A System for Citations Retrieval on the Web

Dalibor Fiala

2003

University of West Bohemia in Pilsen

Faculty of Applied Sciences

Department of Computer Science and Engineering

DIPLOMA THESIS

Pilsen, 2003 Dalibor Fiala

University of West Bohemia in Pilsen

Faculty of Applied Sciences

Department of Computer Science and Engineering

Diploma Thesis

A System for Citations Retrieval on the Web

Pilsen, 2003

Dalibor Fiala

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 thesis 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 search strings 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#

Table of Contents

List of Figures

List of Tables

List of Algorithms

Declaration

I hereby declare that this diploma thesis is completely my own work and that I used only the cited sources.

Pilsen, June 15, 2003 Dalibor Fiala

1 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.

1.1 Goal

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 search strings have to be taken into account, so there must be a fuzzy decision whether the results obtained match the query. Unlike other citations retrieval systems, which are based on formulating SQL queries on a vast database of papers, 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.

This thesis suggests implementation of a system for automated citations search. The resulting program called *CiteSeeker* searches the Web for references to specific publications such as

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

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.

1.2 Platform

The whole project is based on the .NET platform and almost exclusively on the C# programming language. This enables to take advantage of the strong .NET Web services support, comprehensive .NET Framework classes library and very intuitive C# object oriented features. Several external freely available utilities, written mostly in C and C++, are used as well in text extraction and decompression. A little piece of C code was created to provide interface to one of this tools.

1.3 Text structure

Sections 2, 3, 4, 5 discuss theoretical aspects of Web search and citations retrieval including common Internet file formats, existing search engines, the issue of fuzzy search and finally, it brings an overview of the interface to the most powerful search engine ever – Google. Section 6 deals with design viewpoints of *CiteSeeker*; Section 7 reveals the internals of its implementation. Results and conclusions are drawn in Section 8. These three sections are the fundamental part of the thesis. The guides to *CiteSeeker* as well as sample screenshots, inputs and results are placed in appendices.

1.4 Acknowledgements

Many thanks belong to the supervisor of this thesis, Prof. Václav Skala, whose inexhaustible stream of ideas served as a driving engine and helped improve the functionality of *CiteSeeker*. The author is also very grateful to his brother Robert who had to abandon the computer and sleep in another room for a couple of months. Last but not least, it has to be mentioned that Microsoft Visual Studio .NET turned out to be a very pleasant tool to program, debug and deploy such an extensive application.

This project was supported by the Ministry of Education of the Czech Republic - Project MSM 235200005 and Microsoft Research Ltd. (UK).

2 Web Document Formats

2.1 HTML

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 (SHTML, DHTML, XML, etc.). HTML means Hypertext Markup Language. Their contents are plain text containing *tags* (markup), which are special strings bearing semantic information on what the text they enclose actually mean. HTML is standardized by W3C (World Wide Web Consortium) and its latest version is 4.01 [38]. It has about 90 tags (elements) denoting paragraph, header, title, list and others. The fundamental element for hypertext is a link to another Web page or image – simply a connection to other documents. In particular, HTML documents cooperate well with the HTTP protocol via which they are mostly transmitted on the Web.

For the purpose of citations retrieval HTML files can be treated like any other plain text files. They do not need to be parsed – tags and their attributes do not have to be processed separately from the actual content. That would only waste time but it would bring nothing significant. Thus, markup language files do **not** represent **problematic** document formats. Unfortunately, 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 in their processing.

2.2 RTF

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 [39] or those tools might simply be created, the irrelevant information can easily be ignored in either case. None of these documents links to any other. They are **not problematic**, either.

2.3 PDF & PS

The vast majority of online research papers are in Portable Document Format (PDF) or in PostScript (PS). PDF is a file format used to represent a document in a manner independent of the application software, hardware, and operating system used to create it. It ensures that when printed it looks the same as on the screen. Its latest version is 1.3. The key features described in [12] follow:

- It is either a 7-bit ASCII or a binary file.
- Text and images are compressed (JPEG, LZW, Run Length, Flate…).
- It is font independent. If a font used in a document is not available on the computer where the document is viewed a master font will simulate the original font to maintain the colour and formatting using font descriptor information included in the document.
- It is randomly accessed. Using the cross-reference table stored at the end of the file, the time needed to view an arbitrary page is nearly independent of the total number of pages in the document.

- It is updated incrementally. Later changes are appended to the end leaving the current data intact. This enables very fast modifications and it is also possible to undo saved changes.
- It is extensible. Reader applications that understand earlier PDF versions will mostly ignore features they do not implement and will not break
- Various objects may be included in the file such as hypertext links or sounds.
- Encryption is supported. Documents may be encrypted via symmetric RC4 algorithm and MD5 hash function to protect their contents from unauthorized access. They might then have an owner and a user password. The owner can specify operations that are restricted for the user – printing, copying text and graphics out of the document, modifying it, etc.
- Linearized PDFs enable efficient incremental access in a network environment. When data for a page of a PDF document is delivered over a slow channel, the most useful data should be displayed first. For instance, the user can follow a link before the entire page has been received and displayed.

PostScript is a page description language – a programming language designed to describe accurately what a page looks like. It is interpreted, stack-based and device independent. All PostScript printers contain a processor that interprets PostScript code and produces graphical information out of it. A **PDF** file is actually a PostScript file which **has** already **been interpreted** [41]. The latest version is PostScript 3. PostScript is vector oriented. It treats all graphical information as collections of objects rather than as bit maps. It is the standard for desktop publishing because it is supported by imagesetters – very high-resolution printers used to produce final state publication copies.

2.3.1 Text extraction tools

As the actual text in PS and PDF files that would be printed is, in general, not readable from the source, these file types are **problematic**. 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. **All** of them **require** the support of **Ghostscript**, an open source PostScript interpreter [14]. The utilities are following:

- **Pstotext** – an open source program, written in C and in PostScript. It converts PS and also PDF files into text using OCR heuristics [40].
- **PreScript** – part of New Zealand Digital Library project. It is an open source program as well, written in Perl or in Python (both requiring the corresponding interpreter) and in PostScript. It extracts text or HTML from PS and also from PDF files [15]. The algorithms used are described in [4].
- **Pdftotext** – part of Xpdf, an open source project written in C and in C++. It converts PDFs into text [20].

The reliability of freely available software is by far not 100 % as will be shown in Section 8.1.

2.4 Archives

Files on the Web are often encoded (compressed, packed) to reduce their size and thus to make their accessibility easier. They may be compressed individually, or they may be placed into archives (archived) along with other packed files. It is essential for a search machine to be able to unpack compressed files so as to access the information in them. The frequent archive types are:

- Zip – It can store many files and directories. Variations and combinations of probabilistic, LZW, Shannon-Fano and Huffman compression algorithms are used [46]. The unpacking program pkunzip is shareware [42]; UnZip is open source [43]. These archives always have the extension ZIP.
- Gzip – It can store only one file without directory structure. It uses LZ77 compression algorithm. The packing/unpacking program is gzip [44], open source as well. The common extensions of gzip archives are GZ or Z.
- Tar – This is not a compressed archive in fact. It merely concatenates a number of files and directories into a single file. This file is then often compressed by gzip. Tar extensions are TAR and TAR.GZ, TGZ or TAZ when gzipped. The open source tar utility [45] can automatically invoke gzip when correct options are set.

3 Search Engines

Information in this chapter comes from [16], [17], [23] if not stated otherwise.

3.1 Web scope

In all the text below, the terms *server* and *web site* are considered as synonyms. Thus, www.mit.edu and web.mit.edu are two different *servers* with the *documents* that reside on them being referred to as W*eb pages*. The number of Internet *hosts*, i.e. machines connected to the Internet directly or via dial-up, was about 180 million in January 2003 as can be seen at Figure 3.1. See [24] for the methodology used in *host* counting. Of course, not all of the Internet *hosts* provide Web services. The total number of Web servers estimated by Netcraft [48] was 40 million approx. in April 2003.

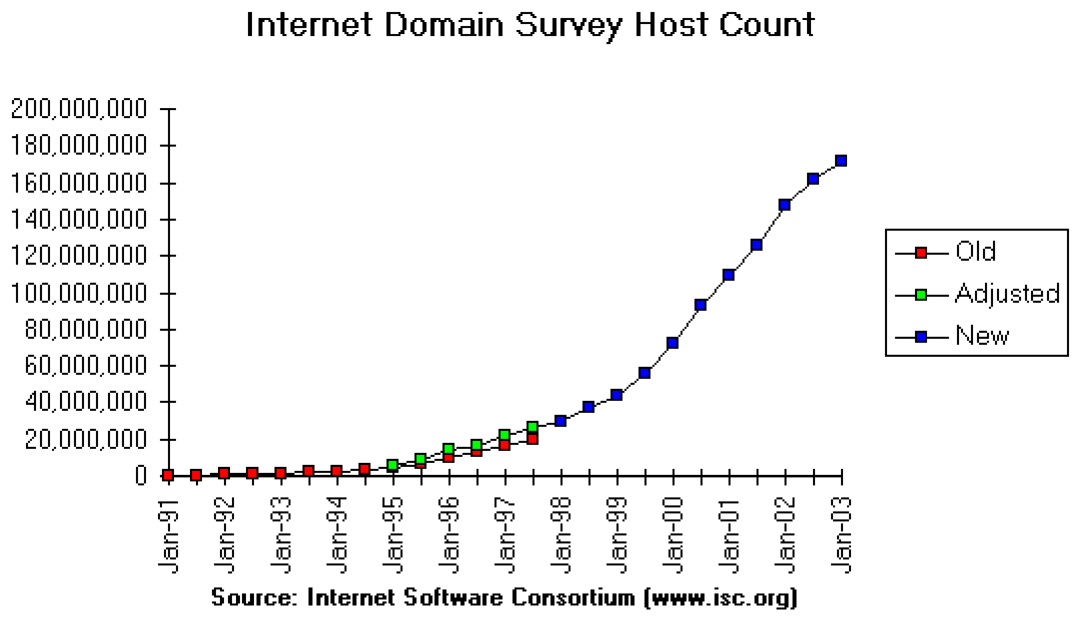


Figure 3.1: Internet growth (source: [24])

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]. The method used was random sampling of IP addresses, then removing servers behind firewalls, requiring authorization or containing no content. Then 2 500 random servers were crawled trying to get all pages on them. The resulting number of pages derived from the average number of pages on the random servers and their estimate of all servers. They also found out that there was 6 TB (terabytes) of textual content on the Web and that 6 % of the pages were scientific or educational. According to their research none of the search engines covered more than 16 % of the Web.

3.1.1 Current Web size

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 3.2.1) 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 x 10^{10}). Again, retaining the relations from [1] that would mean 75 – 150 TB of textual information. The numbers corresponding to scientific and educational pages are then 600 - 1 200 million Web pages and 4.5 – 9 TB of textual content. These are well the scopes that everyone searching the World Wide Web must face.

3.2 General search engines

The Web growth is exponential and bibliographic control does not exist. To find a particular piece of information the use of a search engine is a necessity. The term *search engine* is rather universal; a finer division may be used:

- *search engines*
- *directory services*
- *metasearch engines*
- *search providers*

In this grouping *search engines* such as Google, AltaVista, Excite, HotBot, Lycos do keyword searches against a database, *directory services* (Yahoo!, LookSmart, Britannica, The Open Directory) sometimes also called *subject guides* offer information sorted in subject directories, *metasearch engines* (also *metacrawlers*) such as Dogpile, Mamma, Metacrawler or SawySearch send search requests to several *search engines* and *search providers* provide the underlying database and search engine for external partners. Examples of *search providers* might be Inktomi and Fast Search.

Directory services are fine for browsing general topics but for finding specific information *search engines* are more useful. Various factors influence the results from each. In particular, the size of their database, frequency of its update, search capability and design (algorithms) resulting in different speed have to be taken into account. The best use of *metasearch engines* is submitting queries on obscure items to find out if something can be found on the Web at all. One of the current Inktomi partners is MSN Web Search.

The categories listed above are not completely disjunctive sets. For instance, Google provides its own directory service or Yahoo! invokes Google to refine its services. These partnerships make way to a new name – *portal*. This implies a starting point for all uses of the Web.

3.2.1 Google

Due to Google's superiority to other *search engines* in almost all features we are going to take a closer look at it as a representative of this category.

It launched officially on September 21, 1999 with Alpha and Beta test versions released earlier. Since then it has pushed through with its relevance linking based on pages link analysis (the patented PageRank method), cached (archived) pages and a rapid growth. In June 2000 it announced a database of over 560 million pages and they moved up their claim up to 3 billion by November 2002. As of April 25, 2003 the number is 3 083 324 652 Web pages [25].

Today it has more than 800 employees who speak 34 languages, more than 60 of them having a PhD. degree. 12 Google offices are spread worldwide. About 200 million queries are answered a day. Each search takes fractions of a second, which is achieved by the computing power of a cluster of 10 000 networked PCs running on Red Hat Linux [25].

Googlebot

Googlebot is Google's web-crawling robot written in C++ programming language. It collects documents from the web to build a searchable index for the Google search engine. It follows HREF and SRC links on the Web pages. It is registered with the Web robots database [32] and it obeys the *Robot Exclusion Standard* [31] so it is possible to prevent it from visiting a site or a particular URL.

Databases

The main Google database consists of four parts:

- *indexed Web pages*
- *daily reindexed Web pages*
- *unindexed URLs*
- *other file types*

Indexed Web pages are Web pages whose words have been indexed, i.e. some records have been made about what terms and how many times they occur on a specific page. Typically, the terms are sorted descending as in an inverted index.

Daily reindexed Web pages are the same, except that Google reindexes them “every day”. These pages display the date they were last refreshed after the URL and size in Google's results.

Unindexed URLs represent URLs for Web pages or documents that Google's spider (Googlebot) has not actually visited and has not indexed.

Other file types are Web-accessible documents that are not HTML-like Web pages, such as Adobe Acrobat PDF (.pdf), PostScript (.ps), Microsoft Word (.doc), Excel (.xls), PowerPoint (.ppt), Rich Text Format (.rtf) and others.

The percentage of these four categories as of December 2001 is shown at Figure 3.2, which originates from [26].

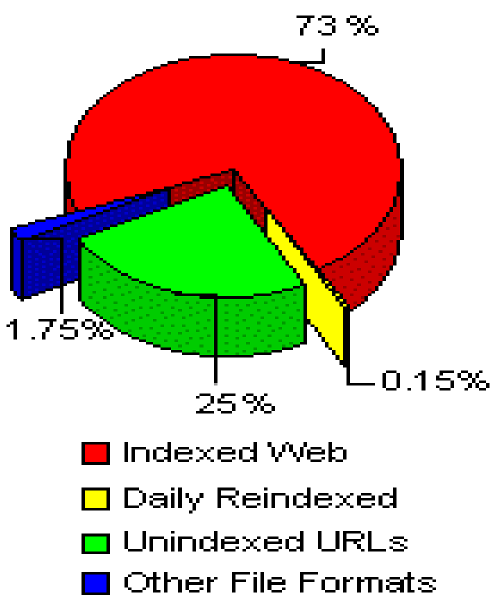


Figure 3.2: Main Google database breakdown (Dec 2001, source: [26])

With regard to Section 2 there is another interesting analysis made by the same author whose results can be seen at Figure 3.3.

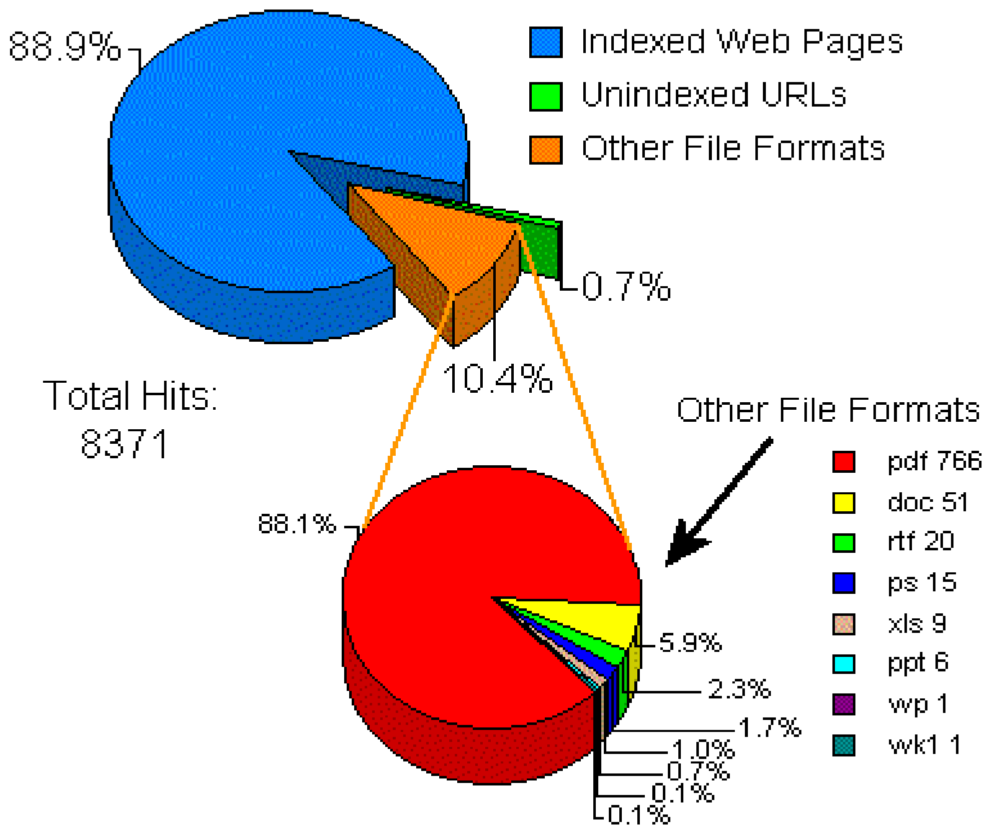


Figure 3.3: Analysis of Google searches (March 2002, source: [26])

The upper pie chart represents a total of 8 371 results from 25 one-word searches on Google. *Indexed* and d*aily reindexed Web pages* were counted together and are the largest slice. The lower pie chart shows the subdivision of the o*ther file types* with corresponding file extensions and the actual number of results for each type. **PDFs are by far the most numerous.**

In addition to the main database, Google has image, news and several topic databases (compare with Section 5) and runs a directory service as well.

Comparison with other search engines

The first feature to compare one may think of is the size of the databases. While it is difficult to measure the absolute database size and the search engines providers have to be trusted in what say, it is possible to determine relative database sizes [27]. The numbers in the chart (Figure 3.4) represent the number of (HTML-like) Web pages found for 25 single word queries.

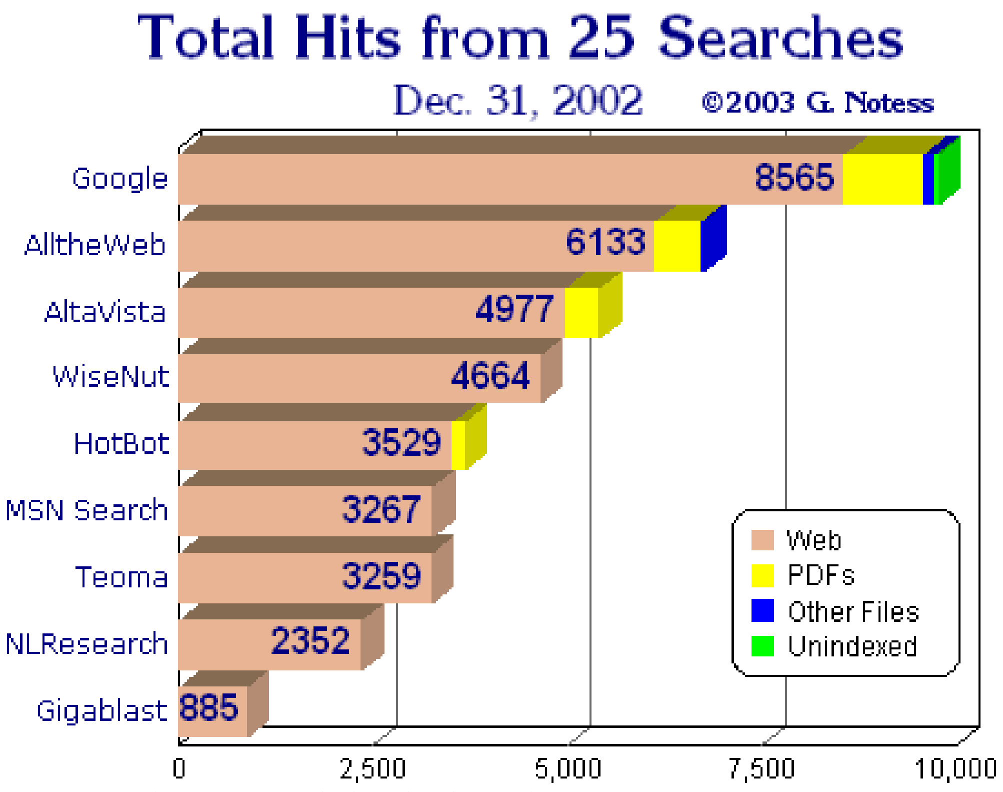


Figure 3.4: Relative database sizes (Dec 2002, source: [27])

It is easy then to recalculate the results into absolute numbers provided we consider Google's 3 083 million documents as 100 %. The resulting absolute numbers of documents in search engines' databases are in Table 3.1. Unlike Figure 3.4, though, relative numbers are the total numbers of hits (verified results), i.e. with all possible URLs included.

Search engine	**Total hits in test**	**Documents in database [millions]**
Google	9 732	3 083
AlltheWeb	6 757	2 141
AltaVista	5 419	1 717
WiseNut	4 664	1 478
HotBot	3 680	1 166
MSN Search	3 267	1 035
Teoma	3 259	1 032
NLResearch	2 352	745
Gigablast	2 352	745

Table 3.1: Absolute database sizes (derived from Figure 3.4)

The methodology used to conclude the numbers in Figure 3.2 through 3.4 can be seen at [28].

Now that we know how big the databases of search engines are and what they consist of, the second relevant feature is how often their databases are refreshed. That means, in what intervals the crawler visits one particular Web page. (The crawler is a program that "crawls" the Web, downloads documents that are then indexed by other programs or even humans.) Again, there exists some analysis [27] summarized in Table 3.2, which is a selection of the most interesting search engines. The analysis evaluated 5 searches with 13 – 24 matches per search engine for specific pages that were updated daily and reported that date.

Search engine	**Newest page found**	**Oldest page found**	**Rough average**
Google	1 day	53 days	1 month
MSN	1 day	36 days	2 weeks
AltaVista	1 day	112 days	3 months
Gigablast	6 days	172 days	4 months

Table 2.2: Databases' freshness (as of Oct 20, 2002, source: [27])

Surprisingly, MSN turned out to be the most "fresh" search engine with the average crawler visits interval of about 2 weeks. This may be in relation with the nature of the Web pages searched for or the time that the analysis was made. Google themselves claim to reindex pages within 6 to 8 weeks [29] or every 4 weeks [30]. Obviously, all search engines are "pictures of the past", none of them performs a real-time Web search.

Remark

One interesting, undocumented Google search function is the possibility to find out how many pages within a particular domain it has actually indexed. The special *site:* term must be used along with a term from the domain name. For example, *wscg site:wscg.zcu.cz* will retrieve the most comprehensive set of results (1 070 as of May 1, 2003).

Google summary

In tables 3.3 and 3.4 Google's strengths and weaknesses are summarized. The conclusion was made from [17] and from personal experience. See also Table 5.2 for an overview of special search terms.

Strength	Description
size	It has the largest database including many file types.
relevance	Pages linked from others with a greater weight given to authoritative sites are ranked higher (PageRank analysis).
cached archive	It is the only search engine providing access to pages at the time they were indexed.
freshness	The average time of page reindexing is one month.
special query terms	It offers a number of special query terms enabling very specific searches.

Table 3.3: Google's strengths

Weakness	Description
limited Boolean search	The full Boolean searching with the ability to nest operators is not supported.
case insensitivity	No case-sensitive search is possible.
no truncation	There is no automatic plural or singular searching or search for words with the same word root. There is only one way of using wildcards such as "*" in the phrase "pay attention * fire" where it matches any word in that position. (Very useful in finding the correct English prepositions.)
no similarity	Google searches for exact, not for similar words.
size limited indexing	It indexes only first 101 kB of a Web page and about 120 kB of PDFs.

Table 3.4: Google's weaknesses

3.3 Specialized search engines

Two specialized search engines related to citations retrieval from scientific papers will be taken a closer look at:

- ISI Web of Science [33]
- ResearchIndex (formerly CiteSeer) [18]

3.3.1 ISI Web of Science

ISI Web of Science (ISI stands for Institute for Scientific Information) enables users to search a database consisting primarily of papers from about 8 500 research journals. In addition to journals, specific Web sites are also included in the database. See [10] and [11] for information on how the journals and Web sites are selected. The database covers 1978 to date, but only the 1991+ portion has English language abstracts. This amounts to approximately 70 % of the articles in the database. There are weekly updates, with items usually appearing 3 to 8 weeks after publication [34]. Its important feature is the *cited reference searching*. Citations mean later references citing an earlier article. Users can search for all references to specific papers, authors or even keywords. ISI Web of Science is a **commercial product**.

3.3.2 ResearchIndex

On the contrary, services as well as the full source code of ResearchIndex are **freely available**. ResearchIndex uses search engines (with queries "publications", "papers", "postscript", etc.) and crawling to efficiently locate papers on the Web. Start points for crawling may also be submitted by users who would like to have their documents indexed. It may take a few weeks after submitting to happen so. Its database is continuously updated 24 hours a day. Unlike ISI Web of Science, the citation index is constructed in a **fully automated** way – no manual effort is needed. That may cause a broader scope of literature to be indexed.

Operating completely autonomously, ResearchIndex works by downloading papers from the Web and converting them to text. It then parses the papers to extract the citations and the context in which the citations are made in the body of the paper, storing this information in a database. ResearchIndex includes full-text article and citation indexing, and allows the location of papers by keyword search or citation links. It can also locate papers related to a given article by using common citation information or word similarity. Given a particular paper, ResearchIndex can also display the context of how subsequent publications cite that paper [2].

ResearchIndex downloads Postscript or PDF files, which are then converted into text using PreScript from the New Zealand Digital Library project [15]. It checks that the document is a research document by testing for the existence of a reference or bibliography section.

Once ResearchIndex has a document in usable form, it must locate the section containing the reference list, either by identifying the section header or the citation list itself. It then extracts individual citations, delineating individual citations by citation identifiers, vertical spacing, or indentation. ResearchIndex parses each citation using heuristics to extract fields such as title, author, year of publication, page numbers, and the citation identifier. ResearchIndex also uses databases of author names, journal names, and so forth to help identify citation subfields.

Citations of a given article may have widely varying formats as shown at Figure 3.5. Much of the significance of ResearchIndex derives from the ability to recognize that all of these citations might refer to the same article. Also, ResearchIndex uses font and spacing information to identify the title and author of documents being indexed. Identifying the indexed documents allows analyzing the graph formed by citation links, which results in abundant citation statistics.

Aha, D. W. (1991), Instance-based learning algorithms, Machine Learning 6(1), 37-66.
D. W. Aha, D. Kibler and M. K. Albert, Instance-Based Learning Algorithms. Machine Learning 6 37-66, Kluwer Academic Publishers, 1991.
Aha, D. W., Kibler, D. & Albert, M. K. (1990). Instance-based learning algorithms. Draft submission to Machine Learning

Figure 3.5: Varying citations of the same article (source: [2])

Internals

Several classes of methods for identifying and grouping citations to identical articles are applied [2]:

- String distance measurements, which consider distance as the difference between strings of symbols.
- Word frequency measurements, which are based on the statistics of words that are common to each string (TFIDF – Term Frequency vs. Inverse Document Frequency, common in information retrieval).
- Knowledge about subfields or the structure of the data
- Probabilistic models, which use known bibliographic information to identify subfields of citations

Furthermore, algorithms for finding related articles are used:

- word vectors, a TFIDF scheme used to locate articles with similar words
- distance comparison of the article headers, used to find similar headers
- Common Citation vs. Inverse Document Frequency (CCIDF), which finds articles with similar citations

3.3.3 Conclusions

Both ISI Web of Science and ResearchIndex do citation indexing of research publications, the first needing manual effort, the latter fully automatized but devoted primarily to computer science literature. There exist a number of other specialized (scientific) search engines such as OJOSE [35], Scirus [36] or Phibot [37]. OJOSE is a metasearch engine, which submits queries to scientific databases, online journals or other search engines. Scirus and Phibot index research Web sites and electronic journals. Phibot offers improved relevance algorithms and indexes some of the Web sites in 15-minute intervals! None of these three search engines provides services related to citations retrieval. ResearchIndex, developed in NEC Research Institute, Princeton is by far closest to the objective of this thesis.

4 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. The difference may consist in the order of letters, in missing, redundant or completely distinct letters (slips), or in missing or redundant word separators (usually spaces). The above mentioned is the expected dissimilarity between input strings and those found. But, in general, each pair of strings (input and found) may have two arbitrarily diverse members. Table 4.1 shows an example of such string pairs.

algorithm	algoritm
line clipping	line cliping
facial expression analysis	facialexpression analysis
three-dimensional object construction	three dimensional object construction
approximating shortest paths	aproximating schortest pahts

Table 4.1: Fuzzy search string pairs

It is desirable that all of the contrasting strings be recognized as “the same”, which would be impossible when using exact match search. Note that all of the strings are in lower case, i.e. the comparison is case insensitive. Also all word separators are supposed to have been converted into a single space beforehand. Obviously, 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.

4.1 Exact search

A commonly used exact search algorithm, in which all of the characters (or symbols) of the search string must match the corresponding characters in the substring of the text that is searched, is the KMP algorithm with time complexity of O(N + T), where N is the search string length and T is the length of the text to be searched [9]. Unlike exact search we need a metric of how similar the search string and the substring are.

The method of comparing two strings used in our project is discussed in [3] and [5] and the algorithm is applied in GNU diff 2.7 as stated in [47]. A short summary is presented below.

4.2 Edit graph

Let A and B be two strings of lengths N and M composed of characters $a_1a_2\ldots a_N$ and $b_1b_2\ldots b_M$. The *edit graph* for A and B has a vertex at each point in the grid (x, y), $x \in [0, N]$ and $y \in [0, M]$. The vertices of the edit graph are connected by horizontal, vertical, and diagonal directed edges to form a directed acyclic graph. If $a_x = b_x$ then there is a diagonal edge pointing from (x – 1, y – 1) to (x, y). See Figure 4.1 for an edit graph of the strings “SKALA“ and “ACULA“. Note that the strings might have various lengths, thus implying a rectangular grid.

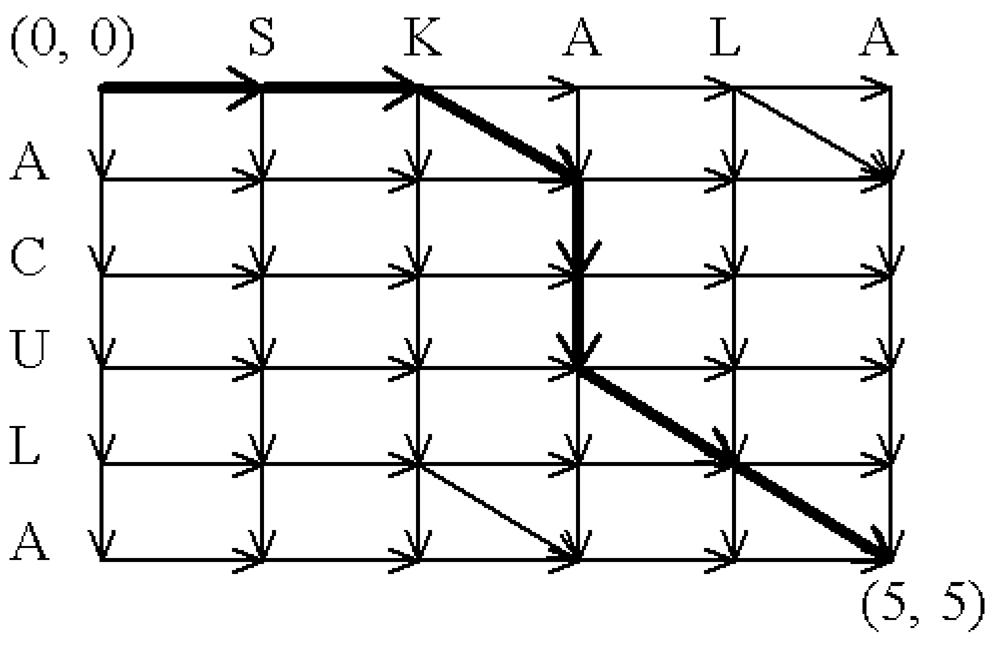


Figure 4.1: Edit graph

4.2.1 Subsequences and edit scripts

A *subsequence* of a string is any string obtained by deleting zero or more characters from the given string. A *common subsequence* of two strings is a subsequence of both. An *edit script* for strings A and B is a set of insert and delete commands that **transform A into B**. The *delete command* ''x*D*'' deletes the letter a_x from A. The *insert command* ''x I $b_1b_2...b_t$" inserts the sequence of characters $b_1...b_t$ immediately after a_x. All the commands refer to the original positions within A, before any changes have been made. The **length** of a script is the **number of characters inserted and deleted**.

Each diagonal edge ending at (x, y) gives a symbol, a_x (= b_y), in the common subsequence. Each horizontal edge to point (x, y) corresponds to the delete command ''xD'', and a sequence of vertical edges from (x, y) to (x, z) corresponds to the insert command, ''x I $b_{y+1}...b_z$''. Thus the number of vertical and horizontal edges in the path from (0, 0) to (N, M) is the length of its corresponding script and the number of diagonal edges is the length of its corresponding subsequence. For the path at Figure 4.1 the edit script is "1D 2D 3ICU" and the common subsequence is "ALA".

4.2.2 Shortest edit script

The problem of finding a *shortest edit script* (SES) is equivalent to finding a path from (0, 0) to (N, M) with the minimum number of non-diagonal edges.

A modification of the basic greedy algorithm is running it "simultaneously" in both the forward and backward directions until furthest reaching forward and reverse paths starting at opposing corners overlap. The procedure only requires O((M + N) D) time, where M and N are strings' lengths and D is the length of the SES. See [3] for details.

4.2.3 Similarity

Let us now return to Figure 4.1 again. The path shown is the minimum path from (0, 0) to (5, 5) with respect to the number of non-diagonal edges. Thus, the edit script "1D 2D 3ICU" is the SES whose length is 4. The *dissimilarity* of the two strings can be determined 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:

$$\mathrm{sim}(A, B) = 1 - D / (M + N) \qquad (4.1)$$

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. Thus using (4.1), sim("SKALA", "ACULA") = 1 – 4 / (5 + 5) = 0.6 (60 %).

4.3 Fuzzy string comparison (fstrcmp)

Sim(A, B) is the output of ***fstrcmp()*** function, which was acquired at [47]. 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. See Table 4.2 for outputs of *fstrcmp()* with *limit* set to 0.0 in all cases.

algorithm	algoritm	0.9412
line clipping	line cliping	0.9600
facial expression analysis	facialexpression analysis	0.9804
three-dimensional object construction	three dimensional object construction	0.9730
approximating shortest paths	aproximating schortest pahts	0.9286

Table 4.2: Strings and their similarities

We provide a wrapper for this function – **fstrcmp.exe**. It has three inputs: *limit*, *A*, *text*. The first two parameters (*limit*, *A*) are obvious. 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 P (*fstrcmp()* is invoked P times then), the resulting time cost of a fuzzy search for *A* in *text* is

$$O(N\,D\,T) \qquad (4.2)$$

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 [19], 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.

5 Google Web APIs

The **Google** search engine provides free APIs for its services.[21]. They are available for non-commercial use only. They enable software developers to query more than 3 billion Web documents directly from their own computer programs. Google uses the SOAP and WSDL standards so that programs written on any platform supporting web services can communicate with it.

SOAP (Simple Object Access Protocol) is an XML based protocol for exchange of information in a decentralized, distributed environment. **WSDL** (Web Services Description Language) is an XML format for describing network services as a set of endpoints operating on messages containing either document-oriented or procedure-oriented information. **Web services** are programmatic interfaces for application-to-application communication via the World Wide Web.

To be able to use the Google Web APIs services the appropriate WSDL file (GoogleSearch.wsdl) must first be obtained. Second, a Google Account has to be created, which results in receiving a licence key. The Google Account and license key entitle the holder up to 1 000 automated queries per day. Finally, the developer's program must include the license key with each query submitted to the Google Web APIs service.

5.1 Services

Google Web APIs services support three types of requests: *search requests*, *cache requests* and *spelling requests*. *Search requests* submit a query string and a set of parameters to the Google Web APIs service and receive a set of search results in return. Cache requests submit a URL and receive the contents of the URL when Google's crawlers last visited the page. The content is in plain text (in fact, it is HTMLized). Spelling requests submit a query and get a suggested spell correction for the query in return. Below, we will describe only the search requests, which are used in *CiteSeeker*.

5.2 Search requests

All information below and more details can be found at [22]. The syntax of a Google search request is following:

```
Google.GoogleSearchResult
Google.GoogleSearchService.doGoogleSearch(string key, string q, int start, int maxResults,
                                          bool filter, string restrict, bool safeSearch, string lr,
                                          string ie, string oe)
```

The parameters semantics is summarized in Table 5.1 below.

key	Google Account key, required to access the Google service.
q	The actual query. See Query terms for details.
start	Zero-based index of the first desired result.
maxResults	Number of results desired per query. The maximum value per query is 10.
filter	Hides very similar results and results coming from the same Web host.
restrict	Restricts the search to a country like "Ukraine" or a topic like "Linux."
safeSearch	Enables filtering of adult content in the search results.
lr	*Language Restrict* - Restricts the search to one or more languages.
ie	*Input Encoding* – Ignored, only UTF-8 is supported.
oe	*Output Encoding* – Ignored, only UTF-8 is supported.

Table 5.1: Google search parameters

5.3 Query terms

By default, all the terms in the query string are logically ANDed. Also, the order of the terms in the query will impact the search results. Common words (conjunctions, prepositions, articles as well as some single digits and single letters) are ignored. These words are referred to as *stop words*. Enclosing *stop words* in double quotes prevents Google from ignoring them. An example might be the phrase “what is in the box”. An alternative way is to precede them with a plus sign (+).

All non-alphanumeric characters that are included in a search query are treated as word separators. The only exceptions are the double quote mark ("), plus sign (+), minus sign (-) and ampersand (&). The ampersand character (&) is treated as a normal character in the query term in which it is included, whereas the remaining exception characters correspond to search features mentioned in the below table of the most important special query terms (Table 5.2).

Capability	Example	Description
inclusion	czech +beer	It enforces "beer" to be included in the search.
exclusion	british –food	It enforces "food" to be excluded from the search.
phrase	"to be or not to be"	It searches for complete phrases.
boolean OR	paris OR london	It retrieves pages including either "paris" or "london".
site restriction	opengl site:wscg.zcu.cz	It searches only within a specific web site.
title search	intitle:visualization	It restricts the search to documents containing "visualization" in the title.
URL search	inurl:visualization	It restricts the search to documents containing "visualization" in their URLs.
file type	graphics filetype:doc	Results include only documents with the specified extension.
back links	link:www.zcu.cz	It lists web pages that have links to www.zcu.cz.
cached results	cache:www.foo.com/foo1.rtf	It returns the cached HTMLized version of the desired document.

Table 5.2: The 10 most important special query terms

There are 18 special query terms in total.

5.4 Filters and restricts

If **filter** parameter is true, very similar results are hidden and if multiple results come from the same Web host, then only the first two are returned. The **restrict** parameter enables searching sites only within particular countries or related to a topic. Currently, 241 countries and 5 topics (US Government, Macintosh, Linux, BSD Unix, Microsoft) are supported. Another kind of filter is the **safeSearch** parameter set to true, which excludes sites with adult content from the results. The **lr** parameter forces Google to search for documents within one or more specified languages. At present, there are 28 languages that can be chosen from. The **restrict** and **lr** parameters may be combined together.

5.5 Limitations

Due to optimization of Google's infrastructure for end users there are rather strict limitations for Google Web APIs developers. These limitations are said to be strongly reduced in the future. Table 5.3 below shows limits of the latest (August 2002) release of the APIs. However, experiments have shown that the bounds may be exceeded. The last two limits mean that 100 queries may be submitted a day, each with 10 results at most.

Feature	Limit
max. search request length	2048 B
max. number of words in the query	10
max. number of *site:* terms in the query	1
max. number of results per query	10
max. number of results per day	1000

Table 5.3: Google Web APIs limitations

5.6 Results

The response to each search request is of type Google.GoogleSearchResult, which is a structured type. Items of this structure are displayed in Table 4.4.

Type	Item	Semantics
bool	**documentFiltering**	Has the search been filtered?
string	**searchComments**	Such as “Stop words were removed from the search.”
int	**estimatedTotalResultsCount**	The estimated total number of results that **exist** for the query.
bool	**estimateIsExact**	Is the estimate the exact value?
Google.ResultElement[]	**resultElements**	An array of result items.
string	**searchQuery**	The value of **q** parameter in the request.
int	**startIndex**	1-based index of the first result in **resultElements**.
int	**endIndex**	1-based index of the last result in **resultElements**.
string	**searchTips**	Suggestions on how to use Google.
Google.DirectoryCategory[]	**directoryCategories**	An array of directory categories corresponding to the results.
double	**searchTime**	The total server time in seconds.

Table 5.4: Google.GoogleSearchResult structure

The Google.ResultElement type is the actual result with document snippet and URL. It is also a structure consisting of the following elements:

Type	Item	Semantics
string	**summary**	If the result is in a directory category, the directory summary appears here.
string	**URL**	The absolute URL of the document.
string	**snippet**	A snippet (extract) which shows the query in context of the document where it appears.
string	**title**	The title of the search result
string	**cachedSize**	Size in kB of the cached version of the URL.
bool	**relatedInformationPresent**	Is the *related:* query term supported for this URL?
string	**hostName**	When filtering enabled the second result from the same host includes its host name.
Google.DirectoryCategory	**directoryCategory**	The result's directory category.
string	**directoryTitle**	The corresponding directory title.

Table 5.5: Google.ResultElement structure

5.7 Conclusion

Google Web APIs services are a very powerful tool for automatized querying the most effective search engine on the Internet. The current limitations of the services provided (the maximum number of 1 000 results a day is of most concern) are expected to be deprecated soon. The greatest asset compared to creating requests as part of URLs' queries and thus connecting to a standard Internet resource (an example might be connecting to http://www.google.com/search?q=graphics) is not having to parse the HTML code of the response. Instead, the response is an object with an exactly defined structure, which enables to process results very fast. Likewise, there is no problem with constructing the URLs. Inputs for the search engine are simply parameters of a method.

6 CiteSeeker - Design

6.1 Problems

The core of *CiteSeeker* is a Web crawler, thus the first obvious problems are related to the Web structure. A sample structure is depicted at Figure 6.1. In fact, it is a directed graph, but the directions are omitted here for simplicity. Each server is a graph of documents. Dashed lines are links which may introduce loops inside a server as well as between servers when directed accordingly. The optimal case for *CiteSeeker* is to traverse a server's documents as a tree.

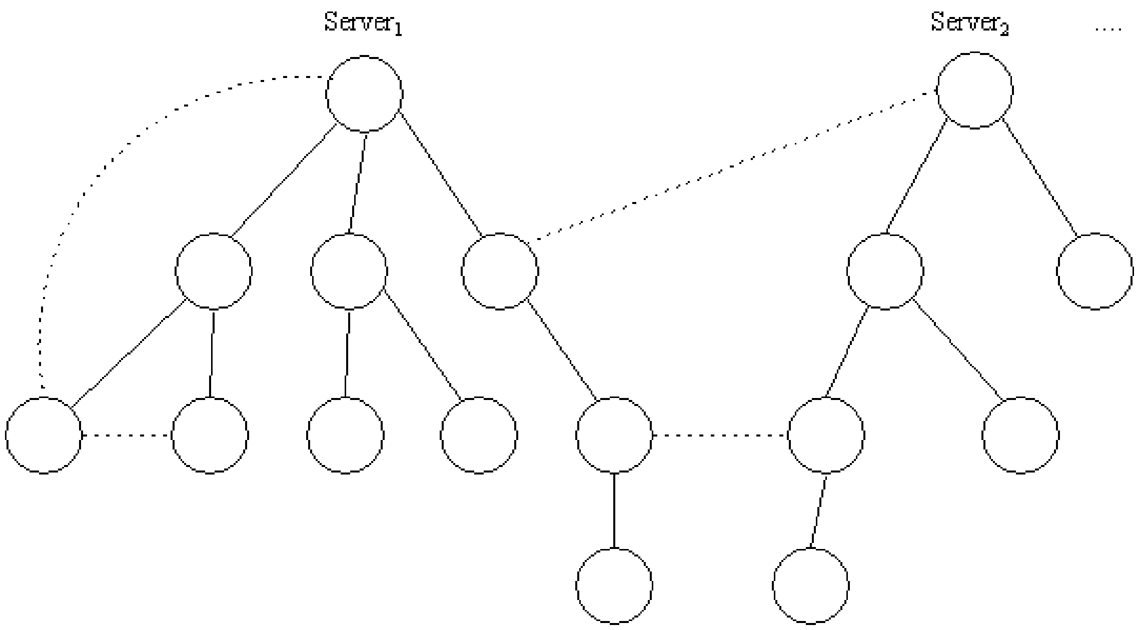
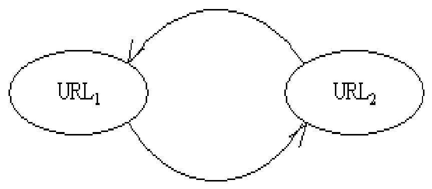


Figure 6.1: Web structure

CiteSeeker is not aimed to process the retrieved information any further (such as building a database upon it), which would require additional hardware and software resources and would be a complicated task. There is no need to store the documents searched, which implies no difficulties with insufficient (static) memory as to the contents of the documents themselves. The information retrieved should have a very simple form and hereby it should enable some kind of postprocessing.

But the obvious trouble arises. As shown at Figure 6.1, two documents on the Web may link to each other, which would lead to an indefinite loop for the crawler as it moves forward using references to other files. Such a little loop may be easily discovered but there might occur loops including thousands of documents (see Figure 6.2).



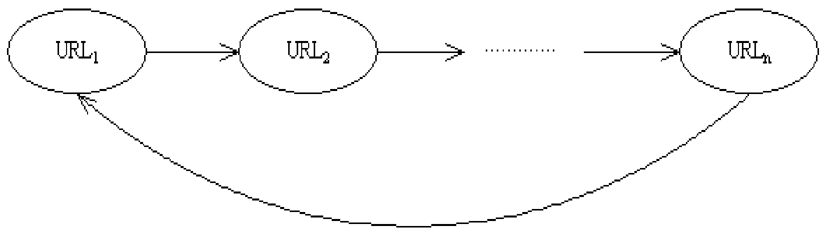


Figure 6.2: Document loops

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.), it will, theoretically, have searched all the documents on the Web by some time point. (In reality it is impossible because documents are dynamically created and removed and some of them may not be in the link structure at all.) If we take the lower bound of Web size from Section 3.1.1 for granted, then there are about 10^{10} URLs, each identifying one document. Suppose a URL is a 50 B string in average. Then the total space required to store these URLs is

$$10^{10} \text{ URL x } 50 \text{ B} = 5 \text{ x } 10^{11} \text{ B} \approx 500 \text{ GB} \qquad (6.1)$$

Thus, to keep track of as many documents searched as possible URLs have to be managed in a more economical way. Besides space difficulties, there is a minor problem with ambiguity of the system inputs. The initial conditions (names of authors or papers) may be incomplete or inaccurate, which can be solved using standard methods introduced in Section 4.

6.2 Avoiding loops

6.2.1 Data structures

The entire organization is depicted at Figure 6.3. A few data structures have to be introduced: *Pending Servers*, *Pending Documents*, *Completed Servers*, *Completed Documents*. The terms need to be explained.

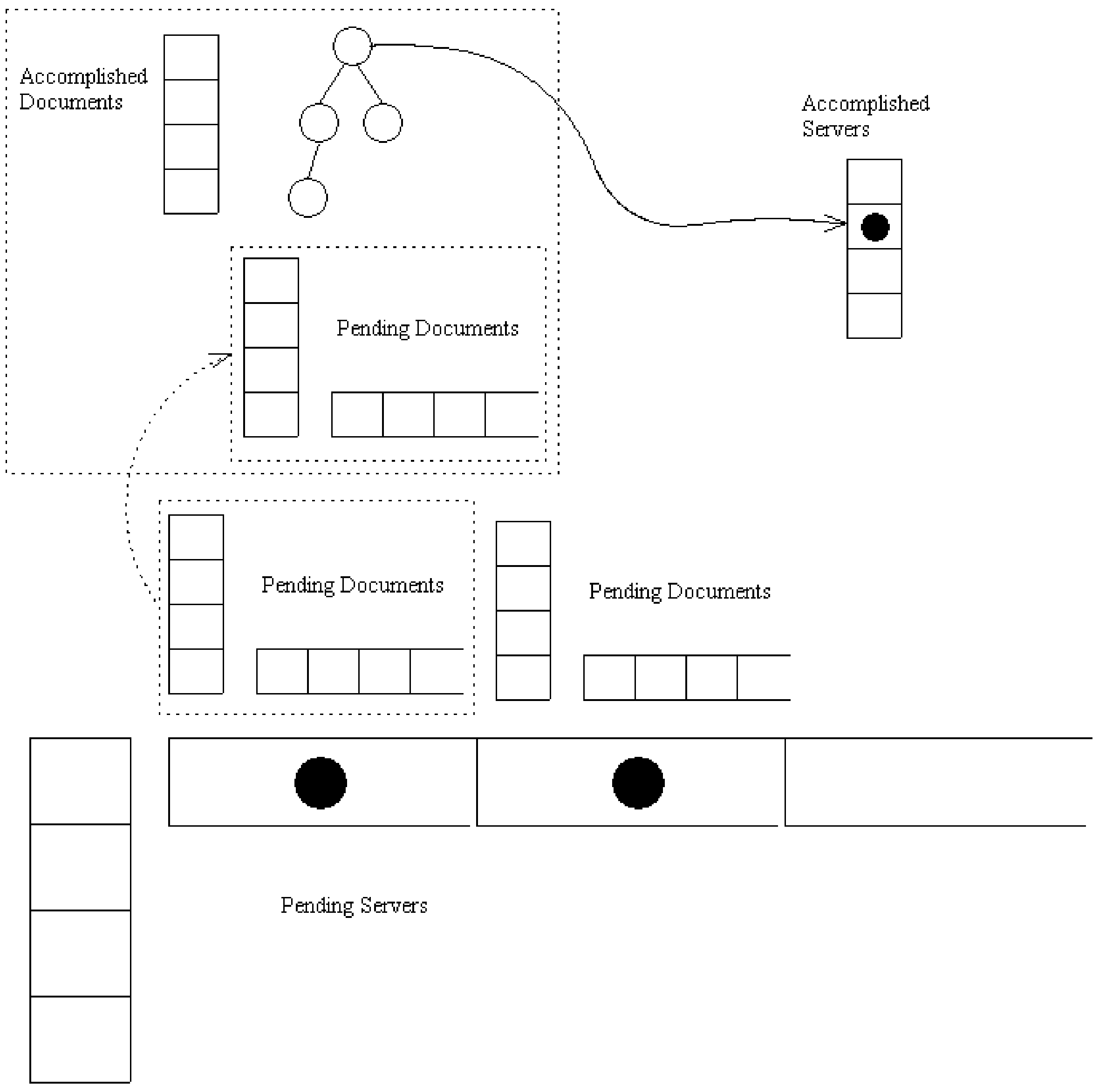


Figure 6.3: Fundamental data structures

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.

6.2.2 Activity

A typical procedure of *CiteSeeker* activity concerned with finding as many documents as possible (a kind of Web crawling) looks like this.

1) 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. Even if the user sets a particular document as a root, only its server will be considered as a start point. Yet the document itself will still be taken into account via *Pending Documents* as described further.

2) 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 *d*ocuments of "their" server in the *Pending Servers* queue provided there is one. In the opposite case, the server is created and enqueued, first.

3) Having been handled, each document (its URL) from the current server is added to the *Completed* (accomplished) *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 6.4.

4) 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 accomplished servers table, it will be ignored.

6.3 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 *d*ocuments 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:

$$\begin{aligned}\text{total URLs} = {} & \text{pending servers} + \text{completed servers} + \\ & \text{pending documents} + \text{completed documents}\end{aligned} \tag{6.2}$$

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 3.1.1. Again, if a URL is a 50 B string then the memory requirements to store all of them are approximately

$$(K_1 \text{ x } 10^6 + K_2 \text{ x } 10^6 + K_3 \text{ x } 10^6 + R) \text{ URL x } 50 \text{ B} \approx 500 \text{ MB} + R \text{ x } 50 \text{ B} \qquad (6.3)$$

where K_1 is the number of pending servers in millions, K_2 is the number of completed servers in millions, K_3 is the number of completed documents in 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 (6.1). 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 got 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.

6.4 Documents tree

Now that there are no loops which could trap the crawler another problem arises: 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 [8, 9]. 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 [8] and running out of memory would be very fast. Figure 6.4 shows a binary tree and the order in which nodes are traversed with breadth-first search. If the current node is 6, only nodes 7 – 13 are in memory (in the queue). The numbers on the right-hand side count nodes at the corresponding level.

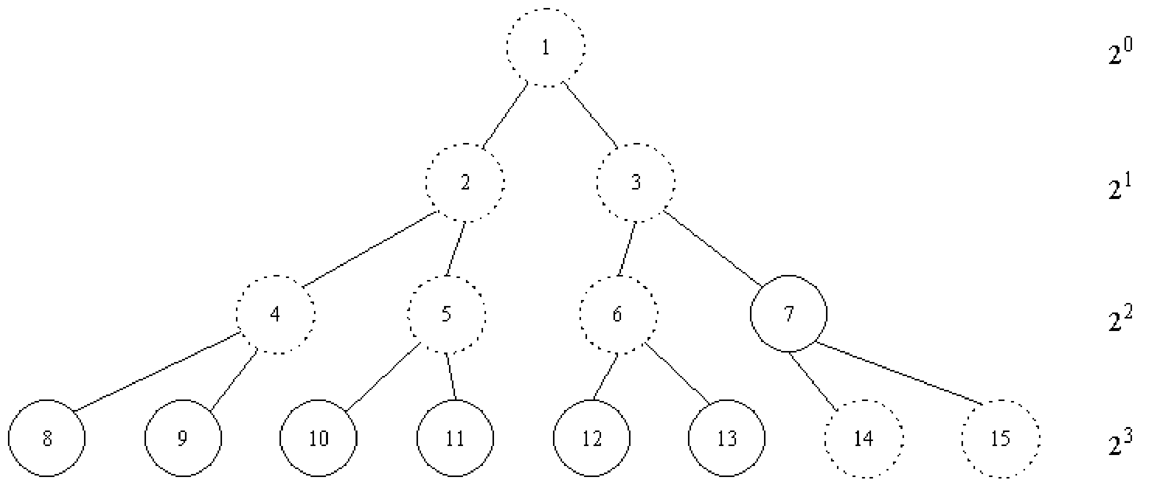


Figure 6.4: Breadth-first search

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 as shown at Figure 6.5 – the nodes on stack and the nodes referenced by them. The nodes traversal order (and their number at the same time) is stated in their middle whereas the removal order is next to them. At the point that the node 4 is being searched, nodes 7, 8 and 10 are not in memory. On tracking back to the root node 1, nodes 4, 3, 5 and 2 are released. They are no more useful. At this stage only nodes 6 and 9 have not been searched yet. 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.

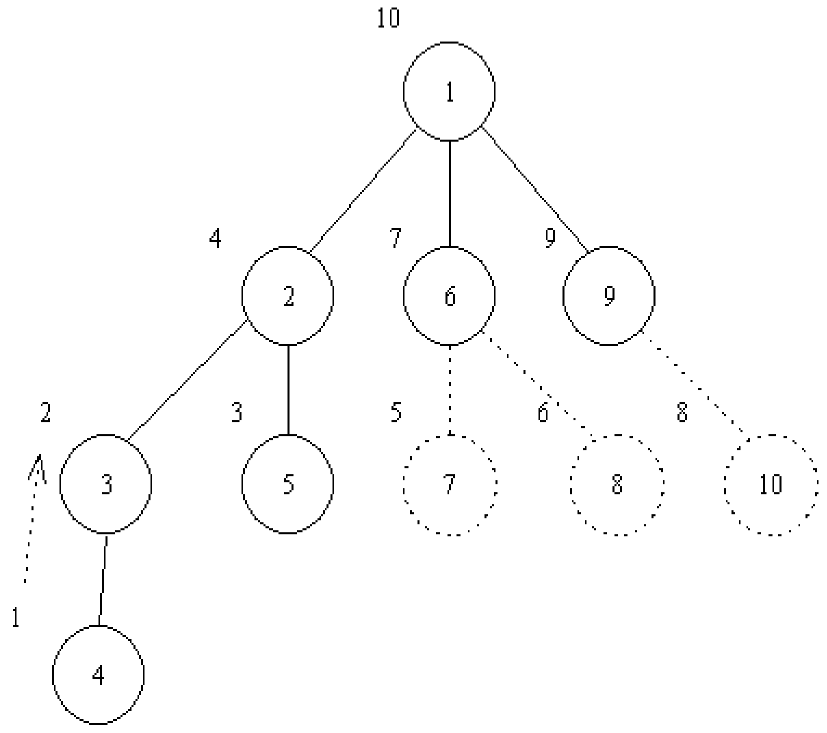


Figure 6.5: Depth-first search

Although the time complexity in either case is O(n) where n is the number of nodes [8], 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.

6.5 Object decomposition

After making the previous analyses it is rather straightforward to break the problem statements down into objects and to determine their rough static and dynamic behaviour which will be described using UML (Unified Modelling Language) diagrams.

6.5.1 Static behaviour

The basic classes breakdown in terms of a UML class diagram is depicted at Figure 6.6. Exception classes and GUI classes are left out. Only the most important attributes are shown, methods are hidden. Note that the blue classes have no dependency on others. They have only static methods, i.e. they are never instantiated.

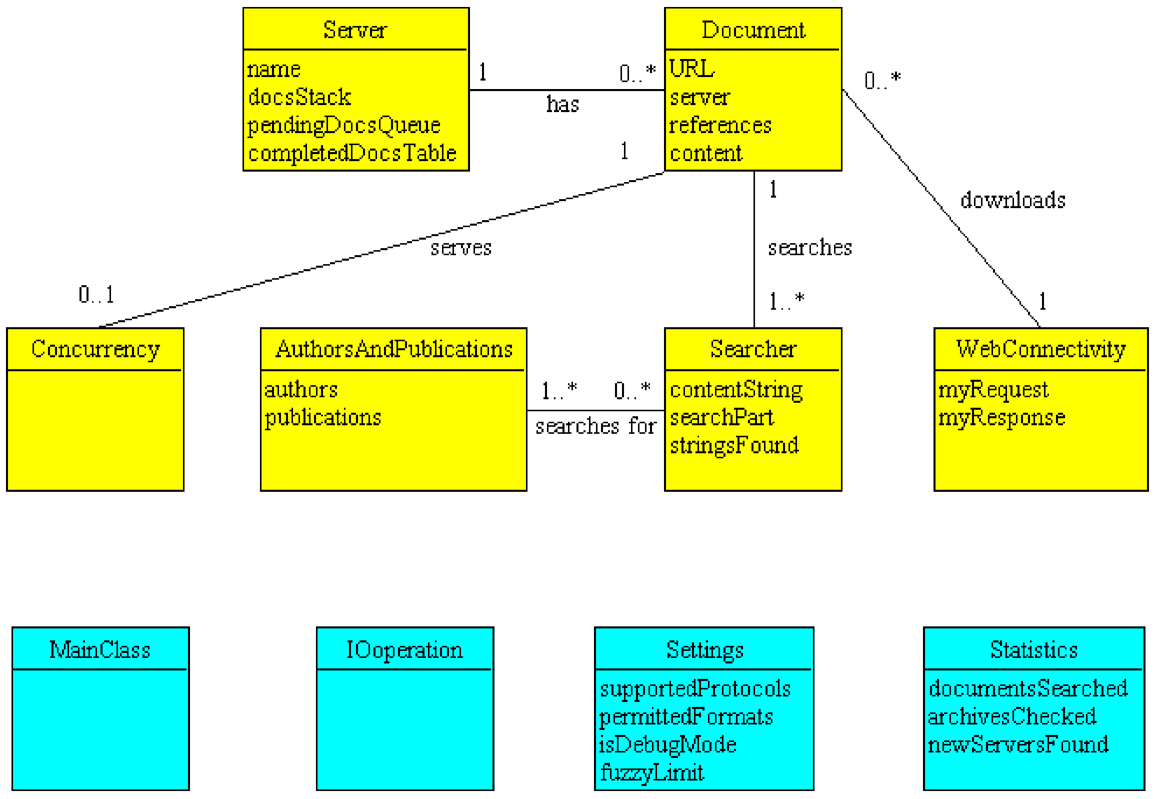


Figure 6.6: Simplified class diagram

6.5.2 Dynamic behaviour

Figures 6.7 – 6.11 depict state diagrams of the individual classes' objects. Figure 6.12 shows a typical scenario in a sequence diagram. Note that the class AuthorsAndPublications represents rather a *struct* data type. Therefore, an object of this class has no dynamic behaviour.

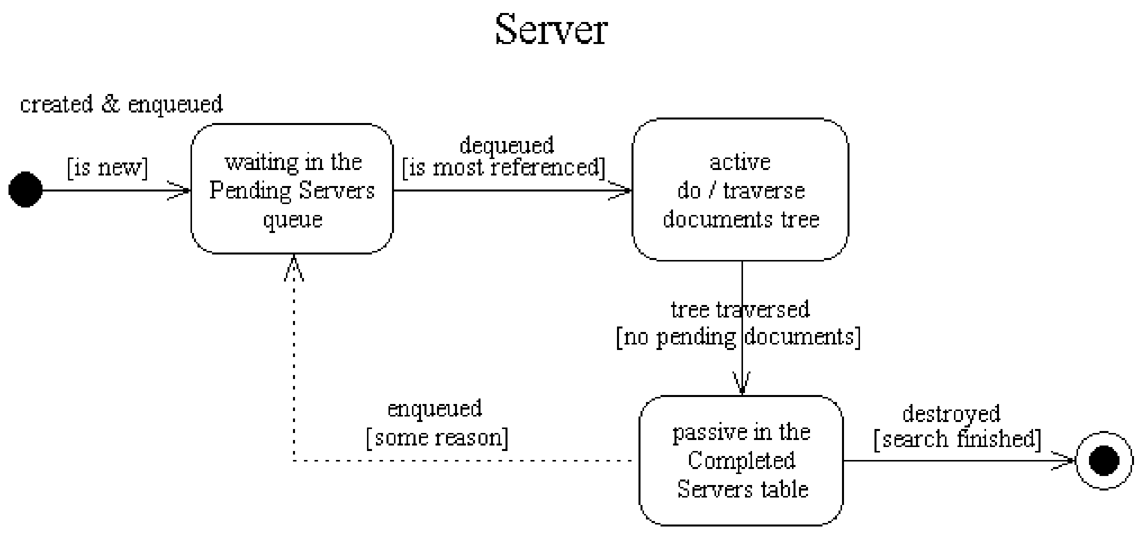


Figure 6.7: Server state diagram

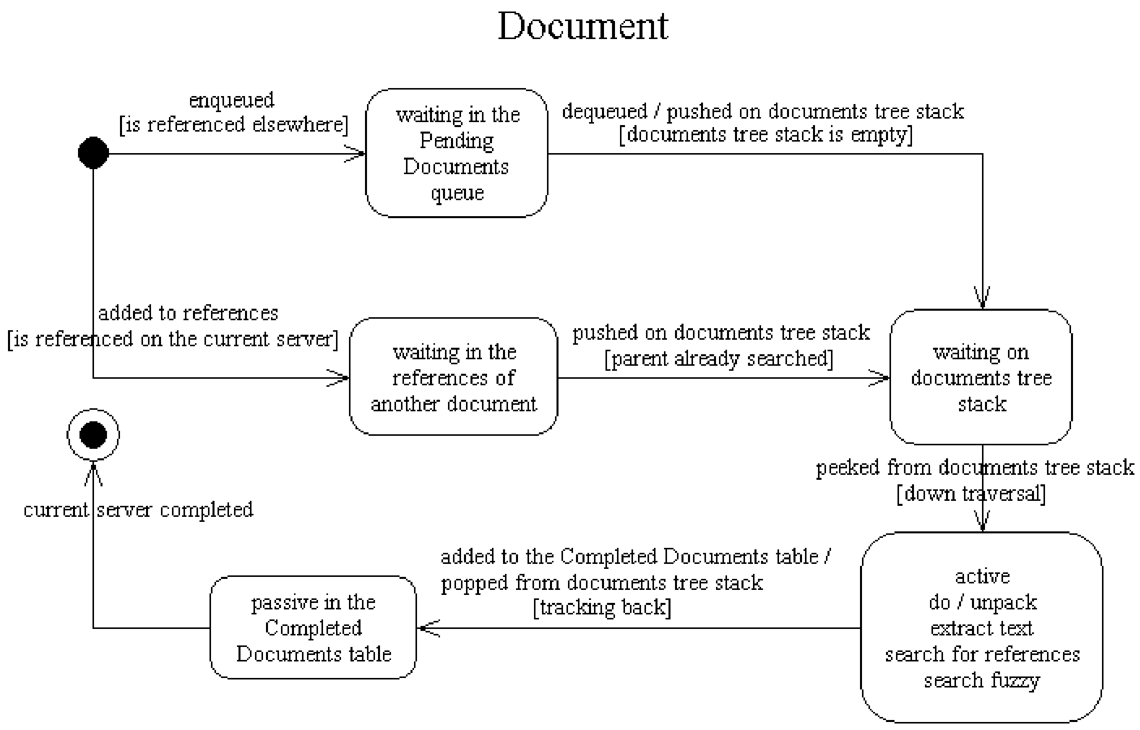

Document
enqueued
[is referenced elsewhere]
waiting in the Pending Documents queue
dequeued / pushed on documents tree stack
[documents tree stack is empty]
added to references
[is referenced on the current server]
waiting in the references of another document
pushed on documents tree stack
[parent already searched]
waiting on documents tree stack
peeked from documents tree stack
[down traversal]
current server completed
added to the Completed Documents table /
popped from documents tree stack
[tracking back]
passive in the Completed Documents table
active
do / unpack
extract text
search for references
search fuzzy

Figure 6.8: Document state diagram

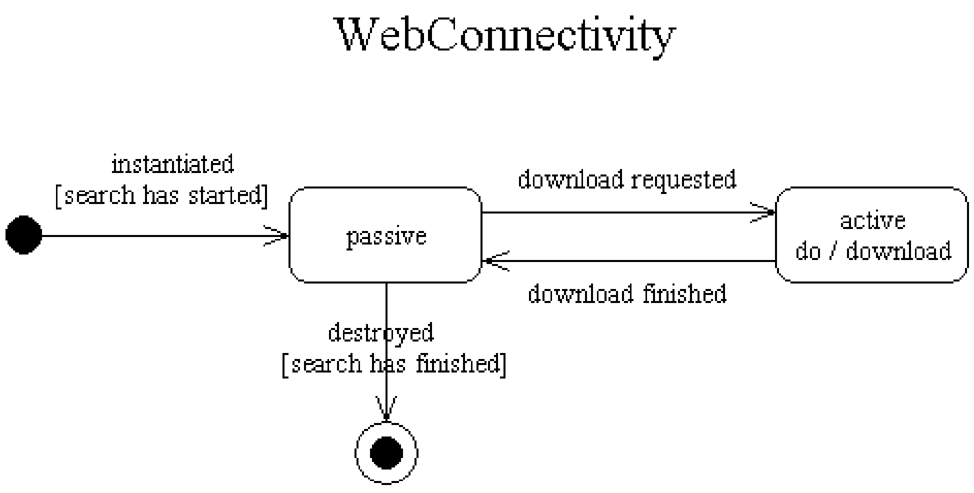


Figure 6.9: WebConnectivity state diagram

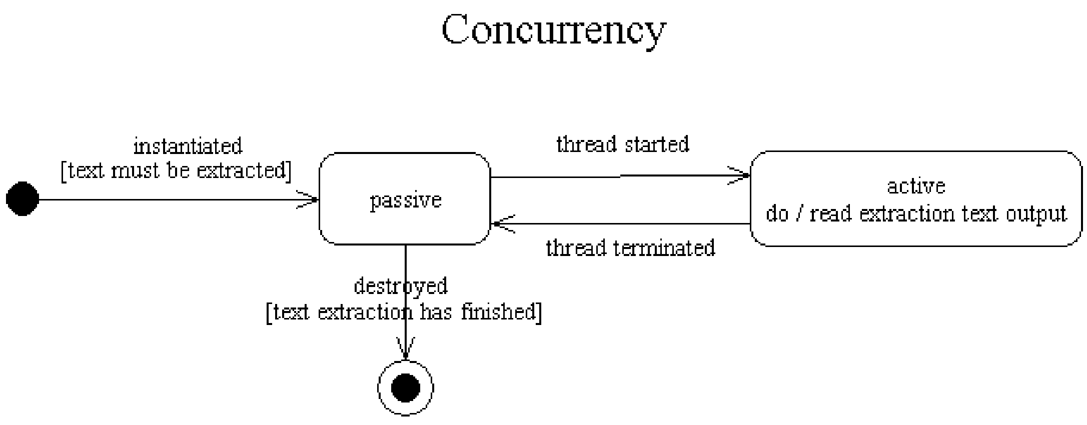


Figure 6.10: Concurrency state diagram

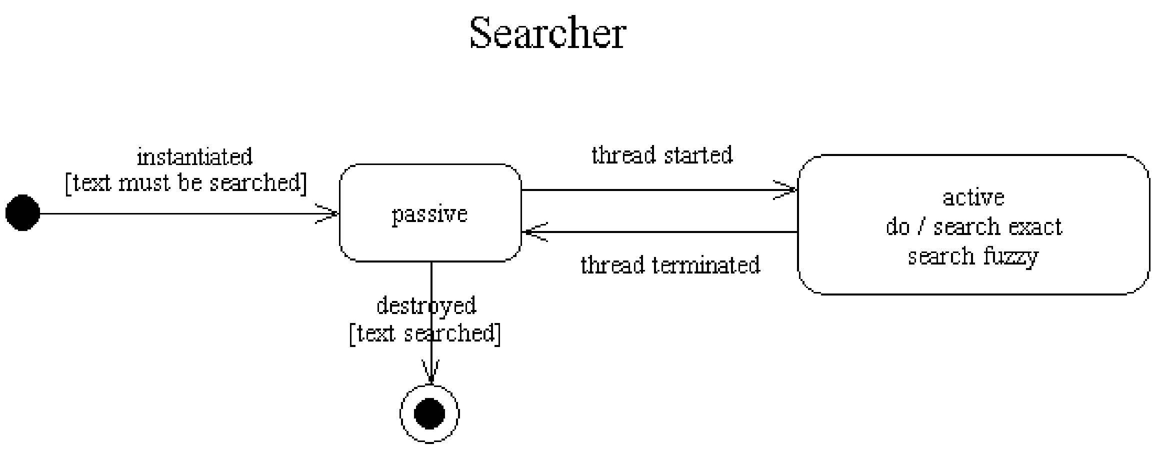


Figure 6.11: Searcher state diagram

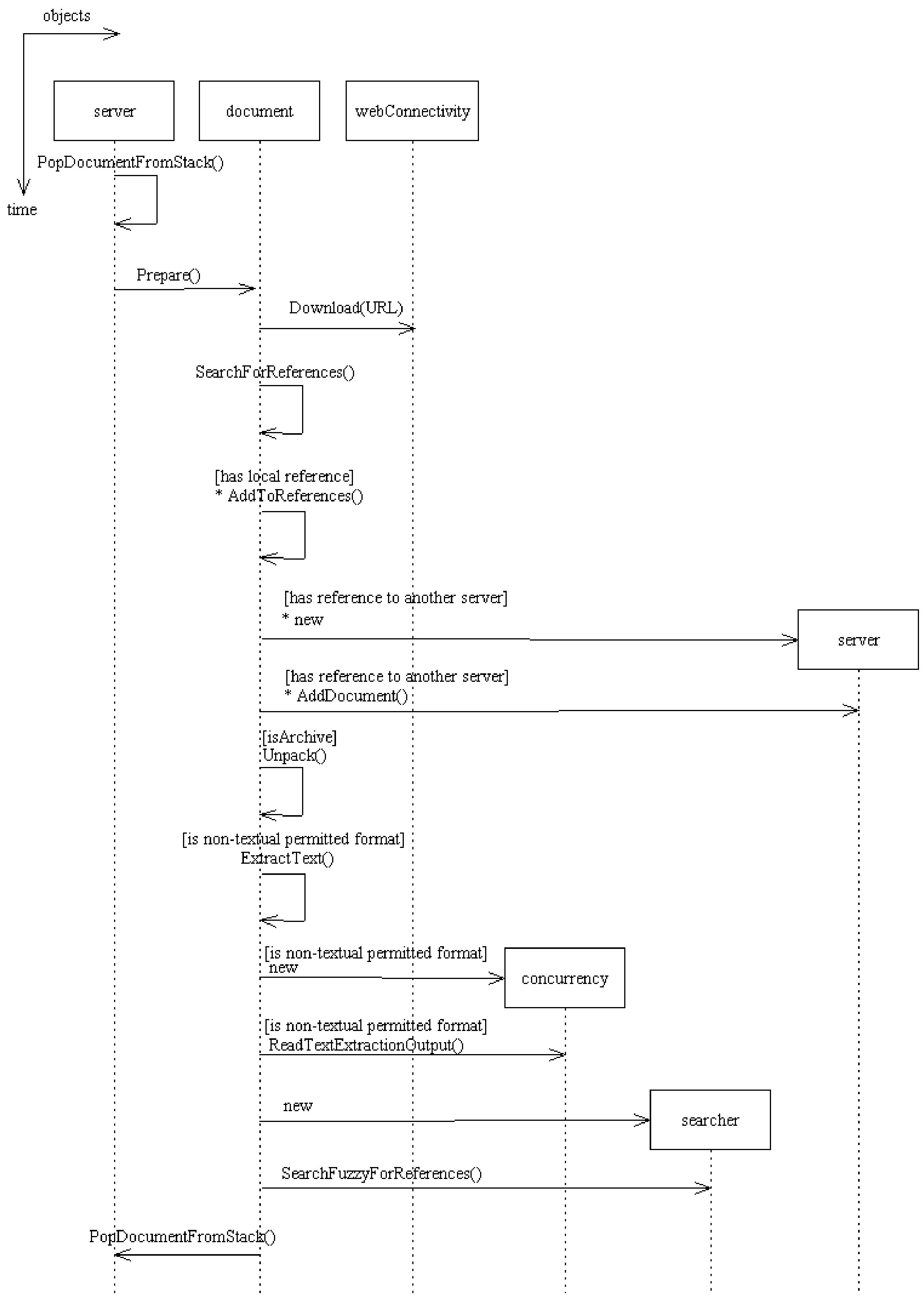


Figure 6.12: Sequence diagram of a typical scenario

7 CiteSeeker - Implementation

7.1 Structure of C# project

The CiteSeeker project in C# consists of two namespaces:

- CiteSeeker
- Google_Web_APIs_Demo.Google

Google_Web_APIs_Demo.Google's source code was obtained from [21]. It provides functionality related to communicating with Google via its Web APIs services. See Section 5 for more information on this.

The CiteSeeker namespace is spread across the following source code files:

- AboutForm.cs
- AssemblyInfo.cs
- Concurrency.cs
- Document.cs
- Main.cs
- MainForm.cs
- MemoryUsageForm.cs
- SettingsForm.cs
- Statistics.cs

It encompasses 18 public classes summarized in Table 7.1.

Category	Classes
main functionality	AuthorsAndPublications Concurrency Document MainClass IOoperation Searcher Server Settings Statistics WebConnectivity
graphical interface	AboutForm MainForm MemoryUsageForm SettingsForm
exceptions	InvalidReferenceException FileEmptyException SearchSuspendedException SkipCurrentServerException

Table 7.1: Classes of namespace CiteSeeker

7.2 Non-C# components

CiteSeeker uses a number of existing utilities. See Appendix B for their exact names and versions. The completely external tools are:

- Ghostscript
- Pstotext
- Pdftotext
- Pkunzip
- Gzip
- Tar

Ghostscript is used indirectly via Pstotext and Pdftotext to extract text from PS and PDF files. Pkunzip, Gzip and Tar help unpack compressed files. Some of these programs enable flushing the files they should process into their standard input, others do not. *CiteSeeker* must also take into account that PDFs, for instance, require random access and therefore cannot be accepted through the standard input by the text extraction utilities. All programs allow redirecting their standard output, but there were problems with DOS-based programs (Pkunzip, Tar) where the pipe between them and the .NET-based application (*CiteSeeker*) had an unreliable behaviour in both directions. In these cases temporary text files were used for both inputs and outputs. Otherwise, when it was possible, pipes were created so as to speed up the external communication.

CiteSeeker's sources also include files fstrcmp.c and fstrcmp.h that were compiled with Borland C++ 5.5.1 compiler into **fstrcmp.exe**. Compare with Section 4.3. *CiteSeeker* passes two arguments on the command line to **fstrcmp.exe**: name of the input and output file. The input file contains three values in accordance with Section 4.3. From the output file *CiteSeeker* will learn whether and what similar string was found and what the similarity is. Both of the files are temporary – they are removed after each *CiteSeeker* – fstrcmp session.

7.3 Classes

In this section the functionality of all classes from the first category in Table 7.1 (except AuthorsAndPublications, which is rather a *struct* type) will be explained. The classes with GUI (Graphical User Interface) will be omitted as well as the exception classes, all of which are derived from the standard Exception class of .NET Framework. The methods whose names are not preceded by a class name are methods of the class in the section title.

7.3.1 MainClass

This class has only static methods. Their description follows.

Start()
It is a start point for each search. The main working thread (not serving GUI) runs this method. After the method has finished, the main working thread terminates. This method creates the WebConnectivity object that will persist until the current search finishes or suspends. It calls all methods reading input parameters and then invokes *IterateThroughAllServers()* for a new search or *ResumeSearch()* to resume a suspended search.

IterateThroughAllServers()
It iterates through all servers in the queue (*Pending Servers)*. It calls *Server.PopMostReferencedServer()* to pick up the server with the most references. It invokes *Server.SearchDocuments()* upon each server dequeued.

ReadStayWithinDomains()
It opens the file INPUTS\staywithindomains.txt if it exists and reads its content **case sensitively**. The valid strings (URLs or "wildcard" URLs) will force *CiteSeeker* to search only those URLs that are in accordance with them.

ReadForbiddendDomains()
It opens the file INPUTS\forbiddendomains.txt if it exists and reads its content **case sensitively**. The valid strings (URLs or "wildcard" URLs) will force *CiteSeeker* to avoid the corresponding URLs. Forbidden domains are taken into account only and only if there are no stay-within domains.

LoadData()
If a suspended search is resumed, this method is invoked and, in turn, it calls *IOoperation.LoadDataStructures()*, in which the appropriate data stored on disk is loaded into memory, and it returns the current server.

ReadStartPoints()
It opens the file INPUTS\startpoints.txt (it must exist) and reads its content **case sensitively**. Yet the file can be empty. The start points are read from file but they can be obtained from Google via *IOoperation.SearchWithGoogle()* as well if the user turns on this feature (see Appendix A). The Google start points are delimited from others by "BEGIN OF GOOGLE OUTPUT" and "END OF GOOGLE OUTPUT" strings in the output window.

ReadSearchStrings()
It opens and parses the file INPUTS\searchstrings.txt and places the search strings (authors and publications – their format will be described later) into the appropriate data structures calling *InitializeKeywords()*. The file has to exist and has to contain valid values. The content of the file is transformed into lower case so the search strings are **case insensitive**!

ResumeSearch()
It starts the search with the current server, i.e. it calls *Server.SearchDocuments()* upon the current server acquired from *LoadData()*.

Initialize()
It makes all necessary initializations when the search begins such as resetting statistics, loading default settings values and setting menu items.

Finalize()
It frees resources when the search has finished or has been suspended. In case of the latter it also stores *Pending Servers* queue and *Completed Servers* table disk so that the search could be resumed later. (*Pending Servers* table need not be stored, it would be rebuilt from the queue then.) At last, it clears all of the three global data structures above, it resets menu items and objects associated with Settings class.

InitializeKeywords()
It parses the file with search strings and makes lists of authors and publications out of them. These are then put into a list of objects of class AuthorsAndPublications, which is made public for others via Settings class. If the class has no valid inputs, an exception is thrown.

ContainsOnlyDelimiters()
It is a helper method for the previous one which helps determine valid inputs. It returns true if a string is composed only of the characters specified.

7.3.2 IOoperation

This class provides static methods that write to or read from files, write to the output window, which the user can immediately see, or communicate with Google.

WriteLine()
It writes a line of text to the text box of the main window and also into the debug file (LOGS\debug.log) provided the debug mode is turned on. Both of the output "devices" are locked before to ensure that no other thread could write there at the same time.

WriteError()
It writes a line of text (error message) into the errors file (LOGS\errors.log). This file is made thread-safe before as well.

CountTime()
It always calls *Statistics.CountTime()* to determine the time elapsed between two points and it prints the result on the screen and possibly into the debug file.

CountDocs()
It invokes *Statistics.CountDocs()* to print miscellaneous statistics about documents processed. It also increments a count to store global objects on disk (into DATA\currentServer.dat, DATA\completedServersTable.dat and DATA\pendingServersQueue.dat) at the intervals specified by *Settings.flushInterval* and to write their size into the debug file.

BuildGoogleQuery()
The query for Google is formulated. here and returned as a string. The query terms are either input by the user in the *Advanced/Settings/Google Query* form or the default ones are used.

SearchWithGoogle()
In invokes Google with the query from the previous method and returns the results (URLs) as an array of start points. The number of results Google should return is set in Settings. Compare with Section 5.3.

SetStartPoints()
Start points for the search are organized accordingly. The pseudo-code algorithm of how this is done is stated below in Algorithm 7.1.

```
for each start point
{
        parse its URL;
        if URL is nonsense, continue with next start point;

        split URL into protocol  + server and path + file;
        if URL's protocol is not supported or it contains not permitted domain or path or
                file format is not allowed
                        continue with next start point;

        if server is not in Completed Servers table
        {
                if server is not in Pending Servers table
                        place it into both;
                if URL is not server name itself
                        add it as document into server's Pending Document queue & table;
        }
}
```

Algorithm 7.1: Setting start points

ParseUri()
It parses a URL and returns it as a string, in which the protocol and host (server) are in lower case and the path and file remain unchanged. URLs are, in general, case sensitive. It is merely up to the Web server's operating system whether or not it distinguishes between lower and upper case. For instance, UNIX/Linux does so while Windows does not.

GetStrings()
It opens a specific file and returns an array of strings from its content. Each line beginning with '#' (with no white space before it) is considered a comment and is left out. The other lines are trimmed (white space is removed) and the empty lines are omitted, too. If necessary, all the strings are transformed into lower case, or invalid URLs are thrown away.

BuildPendingServersTable()
The *Pending Servers* table (a hash table with keys equal to server names) is constructed upon the *Pending Servers q*ueue.

LoadDataStructures()
It reads binary data from disk and makes memory objects out of it (a process called *deserialization*, the opposite is *serialization*). The data on disk is in the files DATA\currentServer.dat, DATA\completedServersTable.dat and DATA\pendingServersQueue.dat and all of the global objects *Pending Servers* queue, *Pending Servers* table (via *BuildPendingServersTable()*) and *Completed Servers* table as well as the current server are instantiated. The table of pending servers could not be simply loaded from a file because the server instances after deserialization would be distinct from those in the queue. This method is called when a suspended search is to go on and returns the current server.

Serialize()
It saves an object in the appropriate file. Strictly said, a graph of objects is stored as all other objects referenced by this object are saved too. The resulting file would not necessarily have to be binary, it might be in a human-readable format such as XML as well. *CiteSeeker* uses only binary serialization because the size of the file (or stream) is the approximate size of an object in memory.

7.3.3 WebConnectivity

This class implements the basic Internet connectivity. As C# and .NET Framework have a strong support of Internet and Web services related issues, it may all be done within a couple of lines of code.

CopyStreamIntoByteArray()
The downloaded stream is copied into an array of bytes. This array is the content of a document and will be searched in due course. As this array is static, its size must be determined beforehand. If the length of the downloaded stream is, for some reason, not known from the HTTP header, the size of the array is set to ten million bytes. Of course, if a document is smaller, it is copied from the original array to another array, which is sized accordingly.

ConnectToNet()
It makes a Web request and gets the response. In C# it is very easy, no knowledge of Internet protocols is needed:

```
// opens connection and gets response stream
WebRequest myRequest = WebRequest.Create(URL);
WebResponse myResponse = myRequest.GetResponse();
Stream myStream = myResponse.GetResponseStream();
```

ReleaseResponse()
It frees resources associated with the Web response. Since the WebConnectivity object persists during the search, the Web response is released as soon as its stream is copied to a byte array.

7.3.4 Server

It is the class representing servers and its relation with the Document class is fundamental for understanding the process of Web crawling (see Table 6.6). 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 via its public static attributes (static attributes and methods are not owned by a particular object instance, but they are owned by the class itself). These attributes are:

- *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.

It should be mentioned that all of the high level data structures above (queue, hash table, stack) are imported from the .NET Framework class library. This approach makes the programmer not worry about their implementation, but on the other hand the behaviour of objects of these classes is fully transparent in some cases from the programmer's point of view. For example, a hash table has some initial capacity which is made equal to the smallest prime number larger than twice the current capacity each time the hash table is occupied to some specified extent. After creating a hash table, there is no way how the programmer could change the capacity (i.e. the number of clusters – buckets – where synonyms, objects with keys producing the same hash code, are stored). But unless the programmer wishes to make some optimizations as to the size of objects, the library classes are very comfortable. Hash tables usually provide an O(1) access to its objects. This is very useful in accessing a particular object in the underlying queue or in finding out whether some object is in the queue.

AddDocument()
A document is added to the server. It means that its URL is enqueued in the *Pending Documents* queue and added to the *Pending Documents* table provided it is neither in the *Pending Documents* nor in the *Completed Documents* tables. In any case, the count of how many times the server is referenced from others is incremented.

PopMostReferencedServer()
This is a static method. It dequeues the most referenced server from the queue of pending servers in O(N) time where N is the queue length.

WaitOnPause()
This static method makes the current thread wait until the user clicks *Search/Pause* and terminates the break.

SearchTree()
It traverses the documents tree iteratively (non-recursively) using a stack and it invokes the actual search method (*Document.Search()*) upon each document it finds. It maintains the structure of the tree that is shown at Figure 6.5. It saves the tree to disk (to DATA\currentServer.dat) at regular intervals.

SearchDocuments()
It dequeues documents from the queue of pending documents and passes them as a root to *SearchTree()*. In case the search is resumed the first document is popped from the documents stack. When all the documents in the queue have been searched, the server's root (default

index file) will be examined. This method also handles the exceptions thrown when the search was suspended (it saves the documents tree) or the server was skipped. See Appendix A for the details of how the user can do that.

CompleteServer()
It “administratively“ completes the server. It adds it to the *Completed Servers* table and removes it from the *Pending Servers* table. (The current server is still in the table of pending servers. This measure was taken with view of more threads searching several servers in parallel. They would then not mix their current servers. This parallelism was not implemented at last.) The name of the completed server is logged to LOGS\completed.log.

7.3.5 Document

Document is a node in the graph such as the nodes at Figure 6.5. 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. Some methods are not commented out, but they are never invoked for various reasons. They will not be stated in this overview. The private attributes are:

- *URL*
- *content*
- *references*
- *server*

URL is the Uniform Resource Locator string of the document. (The more technical term URI – Uniform Resource Identifier should be used in fact), *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. Besides these attributes there is also a static one for a regular expression by means of which the links to other documents in an HTML file will be found. This attribute is static because it is the same for all documents and its compilation takes a while.

Initialize()
It gets the content of the document. It calls *WebConnectivity.ConnectToNet()* and measures the download time. The document (its URL) is then added to the table of completed documents and removed from the queue of pending documents.

Here comes perhaps the trickiest 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. Next time a redirection happens too, but this time it may be to an entirely different page and next time the same. Redirections might be nested as well. This is prone to ending up in an infinite loop of documents, the Web crawler may be “trapped“. If only the original URL is stored, the search engine will never learn the “real“ URL of the resource like in the following example:

original (referenced) URL:	path/
“real” (returned) URL:	path/index.html

The original URL is referenced in a document, the “real” URL is returned by the Web server. If the “real“ URL is referenced somewhere later, it will be accessed and its content (often a directory listing) will be searched again. The path may certainly be a host name and then the situation looks even worse. The current implementation of *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 and *CiteSeeker* is not an exception. The exact method of preventing traps is still an open issue, which will probably have to be solved in the future versions of *CiteSeeker*.

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 this *CiteSeeker* version does not consider queries.

Search()
It processes a downloaded document calling search methods appropriate for the specified file format. It also measures the processing time.

MakeAbsoluteURL()
This method combines a relative URL with its base. URLs in documents may be either absolute (beginning with a protocol) or relative (beginning otherwise) to the server root, current directory, or up-level directories. The relative URLs must be transformed into absolute URLs so that they could be added to the table of completed documents.

IsPermittedFormat()
It returns true if the file in a URL specified has a permitted extension. They can be set by the user in *Advanced/Settings/Search Files*.

CheckWildcardDomains()
It checks whether the server of a document conforms to the "wildcard" stay-within and forbidden domains specified in the corresponding input files. The stay-within domains have a higher priority. The “wildcard“ is the dot at the beginning such as in “.edu“, “.zcu.cz“, etc. If the answer is negative, an exception is raised.

CheckIfPermittedDomain()
The same as before except that the whole URL is compared with full URLs.

ModifyURL()
It throws an exception if the URL is inappropriate (e.g. a mail or news URL). Then it calls *MakeAbsoluteURL()*.

IsSupportedProtocol()
It returns true if a URL's protocol is supported. Currently, these four protocols are supported: "http://", "https://", "file://".

RemoveFragmentsFromURL()
Fragment and query components are removed from the URL.

CheckIfPermittedFormat()
It throws an exception if a URL contains a file with an extension that is not permitted. There is a problem with files that have no extension. There is no way of finding out in advance if the last level of the path is a file with no extension or if it is just a directory with no slash ('/') at the end. This simple heuristics is used: If the bottom path level contains a dot ('.'), it is considered a file.

ExamineReferencedURL()
Each URL referenced in the document is examined here. The URL is transformed into the canonical format (e.g. '%20' changes to ' ' or the host name is converted to lower case, etc.) and the preceding methods are invoked. This method returns an element from the enumeration {OnTheServer, Elsewhere} according to where the URL belongs. If it belongs to another server (elsewhere), it is added to the queue and table of pending documents on that server. If the server object does not exist yet, it is instantiated and enqueued in the *Pending Servers* queue and added to the *Pending Servers* table. Of course, the server must not be completed (located in the *Completed Servers* table).

ScheduleSearchForKeywords()
It starts the threads that will be searching for citations. In this version of *CiteSeeker* only one thread is created for the reasons stated in Section 7.3.7. It waits until the search thread terminates, measures the search time and logs the result to LOGS\success.log in case a citation was found.

ReduceWhiteSpace()
This method reduces all white space ("invisible" characters) in a string into a single space. It is important to evaluate strings such as "white space" and "white   space" as equal.

GetStringFromContent()
It converts a byte array (the document's content) to a string.

IsSafeURL()
It indicates whether the specified URL is "loop-safe". That means that suspiciously long URLs or URLs with too many same segments in a row are skipped. The corresponding constants are set in Settings class.

AddToReferences()
It adds a URL into the list of references (and to the table of pending documents) provided it has not been referenced yet (it is not in the table of pending documents) nor is it waiting in the queue of pending documents. It is an ethical problem whether or not to add to references also

the directories from the URL's path. They are probably not public if they are not referenced themselves (there is no path to them from the root). Therefore, the code that would do that is commented out (the same in *Server.AddDocument()*). Since the last stage in searching a server is constructing a tree from its root, it ensures that all the documents reachable from the root will be searched.

FindZipFileNames()
It returns a string array of names of the files that are packed in a ZIP archive. The output from Pkunzip is read from a temporary file (in TEMP directory) because the redirection of Pkunzip's standard output did not work in the Windows version of *CiteSeeker* (but it did in its console version).

UnzipPdf()
It unzips a PDF file from a ZIP archive to the DOWNS directory via Pkunzip without restoring the possible directory structure. PDF files require random access (see Section 2.3), they cannot be flushed into the standard input of text extraction utilities. Therefore they are unpacked directly to disk.

UnzipPs()
It unzips a PS (or any other except for PDF) file to the standard output and returns it as a string. The output of Pkunzip must also be read from a temporary file.

WriteContentToFile()
It writes the document's content (a byte array) to a specific file in the directory DOWNS.

Unzip()
It calls Pkunzip to unpack an archive. Further methods are then called to process the files extracted – extract text from them and search them. Pkunzip cannot read from the standard input, so the archive must be created on disk. It is taken into consideration as well, that Pkunzip 2.50 does not extract long file names in Windows 2000 (unlike in Windows 95). The rough algorithm employed in this method is shown below in Algorithm 7.2

MakeShortFileName()
It makes a file name comply with the old DOS length restrictions – the name will have eight characters at most and the extension three. It also removes the possible path. This method is used in the preceding one.

FindTarFileNames()
Analogy to *FindZipFileNames()*. It returns a string array of names of the files that are packed in a TAR archive.

UntarPdfPs()
It unpacks one specific file via Tar from a TAR archive to the DOWNS directory while restoring the directory structure. No distinction is made between PDF, PS and other files.

Untar()
It treats a TAR archive in a way similar to that in *Unzip()* and Algorithm 7.2. The difference is that Tar can work with long names and it restores the directory structure which must then be erased. If the TAR archive is combined with GZIP it is automatically filtered through Gzip.

```
{
    update archive statistics;

    // saves archive to disk
    write document's content to file;

    find names of files in archive;

    for each file in archive
    {
        // user can interrupt unpacking
        if search is paused wait until pause terminates;

        if file is directory or has forbidden format
            continue with next file;

        // PDFs must be read from disk
        if file is PDF
            unzip file to disk;
        else
            unzip file to memory;

        if unpacking went wrong
            continue with next file;

        // PDFs and PS are considered as tree leaves
        // therefore they are not searched for references to other documents
        if file is PDF
            extract text from PDF and search it for citations;
        else
        {
            // document's content is refreshed
            make byte array from unzipped file in memory;
            if file is PS
                extract text from PS and search it for citations;
            else
                search file for citations and for references to other files;
        }

        if searching went wrong
            continue with next file;

    }   // for each file in archive

    delete archive;
}
```

Algorithm 7.2: Unzipping an archive

GunzipPdf()
It unpacks a PDF file from a GZIP archive to disk. The archive is removed automatically.

GunzipPs()
It unpacks a PS (or any other except for PDF) file from a GZIP archive to the standard output and returns it as a string. The archive is deleted explicitly.

Gunzip()
GZIP archives contain only a single file with no directory structure (see Section 2.4). Therefore, the algorithm of processing this kind of archives differs from Algorithm 7.1 in that

the main loop (for each file in archive) has one iteration only. The name of the file in the archive is determined form the archive header. Gzip can read from the standard input, but there were problems with the .NET Framework StreamWriter class, so every GZIP archive is saved to disk before.

ReadFromFileToContent()
It is the reverse action of *WriteContentToFile()*. A disk file is read into the document's content.

SearchPs()
It extracts text from a PostScript (PS) file using Pstotext and searches it for citations. For simplicity, PS and PDF files are considered leaves in the tree of documents and they are not searched for references to other documents. The standard input, output and error output of Pstotext are redirected. The PS file (in 7-bit ASCII) is flushed to the standard input. The standard output (with the text extracted) and error output are read by independent threads. The text extraction time is measured and statistics is updated.

ReadOutputAndError()
It creates two threads that read from two streams in parallel and returns the strings read. It is the only method that uses the Concurrency class. This method is invoked in *SearchPs()* and *SearchPdf()*.

SearchPdf()
Analogy to *SearchPs()*. The standard input of Pdftotext cannot be redirected as PDF files require random access (see Section 2.3). PDF files are always read from disk.

SearchForReferences()
The document‘s content in plain text (either in memory or on disk after unpacking an archive) is searched for citations and for references to other documents. The links to other documents are retrieved using a regular expression that dumps the "href" HTML tags. The URLs obtained in this manner are further examined to retain only the relevant ones The references to documents on the same server as the server of the current document are added to the list of references (*AddToReferences()*) and will make up the children nodes in the tree of documents (see Figure 6.5). The links to documents on other servers are processed via *Server.AddDocument()*. As URLs are case sensitive, the document's content cannot be transformed into lower case before. Note that PDF and PS files will never get to this method.

7.3.6 Concurrency

This is a helper class used only in *Document.ReadOutputAndError()*. It has one method only.

ReadFromStreamReader()
This method will be executed by a thread and will read from a stream specified.

7.3.7 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.

SearchFuzzy()
It executes the program fstrcmp.exe described in Section 4.3 and decides whether a search string has been found in a text with at least the minimum similarity given. The communication between *CiteSeeker* and fstrcmp.exe is done via temporary input and output files the names of which are passed to fstrcmp.exe as command line arguments. If the search was successful, the string found along with its similarity to the search string are returned.

Problems arose when more than one thread ran this method. The process running by the program fstrcmp.exe could not be created unless the threads started with little delays between each other. As the delay length would probably be hardware-dependent, the idea of a parallel fuzzy search was abandoned and there is currently only one search thread. The possible solutions would be to rewrite the entire source code of Fstrcmp to C# or to compile the function *fstrcmp()* written in C to a DLL library and to call *fstrcmp()* directly from *CiteSeeker*. The first solution costs time while the latter brings problems related to passing UNICODE strings from C# to ASCII string routines in C. Both alternatives will have to be considered in future versions of *CiteSeeker*.

IsTooClose()
It returns true if a specified position (an index to a 1D array) is too close to some positions in a list. The value for “too close“ can be set in *Advanced/Settings/Search Parameters/Minimum exact search distance*.

SearchForKeywords()
This method 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 3.3.2).

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. The whole simplified algorithm is explained below in Algorithm 7.3.

If we denote N the number of author groups (i.e. lines with authors’ names in INPUTS\searchstrings.txt), M the number of authors in a group (in a line), P the number of occurrences of an author in the document and D the number of publications of that author, the time complexity of Algorithm 7.3 as to the number of fuzzy search invocations is

$$O(N\ M\ P\ D) \tag{7.1}$$

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 (4.2).

```
// it takes all inputs from INPUTS\searchstrings.txt parsed
for all authors and publications
{
        empty list of authors‘ positions;

        // more than one author may be assigned to more than one publication, e.g.
        // author=Brown + Smith
        // publication 1
        // publication 2
        for each author of a set of publications
        {
                // exact search
                while author’s name is found (start from end of document, remember position)
                {
                        // e.g. [Brown97] Brown…
                        if author is too close to previous position
                                continue from this position;
                        // e.g. Brown, Smith…
                        if author is too close to others found
                                continue from this position;
                        add position to list of positions;

                        // user can interrupt search
                        if search is paused wait until pause terminates;

                        // for example, select 200 characters behind Brown
                        determine little search part past author;
                        reduce white space in search part;

                        // fuzzy search
                        // publication 1
                        // publication 2
                        for each publication
                        {
                                if publication is found in search part
                                        log result;
                        }

                        if fuzzy search went wrong
                                continue with next publication;

                }       // while author’s name is found

        }       // for each author of a set of publications

}       //      for all authors and publications
```

Algorithm 7.3: Search for citations

7.3.8 Statistics

This class has only static attributes that enable monitoring the following statistical parameters (not all of them are accessible by the user):

- how many servers were searched
- how many documents were searched
- how many documents were successfully searched (with one or more citations found)
- how many archives were checked
- how many kilobytes were processed (≈ downloaded)
- how many new servers were found
- how long the search takes
- how many PS and PDF files were checked
- how many errors occurred during PS and PDF text extraction
- the number of PDF files processed
- the number of PS files processed
- average time of text extraction from PDF files
- average time of text extraction from PS files

The static methods are:

InitializeStatistics()
At the beginning of each search all statistical parameters are reset.

CountTime()
This method is overloaded (there are two versions with different parameters). It prints the time span between two time values to a file or to the output window.

CountPsPdfTime()
PrintPsPdfTime()
These methods count and print the average time of text extraction from PS and PDF files. They may be inactive in the current build of *CiteSeeker*.

CalculateSize()
It serializes the specified object (or object graph) and returns its length in bytes. On this occasion the object is also saved to disk (directory DATA) if needed.

CountDocs()
This method has two overloads. They print to the output window or to a file (LOGS\debug.log) some statistical parameters, the numbers of objects in the queues and tables and the total physical memory used by *CiteSeeker*. At regular intervals the physical sizes of queues and tables are logged to a file as well.

7.3.9 Settings

This class has a large number of static attributes that are necessary for the search. Some of them are constants (such as names of directories and files) while the others are variables that can be changed by the user via *Advanced/Settings* on the menu, e.g. search parameters. The methods are mostly concerned with setting these parameters.

InitializeSettings()
It is called on every search start. It creates all necessary directories and creates or opens the log files.

CreateDirectories()
It creates the directories LOGS, DATA, DOWNS and TEMP if they do not exist.

InitializeGoogleKeywords()
It creates an initial list of default words for a Google query.

InitializePermittedFormats()
It creates an initial list of file formats permitted by default.

CreateOrOpenFile()
It creates or opens for appending the file specified.

Finalize()
Called on every search end. It closes the debug log file if opened.

7.4 Inputs and outputs

Inputs and outputs to and from *CiteSeeker* including temporary files are located in subdirectories of the CiteSeeker folder. The full structure looks like this:

- DATA
  - completedServersTable.dat
  - currentServer.dat
  - pendingServersQueue.dat
- DOWNS
- INPUTS
  - forbiddendomains.txt
  - searchstrings.txt
  - startpoints.txt
  - staywithindomains.txt
- LOGS
  - completed.log
  - debug.log
  - errors.log
  - success.log
  - summary.log
- TEMP

As *CiteSeeker* runs on Windows only, the file and folder names can be in lower or upper case arbitrarily.

The directory DATA contains files with serialized objects – the table of completed servers, queue of pending servers and the state of the current server (tree of documents, table of completed documents, queue and table of pending documents). They are used if the search is resumed later.

The folder TEMP will contain temporary text files used in the communication with external programs. The user should have no knowledge about them. The files downloaded are stored to the DOWNS directory. They are deleted once they have been searched.

7.4.1 Inputs

See Appendix E for input examples. All input files can have lines commented out by placing a hash mark ('#') at the beginning of the lines (with no white space before). These lines along with blank lines are ignored when *CiteSeeker* parses the input files.

Searchstrings.txt is required. It contains the authors and publications whose citations will be searched for. They are case insensitive. The lines of authors must start with "author=" followed by one or more authors delimited by a plus sign ('+'). All the lines below the line of authors that do not start with "author=" are the corresponding publications. (The number of publications for a group is independent of the number of authors in that group.) Another author or group of authors starts with "author=" again, etc. As shown in Algorithm 7.3 the more authors in a group the longer the search. The author's names should be as short as possible because they are searched for with exact match.

Startpoints.txt is required although it may be empty (start points are intended to be acquired from Google, for instance). It comprises the URLs where the search shall begin. It may be servers, servers with paths or even servers with paths and files. In any case, the protocol must be stated too. The URLs are case sensitive.

Staywithindomains.txt and forbiddendomains.txt are optional and include the URLs that the search should be restricted to, or that should be avoided, respectively. In addition to start points, "wildcard" domains such as ".edu" may be used here.

7.4.2 Outputs

CiteSeeker outputs are located in the LOGS folder. See Appendix E for examples of outputs. The date and time are logged to log files on each search start.

The citations found are logged to success.log. Each line has the format of a line number, the URL where one or more citations were found, a paper header (first one hundred characters of the paper with that URL where the title and author are supposed to be), a publication to search for, the publication found and their similarity. The last three elements are repeated provided citations of more publications were found. All the components are delimited by tabs, so that this file might be easily imported to Microsoft Excel, for example.

Debug.log is more or less a mirror of what the user sees in the main window of *CiteSeeker*. Information on the physical size of objects is logged here periodically. The generation of this file can be forbidden by the user. Summary.log brings some statistics of a terminated search. Completed.log is a list of servers searched. Errors.log is a list of errors that occurred during the search. There is a URL and an error message (with the name of the method that handled the error) delimited by a tab in each line.

7.5 Communication flow

The scheme of a communication flow between *CiteSeeker* and its surroundings is depicted at Figure 7.1.

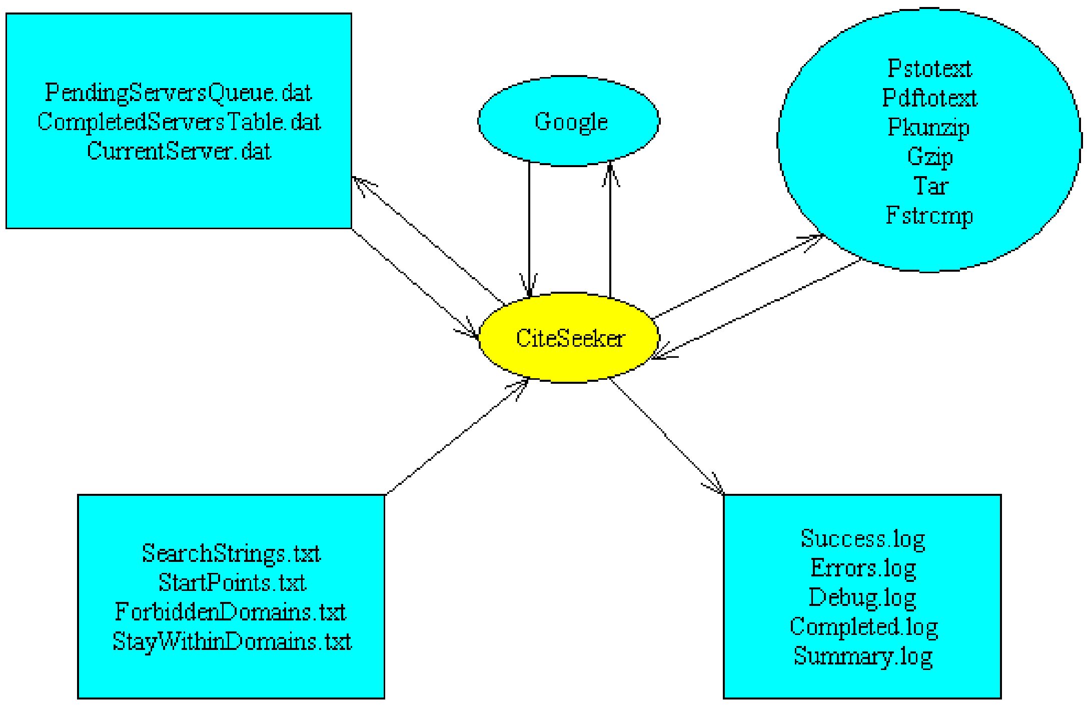


Figure 7.1: Communication flow

8 Conclusions

This thesis 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 even submit queries to Google! Some results and their aspects are presented below.

8.1 Results and time cost

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.

Execution time	3:01:51:382
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)

Table 8.1: Searching http://wscg.zcu.cz

Execution time	3:29:40:468
Documents searched	1 548
Documents successfully searched	13
New servers found	135
Kilobytes processed	840 988
Archives checked	290
PS and PDF checked	828
Text extraction errors	6
Extracted PS (average time)	288 (17.79 sec)
Extracted PDF (average time)	534 (0.57 sec)

Table 8.2: Searching http://iason.zcu.cz

As can be seen in Table 8.1 *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 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. The results from iason.zcu.cz (Table 8.2) can be interpreted by analogy.

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, which uses OCR techniques and is incorporated in the well known GSview application [14], is documented in Table 8.3 in which pstotext extracts text not only from PS files but also from PDF files.

Execution time	5:02:10:116
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)

Table 8.3: Searching http://wscg.zcu.cz without pdftotext

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 (see Appendix C for details). The search took now about 5 hours with less success than in Table 8.1. 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 2.3.1) for extracting text from PS files, but it is assumed that it might speed up the search significantly. This would be a possible improvement in the future versions of *CiteSeeker*.

8.2 Memory cost

Table 8.4 shows an example of memory usage and sizes of objects found out during a test.

Object	Elements	Size [B]
Pending Documents queue	0	145
Pending Documents table	71	5 276
Completed Documents table	200	12 680
Documents tree height	3	19 817
Pending Servers queue	24	7 448
Pending Servers table	25	26 784
Completed Servers table	0	240
Physical memory usage		47 800 320

Table 8.4: Example of memory usage

The sizes of objects cannot simply be added because they are partly in overlay (on the same data), e.g. queue and table of pending servers. The total memory used by *CiteSeeker* is rather high due to graphical interface (particularly in System.Windows.Forms.dll) and other resources loaded into memory as part of .NET Framework support for the application. Another test indicated that the queue of pending documents with 10 elements had a size of 631 B and the table of completed servers with 2 elements had a size of 293 B. Using linear extrapolation we can make some estimate of the objects' sizes for 40 million servers and 10 billion documents from Section 3.1. A queue of 40 million pending servers would then have about 12.4 GB (if they had the same number of documents as the server in Table 8.4), a table of 40 million completed servers roughly 5.8 GB, a queue of 250 pending documents (10 000 / 40 = 250) would have just about 15 kB, a tree of 250 documents 70 kB approximately, etc.

8.2.1 Case study

The following tables and figures deal with deeper analysis of 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 8.5 has a similar meaning as Table 8.1. Although the number of documents searched is significantly larger than in Table 8.1, the size of data processed is almost the same. The number of PS and PDF files checked is smaller and the total execution time is still reasonable – under five hours.

Execution time	4:42:22:484
Documents searched	17 899
Documents successfully searched	0
New servers found	2 785
Kilobytes processed	727 962
Archives checked	127
PS and PDF checked	338
Text extraction errors	11

Table 8.5: Searching http://www.siggraph.org

Table 8.6 shows a 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 grows logically, the table of pending documents grows as well because of documents with different original and real URLs (see Section 7.3.5) that remain in the table. The height of the documents tree first increases and then decreases as is typical for depth-first search (see Figure 6.5). Both the queue and table of pending servers expand as new servers are encountered during the search.

Figure 8.1 illustrates the dependency of data structures' sizes on documents searched in Table 8.6. The data structures are partly in overlay thus the same data may be included in the size. Figure 8.2 shows that *CiteSeeker*'s memory usage was growing at the beginning but later it became stable.

Figure 8.3 depicts the dependency of the size of both the queue and table of pending servers. The hash table is clearly more memory demanding but note that it does involve the current server whereas the queue does not (see Section 7.3.4). Finally, Figure 8.4 shows that the size of *Completed Documents* table increases linearly with the number of elements contained. It is the behaviour of any hash table with null values such as *Pending Documents* table or *Completed Servers* table (compare also with Section 7.3.4).

Documents searched	100	500	1 027	4 820	10 757	17 862
***Pending Docs* queue**	0	0	0	0	0	0
Size [B]	145	145	145	145	145	145
***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]	14 981	74 767	97 508	402 766	791 622	1 371 240
***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
***Completed Servers* table**	0	0	0	0	0	0
Size [B]	240	240	240	240	240	240
Memory usage [B]	24 358 912	29 646 848	58 949 632	107 433 984	107 184 128	106 713 088

Table 8.6: Memory cost samples when searching http://www.siggraph.org

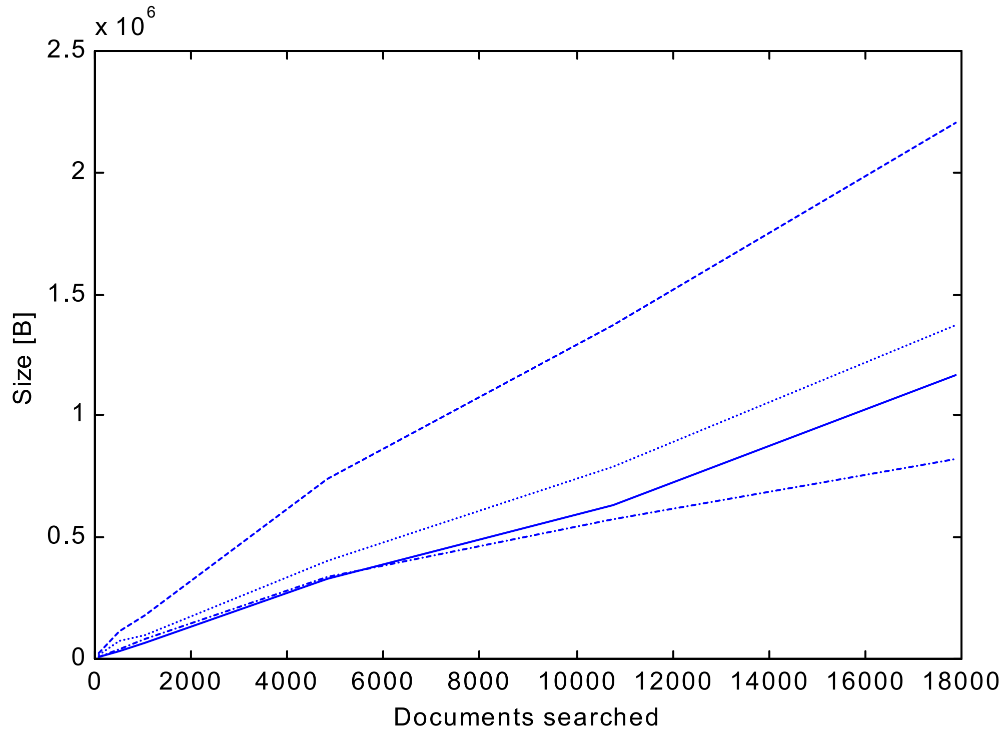


Figure 8.1: Size of data structures when searching http://www.siggraph.org
solid line: *Completed Documents* table
dotted line: Documents tree
dashdot line: *Pending Servers* queue
dashed line: *Pending Servers* table

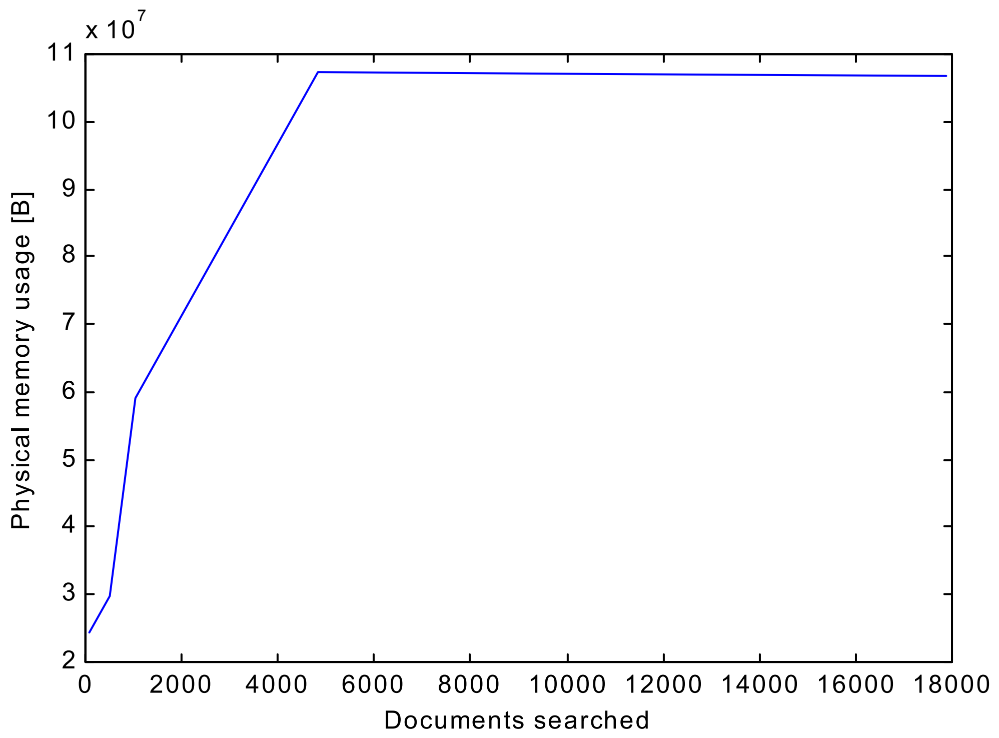


Figure 8.2: Memory used by *CiteSeeker* when searching http://www.siggraph.org

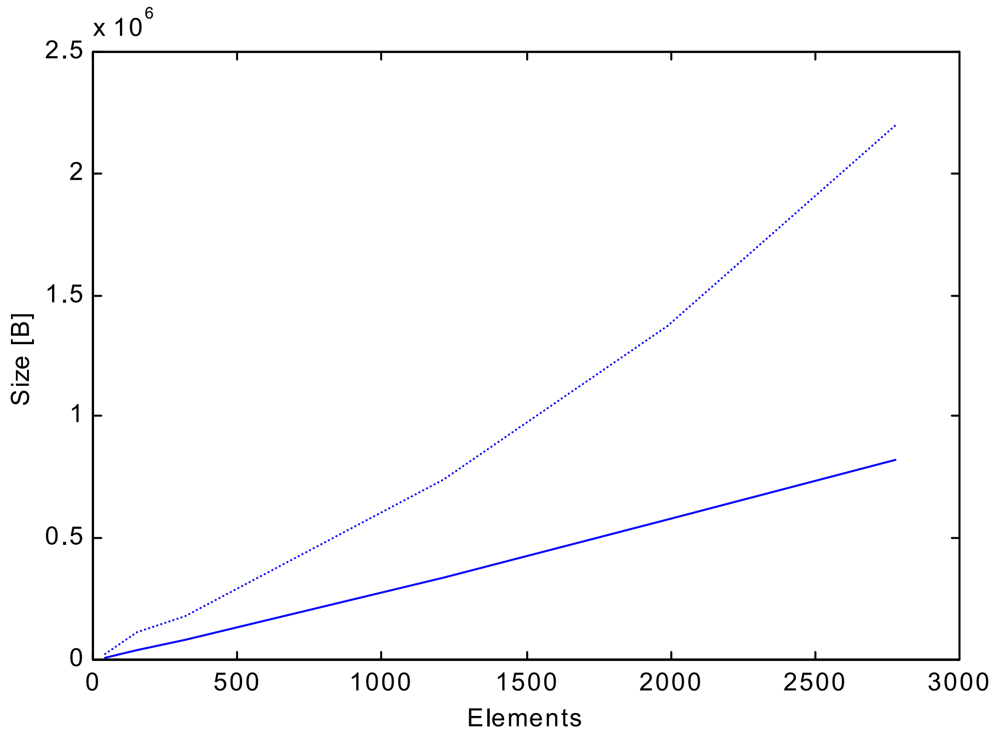


Figure 8.3: Size of *Pending Servers* data structures when searching http://www.siggraph.org
solid line: queue
dotted line: table

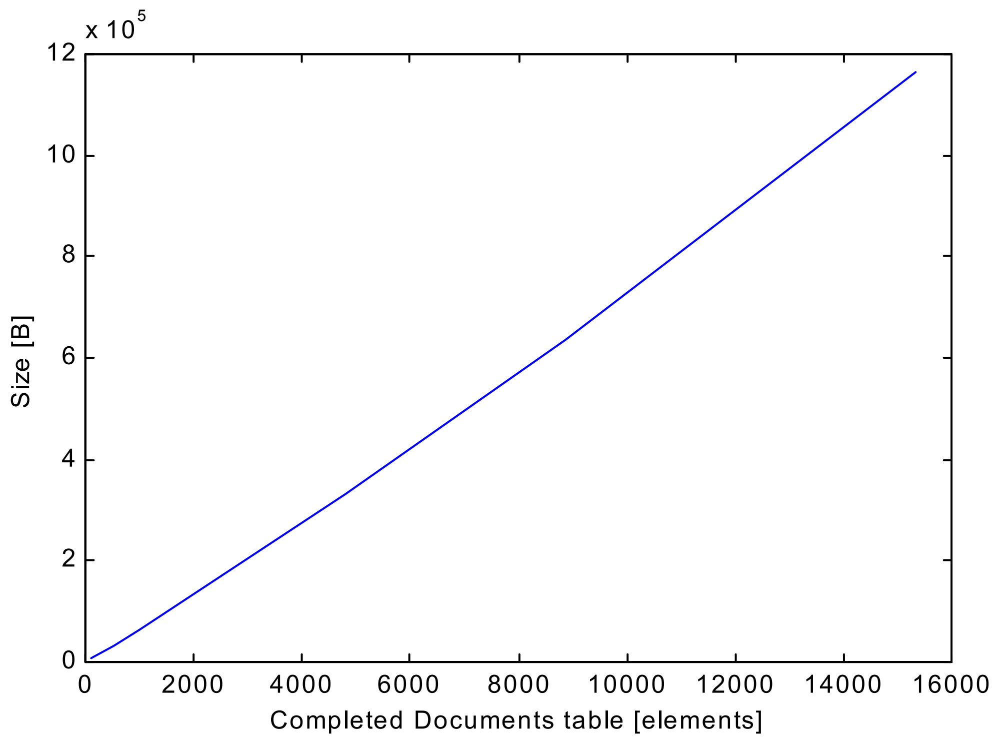


Figure 8.4: Size of *Completed Documents* table

All values in Section 8.1 and 8.2 were obtained from *CiteSeeker* compiled in debug mode.

8.3 Possible improvements

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 were not yet 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. By compiling something like fstrcmp.dll, for instance. See Section 7.3.7.
- Enhance reliability with dynamic or redirected Web pages. See section 7.3.5.
- Use PreScript instead of pstotext. See Section 8.1.
- 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.

List of Abbreviations

ASCII	American Standard Code for Information Interchange
API	Application Interface
ASP	Active Server Pages
CCIDF	Common Citation vs. Inverse Document Frequency
DHTML	Dynamic HTML
DOS	Disk Operating System
GNU	GNU’s Not Unix
GUI	Graphical User Interface
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IP	Internet Protocol
ISI	Institute for Scientific Information
JPEG	Joint Photographic Experts Group
LZW	Lempel Ziv Welch
MD5	Message Digest 5
MSN	Microsoft Network
OCR	Optical Character Recognition
PDF	Portable Document Format
PHP	Hypertext PreProcessor
PS	PostScript
RC4	Rivest Cipher 4
RTF	Rich Text Format
SES	Shortest Edit Script
SHTML	Server-side HTML
SOAP	Simple Object Access Protocol
SQL	Structured Query Language
TAR	Tape ARchiver
TFIDF	Term Frequency vs. Inverse Document Frequency
UML	Unified Modelling Language
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
UTF	Unicode Transfer Format
XML	Extended Markup Language
W3C	World Wide Web Consortium
WSDL	Web Services Description Language
WWW	World Wide Web

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] Archer T.: Myslime v jazyku C# - knihovna programatora, Grada 2002

[7] Gunnerson E.: Zaciname programovat v C#, Computer Press, Praha 2001

[8] Kucera L.: Kombinatoricke algoritmy, SNTL, Praha 1989

[9] Rychlik J.: Programovaci techniky, Kopp, C. Budejovice 1993

[10] Testa J.: The ISI Database: The Journal Selection Process, http://sunweb.isinet.com/isi/hot/essays/selectionofmaterialforcoverage/199701.html - 2

[11] Testa J.: Current Web Contents: Developing Web Site Selection Criteria, http://sunweb.isinet.com/isi/hot/essays/selectionofmaterialforcoverage/23.html

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

[13] C# Language Specification, Standard ECMA-304, http://www.ecma.ch/

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

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

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

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

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

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

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

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

[22] http://www.google.com/apis/reference.html

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

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

[25] Google: http://www.google.com/

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

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

[28] Search Engines Showdown: Size Comparison Methodology: http://www.searchengineshowdown.com/stats/methodology.shtml

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

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

[31] Robot Exclusion Standard: http://www.robotstxt.org/wc/exclusion.html - robotstxt

[32] Database of Web Robots, Overview: http://www.robotstxt.org/wc/active/html/index.html

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

[34] Science Citation Index help, Web version: Princeton University: http://www.princeton.edu/~biolib/instruct/SCI.html
[35] Online JOurnal Search Engine: http://ojose.lanners.org/
[36] Scirus – for scientific information: http://www.scirus.com/
[37] Welcome to Phibot: http://phibot.org/new/Search/Main
[38] W3C HTML Home Page: http://www.w3.org/MarkUp/
[39] SWISH++: http://homepage.mac.com/pauljlucas/software/swish/
[40] The pstotext program: http://www.research.compaq.com/SRC/virtualpaper/pstotext.html
[41] Print Center Features – Adobe PostScript vs. Adobe PDF: http://www.adobe.com/print/features/psvspdf/main.html
[42] Download: http://www.shamrock.de/cgi-bin/download.pl?pkunzip.exe
[43] Info-ZIP's UnZip: http://www.info-zip.org/pub/infozip/UnZip.html - Release
[44] The gzip home page: http://www.gzip.org/
[45] tar – GNU Project – Free Software Foundation (FSF): http://www.gnu.org/software/tar/tar.html
[46] PKWARE – Enterprise Solutions – White Papers: http://www.pkware.com/products/enterprise/white_papers/appnote.html
[47] http://search.cpan.org/src/MLEHMANN/String-Similarity-0.02/fstrcmp.c
[48] Netcraft: Web Server Survey Archives: http://news.netcraft.com/archives/web_server_survey.html

Appendix A: User's Guide

The actual program executable is the file CiteSeeker.exe.

The main menu of the application, its submenus and tabbed pages have the following structure:

- *Search*
    - *New*
    - *Resume*
    - *Suspend*
    - *Pause*
    - *Quit*
- *Advanced*
    - *Settings*
        - *Running Mode*
        - *Search Parameters*
        - *Search Files*
        - *Google Query*
        - *Display Parameters*
    - *Memory Usage*
    - *Skip Current Server*
- *About*

About
The *About* item brings up copyright and program version information.

Search
New starts a new search. All of the input files are read and search parameters are constructed from them. In case there is some data in DATA directory from a previous search, it will be **overwritten**! For this reason, it is recommended that backup copies of DATA files be made before starting a new search provided the user would like to have several search positions stored.

Resume resumes a suspended search. In general, it reads input files as well and, in addition, it loads data stored in DATA folder by the previous search into memory. Thus, the current search continues from the point that was recorded last either in the same or earlier program sessions.

Suspend suspends the current search. All data kept in memory is saved to DATA folder and the search can be resumed via *Search/Resume* in the future. The actual suspension may take a while because the search is allowed to suspend only when going from one document to another.

Pause requests or terminates a search pause. It may also take a little while for the actual pause to occur although there are more checkpoints in the program than for *Suspend*. Pausing a search enables the *Advanced/Memory Usage* menu item. When the search is paused and the user decides to suspend it, he or she must first press *Suspend* and than *Pause* to terminate the pause and suspend the search.

Quit terminates the application.

Advanced

Memory Usage brings up a form with information on physical memory used by the program and about how large current objects are. See Appendix D for a screenshot of this form. This menu item is enabled only when the current search is paused.

Skip Current Server enables skipping the current server and popping the next one from the queue of pending servers. This may be particularly useful when *CiteSeeker* gets trapped and iterates infinitely in a loop of documents. The skip is not immediate, the current document must be completed first.

Settings cannot be changed when a search is in progress. Otherwise, parameters divided into five categories may be modified.

Running Mode

Run in debug mode: If checked, debug.log file will be created in LOGS folder. It is a mirror of the program window output with some additional information about objects' sizes.

Invoke Google: If checked, a request will be sent to Google and its results will be used for getting perspective Web sites (servers) which will be searched. The actual Google query must be built on *Google Query* tab page.

Maximum results: This is the highest number of results Google should return. The upper bound is 1000 for one search a day. See Section 5 for more details.

Filter: If checked, Google filters its results. For instance, it hides very similar results, it returns only two results coming from the same server, etc. See Section 5 for more details.

Google Account key: A unique key must be set here, so that communication with Google could be possible. See Section 5 for information on how to obtain this key or use the default key.

Search Parameters

See Appendix D for a screenshot of this page.

Search Method is either *Fuzzy* or *Exact*. Only the method of searching for publications will be influenced by this choice. Names of authors are always searched for by the exact method. In general, the fuzzy search method is much slower. For more information on search methods, see Section 4.

Fuzzy search limit is the minimum similarity desired from (0.0; 1.0> between a publication in searchstrings.txt and the one found on the Web. **Increasing** this number **will speed up** the program but it may also cause some **perspective documents** to be **omitted**. Similarity 1.0 means an exact search, in fact. For more information on similarities, see Section 4.

Fuzzy search part length: This is the length in characters of that part of the text that fuzzy search should be applied to. The fuzzy search part begins immediately past the author's name found. This number **significantly affects** the program **speed**! The larger the length, the slower the search.

Minimum exact search distance is the minimum distance in characters between two occurrences of authors (the same or different names) which causes two searches for publications to be performed. In other words, if two authors are found and the distance between them is less than this number, they are considered as authors of one publication, and only one search for publications occurs in the text past them.

Data flush interval determines how often is the data in memory saved to disk If this number is n, then the data is stored after completion of each n-th document. This number **enhances security** (in case of a system crash less data is lost) but also **worsens performance**.

Search Files
Permitted file formats are set here. Corresponding file extensions are listed in the following:

- HTML, XML (htm, html, shtml, dhtml, xml)
- PHP (php, php3, php4, asp, aspx)
- TXT (txt)
- RTF (rtf)
- DOC (doc)
- PPT (ppt)
- PDF (pdf)
- PS (ps)
- ZIP (zip)
- GZ (gz, z)
- TAR (tar, tgz, taz)

Google Query
This page is enabled only when *Running Mode/Invoke Google* is checked. The maximum number of ten query terms is permitted, which is implied by Google Web APIs services limitations shown in Table 5.3. Terms can be logically ANDed or ORed. Each term is input in one text box. and is automatically treated as a phrase if it consists of more words. Thus, there is no need to enclose them in double quotes. Special query terms may be entered here as well. See Table 5.2 for the list of the most important special query terms. See Appendix D for a screenshot of *Google Query* page.

Display Parameters
Window buffer size sets how many lines of output can be scrolled up or down in the main window. The greater the number, the **greater** the **memory requirements**.

Swap buffers interval determines how often the main window front buffer shall be refreshed and replaced with the back buffer. If this number is n, the front buffer will be refreshed with each n-th output line exceeding the window buffer size.

Default values
The user can set default values by clicking the "Load defaults" button in *Settings*. These values are also set during program startup when no configuration file (CiteSeeker.cfg) has been found. The program parameters' default values are as follows in Table A1:

Parameter	**Default value**
Running Mode/Run in debug mode	true
Running Mode/Invoke Google	false
Running Mode/Maximum results	10
Running Mode/Filter	true
Running Mode/Google Account key	FTIIRPBQFHK5yVDfHA9zTuMEdvulOQ2O
Search Parameters/Search Method	Fuzzy
Search Parameters/Fuzzy search limit	0.75
Search Parameters/Fuzzy search part length	200
Search Parameters/Minimum exact search dist.	30
Search Parameters/Data flush interval	100
Search Files	all displayed file formats checked
Google Query	computer graphics visualization parallel processing
Display Parameters/Window buffer size	300
Display Parameters/Swap buffers interval	50

Table A1: Default values for *CiteSeeker*

Appendix B: Installation

The entire CiteSeeker directory on the CD must be copied to a **local** disk on a computer with installed .NET Framework. The disk is required to be local because of strict .NET security permissions. The CiteSeeker directory, which is the current working directory of the application, will include following files and folders:

```
DATA
DOWNS
INPUTS
LOGS
TEMP
xpdfrc
CiteSeeker.cfg
gsdll32.dll
pstotxt3.dll
CiteSeeker.exe
fstrcmp.exe
gswin32c.exe
gzip.exe
pdftotext.exe
pkunzip.exe
pstotxt3.exe
tar-11~1.exe
gsdll32.lib
```

The directories DATA (for serialized data), DOWNS (downloaded data), LOGS (log files, i.e. search results), TEMP (temporary files) and the file CiteSeeker.cfg (configuration file) do not necessarily have to be present. In case of their absence they are created or default configuration values are used. Of course, if DATA is missing, the search cannot be resumed.

Furthermore, Ghostcript must be installed on the local computer. Paths to its fonts **must** be set in xpdfrc configuration file so that Pdftotext could have access to non-embedded fonts. (Pstotext will find Ghostscript fonts automatically from the registry entries.) The user can simply replace the paths in the existing sample xpdfrc file. Ghostscript installation file can be obtained from Ghostcsript folder on the CD or from [14].

Below, you will find a summary table B1 of external utilities needed by *CiteSeeker* and their versions used in the testing stage.

Utility	Version
GHOSTSCRIPT	7.04
PSTOTEXT	1.8h
PDFTOTEXT	2.02
PKUNZIP	2.50
GZIP	1.2.4 Win32
TAR	1.12
FSTRCMP	

Table B1: Utilities used

Note: Pstotext's source code was slightly modified. See Appendix C for more details.

Appendix C: Programmer's Guide

1)

Figure C1 shows the structure of *CiteSeeker* project in C# in this top-down hierarchy: project, namespaces, files, classes.

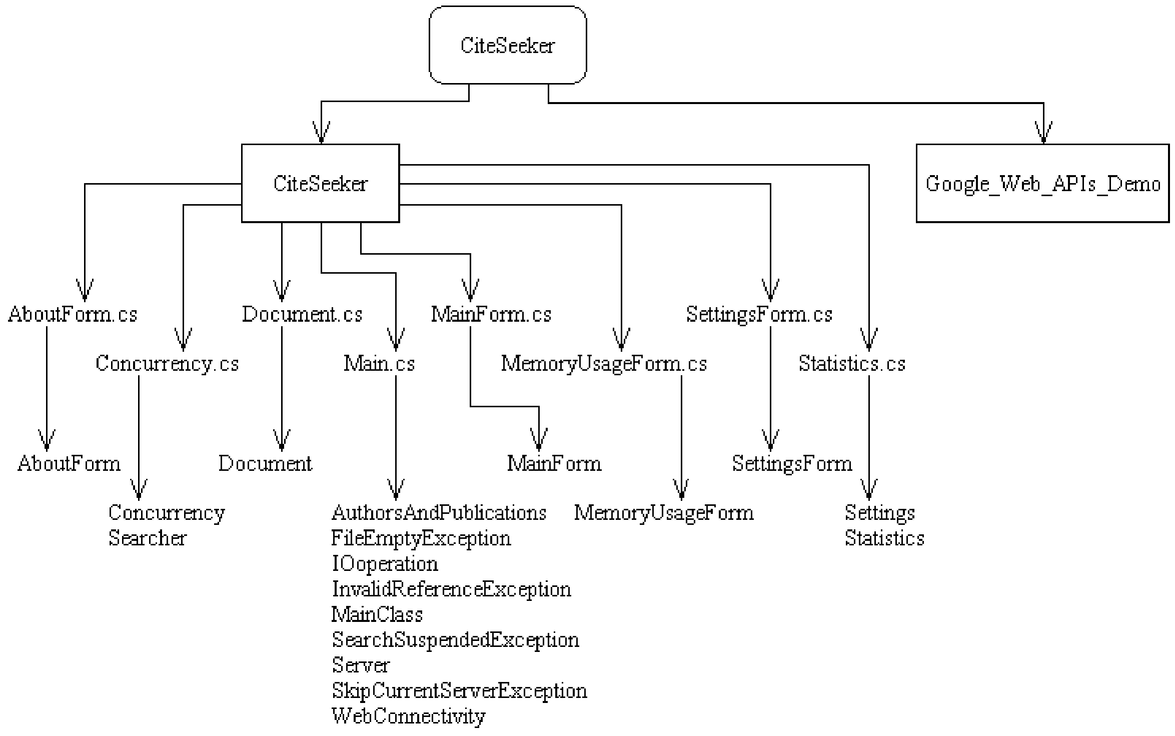


Figure C1: Project structure

2)

This section is an overview of the classes that have some functionality not related to GUI and their methods and constructors.

public class Concurrency

public Concurrency(StreamReader streamReader)
public void ReadFromStreamReader()

public class Document

static Document()
public Document(Server server, String URL)
void AddToReferences(String URL)
public static void CheckIfPermittedDomain(String URL)
public static void CheckWildcardDomains(String serverName)
DocumentLocation ExamineReferencedURL(ref String URL)
String[] FindTarFileNames(String archive)
String[] FindZipFileNames(String fileName)
String GetStringFromContent()

```
void Gunzip()
void GunzipPdf(String archive)
String GunzipPs(String archive)
void Initialize(WebConnectivity web)
static bool IsPermittedFormat(String URL)
public static bool IsSafeURL(String URL)
public static bool IsSupportedProtocol(String URL)
void MakeAbsoluteURL(ref String baseURL, ref String URL)
String MakeShortFileName(String name)
void ModifyURL(ref String URL)
void ReadFromFileToContent(String file)
void ReadOutputAndError(StreamReader stdOutput, StreamReader stdError, ref String output,
                        ref String error)
public static String ReduceWhiteSpace(ref String contentString)
void RemoveFragmentsFromURL(Uri myUri, ref String URL)
void ScheduleSearchForKeywords(String contentString)
public void Search(WebConnectivity web)
void SearchForReferences(String file, String contentString)
void SearchPdf(String file)
void SearchPs(String file, String contentString)
void Untar()
void UntarPdfPs(String archive, String file)
void Unzip()
void UnzipPdf(String archive, String file)
String UnzipPs(String archive, String file)
void WriteContentToFile(String file)
```

public class IOoperation

```
static String BuildGoogleQuery()
static void BuildPendingServersTable()
public static void CountDocs(Server server)
public static void CountTime(DateTime startTime, DateTime endTime, String description)
public static String[] GetStrings(String file, bool isURL)
public static Server LoadDataStructures()
static String ParseUri(String URL)
public static String[] SearchWithGoogle()
public static void Serialize(object graph, String file)
public static void SetStartPoints(String[] startPoints)
public static void WriteError(String error)
public static void WriteLine(String output)
```

public class MainClass

```
static bool ContainsOnlyDelimiters(String s, char c)
static void Finalize()
static void Initialize()
static void InitializeKeywords()
static void IterateThroughAllServers(WebConnectivity myWeb)
static Server LoadData()
static int ReadForbiddendDomains()
static int ReadSearchStrings()
static int ReadStartPoints()
static int ReadStayWithinDomains()
static void ResumeSearch(ref Server currentServer, WebConnectivity myWeb)
public static void Start()
```

public class Searcher

```
public Searcher(String contentString, int start, int end, String URL)
```

```
bool IsTooClose(int position, ArrayList positions)
public void SearchForKeywords()
int SearchFuzzy(double minSimilarity, String searchString, String contentString, ref String output)
```

public class Server

```
static Server()
public Server(String name)
public void AddDocument(String Uri)
void CompleteServer(bool isSkipped)
public static Server PopMostReferencedServer()
public void SearchDocuments(WebConnectivity web)
void SearchTree(Document document, WebConnectivity web)
public static void WaitOnPause()
```

public class Settings

```
public static void CreateDirectories()
static StreamWriter CreateOrOpenFile(String file)
public static void Finalize()
public static void InitializeGoogleKeywords()
public static void InitializePermittedFormats()
public static void InitializeSettings()
```

public class Statistics

```
public static long CalculateSize(object graph, String file)
public static void CountDocs(Server server, StreamWriter aus, bool isFlush)
public static void CountDocs(Server server, bool isFlush)
public static void CountPsPdfTime(bool isPdf, DateTime startTime, DateTime endTime)
public static void CountTime(DateTime startTime, DateTime endTime, String description)
public static void CountTime(DateTime startTime, DateTime endTime, String description,
                             StreamWriter aus)
public static void InitializeStatistics()
public static String PrintPsPdfTime(bool isPdf)
```

public class WebConnectivity

```
public byte[] ConnectToNet(String URL, ref String fullUri)
void CopyStreamIntoByteArray()
public void ReleaseResponse()
```

3)

The file pstotxtd.c in pstotext originally included these two lines:

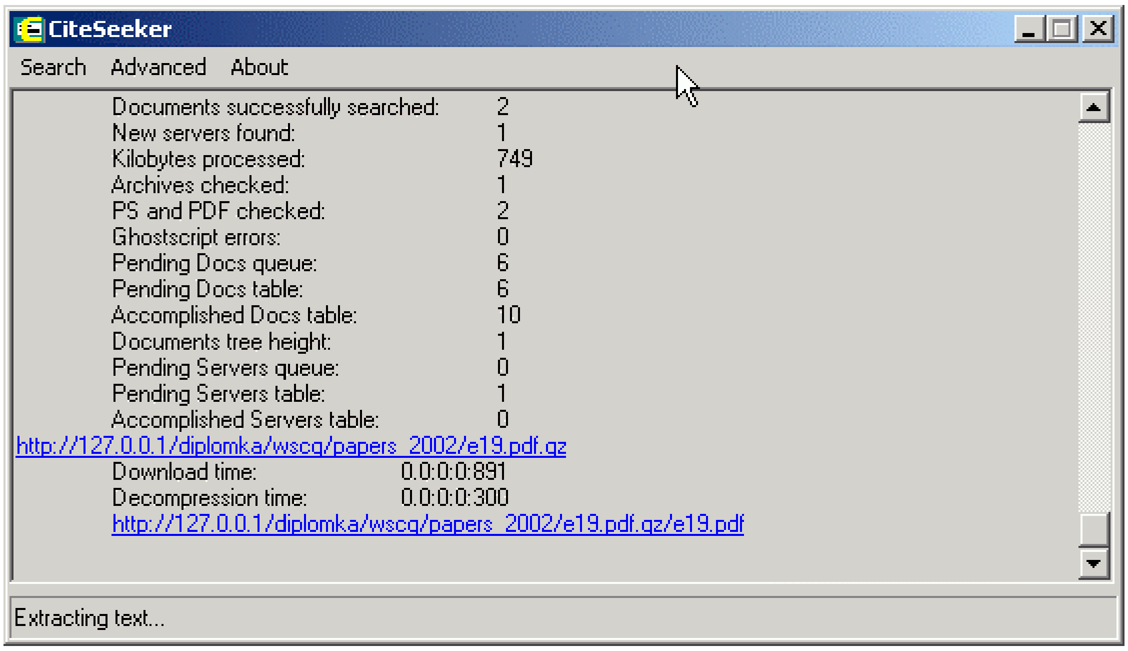
```
#define LINELEN 2000
gs = fopen(gstemp, "r");
```

They were changed to

```
#define LINELEN 2048
gs = fopen(gstemp, "rb");
```

The first change has no effect while the latter allows accepting binary PDF files that include a '^Z' character. As pstotext is no longer used for extracting text from PDFs in *CiteSeeker*, these modifications are irrelevant.

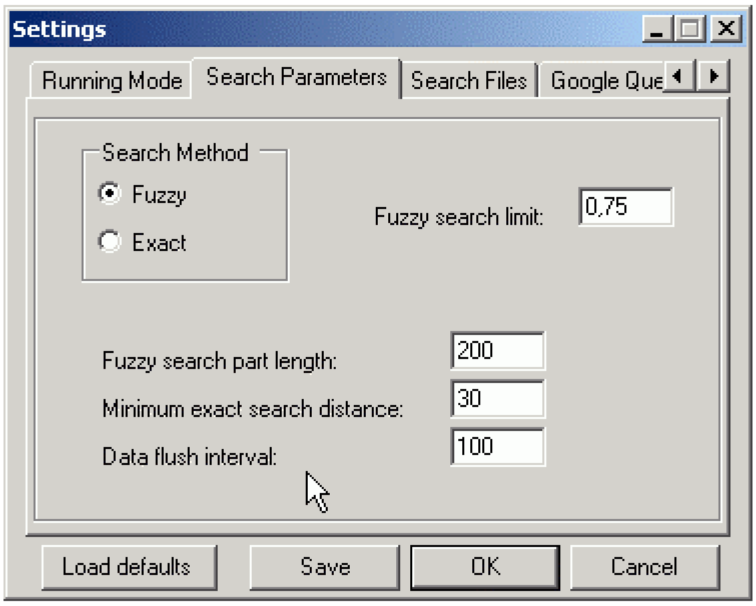
Appendix D: CiteSeeker Screenshots



```
Documents successfully searched:    2
New servers found:                  1
Kilobytes processed:                749
Archives checked:                   1
PS and PDF checked:                 2
Ghostscript errors:                 0
Pending Docs queue:                 6
Pending Docs table:                 6
Accomplished Docs table:            10
Documents tree height:              1
Pending Servers queue:              0
Pending Servers table:              1
Accomplished Servers table:         0
http://127.0.0.1/diplomka/wscg/papers_2002/e19.pdf.gz
Download time:              0:0:0:0:891
Decompression time:         0:0:0:0:300
http://127.0.0.1/diplomka/wscg/papers_2002/e19.pdf.gz/e19.pdf

Extracting text...
```

Figure D1: Application main window



Search Method: (● Fuzzy) ○ Exact

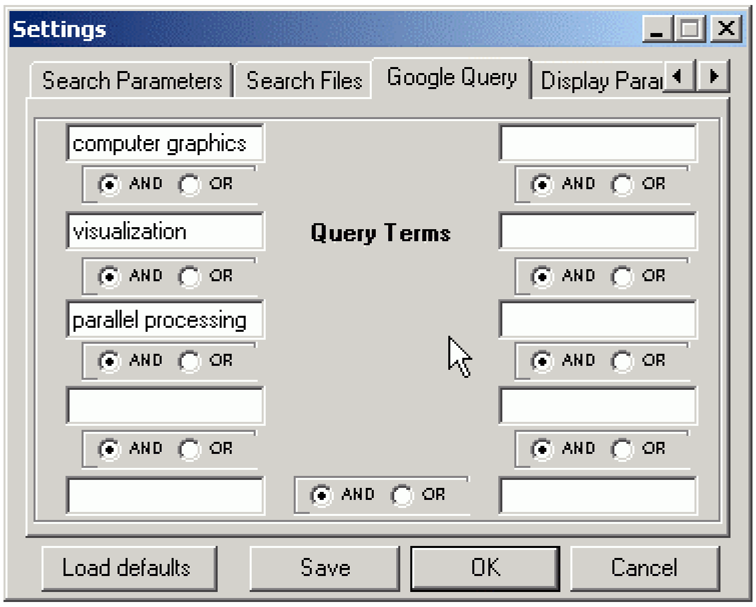
Fuzzy search limit: 0,75

Fuzzy search part length: 200

Minimum exact search distance: 30

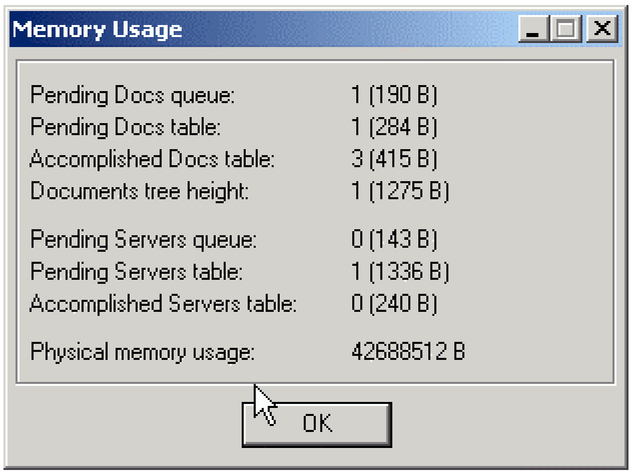
Data flush interval: 100

Figure D2: Settings form – search parameters



computer graphics		
[✓ AND] OR		[✓ AND] OR
visualization		
[✓ AND] OR		[✓ AND] OR
parallel processing		
[✓ AND] OR		[✓ AND] OR
[✓ AND] OR		[✓ AND] OR
	[✓ AND] OR	

Figure D3: Settings form – Google query



Pending Docs queue:	1 (190 B)
Pending Docs table:	1 (284 B)
Accomplished Docs table:	3 (415 B)
Documents tree height:	1 (1275 B)
Pending Servers queue:	0 (143 B)
Pending Servers table:	1 (1336 B)
Accomplished Servers table:	0 (240 B)
Physical memory usage:	42688512 B

Figure D4: Memory Usage form

Appendix E: Sample Inputs and Outputs

1) **searchstrings.txt**

```
#	2002

author=Bosch + Agarwal
Three-Dimensional Object Construction Using a Self-mixing Type Scanning Laser Range Finder
Approximating shortest paths on a convex polytope in three dimensions
author=Shen
Computer modeling, analysis, and synthesis of dressed human

#	2001

author=Kokaram + Korotov + Kanade
Detection and Removal of Line Scratches in Degraded Motion Picture Restoration
Acute Type Condition for Tetrahedral Triangulations and the Discrete Maximum Principle
Recognizing action units for facial expression analysis
```

2) **startpoints.txt**

```
http://www.acm.org
http://www.siggraph.org/
http://mambo.ucsc.edu/psl/cg.html
http://graphics.cs.ucdavis.edu/GraphicsNotes/
```

3) forbiddendomains.txt

```
.com
.uk
http://wscg.zcu.cz
http://www.math.psu.edu/dna/graphics.html
.stanford.edu
http://iason.zcu.cz/~skala
```

4) success.log

```
**********          11.6.2003  23:47:42      **********
1          http://127.0.0.1/diplomka/wscg/papers_2002/a11.pdf      3d-shape reconstruction based on
radon transform with application in volume measurment chuchart p        bosch: three-dimensional
object construction using a self-mixing type scanning laser range finder  r. and lescure, m., "three-
dimensional object construction using a self-mixing type scann  0.7556
2          http://127.0.0.1/diplomka/wscg/papers_2002/g31.zip/g31.pdf      computing geodesic distances
on triangular meshes marcin novotni and reinhard klein insitut fTĘur      agarwal: approximating
shortest paths on a convex polytope in three dimensions             r. varadarajan. approximating
shortest paths on a convex polytope in  0.7536
3          http://127.0.0.1/diplomka/wscg/papers_2002/a47.ps.gz/a47.ps   line scratch detection on
digital images: an energy based model d. vitulano v. bruni 1 p. ciarli       kokaram: detection and
removal of line scratches in degraded motion picture restoration, detection and re- moval of line
scratches in degraded mo- tion picture resto      0.9231
```

5) summary.log

```
**********          30.4.2003  17:12:9       **********
The search has been suspended.
          Execution time:             0.0:3:46:535
          Documents searched:                    3
          Documents successfully searched:       3
          New servers found:                     1
          Kilobytes downloaded:                  1360
          Archives checked:                      2
          PS and PDF checked:                    3
          Ghostscript errors:                    0
          Pending Servers queue:                 0 (143 B)
          Pending Servers table:                 1 (1269 B)
          Accomplished Servers table:            0 (240 B)
          Physical memory usage: 43753472 B
```

Registration Form

I agree with my thesis being used for non-circulating loans in the University Library of UWB in Pilsen.

Pilsen, June 15, 2003

Dalibor Fiala

The borrower certifies by signature to have used this thesis for studying purposes and to cite it within references.

Name	Faculty/Department	Date	Signature