (M + N) \quad (1)$$

where M is length of A, N is length of B and D is their SES length.

Obviously, sim(A, B) = 1 when A and B are the same and sim(A, B) = 0 when A and B are completely different.

Sim(A, B) is the output of fstrcmp() function, which was acquired at [30]. In addition to A and B, this function has a third parameter – limit. If sim(A, B) acquired during computation drops below limit, execution is stopped. This avoids analyzing strings that can no longer be as similar as requested.

We provide a wrapper for this function – fstrcmp.exe. It has three inputs: limit, A, text. The first two parameters (limit, A) are evident. The text parameter is a string which will be searched for A. It will not be compared to A as a whole. Instead, strings of the same length as A are extracted from text starting at position 0 in text with shifts by one character. Of course, towards the end of text the extracted string will be shorter than A. Each such string is passed as parameter B into fstrcmp(A, B, limit). Provided the length of text is T (fstrcmp() is invoked T times then), the resulting time cost of a fuzzy search for A in text is

$$O(N D T) \quad (2)$$

where N is the search string length (it is 2 * length(A), in fact), T is the text length (which will be searched) and D is the SES length for A and B extracted from text before each invocation of fstrcmp(A, B, limit). Apparently, D may vary on each invocation of fstrcmp() but the relations remain the same.

An alternative to fstrcmp() is Agrep [13], a utility which also provides a kind of fuzzy (approximate) search based on a non-deterministic finite state machine. Unfortunately, it has some severe limitations: The search string must not be more than 32 bytes long, and the number of errors in it must not exceed 8.

V. CITESEEKER – DESIGN

A. Problems with Web Crawling

The core of CiteSeeker is a Web crawler, thus the first obvious problems are related to the Web structure. Each Web server is a directed graph of documents. Links among documents may introduce loops within a server as well as among distinct servers when directed accordingly. The optimal case for CiteSeeker is to traverse a server's documents as a tree.

To avoid loops a mechanism of "memorizing" the documents already searched must be implemented. That means storing the documents URLs in some way. Collecting each individual URL visited would cause similar problems as gathering the documents contents – insufficient space as CiteSeeker performs a long-term search (hours, days, weeks, months, etc.). If we take the lower bound of Web size from Section III for granted, then there are about 10^{10} URLs, each identifying one document. Suppose a URL is a 50 B string on average. Then the total space required to store these URLs is

$$10^{10} \text{ URL} \times 50 \text{ B} = 5 \times 10^{11} \text{ B} \approx 500 \text{ GB} \quad (3)$$

Thus, to keep track of as many documents searched as possible URLs have to be managed in a more economical way.

B. Data Structures

The entire organization is depicted in Figure 1. A few data structures have to be introduced: Pending Servers, Pending Documents, Completed Servers, Completed Documents. The terms need to be explained.

Pending Servers is a queue of the servers to be searched (or, actually, the documents residing on those servers). Initially, it is an ordered set of start points for the search engine. Thus, it may also be referred to as a roots queue. It is a queue without duplicities, so there is an underlying hash table to avoid them. This hash table has a server's URL as its key and a reference to the same object in the queue as its value. Pending Documents is a queue of the documents that have to be searched. All of these documents are on one particular server. Their URLs may be relative to that server. It is a queue without duplicities as well. Completed Documents is a hash table of the documents on one server that have already been searched. Completed Servers is a hash table of the servers that have been "entirely" searched and are now "asleep". They may also be referred to as "skeletons". The "entirety" of the search will be explained below.

C. Web Crawling Activity

A typical procedure of CiteSeeker activity concerned with finding as many documents as possible looks like this.

- A server is popped from the queue of pending servers. At the beginning the queue contains servers (their URLs) provided by the user or obtained from external resources such as invoking another search engine.
- Once a server has been selected the search engine starts crawling it from its root. Every document is searched for search strings (citations) as well as for links to other files. Strictly said, only those documents that are placed on the server currently being searched are processed. The others (again, their URLs) are added to the pending documents of "their" server in the Pending Servers queue provided there is one. In the opposite case, the server is created and enqueued, first.
- Having been handled, each document (its URL) from the current server is added to the Completed Documents table. In this way it is ensured that the document shall never be processed again if referenced from within the same server. In this manner a tree of "all" documents relative to one server is being constructed. More about this tree will be said in Section V.E.
- When no more files on the server have been found, it is checked whether there are some records in the Pending Documents – files that need to be searched. This is also done in conjunction with the completed documents so that no double processing of a file could be possible. When a server has been completely searched, i.e. no new links to relative documents have been found and the pending documents queue is empty, the server is declared as "entirely searched" and is set "asleep". That means its URL is added to the completed servers. These "skeletons" will never be "resurrected" again. So if a document is encountered during further crawling whose server is listed in the completed servers table, it will be ignored.

D. Sparing Space

In CiteSeeker a server is "entirely" searched even if there may be undiscovered documents which will perhaps be referenced later from other servers. This is a trade-off between accuracy and space requirements. Briefly, documents' URLs are kept in memory as long as the search is running on their server (let alone their possible presence in the pending documents of the pending servers before their server is processed), then they are released and are represented as a whole only by the server's URL. In this way we spare a significant number of URLs as their total number is given by this formula:

$$\text{total URLs} = \text{pending servers} + \text{completed servers} + \text{pending documents} + \text{completed documents} \quad (4)$$

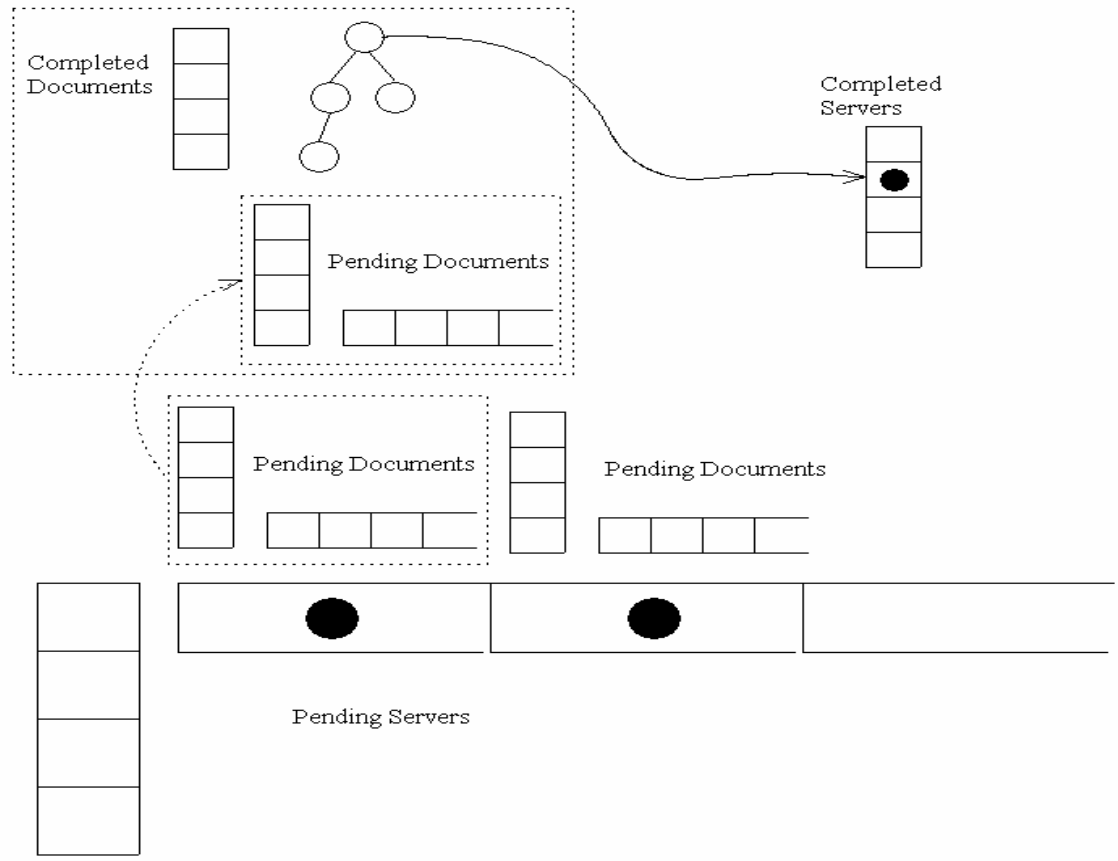

Figure 1- Fundamental data structures

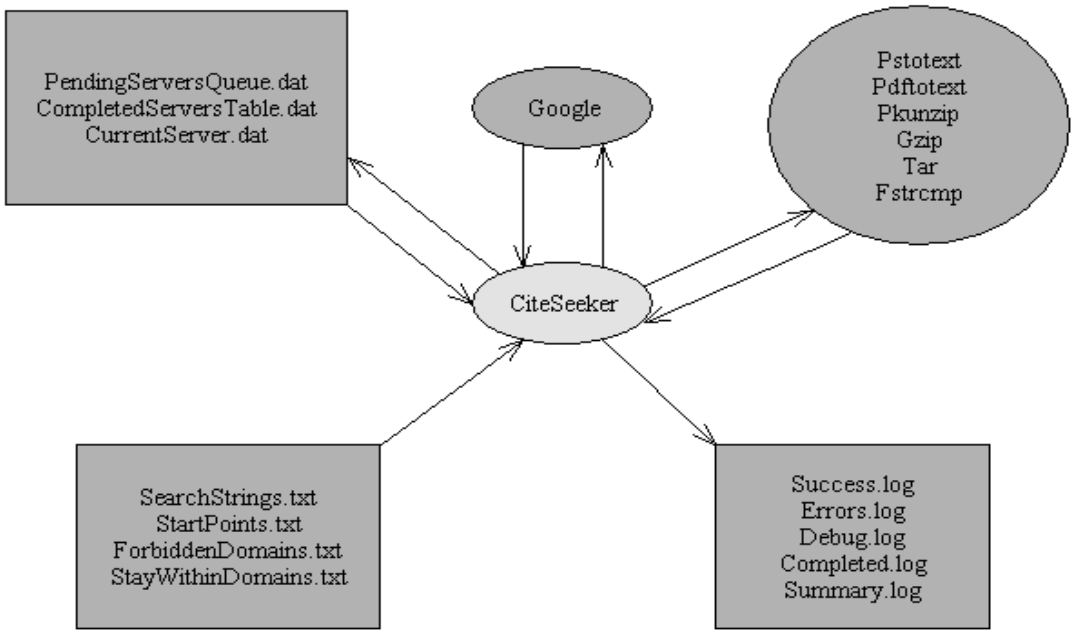

Figure 2- Communication flow

Note that the first two terms (pending servers and completed servers) are thought to be global objects whereas the completed documents are local objects (relative to the current server which is being searched) and the pending documents are both (they are present on the current server as well as on the pending servers). The resulting number may differ in relation with how much activity is done in parallel. The number of pending servers, completed servers, pending documents and completed documents might be in millions each, say tens of millions of URLs at most to comply with the numbers stated in Section III. Again, if a URL is a 50 B string then the memory requirements to store all of them are approximately

$$(K_1 \times 10^6 + K_2 \times 10^6 + K_3 \times 10^6 + R)\ \text{URL} \times 50\ \text{B} \approx \approx 500\ \text{MB} + R \times 50\ \text{B} \quad (5)$$

where K_1 is the number of pending servers, K_2 number of completed servers, K_3 number of completed documents (each represents the order of millions) and R is the number of pending documents.

When R = 0 this is 10^3 times less than with the brute force method in (3). We can manipulate the pending documents and keep R arbitrarily low or high. In case of insufficient memory they are simply not added to the table. On the contrary, if there is space enough the servers that would normally be placed to the "skeletons" might be enqueued in the Pending Servers again in order to collect new relative URLs on their way to the queue head. This would be particularly useful for the very first servers to be searched because they have no or only few relative documents. Likewise, a temporarily unavailable server or an unavailable document may be enqueued once more (or even a couple of times) according to how much space is at our disposal. So the total cost may adaptively change and might be as little as hundreds of megabytes.

E. Documents Tree

How to traverse this tree of documents on one server? All the documents must be searched whatever the order. So there is no need for the tree to be balanced in any way.

If breadth-first search is used the helper data structure is a queue [6]. There are right siblings, children of left siblings and children of the current node (document) in the queue. In general, breadth-first search requires the more space the broader the tree. Specifically, if each node has a fixed number of children, the number of nodes at the same level grows exponentially with the level number [6] and running out of memory would be very fast. If depth-first search (backtracking) is used instead, the underlying data structure is a stack. There is no difficulty with the tree breadth whatsoever. Only a part of the tree needs to be stored at a time – the nodes on stack and the nodes referenced by them. Note that the only content of nodes is a URL of an individual document. In general, at a time there is only the current node, its direct ancestors and their children in memory. Of course, even this method may fail in case of very high trees (the worst case is a simple linked list). Then some of the nodes must easily be thrown away without searching.

Although the time complexity in either case is O(N) where N is the number of nodes [6], the space complexity depends on the shape of the tree. Depth-first search has difficulties with high trees, breadth-first search with broad trees. High document trees on a server are supposed to be less frequent. In addition, the depth-first search enables a faster access to tree leaves where PS and PDF files often reside. Thus, CiteSeeker uses depth-first search.

VI. Some Notes on Implementation

As C# and .NET Framework have a strong support of Internet and Web services related issues, connecting to the Internet may be done within a couple of lines of code. The whole functionality is provided by 18 public classes, the most important of which will be mentioned only. Briefly, they can be derived by object decomposition from Figure 1.

A. Server

Server is the class representing servers and its relation with the Document class is fundamental for understanding the process of Web crawling. Its private attributes are:

- name
- pendingDocsQueue
- pendingDocsTable
- completedDocsTable
- docsStack
- references

Name is the protocol and hostname terminated by a slash by default. PendingDocsQueue is a queue of documents that should be processed. Documents are enqueued in this queue when links to them are found on other servers. (In fact only the documents' URLs are enqueued. The document instances are not created until they are dequeued.) PendingDocsTable is a hash table upon this queue with URLs as its keys. It ensures that the URLs in the queue are unique. CompletedDocsTable is a hash table of URLs of those documents on this server that have been searched already. DocsStack is a stack for the depth-first traversal of the documents tree. References is a count that counts how many times this server (or a document on it) has been referenced from other servers (or documents on them). It is the priority of the server in the Pending Servers queue.

Global variables are defined in this class as well:

- pendingServersQueue
- pendingServersTable
- completedServersTable

PendingServersQueue is the queue of servers that shall be searched, pendingServersTable helps avoid duplicate servers and access them quickly when a document is added to them and completedServersTable is a hash table with names of servers already searched.

B. Document

Document is a node in the Web graph. The tree root (a server's default index file) is a document too. This class provides methods that deal with unpacking, text extraction, finding references to other documents and so forth. The private attributes are:

- URL
- content
- references
- server

URL is the document's unique identifier, content is the document's content in a byte array. References is a list of references to other documents that were found in the content. Server is a reference to the server object to which the document belongs.

After downloading a document, its URL is added to the table of completed documents and removed from the queue of pending documents. Here comes a tricky part of the program. After the download the original document's URL and the one returned from the Internet resource are compared. The string below shows the components of the most comprehensive URL:

protocol :// host : port / path / file # fragment ? query

The problem is that the original and returned URLs may differ not only in the fragment or query components, which CiteSeeker automatically removes from both URLs, but also in the protocol, host, path and file components. This happens when a Web page redirects the request to another Web page. If CiteSeeker added only the URL returned to the table, it would mean that the original URL may be accessed later again. If only the original URL is stored, the search engine will never learn the "real" URL of the resource.

CiteSeeker remembers both of the URLs, which has a negative impact on the size of the table of completed documents. In general, Web robots have problems with dynamic Web pages. The exact method of preventing visits to URLs already visited would involve storing unique keys of documents contents, which slows down the operation.

Removing the query component is very sensitive with dynamic Web pages such as PHP and ASP, which often accept query parameters to eventually provide pages with various content. If the parameter is removed, the dynamic page mostly uses a default one. Leaving the queries would mean an enormous growth of the amount of URLs that would have to be added to hash tables. Moreover, nothing is known about the content of dynamic pages in advance. All these URLs would have to be accessed and only the Content-Type in HTTP header could tell us something. Though it is not always present in the header and it is not very exact. So there is a risk of downloading too many irrelevant files. For these reasons CiteSeeker does not consider queries.

C. Searcher

This class does the actual searching (exact and fuzzy) of documents for citations. It works only with the final state of documents – their plain text converted into lower case.

The actual search routine combines exact and fuzzy search methods to quickly find citations of particular papers in the text of a document. The basis is to make a fast decision which throws away irrelevant documents and then to examine the perspective documents in detail. Originally, we wanted to search fuzzy only the references section of a paper. If the references section was not found, the document would be skipped (the fast decision). However, it might be very tricky to rely that the references sections in articles have always the same form and that they begin with "References" or "Bibliography" titles. The documents themselves would have to be analyzed using artificial intelligence techniques like in ResearchIndex (see Section III.B).

At last, we chose this approach: If an author's name is not found in the document with exact search, the document is ignored (fast decision). Otherwise, a little part of the document past the author's name is searched fuzzy for publications by this author. In this way, not only citations are found, but also documents where the author's name and the publication title are next to each other. But that may be useful as well.

If we denote N the number of author groups, M the number of authors in a group, D the number of publications of that author (user inputs) and P the number of occurrences of an author in the document, the time complexity of this algorithm as to the number of fuzzy search invocations is

$$O(N\ M\ P\ D) \qquad (6)$$

It is simply four nested loops. Of course, M, P, D should rather be considered average values. The complexity of the fuzzy search itself depends on the length of the publication name and the search part length. See (2).

VII. Inputs and Outputs

The basic communication scheme of CiteSeeker is depicted in Figure 2. The TXT files are the primary input for CiteSeeker. They represent authors and their publications whose citations should be found, start points (a list of URLs) which the search will start from and "geographic" restrictions for the search. The LOG files are text files with search results – URLs where citations were found along with the similarity of publications cited and searched for, error and debug messages, a list of servers completed (searched) and some statistical summary (time and numbers).

Serialized objects are stored to the DAT files. These files can be used later when a suspended search is resumed. Google search engine may be invoked at the beginning to get some start points. The last group are external utilities for text extraction, archive decompression and fuzzy search. The Web may be thought of as lying in the background.

VIII. Results

The following two tables demonstrate the capabilities of CiteSeeker used to find citations of 129 publications

by one author on two servers. CiteSeeker was running on a machine with two Intel 447 MHz processors, 1 GB RAM and Windows 2000 on May 15, 2003.

Table I- Searching http://wscg.zcu.cz

Execution time	3 hrs 01 min
Documents searched	1 335
Documents successfully searched	8
New servers found	82
Kilobytes processed	794 031
Archives checked	270
PS and PDF checked	811
Text extraction errors	8
Extracted PS (average time)	255 (19.17 sec)
Extracted PDF (average time)	548 (0.57 sec)

As can be seen in Table I CiteSeeker completely searched the server wscg.zcu.cz in about three hours, processed 1 335 documents (in 8 of them one or more citations were found) with the total size of 794 MB approximately. Links to documents on 82 different servers (including wscg) were found. 270 of the documents were archives. 811 PS and PDF files were checked and 8 errors (1 %) occurred during the text extraction. (This is the official number derived from the return codes of text extraction programs. The actual number is estimated to be much higher. The correctly extracted text is not exactly what would be seen in a viewer, either. Slight differences must always be taken account of.) The average PS extraction time was 19.17 sec while the average PDF text extraction time was only 0.57 sec.

Although the Internet connection speed (roughly 100 kB / sec) had its influence on the resulting time, it is obvious that extracting text from PostScript files makes up 40 – 45 % and from PDF files only 2 – 3 % of the total search time. The poor performance of pstotext is documented in Table II in which pstotext extracts text not only from PS files but also from PDF files.

Table II- Searching http://wscg.zcu.cz without pdftotext

Execution time	5 hrs 02 min
Documents searched	1 343
Documents successfully searched	6
New servers found	82
Kilobytes processed	794 132
Archives checked	270
PS and PDF checked	818
Text extraction errors	61
Extracted PS (average time)	261 (18.77 sec)
Extracted PDF (average time)	496 (10.42 sec)

In this test, which was run on April 24, 2003 on the same computer a slightly modified pstotext was used to improve text extraction from PDFs. The search took now about 5 hours with less success than in Table I. 61 errors (7.5 %) occurred during text extraction (again, experiments have shown that the actual error rate is twice as high at least). The average time of text extraction from PDFs was 10.42 sec, which made up about 28.5 % of the resulting time in total. Thus, as to the text extraction from PDF files, pstotext is at least 10 times slower than pdftotext. No experiments were made with PreScript (see Section II.A) for extracting text from PS files, but it is assumed that it might speed up the search significantly.

A. Memory Cost

Next example is a search on http://www.siggraph.org performed on June 9, 2003 on a machine with an Intel 398 MHz processor, 500 MB RAM and Windows 2000.

Table III shows the time development of memory used. The time variable is given by the number of documents searched, which were sampled six times. Each data structure is represented by a row of numbers of elements included at those six time points and a row of its corresponding sizes in bytes. The table of completed documents enlarges logically, the table of pending documents increases as well because of documents with different original and real URLs (see Section VI.B) that remain in the table. The height (and size) of the documents tree first increases and then decreases as is typical for depth-first search. Both the queue and table of pending servers expand as new servers are encountered during the search.

The data structures are partly in overlay thus the same data may be included in the size. Note the size of both the queue and table of pending servers. The hash table is clearly more memory demanding but it does involve the current server whereas the queue does not (see Section VI.B). Sizes of other objects that do not change are not shown.

IX. CONCLUSIONS

This paper introduced CiteSeeker, a tool for automated citations retrieval on the Web using fuzzy search techniques. CiteSeeker is based on the .NET platform and is almost entirely written in C#. However, it uses a number of external utilities that help handle non-textual documents such as archives, PostScript or PDF files, etc. Inputs for CiteSeeker and its outputs are text files, but CiteSeeker also provides a comfortable graphical user interface, which allows the user to set many search parameters or submit queries to Google. CiteSeeker is available for download at [32].

CiteSeeker has shown its strengths in searching for citations on several “safe” servers, however, it did encounter problems when crawling the “farther” Web where it had difficulties especially with dynamic Web pages. CiteSeeker may be particularly useful for searching servers with conference papers (such as wscg.zcu.cz) that have not yet been crawled by a conventional search engine. As a file name and path is also a URL, CiteSeeker can also search a local disk or CD provided the documents link to each other.

The following list enumerates possible improvements:

- Create more search threads.
- Enhance reliability with dynamic or redirected Web pages. See section VI.B.
- Use PreScript instead of pstotext. See Section VIII.
- Add database support. Currently, CiteSeeker is limited by physical memory or virtual memory paging file. Some tables might be located in a database.
- Enhance the site selection heuristics, in general

This work has been partly supported by grants No. MSM 235200005 and ME494.

REFERENCES

[1] Lawrence S., Giles C. L.: "Accessibility of Information on the Web", *Nature*, Vol. 400, pp. 107 – 109, July 8, 1999

[2] Lawrence S., Giles C. L., Bollacker K.: "Digital Libraries and Autonomous Citation Indexing", *IEEE Computer*, Vol. 32, No. 6, pp. 67 – 71, 1999

[3] Myers E.: "An O(ND) Difference Algorithm and its Variations", *Algorithmica*, Vol. 1, No. 2, pp. 251-266, 1986

[4] Nevill-Manning C. G., Reed T., Witten I. H.: "Extracting Text from PostScript", *Software Practice and Experience*, Vol. 28, No. 5, pp. 481 – 491, 1998

[5] Ukkonen E.: "Algorithms for Approximate String Matching", *Information and Control*, Vol. 64, pp. 100 - 118, 1985

[6] Kucera L.: "Combinatorial Algorithms" (in Czech), SNTL, Praha 1989

[7] Adobe Systems Incorporated: Portable Document Format Reference Manual, Version 1.2, November 27, 1996

[8] Ghostscript, Ghostview and Gsview: http://www.cs.wisc.edu/~ghost/index.htm

[9] NZDL:PreScript: http://www.nzdl.org/html/prescript.html

[10]Kansas City Public Library - Introduction to Search Engines: http://www.kclibrary.org/resources/search/intro.cfm

[11]Google Review on Search Engine Showdown: http://www.searchengineshowdown.com/features/google/review.html

[12]Computer Science Papers NEC Research Institute CiteSeer Publications ResearchIndex: http://citeseer.nj.nec.com/cs

[13]AGREP, an approximate GREP: http://www.tgries.de/agrep/

[14]Xpdf:Download: http://www.foolabs.com/xpdf/download.html

[15]Google Web APIs – Home: http://www.google.com/apis/

[16]Search Engine Showdown Reviews: http://searchengineshowdown.com/reviews/

[17]Internet Software Consortium: http://www.isc.org/

[18]Google Database Components: http://searchengineshowdown.com/features/google/dbanalysis.shtml

[19]Freshness Showdown: http://www.searchengineshowdown.com/stats/freshness.shtml

[20]Remove Content from Google's Index: http://www.google.com/remove.html

[21]Google Information for Webmasters: http://www.google.com/webmasters/4.html

[22]ISI Web of Science: http://www.isinet.com/isi/products/citation/wos/index.html

[23]SWISH++: http://homepage.mac.com/pauljlucas/software/swish/

[24]The pstotext program: http://www.research.compaq.com/SRC/virtualpaper/pstotext.html

[25]Print Center Features – Adobe PostScript vs. Adobe PDF: http://www.adobe.com/print/features/psvspdf/main.html

[26]Download: http://www.shamrock.de/cgi-bin/download.pl?pkunzip.exe

[27]Info-ZIP's UnZip: http://www.info-zip.org/pub/infozip/UnZip.html

[28]The gzip home page: http://www.gzip.org/

[29]tar – GNU Project – Free Software Foundation (FSF): http://www.gnu.org/software/tar/tar.html

[30]http://search.cpan.org/src/MLEHMANN/String-Similarity-0.02/fstrcmp.c

[31]Netcraft: Web Server Survey Archives: http://news.netcraft.com/archives/web_server_survey.html

[32]CiteSeeker Home Page: http://home.zcu.cz/~dalfia/thesis.htm

Table III- Memory cost samples when searching http://www.siggraph.org

Documents searched	100	500	1 027	4 820	10 757	17 862
***Pending Docs* table**	96	496	403	870	1 959	2 688
Size [B]	5 247	32 865	26 775	56 689	143 779	204 611
***Completed Docs* table**	102	505	1 005	4 822	8 857	15 343
Size [B]	6 463	31 423	63 099	330 010	636 857	1 165 061
Documents tree height	9	26	25	66	39	3
Size [B]	3126	10334	7489	15922	10841	1423
***Pending Servers* queue**	39	149	319	1 214	1 984	2 781
Size [B]	11 346	40 405	85 416	333 245	574 126	819 907
***Pending Servers* table**	40	150	320	1 215	1 985	2 782
Size [B]	25 921	115 316	183 918	741 477	1 375 067	2 204 448
Memory usage [B]	24 358 912	29 646 848	58 949 632	107 433 984	107 184 128	106 713 088