CHALMERS

Automated Search and Management for Geographical Web Services

Master of Science Thesis in the Programme Computer Science: Algorithms, Languages and Logic

DANIEL SIDSTEN

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, March 2012

Methods for search and management of Geographical web services of OGC type are examined in this report.
OGC is an open standard organization for geographical services.

DANIEL SIDSTEN

Examiner: Erland Holmström

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Cover:
A picture of computerized search for geographical data

Department of Computer Science and Engineering
Göteborg, Sweden March 2012

Abstract

This thesis holds the result of a study in focused web-information search, retrieval and management of retrieved data. Focused web-information search means that the search is exclusive for one type of content or topic. The only topic examined in this search is OGC web-services. More specifically, URLs to OGC web-services are searched for.
Of course, many of the techniques utilized and discussed in this thesis are of possible interest also for other topics.
The OGC web-services are basically services for online geospatial data management, such as online maps and coverage data.

In detail this study examines how one can effectively locate a large amount of these OGC-services on the web. That is some techniques, such as meta-search and web-crawling are examined. Also after an evaluation of the methods examined, the most effective methods are combined into a search-system prototype. The search-system is made for autonomous, effective and stable retrieval and management of the location of the OGC web-services.

This work was carried out at Carmenta AB. The testing of retrieval techniques in this thesis have generated the location of a total of approximately 2600 online high-quality OGC-services. The prototype have only been partially developed, but it is estimated to be capable of discovery of approximately 3000 online services in a session of 33 hours.

Keywords: OGC web-services, URLs, Geospatial Data

Contents

Preface

Firstly I would like to thank my supervisor at Carmenta, Mats Olsson, for helping me get started with the project and for other advise. Thanks to him I found out about the Yahoo! search-API BOSS. Also his support and feedback have been helpful during especially the method-testing and prototype development periods of the project.

Secondly I would like to thank my supervisor and examiner at Chalmers, Erland Holmström, for helping me with the work on this report. His feedback have been important for making the structure and content of the report much better. Also his initial support in finding other related thesis-work was most appreciated.

Thirdly I thank Carlos Castillo and all other people involved in the WIRE-project. Without WIRE, much of the work and research in this thesis, would not have been possible. Not only WIRE, but also the PhD. thesis on Effective Web Crawling by Carlos Castillo, have been very useful for this master-thesis work.

Finally I would also like to thank all people at Carmenta AB, for being friendly and helpful to me at all times during the thesis-work. Also I would like to thank many others not mentioned previously, like Alex Chudnovsky, for making such a useful open-source HTML-parser for the .Net-platform.

List of abbreviations

URL	Uniform Resource Locator, web-resource reference
HTTP	Hypertext Transfer Protocol, communication-protocol for web-requests
API	Application Programming Interface, used for communication between softwares
IETF	Internet Engineering Task Force, standards organization for the web
HTML	Hypertext Markup Language, a web-standard for text-markup
XML	eXtensible Markup Language, a rule-set for machine-readable documents
GIS	Geographic Information system, a system for handling of geospatial data
OGC	Open Geospatial Consortium
WMS	Web Map Service, an OGC-standard
WFS	Web Feature Service, an OGC-standard
WCS	Web Coverage Service, an OGC-standard
WPS	Web Processing Service, an OGC-standard
BOSS	Build your Own Search Service, an API to Yahoo!'s search engine
WIRE	Web Information Retrieval Environment, an open-source web-crawler
OPIC	Online Page Importance Computation, an algorithm for deciding crawl-order
HMAC	Hash-based Message Authentication Code, a cryptographic verification construct
SHA1	Secure Hash Algorithm 1, a cryptographic hash-function.
USD	United States Dollar, a currency
RFC	Request For Comments, standards documentation published by IETF

Chapter 1

Introduction

introduction: The search for geospatial data

Geospatial data is the type of geographical data, which is used by a Geographic Information System (GIS). GIS-systems are computer based and capable of manipulating and analysing the geospatial data, and outputting it in a more pleasant format for humans to use. One such format is a layered map, which basically consists of an image with separate layers for different types of data, such as land-data, sea-data and roads. Figure 1.1 shows some typical layers in a map produced by a GIS.

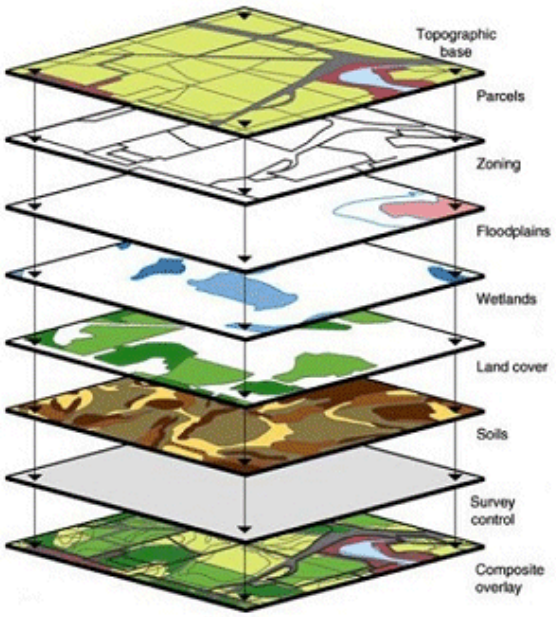

Figure 1.1: GIS layered map

There are many web-based applications for GIS. These applications enable access to web-services containing sources for geospatial data. There are however many standards and formats available for the structure of this data. In 1997, a standards organization for geospatial data called OGC, began developing standards for GIS web-services.
Since then, OGC's standardization of web services has led to an ever-increasing number

of geospatial data sources on the Internet. Since the standards are open, there exists a number of available different implementations at many different locations around the world. Many new opportunities for products and services could emerge from all these sources. Since there is no requirement to publish the location of a new web-service for OGC or any other organization, there is no easy way to find these services.

This thesis work, involves finding, evaluating and developing new methods for automated search and management for geographical web services of OGC-type. The work also includes a prototype implementation using the developed methods, and a final evaluation of the estimated results of the prototype.

Background

In 1994 The Open Geospatial Consortium (OGC) was formed. It is an international voluntary consensus standards organization. OGC is involved with developing open standards for geospatial content and service, as well as GIS data- processing and -sharing. [Wiki 02]

A GIS may perform geospatial data-analysis. This type of analysis applies statistical or any other informational analysis technique, on geographically based data. [Wiki 03] The most basic types of analysis is map-overlay and simple buffering. In map-overlay, two or more maps or map-layers are combined using a set of predefined rules.
Simple buffering is a proximity analysis, which is typically used for identifying regions of a map within a specified distance of one or more features, such as towns, roads, etc. [Wiki 01]

In 1997 OGC became involved in developing standards for web-mapping, after Allan Doyle had published a paper about an idea for a "WWW Mapping Framework". A task force for designing a standard protocol using Doyle's ideas was established by OGC. The first results of the task force were demonstrated in September 1999.

In April 2000, OGC released the first version of a GIS-standard for web-based map-services, called Web Map Service (WMS). [Wiki 04]
Since then, the number of available map-services on the web have rapidly increased.
Today WMS is a widely supported standard for map-formated GIS data. Also several other standards for geospatial web-services have been developed by OGC.

Problem

The problem handled by this thesis is, how one can find and manage OGC-services on the web in an automated manner.
An OGC-service can be accessed by many different URLs (Uniform Resource Locator), depending on details in the request. Since every OGC-service must, by the OGC-standards, have a capabilities-document, this type of URL is the most useful and therefore an URL for an OGC-service which points to the capabilities document is regarded as an OGC-URL in this thesis. (the capabilities-document specify in detail, the actual capabilities of the service).

The search for OGC-services identified by OGC-URLs, involves investigation of different methods for discovery and a further analysis of these methods, followed by an analysis of a suitable way to automate the methods for discovery. Eventually the second analysis-step could lead to an idea for an automated agent, which then should be developed.
The development of this agent should only be a simple prototype, demonstrating the effectiveness of its underlying methods.

Purpose

The purpose of the thesis work is to find, evaluate and eventually develop new methods for the discovery and management of geographical web services. Today the discovery of these is somewhat difficult and the management is work intensive.
The idea is that some kind of automated engine or web crawler should be able to automate the process of finding, updating and adding new URLs to these services for geographical data. This would increase the effectiveness of Carmenta's map-catalog and offer a service of higher quality for its costumers.

Scope

The scope of this master-thesis work is concerned with OGC-standardized geospatial data. The type of this data is limited to mainly three types of OGC web-services. These are: WMS (Web Map Service), WFS (Web Feature Service) and WCS (Web Coverage Service). In particular the focus is set on WMS-data, since this type of geospatial-data is the most commonly found.
This means that the standards for these three services, will be examined and active search and identification of OGC-services are to be based on the standard protocols for them, i.e., services like WPS might be found accidentally, but the intention is not to actively search for other OGC-services.
Should other OGC-services be found, they are kept and regarded as a fourth type of OGC-service, known as simply OGC-services.

Since WMS, WFS and WCS are known to be much more common than the other services, the search for them will not limit the amount of services found significantly. Thus the OGC-material found will still form an adequate base for testing of the agent that is to be

developed.

The search for geospatial data is limited to the HTTP-protocol. It is not in any way limited in the Internet address-range, i.e., if possible the entire world wide web may be considered for this search.
No search in web-forms will be explicitly examined, so called deep-web crawling. Some crawlers for search engines do actually perform this type of search, so deep-web-search will be indirectly examined in the meta-search parts.

Chapter 2

Overview of OGC-standards and web-crawling techniques

In order to understand the topic examined it was necessary to study some of the specifications for the OGC-standards. Also in order to understand the possibilities for discovery of web-based information, some study of web-crawling techniques were performed.

2.1 OGC Standards

Open Geospatial Consortium (OGC), is an international organization, that develops standards for geospatial data systems. In this thesis the protocols for the three open OGC-standards WMS, WCS and WFS were more closely examined.

2.1.1 WMS

Web Map Service (WMS) is a protocol standard, developed by OGC in 1999. It is used for exchanging geo-referenced map images over the Internet. The images are generated by a map-server, which is using data from a GIS database.
WMS-services can be acquired through HTTP-requests on the Internet, using a Uniform Resource Locator (URL). The URL should conform to the IETF RFC 2616, and must support GET-requests, POST is optional.
Currently there are two type of requests that are mandatory according to the WMS-standard. These are GetCapabilities and GetMap -requests. [WMS 1.3]

The GetCapabilities-request returns the meta-data of the information content of the service, and information on which request parameters that are acceptable. The format of the meta-data is ASCII-encoded XML as default, but any other format may be used. If a non-default content-type is used, the server must specify the content-type in the HTTP-response.
For any GetCapabilities-request, it is required for the URL to contain the parameters "service=WMS" and "request=GetCapabilities". [WMS 1.3]
For the versions of 1.0.0 and older the required parameters are
"request=capabilities" and "wmtver=X", where X is the version of the protocol to use.

[WMS 1.0]
For the GetMap-request, nine parameters are mandatory. In this thesis, the focus is set on the GetCapabilities-request, since the XML-document contained in its response is easier to analyze than a picture given by the GetMap-request.

2.1.2 WFS

Web Feature Service (WFS) is a standard for an interface for describing data manipulation operations of geographic features. These data manipulation operations include: Get or query, based on spatial and non-spatial constraints, create new feature instants, delete feature instances and update feature instances. In total there are 11 operations defined in the standard. These are of five types: discovery, query, locking, transaction and stored query.
The discovery operations are most important for the study in this thesis, since they are mandatory to implement and give enough information to identify a service as a WFS-service. Just like for WMS, there is a discovery operation called GetCapabilities, and it is requested via HTTP, in a similar way. The response from any WFS-request should be in either XML-format or as key-value pairs. [WFS 2.0]

2.1.3 WCS

Web Coverage Service (WCS) is a standard for an interface which allows requests for geographical coverages on the web. A geographical coverage is an object in an area, which can be spatially analyzed. The three core operations for WCS are: GetCapabilities, DescribeCoverage and GetCoverage. [WCS 2.0]
The operation mostly used in the search is the GetCapabilities-operation, for the same reason as for the other two standards examined. The structure of its request is also similar to the WMS and WFS counterparts.

2.1.4 Summary

Since all three of the sought OGC-services require all implementations to support the GetCapabilities operation, it seems like a good strategy to look for URL:s containing the string "request=GetCapabilities" in order to identify an OGC-service.
The service-parameter is also mandatory for each request, and is built up in a straight-forward way as "service=WMS", "service=WCS" and "service=WFS" for the three types of services.
Hence it is easy to separate the results obtained when searching using the GetCapabilities request-string. In the case of a missing service-parameter, there is always the option to perform a check of the content in the response.
This should not be difficult, since the default data-format for any response is XML in ASCII-format, which easily can be parsed by an automated agent.

2.2 Crawling the web

Web-crawling is the process of given a starting page, gathering all the links on that page and in some kind of order visit links found on that page. There are two base types of web-crawlers, batch-crawlers and steady-crawlers.
Batch-crawlers crawl for a period of time and then stop, after which a batch have been downloaded. The batch-crawl ends after some threshold value have been reached. This could be a time-period, a set maximum depth (max distance in links from starting page), or maybe a set maximum number of pages downloaded.

The steady crawlers, keep crawling all the time and revisit pages already crawled with some interval, since all pages on the web are subject to change. In this way, the content saved by steady crawlers keep changing all the time, and usually only the most recent material is saved. [Cho 01]

2.2.1 Deep-web crawling

Deep-web crawling is a type of non-standard web-crawling where search is carried out by manipulating web-forms with some input. Crawlers with this capability can gain access to material in databases, which may only be reached using web-forms.
This material is considered to be located in the deep or hidden web, since it is not explicitly linked to by any page.

2.2.2 Focused web-crawling

Focused web-crawling as opposed to general web-crawling is restricted to a certain type of data or topic. Where a general web-crawler downloads all content, a focused crawler will only save pages with relevant content to the specified topic.
The major difference in practical design between a focused crawler and a general crawler is that a focused crawler has a different ordering-metric. Also a focused crawler will usually automatically delete the content of pages downloaded which are off topic.
The aim of this thesis can be seen as the search for a focused crawler with the topic set to OGC-URL data.

Chapter 3

Method

The method is split on two parts. The first part is the analysis-part and the second is the prototype-development-part.
In the first part two things are examined. Firstly the analysis examines how to find OGC-services by different discovery-processes, and secondly how to automate these discovery-processes.

The method for examining discovery-processes is an iterative process, in which a first a process for finding OGC-services is discovered, and then evaluated. If the evaluation indicates that the process was useful, then a slight modification of the process may be carried out and then the process is run and evaluated again.
The slight modification may involve some minor type of development, like modifying a batch script or writing a simple testing-function.
The evaluation of a discovery-process in this context is a type of effectiveness-analysis.
The discovery-processes evaluated in chronological order are: Skylabs-Harvest, Google-Meta-Search, Yahoo! BOSS, One Step Brute-force Crawl and Web Information Retrieval Environment.

The method of finding out how to automate the discovery processes is much more difficult than the method of simple discovery. The idea examined in this thesis is that a type of manager (an automated agent) could combine the discovery-processes examined previously in a self-balancing feedback system.
This automated agent could be made of components consisting of meta-search- and Web-crawling -techniques. The idea for an automated agent, which is estimated as most effective, is developed as a prototype.

3.1 Analysis of discovery-processes

This section further explains the five different discovery-processes. More specifically it explains why a process was chosen for evaluation and how the result for the process was obtained. This thesis explains basically three types of discovery processes. These are listing-pages, meta-search and Web-crawling.

Listing-pages

Using listing-pages refers to simply downloading all OGC-links from pages which list very many of them. This implies that there is an easy way to find these listing-pages by using a search-engine or similar technology.
An alternative is if the URL to the listing-page is static, i.e., never changing. Then it doesn't matter if the listing-page is hard to find, since once found, the link to it will forever work.
Since we can never assume that an URL is static, the second option was not examined in this thesis, i.e., only listing pages which are very easy to find where examined.

Meta-search

Meta-search processes, uses some already existing search engine in a sophisticated manner, in order to extract the related information from the index of that search engine. Meta-search implies however that we at least know some word which seems useful to search for. Without search-words this process-type is useless.
The usage of already existing search-engines in an automated manner introduces another problem. That is avoiding crawler traps. All major search-engine companies have crawler traps, since automated usage of their search-engines is usually against their policies. Hence, finding a useful search engine without crawler trap is difficult. There are two ways to avoid the crawler traps of these search engines.
Either in a stealthy manner, concealing the robot and make it look like a human user, or just revealing the robot and cooperate with the search engine-company using one of their APIs and possibly paying them a small fee.

Web-crawling

The crawling discovery-processes relates to extracting more content from a seed-page than otherwise would be possible with simple download. It can also find new interesting listing-pages, or other useful pages which may lead to more OGC-links being discovered. This type of process cannot be examined as a first method, since it requires the knowledge of already existing seed-pages, i.e., pages where the crawl will start. These seed-pages should be selected carefully, else the result might be poor.

3.1.1 Skylabs-Harvest

This process was the first considered. It is the only type of listing-pages process that was examined.
Skylab Mobilesystems Ltd. is the name of a company which specializes on mobile Geographic Information Systems (GIS). This company have constructed two large listing-pages for WMS-services mainly for their costumers convenience. [SkyLabs 01] [SkyLabs 02]
These pages are among the most well known listing pages for OGC WMS-services. In fact if one searches for "public wms list" using the Google-search engine, the first result is Skylab's old listing-page.
Their new page is linked from this old page, and is not yet as well known.
The actual downloading was performed by a simple HTTP-harvest-robot which downloads all links on a page into a text-file.

3.1.2 Google-Meta-Search

This was the first meta-search process examined. Google is currently the most popular search engine for the web and also is the search engine which indexes the largest part of the web. [WWWS 01]
Therefore the Google search engine should be a good target for meta search.
The idea on how to use Google for meta-search, comes from a blog on WMS-service-mining. [WMS Mining 02]
A series of URLs to Google's search engine was constructed automatically in the following way:
"www.google.com/search?q=allinurl:+"request=getcapabilities"&start=x".
Where x is the offset of the results from the first result. The offset-value was the only difference between the URLs constructed, and it was set to values from 0 to 990, with a difference of 10 between two consecutive URLs. ($x \in \{0, 10, 20, ..., 990\}$)
These URLs were made to use the Google engine for search of "request=getcapabilities" strings in URLs. This is a good approach since all URLs to WMS-services should contain this string, if OGC-protocol for WMS of version 1.3.0 is used. Actually this request will find other types of OGC-services too, since this parameter is not specific for WMS.

After a page had been manually downloaded from Google, it was parsed for OGC-URLs using a standard offline HTML-parser.
This parser regarded URLs which contained the strings "request=capabilities" or "request=getcapabilities" as OGC-URLs. (i.e., this test was using an early version of the parser for OGC-services, which means that some valid results could have been missed.)

The apparent lack of any type of robot search-API for the Google search-engine, means that one would need to, either use it stealthy or find some other, more suitable search-engine.
A stealthy usage of the Google search-engine would probably severely impair the performance of the robot. For a robot to be stealthy it would probably have to mimic human behaviour, which would exclude multiple parallel connections to the search-engine.

A human using a web-browser following links cannot be as fast as a computer, which may perform multiple parallel connections to the search-engine. However, since Google does not allow the usage of automated robots for their services, the development or testing of a stealthy Google meta-search-robot, is not a part of this thesis. [Google 01]
All URLs in this test were generated automatically, but the download of the content was performed manually.

3.1.3 Yahoo! BOSS

This was the second meta-search process examined. It is based on Yahoo!'s search engine. Yahoo! does have an API, which allows direct usage of their search-engine for a small traffic-fee. This API is called Build your Own Search Service (BOSS).
The BOSS API is a restful API, which is not at all difficult to implement and supports OAuth for verification. This verification makes use of a HMAC-SHA1 algorithm. Since BOSS is a non-free API, the verification is very important. For every connection to BOSS, a small fee has to be payed for the traffic.
Despite being non-free, the BOSS API is important, since it gives fast and well-formated results from one of the web's largest search-engines.
The fee is also currently very small, about 0.8 USD for every 1000 queries.

For the purpose of testing the effectiveness of meta-search with the Yahoo! search-engine, a total amount of 158 test-searches were carried out. The test-searches were run at two different occasions. The reason behind this was an update of the BOSS-API, which required an implementation change. And so all tests were run again, to test if the search still worked after the implementation change.
At the first occasion, 67 test-runs were made. These test-runs were performed between the 2nd of may and the 4th of may. The same 67 queries where then used again in a new run on the 5th of may. On the 6th of may another 24 queries were used in BOSS and added to the second set of results. Hence there are results from the old run for 67 queries and from the new one for 91 queries.

In URL search

Just like for Google, there is the possibility to search for information in URL-references, there are two versions of the URL-search for Yahoo!.
These are "instreamset:url:" and "-inurl:" -searches.
The first type will only include the pages for the URLs hit. The latter type will search URL-references for pages and also include pages which contain URL-references hit.

Two comparisons with Google was carried out, where the words for the parameters for OGC-capabilities-requests were used. The first search used the words "request", "service" and "wms. The second used the words "request=getcapabilities" and "service=wms". The result is shown in table 3.1. Both filtered and unfiltered results are shown. The table show that Yahoo! finds less than half as many services using this method, as compared to Google.
It also shows that Google has much more duplicates in their search-results.

In total there are no more than about 500 WMS-services that are found by one search-engine, using this method. This means that the in-URL method should be complemented using standard queries. The selection of these queries is however a non-trivial problem.

It can also be concluded that most capabilities-documents are not indexed by search engines, or they are hidden due to sensitive content restriction policies. In either way, there should be plenty of pages available, which link to the capabilities-documents.
The first in-URL method for Yahoo! does not appear to work for the query word "request=getcapabilities", so the result count for this part is 0. This can be due to some content-restriction policy from Yahoo!.

Search Engine	1	2	Unfiltered 1	Unfiltered 2
Google	505	522	829 000	223 000
Yahoo!	207/773	0/583	3080/737 000	0/7 210

Table 3.1: In URL search comparison with Google

Multi-threaded downloader

Since the Yahoo!-search engine perform poor for searches in HTML-hypertext references for URLs, the results fetched are usually links to pages which contain the content sought. Hence these pages must also be downloaded. For every query, a limit of 2000 total results have been set. Despite this limit every step will generate a lot of content that must be downloaded and parsed. In order to speed up the discovery-process a multi-threaded downloader (MTD) was developed. The MTD uses a set number of threads. Each thread has a downloader assigned, which has a queue of URLs. All URLs inputted into the MTD are split as evenly as possible on its downloaders. Each downloader also has a set timeout, after which it will abort the connection. This was made in order to avoid the MTD to get stuck on some slow download. The downloaders will resolve HTTP-redirects up to a number of steps and may use the range-header for HTTP to limit the size of pages downloaded. Also all downloaders support chunked HTTP-transfer encoding.
The setting for the BOSS-experiment was 10 threads for the connections to BOSS and 50 threads for the download of pages pointed at by BOSS. Both MTD:s used 30 seconds timeout and a 400kB page-size limit. Since the number of URLs to download in the second step was usually much larger than the URLs used for BOSS, there was a need for more threads for the second part. The page-size limits and timeouts settings are defaults which have been tested to be fair trade offs between speed and coverage.

Search query selection

Since there is no way to search in the HTML-hypertext references in standard search-mode, using the Yahoo! search-engine, there is no trivial way to find suitable query-words.
If this was possible, then one could simply only use the protocol-parameters specified by OGC for the search.
The set of search-queries used for the BOSS-experiment (Q), consists of queries added at

four different occasions. ($Q = Q_1 \cup Q_2 \cup Q_3 \cup Q_4$)
All of the 403 queries in Q are made from 37 words (W).
These words were added in succession at four occasions, as the experiment was carried out. ($W = W_1 \cup W_2 \cup W_3 \cup W_4$)
Due to some restrictions, the number of examined queries in Q, were only 91 (23 %).
The initial selection of queries (Q_1) is based on the protocol-standards for OGC-services and the URLs found by previously examined discovery-processes. The later selection of queries is based on the results from previous queries.

Initial search-word selection (W_1): The first elements of W, were constructed using five mandatory protocol-parameters for WMS, WFS and WCS -services.
Although it is not possible to search in hyper-text references, it makes sense to include these words, since it is common that the reference-text for an URL is similar to the link-text, which is search-able. The search-tests performed using the BOSS-API, searches all of the non-HTML text for the pages. The parameters used were: request=getcapabilities (A), service=wms (B_wms), service=wcs (B_wcs) and service=wfs (B_wfs), request=capabilities (C). $W_{P1} \subset W_P = A \cup C \cup B_{wms} \cup B_{wfs} \cup B_{wcs}$
As mentioned in section 2.1.1 and 2.1.4, these parameters are mandatory for the services searched for, and should therefore be useful for finding OGC-services.

Also frequency analysis was performed on the components of OGC-links gathered by previous methods for discovery (SkyLabs and Google MS). A component was regarded as a part of an URL separated by a /-character. For more information on how this was done, see section 3.5.2.
A list was made, sorted on actual component-frequency, from which the top 7 most promising looking words were manually selected (W_{C1}). These are regarded as C_1 to C_7. ($W_{C1} \subset W_C = \bigcup_{k=1}^{7} C_k$)
This simple analysis can reveal if an OGC-link typically makes use of some predefined folder-structure, which may be the case whenever a third-party web-based GIS-application is being used.
For instance, OGC-services which uses the MapServer application typically has an URL which contains the string "mapserver". This string was included as C_3 [MapServer]
To sum up, the initial search-words were: $W_1 = W_{P1} \cup W_{C1}$

Q_1: Having selected the initial set of 12 search words (W_1), the first set of 57 search-queries $Q_1 \subset Q$, were constructed, using combinations of the words in W_1.
The maximum number of search-queries to get from W_1, is the size of its power-set $|p(W_1)| = 2^{|W_1|} = 2^{12} = 4096$.
Since there was no time to examine 4096 queries in BOSS, and it would also cost to much, the number of queries was reduced in this way:
No queries consisting of more than 2 search-words were made, since already combining two words reduced the number of results significantly.
Just for the purpose of acquiring OGC-links, one could make 12 searches for each of the search-words alone in each string. This does however lead to potentially very many results in each search, and the download and parsing of the results obtained would prob-

ably take too long time.
If one only takes search-strings of size 1 and 2, that would lead to $\binom{12}{1} + \binom{12}{2} = 12 + 66 = 78$ queries for Q_1.
The number of queries was limited further by realizing that combining the words in W_C with each other is probably not a good strategy, since they turned out to be more specific than the the words given by the protocol restrictions (W_P). (i.e., they had much fewer search-results when searched on alone)
Actually the average number of results for a single word from W_{P1} was about 40 times higher than for a word from W_{C1}.
So $\binom{|W_{C1}|}{2} = 21$ queries could be removed from the 78, making it 57.
For all queries used in the test of BOSS, it is made in one of following four ways:

- a protocol-word, $s \in W_P$
- two different protocol-words, $s_1 \wedge s_2 \in W_P, s_1 \neq s_2$
- a frequency-analysis word, $s \in W_C$
- one protocol word and one frequency-analysis word, $(s_1 \in W_P) \wedge (s_2 \in W_C)$

Q_2**:** After 36 of the search-cases from Q_1 had been tested, the query set was expanded with Q_2, adding 36 more queries. A total of 6 new words $W_{C2} \subset W_C$ were selected using a new frequency analysis in a similar way as before. Just as before, the frequency-analysis was carried out on all previously found URLs.
Q_2 was created by combining the 6 new search-words with the protocol-words (W_P). The size of the new set was: $|Q_2| = |W_{C2}| + |W_{C2}| \cdot |W_P 1| \ = 6 + 6 \cdot 5 = 36$

Q_3**:** After 42 of the search-cases in $Q_1 \cup Q_2$ had been tested, another set of search-words W_{C3} was made. This set was constructed using frequency analysis on words found in the content of web-pages listing OGC-links. A total of 12 words were used to construct the search set Q_3, of size $|Q_3| = |W_{C3}| + |W_{C3}| \cdot |W_{P1}| = 12 + 12 \cdot 5 = 72$

Q_4**:** Finally, 238 more queries were added by including 7 non-mandatory parameters of request to OGC-URLs, following the OGC-protocols for WMS, WCS and WFS.
($W_{P2} \subset W_P$)
The total number of queries for Q_4 were:
$|Q_4| = |W_{P2}| + \binom{|W_P|}{2} - \binom{|W_{P1}|}{2} + |W_{P2}| \cdot |W_C| = 7 + 66 - 10 + 175 = 238$

Search space size restrictions: In order to further limit the number of test-cases from the initial constructed 403 ($|Q| = |Q_1| + |Q_2| + |Q_3| + |Q_4| = 57 + 36 + 72 + 238 = 403$), some queries were excluded from test in BOSS.
This exclusion was based on the total number of estimated results given by BOSS.
It was not initially known that the estimated results given by BOSS was a large overestimation, so the upper limit was first set to 4000 results. It was later increased to 400 000,

since it became clear that BOSS on average gives an overestimation of about 65 times more than the actual unique count of pages.
This overestimation can be due to bad results not being filtered away initially, or very similar results being included.

A single request to the BOSS-API may give up to 50 results, so the expected maximum number of requests for a query would be: $r_c = \frac{400000}{50 \cdot 65} \approx 123$
Taking this into account , the expected maximum cost for each query was: $\frac{0.8}{1000} \cdot r_c \approx 0.1$ USD. [Yahoo 02]
This seems like a decent upper limit for the cost. If all queries selected, would be used in BOSS, the total cost would as most be about 40 USD. Since this is too much, the queries which had an estimation of more than 400 000 results, were not examined at all. A total of 15 cases were removed based on this criteria. 6 of them from Q_1 and 9 from Q_2.

For Q_3 the results obtained had a very low quality, so the tests were aborted after the first group of 5 queries had been examined. The reason behind this was probably that most of the queries in W_{C3} contained special characters that BOSS filtered away, leading to bogus results.

For Q_4 only the single word queries were used, since all of them had a results count that was under the upper limit. Since the reason behind the combination of words, is to reduce the total results count to get under the limit, all the 231 cases for queries using combined words for Q_4 were discarded.
The total number of examined queries in BOSS was: $(57 - 6) + (36 - 9) + 5 + 7 = 90$

Order of queries examined: The order at which the test-cases were examined might seem a bit odd, but this is mainly due to the fact that we did not at the time, know how incorrect the BOSS-estimation of total-search results was.
That is, we kept increasing the limit for total-results. The test-cases were initially grouped by their total-result size reported by BOSS and examined close together with queries of similar size. There were a total of four groups initially, in the ranges:
$[500, 2000]$, $[100, 499]$, $[2001, 3999]$ and $[1, 99]$. The most interesting query-groups were examined first. Which groups were most interesting were determined by the relative difference between the groups average of total results and the average of total results for all queries (i.e., the groups which had an average size closest to the total average size, were examined first.) The values for this was 1.59, 1.96, 4.42 and 9.30 for the four initial ranges respectively.

The order within each group was random. At first time the upper limit for total results for a query was set to 4000 only.
The final suitable size of the limit ended on 400 000, at which the actual real total-results in BOSS, rarely exceeded 2000.
The query-selection-order is further explained by table 3.2. Due to an error in implementation, query case 62 is same as case 67, i.e., one query is searched on twice, leading to 91 cases with 90 different queries.

Case no.	Origin	Results-limit (k)	Count
1-29	Q_1	4	29
30-36	Q_1	10	7
37-42	$Q_1 \cup Q_2$	20	6
43-47	$Q_1 \cup Q_3$	20	5
48-66	$Q_1 \cup Q_2$	40	19
67-79	$Q_1 \cup Q_2$	400	13
80-86	Q_4	400	7
87-91	$Q_1 \cup Q_2$	400	5
1-91	Q	4-400	91

Table 3.2: Search order of BOSS-queries

3.1.4 One Step Brute-force Crawl

The first Web-crawler process examined. It is a simple one-step Web-crawler, which was made for testing the actual benefits with shallow crawling. This was a simple extension to the already existing multi-threaded downloader used for downloading BOSS-requests. This crawler was made as a test of the difficulties and benefits associated with crawling. For this, the 15 test-cases from the BOSS test, which had the smallest number of result-pages prior at step query case no. 66 were used. The reason for this was that since this crawler has no metrics for excluding irrelevant pages or even detecting pages downloaded earlier, it will quickly download a very large amount of pages, even when given a small set of start pages.
On average the measured number of links followed per page was 36. So if, for instance 28 start-pages were given, then the OSBC-crawler would download about 1000 pages.

3.1.5 Web Information Retrieval Environment (WIRE)

WIRE is the second Web-crawler tested in this thesis. It was developed as a project at the center for Web Research at the University of Chile. The aim was to study the problem of Web search, by creating an efficient search engine. [Car 01]
Since the OSBC-test revealed that crawling might actually be useful for discovery of OGC-URLs, it was decided that another, better crawling method should be tested.

WIRE is open-source licensed and made as a research-project, so it appeared as a decent choice.
WIRE is batch-oriented and made up by several components. For the actual crawling there are four components. These are: Manager, Harvester, Gatherer and Seeder.
The Manager is responsible for long-term scheduling and the Harvester handles short term scheduling and network transfers. The Gatherer parses data and extracts links, while the seeder resolves links for new pages to download. [Car 01]

The WIRE crawler is typically run for a number of cycles. A cycle for WIRE is when these four modules are run in the following order: Manager-Harvester-Gatherer-Seeder. Before any cycle can be run, the seeder must be run once to initialize the crawler, where it takes a newline-separated list of URLs as input.

After each cycle, the crawler may reach pages at a depth of one more than before. If the robots protocol is followed, then the maximum depth reached by the crawler equals the numbers of cycles performed minus two. $D_c = C_c - 2$ This is because the crawler must first download the robots.txt file for all new sites found, and may then in the second cycle, download the actual content, if the robots-protocol allows it.

WIRE is a general crawler, so it cannot be used in a straightforward way for focused search. One can however steer WIRE in the right direction by carefully selecting the seeds to begin with and aborting the crawl at the right number of cycles.
This imposes the difficulty of, given a set of seeds, finding an optimal way of splitting them on groups for crawls, and selecting the crawl-depth for each group. To solve these two problems, we performed a number of experiments with WIRE using the full list of links to results from the BOSS-experiment (P_{t1}) as the base set for the first four crawls.

It was then realized that it is better to use more recently downloaded URLs, so the rest of the crawls used newer results from BOSS (P_{t2}). These seeds were downloaded from BOSS on June 21th, 20 days after the other seed-set was downloaded.

Seed selection from BOSS

As mentioned before, the seeds for WIRE were downloaded from BOSS at two different occasions. Most crawls and analysis of the WIRE-method were performed using the newer seeds (P_{t2}). In order to limit the number of queries used for BOSS seed-fetch, due to the cost of its usage, a selection algorithm (S_{Aq}) was used to find a minimum number of queries to use.
The queries used for getting the seeds (Q_w), were a subset of all queries used $Q_w \subset Q$, when examining the BOSS-method. The problem of finding the minimum number of queries Q_m which hit all sites found by using Q for BOSS, equals the set cover problem, which is NP-complete. [Set Cover]
Since a complete cover is not necessary, S_{Aq} selects a small amount of queries which cover a certain portion of all sites. This portion was set to 75% in the experiment. This limit made the algorithm much faster, setting it lower will speed it up even more. The limit of 75% did however, lead to a decent trade off between speed and coverage.
For a detailed description of S_{Aq} see section B.3.4
Typically a run of S_{Aq} resulted in the selection of 21 queries, which would lead to a reduction of the boss traffic by about 77%, while maximumly reducing the count of total results by 25%. In total it made the second seed-extraction from boss at least 3.25 times more money efficient. This algorithm is very useful, if BOSS-seeds need to be re-fetched for WIRE-crawl using the same queries as before.

WIRE Seed partitioning

Since WIRE makes good use of parallel downloads from different sites, it makes perfect sense to use large groups with many initial seeds for each crawl. This will however make it more difficult to control the crawl, since the measurement of effectiveness will be an average over all the seeds. It will be more difficult to tell which seeds are better suited

for deeper crawls. Also more overhead data might be downloaded whenever large sets of initial seeds are used for WIRE.

For every set of base seeds (S_b), there should be an optimal partitioning of seeds S^o, which contains sets of seeds to run in separate crawls.
Optimal partitioning in this context, is a partitioning which maximizes the total number of OGC-links discovered, given the time T.
The total effectiveness can be defined as $E = \sum_{k=1}^{N} \frac{C^o(S_k,d_k)}{t_k}$, where N is the amount of crawl groups ($N = |S^o|$), S_k is crawl-group k, which takes time t_k to complete, after crawled to depth d_k.
The number of unique OGC-URLs found after crawl of seeds s to depth d is: $C^o(S,d)$

The ideal crawl-depth (d_k^o) for group S_k, might be possible to detect at crawl-time. That is if the OGC-link discovery speed have reached a low value at depth d_v, the crawl may be aborted at that depth.
If we assume that this is possible, then what remains is how to partition S_b into S^o. If we further assume that we have some method R for ranking the seeds by quality, such that they may be ordered by it, then what remains is to figure out how large each of the sets in S^o should be. The grouping of seeds of similar quality together, is desired, since it would reduce download-overhead and make the crawling more controllable.
This is perhaps not a trivial conclusion, but it makes sense to crawl the better seeds to a larger depth than the bad seeds. And since crawl depth is decided per group, it would be best to place seeds of similar quality in the same groups.

The simplest type of partition is a split of S_b on k equally large subsets. Although this may not be optimal. The best partition would consist of exactly M groups, where M is the maximum ideal crawl-depth for a seed in S_b plus one, since there is no need to have more than one group for each different depth d_k.
If one instead of R have a function R_s, that not only ranks the seeds, but also gives some type of score , convertible to desired depth-level, then this is not a difficult problem.
The problem would then be how to compute the desired crawl-depth d_e for a seed s.

The first WIRE experiments that was made, were using Yahoo!'s own ranking methods as an estimate to R. Using this estimation, first a test was made to see the benefits with crawling and if the ranking-algorithm was effective. Four groups were constructed using 7680 seeds. The smallest group contained the 15 best seeds and, were crawled to depth 3, the next group had 8 times more seeds (120 best) and crawled to depth 2, and so on. The idea was that the different crawls would take about the same time. The selection of these seeds, was made using the SA_1 algorithm. See appendix B.3.4 for a detailed description of this algorithm. Table 3.3 shows the result of these first crawls.
It is clear that the most effective crawl group was CN2, and least effective CN4. It is however interesting to note that more OGC-URLs was found in CN4 than CN3, indicating that crawl to depth 3 may be better than depth 2, if the time required is not that crucial.

Rank	Group-size	Groups	ID-Name	Depth	OGC-URLs found	Time (h)
1-15	15	1	CN4	3	465	3.15
1-120	120	1	CN3	2	429	1.33
1-960	960	1	CN2	1	2457	1.18
1-7680	7680	1	CN1	0	1380	1.85

Table 3.3: First WIRE-crawl groups

Ideal crawl depth

Is there a way to compute the ideal crawl depth for a seed s, or is there a way to detect when to abort the crawling of a seed-group S_k?
The first question is hard to answer, but the second one might not be and its answer could give an approximate answer for the first question.

In an attempt to answer the second question after rigorous statistical analysis, we split up the 4400 best seeds given by S_{A1} out of approximately 10000 possible on 60 groups. The amount of hosts totalled about 4800, so more than 90% of the hosts were crawled, since only one seed was initially selected per host. It seemed to be a fair amount, there was no need to crawl all hosts, since up to 25% was skipped anyway by the seed-selection algorithm S_{Aq}.
The 400 first seeds were split evenly on groups of 20 each, and the 4000 rest were split evenly on groups of 100 seeds. All groups were crawled to depth level 3.
The test groups for the 200 first seeds is called CNF, the groups from position 201 to 400 CNG and the last 4000 seeds belong to the group CVX.
Data downloaded by the crawler, was saved for each different depth level and crawl-group, yielding a total of $4 \cdot 60 = 240$ data instances.
Table 3.4 shows the setting and result for these crawl-groups.

Rank	Group-size	Groups	ID-Name	Depth	OGC-URLs found	Time (h)
1-200	20	10	CNF1-10	3	1976	26.35
201-400	20	10	CNG1-10	3	1771	24.00
401-4400	100	40	CVX1-40	3	4469	169.33

Table 3.4: Major WIRE-crawl groups examined

See section 4.1.3 for further information on the analysis of these crawls.

High depth crawl

We made three crawls for a maximum depth set to 5, in order to test if the OGC-link discovery speed would continue to increase after depth-level 3.
Even though three sets form a very small statistical background, these tests took a long time to perform and thus we could not execute many of them.
The test was made by continuing the crawl of three groups from depth 3 to 5. The three test-groups selected was: The one which had the highest value for number of listing-pages found per minute at step 3 ($n_l(3)$) in group CNF (CNF1) and the one with the

highest $n_l(3)$ value from group CNG (CNG2), and also a close to average performing group from CNG as comparison (CNG7).
Table 3.5 shows the high-depth crawls along with their 3-depth counter-parts. As can be

Rank	Group-size	Groups	ID-Name	Depth	OGC-URLs found	Time (h)
1-20	20	1	CNH1	5	1412	4.48
1-20	20	1	CNF1	3	1286	2.45
221-240	20	1	CNGH2	5	235	4.57
221-240	20	1	CNG2	3	145	2.43
321-340	20	1	CNGH7	5	607	4.80
321-340	20	1	CNG7	3	491	2.72

Table 3.5: High-depth WIRE-crawl groups examined

seen, the benefits of crawling to level 5, are small, from 10 to 62 %, when the time required is about the double.

3.2 Analysis of automation-processes

This chapter explains more in detail, the second step of the analysis-part of the method for this thesis. This part of the analysis is divided on two parts. The first one is conceptual design of an automated agent, the second one is test of the principles of the design.
All of the concepts for designs mentioned here are made like feedback systems, utilizing two mayor parts. These parts are the seeder and the manager.
The seeder tells the manager which pages to download or which word-strings to search for, and the manager performs the download and gives the seeder feedback of the result, after which the manager once again receives instructions from the seeder.
Two of the methods examined in this thesis, were by far superior to the other methods and can be implemented in automated agents. These are the BOSS-method and the WIRE-method. These two methods together found 99.87 % of all the live services found by discovery-methods. This is the reason why only these two methods are considered for an automated discovery-agent.

3.2.1 BOSS agent

The pure BOSS-agent works as follows. The seeder has a base set of boss-query-strings Q_0 that are usually good for finding OGC-services. It gives it to the manager, which connects to BOSS and downloads the results of links $P_0 = B(Q_0)$, when searching for Q_0. Then the manager downloads $R_0 = D(P_0)$ from the web, which is all pages pointed to by the URLs in P_0.
After that , the manager sends back a collection of all useful information ($I = U \cup T \cup A$). This includes:

- OGC-URLs ($U_O \subset U$)
- other URLs ($U_N \subset U$)
- words in titles for OGC-URLs ($T_O \subset T$)
- words in titles for other URLs ($T_N \subset T$)
- words in link text for OGC-URLs ($A_O \subset A$)
- words in link text for other URLs ($A_N \subset A$)

Finally the seeder recalculates its knowledge-base using the feedback information (I) received, and computes the new best set of queries to use Q_1, and sends it to the manager. This calculation can be done, using a weighted combination of Info-similarity metric for: URL-parts (U), titles (T) and link-texts (A).

3.2.2 BOSS-WIRE agent

The BOSS-WIRE agent is made in similar way as the pure BOSS-agent. The only difference is that instead of directly downloading R_i from the web, WIRE will handle the download of R_i and of pages R_i' which are linked to from a page $r_i \in R_i$.
Then if many OGC-URLs are found, the crawl continues to the next depth-level until

maximum depth has been reached or the crawl becomes ineffective.
One problem with this method is that crawling with WIRE is a slow process, which would lead to a lot of wait-times for the system. Also data downloaded by WIRE must be periodically erased, so that one will not run out of disk space.

3.2.3 WIRE-WIRE agent

The only difference between this agent and the BOSS-WIRE agent is that the seeder will give links to download to the manager, which in turn excludes the BOSS part and directly gives the links to WIRE.

3.2.4 Evolving agent

Since the differences between the three mentioned agents are small, there is a possibility of an evolving agent. Starting as a pure BOSS-agent, and gradually evolving into a BOSS-WIRE agent and then finally becoming a WIRE-WIRE agent.

The evolution can be time-, step- or progress-based. A time-based agent can evolve at a static pace. For instance changing from step 1 to 2 in a day and step 2 to 3 in two days. The step-based evolution, simply runs each step a number of times before evolving. The final type of evolution is progress-based, which means that the agent evolves when the progress of discovery hits a certain minimal value for each step.
Determination of the constants for the evolution can be performed either experimentally or set dynamically by the agent itself. The latter option would require a more advanced agent.

Perhaps a combination of the three metrics can be used to form a rule for when the agent should evolve. For instance, the agent has a maximum time it may stay in a state, and a maximum number of steps may be performed in the state and also there is a minimum performance required for remaining in the state.

3.3 Prototype Development

The automation-agent chosen for development was the pure BOSS-agent. The reason for this was that this is the type of agent which would require least development and testing time, and thus it was in practice the only agent which was in the time scope for this thesis.
As described in 3.2.1, the agent will use a set of initial queries Q_0 for getting the first results.

After the first results have been received from BOSS, the score for all possible search-words are computed. Then the total BOSS-result count for the $k \cdot 5$ best words ($b_r(w)$) are fetched from BOSS. Where k is the number of queries to use per iteration. This is done in order to determine whether a search-word should be used alone or combined with another search-word.
Then the list containing triplets of type (w_i, r_i, d_i) is updated and saved. This list holds the result-count (r_i) for each word examined (w_i), along with the date when the result-count was fetched from BOSS (d_i).
The date is saved, so that old measures may be updated after a certain time has passed.
There is a certain minimum score required for a search-word to be examined, in order to prevent the list of usable search-words to get filled by less useful words.
The score for queries are then computed, and the k best scoring queries (Q_1) are used in BOSS in the next iteration.

3.3.1 Seed-queries selection

How the seed-queries (Q_S) are selected may effect the performance significantly.
The results from the manual tests of the BOSS-method were used to select five queries which usually gives many good results. The queries selected were "geoserver", "wmsservlet", "esrimap", "wmsconnector" and "request=getcapabilities". The first four of these are words commonly found in URLs to OGC-services, due to implementation details for some commonly used third-party map-server softwares. The last word is taken from the OGC-standards specification.

The four non-OGC-standard words were selected among the seven common words found in the first manual selection of search words, described in section 3.1.3. The "mapserver"-word was skipped, because it only gave 13 OGC-link results when searched on alone, which was much fewer than the others. Two words, "ogcwms" and "arcgis" had too many search-results in BOSS, and were also skipped, since they could not reliably be used to fetch all results for those words. Also the time cost would be too large for these words.

3.3.2 Query-score computation

After the first iteration of queries had been used (Q_0), the results fetched were analysed for frequencies of parts in URLs (U), link-text (A) and title-text (T), in order to assign scores for all search-words found. (W)

The score $p(w)$ for a single-query search word $w \in W$, is computed in the following way:

$$p(w) = (k_U \cdot \frac{C(U_O, w)^2}{C(U, w)} + k_A \cdot \frac{C(A_O, w)^2}{C(A, w)} + k_T \cdot \frac{C(T_O, w)^2}{C(T, w)}) \cdot t_f(w)$$

Where: $C(S, w) = \sum_{v \in S} f_d(w, v)$ and $f_d(x, y) = \begin{cases} 1 & \text{if } (x = y) \\ 0 & \text{otherwise} \end{cases}$

k_U, k_A and k_T are the coefficients associated with score for URL-part (U), URL-text-part (A) and title-text (T), respectively.
By default these are set to 0.25, 0.25 and 0.5. This is due to the fact that title-text is available in lesser extent, and should therefore matter more.
$C(U_O, w)$ and $C(U, w)$ are the count of positive and total occurrences of word w among all words found in URLs.
In the same way, $C(A_O, w)$, $C(A, w)$ and $C(T_O, w)$, $C(T, w)$ are count of positive and total occurrences of w in URL-texts(A) and titles (T).

A positive occurrence for a word w in an URL or URL-text, is when the related URL is an OGC-link. In similar way, whenever a page holds at least one OGC-link, the words in its title will be counted as positive.
The time-factor, $t_f(w)$ for the word w, is an integer in the range $[0, 1]$. It is calculated as follows: $t_f(w) = 1 - e^{-\lambda_t \cdot h(w)}$
The $h(w)$ function returns the number of hours since last time the word w was used in a query-search, or infinity if it has never been used. λ_t is the decay-parameter, which can be computed like: $\lambda_t = \frac{-ln(0.5)}{t_h}$, where t_h is the expected half-time in hours for the web-pages hosting the OGC-links.
A decent value for t_h is 1200, which is equal to 50 days, and is a measurement for the the half-time of the web in general. [Cho 01]

This single-query score is only applicable when the total results $b_r(w)$, for the word considered does not exceed the threshold limit, which is set to $L = 400000$ by default. See section 3.1.3 for an explanation of this limit.
When limit is exceeded, words need to be combined, in order to reduce the number of search-results from BOSS.
In this thesis, only combinations of two words are used, since this was simplest, and usually sufficient for reducing the results-count to get below the limit L.
The score for a combined query using words w_1 and w_2 from W_c is:

$$p(w_1, w_2) = \frac{t_f(w_1, w_2) \cdot (p(w_1) + p(w_2))}{c_d + Max(\sum_{(w_x \neq w_2) \in W_c} 1 - t_f(w_1, w_x), \sum_{(w_x \neq w_1) \in W_c} 1 - t_f(w_2, w_x))}$$

Where c_d is the restriction factor for combined queries. A higher value will lead to single-queries being more prioritized over combined. A value of 2 here means that neither single nor combined are prioritized.
The $t_f(w_1, w_2)$ function gives the time-factor for the query which uses both word w_1 and word w_2, in a similar way as for single queries, i.e., it is based on the time since the query was last used. The denominator includes the maximum value of sum of all the complements of the time-factors, for both of the two words w_1 and w_2.
The reason behind this is, that there will always be some correlation in results when the

same query-words are used again, so in order to limit this, a combined query which has words which recently have been used often in another combined query, will get a lower score.

3.3.3 Improved query-score computation

One mayor flaw of the previous query-score computation, lead to the development of a somewhat more advanced metric. Since the standard score-computation does not take the correlation of specific search-words into consideration, it will repeatedly ask redundant queries to BOSS, i.e., consider an URL of type x/A/B/y, where the words A and B always occur together in the same URL, and x and y are variable. The old metric will assign equally high values for both A and B, leading to queries containing both of them, when it is enough to choose one of them.

The improved score-metric uses the same metric as before, except that it multiplies the result with the inverse correlation value. This value is computed as follows:
The set of highest-scoring words in U is $U_w \subset U_O$. For a potential search word $w_c \in U_w$ among all URL-part words, the inverse correlation value (c_{w_c}) is:

$$c_{w_c} = min_{w \neq w_c \in U_O}(1 - \frac{b_{w,w_c} \cdot m}{j_w})$$

Where:
$b_{w,w_c} = \sum_{U_w^s \in Y} f_c(w, U_w^s) \cdot f_c(w_c, U_w^s)$ and $j_w = \sum_{U_w^s \in Y} f_c(w, U_w^s)$
and U_w^s is the set of URL-part-words in the same URL, in the set of all URLs (Y)

$$f_c(x, S) = \begin{cases} 1 & \text{if } (x \in S) \\ 0 & \text{otherwise} \end{cases}$$

$$m = \begin{cases} 1.0 & \text{if word } w_c \text{ was used in a single query.} \\ c_F & \text{otherwise} \end{cases}$$

The factor m is used for setting the limit of, how much the usage of a combined query may effect the correlation of its components. For single queries it is of course 100 % (1.0), but it is difficult to set this for combined queries. The default value for c_F is set to 75 %. This setting has not been found by experiments, and may be changed in the prototype. It should be somewhere in the range $(0, 1)$.

3.4 Testing Environment

For the purpose of determining how to best, both identify and find OGC-services, a test-framework was written in Microsoft Visual 2010, using the C# programming language.
The test-framework was primarily used for generating statistics for the results of different methods for finding OGC-services.
For more information about the statistics-generation, see section 3.5.
The testing framework consists of modules for HTML-parsing and OGC-service-detection.

3.4.1 HTML-parsing

The parsing of HTML-pages was performed using an open-source library called Majestic-12-HTML-parser. The library was developed for C#.NET with the focus set on high performance. [M12]
The main reason for using a third party parser for the web-pages is that it is not a simple task to actually parse a page for links. Since the format of the pages downloaded by different methods may be very different, it is difficult to come up with a parser which is generic enough and at the same time correct and fast.
The implementation of the Majestic-12 parser was straightforward, and it thus helped speeding up the development of test-methods for the different OGC-discovery experiments.

3.4.2 OGC-service detection

Before any methods of discovering OGC-services can be tested, there is a need to correctly identify new unique OGC-services, and thus separate them from other services and other OGC-services already found. This problem requires methods for identifying OGC-links (classification) and detecting duplicate links.

The problem of detecting a duplicate link to an OGC-link can be considered even harder than identifying the link as an OGC-link. The reasons are many. Firstly one has to define what a duplicate OGC-link is, in this context. Not even this is trivial.
One acceptable definition is that whenever two links for the getcapabilities-request point to the same document, one of them is a duplicate. This is perhaps a too week definition, since there might be slight differences in different stored copies of the same documents, such as additional white-space characters and different version numbers etc.
If one despite this, assumes that there is a function $E(D_1, D_2)$, which given two meta-data documents D_1 and D_2, can tell if D_2 is a duplicate of D_1. Then having only the links to D_1 and D_2, one can see that it is impossible to say for certain that they are not duplicates. There is always the possibility of a site S having a mirror site S', which holds the exact same content as S, except that it has a completely different domain-name. There may also be more than one domain-name mapped to the same site.

The problem of identifying an OGC-service, is not trivial, and there are at least two ways to handle it. These are:

1. Examination of the URL pointing to the service, and compare it with the protocol standards given by OGC.

2. Downloading the data pointed at by the link, and compare the syntax of the content to the OGC-standards.

If the first approach is used then particularly the URL-parameters should be examined. According to the latest versions of the protocol standards for WMS, WCS and WFS, all servers for these services are required to support the getcapabilities-request. An URL for such a request must have a parameter named "request", with the value set to "getcapabilities" or "capabilities".
Furthermore it is also required for any OGC-service URL to have the type of the services specified as another parameter. That parameter should be called "service" and its value should be the name of the OGC-service that the URL points to, which should be either of "WMS" ,"WFS" or "WCS" in this case.

The second approach may be used as a complement for the first one, whenever the estimated probability of the link actually being an OGC-URL is not sufficient. For instance, if a link only has one of the two mandatory components, it may be the result of an erroneous implementation of the protocol or some other service-request URL which appears similar to an OGC-URL. If the URL is an erroneously implemented OGC-URL, the actual service it points to may still be of high quality and following an OGC-standard, so such URLs should not be overlooked. In order to find new unique OGC-services, there are a number of components which are needed. These are:

1. Classifier by URL (C_1)
2. URL-Filter (F)
3. URL-duplication remover (R_1)
4. Content downloader (D)
5. Classifier by Content (C_2)
6. Duplication-removal by content (R_2)

Component no. 1 to 3 are for the URL-examination method, and 4 to 6 are for the data-examination. For the URL-examination method, the components contained, were improved on several occasions.
The reason behind this was, that it became easier to find new ways to remove duplicates once more URLs had been found. There were many types of duplicates which were not thought of as the project started, but which turned out to be important.
All numbers for unique-links in this thesis are computed using the latest versions of C_1, F and R_1. It should be noted however that for most of the tests carried out, the results seemed to be much better at the time, due to poorer duplication detection.

Classifier by URL (C_1)

The task of identifying an URL as an URL pointing to a OGC service, is not that simple. Unless one actually downloads the content that the URL points to, it is not possible to tell for certain whether a link is of OGC-type or not. A link of OGC-type in this context is a link which points to any of the OGC-services WMS, WFS or WCS.

No matter how carefully one examines a link, there will always be false positives and false negatives. In this study I regard the false negatives as worse than the false positives, since a false negative will mean that an existing and potentially new OGC service could be missed. The false positives on the other hand will include some non-OGC links in the result. As long as the number of false positives is not large compared to the true positives, this is not a problem. A later analysis of the downloaded content can remove these links from the collection.

There is however no way to go back to a skipped false negative and add it again, since the amount of data parsed in most experiments is very large.
This is why some of the rules used to detect OGC-links can be considered as weak, and may lead to some false positives.
We compiled some rules using the OGC-protocols for WMS, WFS and WCS into a detection module.
If a link has parameters, the detection module tests the parameter-content of it in three steps. The detection steps performed by the module, searches the URL-string for different parts, which would indicate whether it is of OGC-type.
Here are the steps performed:

1. Check for OGC-service parameter strings (M_s)
2. Check for OGC-string pairs (M_p)
3. Check for OGC-request strings (M_r)

The checks are performed in the order they are listed. If a check succeeds, then a result is returned immediately and the later checks are skipped.

Check no. 1 looks for the existence of the OGC-service parameters with name set to "service", and value set to one of "WMS", "WFS" or "WCS". The existence of any of these three combination of strings, will give a positive result. According to the protocol standards for WMS, WFS and WCS these parameters are mandatory for any URL to these services. [WMS 1.3] [WFS 2.0] [WCS 2.0]

Check no. 2 gives a positive result for an URL u, if for any of the pairs (a, b) in M_p, both a and b are contained in u.
A total of seven pairs are used in this step, these are of three types: version-pairs, protocol-pairs and service-pairs.
The version pairs are the weakest type of checks and they contain a request type key-value pair and the version key. The version parameter is optional by OGC-standard, but it is still used quite frequently.
The request-types used are: getcapabilities, capabilities and getfeature -requests.

The only protocol-pair used is the getmap-request pair for WMS, which uses the mandatory parameters request and layer. This is a strong check, but it is useful, since this type of OGC-link is one of the most commonly found.

The service-pairs used are pairs of request parameters combined with just the name of the service. These are of average strength and will catch some of the cases when the

service-parameter is missing, but the name of an OGC-services exists in the link.

The last check gives a result of uncertain type if the URL u contains any of the three strings in M_r. M_r contains strings for request-parameters. Both versions of the capabilities-requests are used, as well as the getmap-request.

The uncertain result indicates that there is a small chance that the link is of OGC-type. Links of this type may be saved with lower priority and can be tested later, after all ordinary OGC-links.

If all of the three checks fail, then the result of the test is negative, which means that the link with high probability is not of OGC-type.
Since this test is tailored for the discovery of the three most common OGC-services, it may miss an amount of other valid OGC-links.
However, since the focus on this thesis is set for these three services, it is not such a big loss. In fact, the detection of other OGC-services by this method, was never intended. Regardless, other OGC-services that are detected in this way, are kept and not thrown away.

URL-filter (F)

The detection of duplicate URLs requires a standard format for an URL. The actual detection of duplicate URLs is a simple method once all URLs have been converted to this standard format. One can then compare the URL-strings character by character or compare the hashes of the strings.
The filter should also remove invalid URLs, which are URLs that cannot be considered as OGC-URLs. There can be many reasons for an URL being invalid. The filter used by the test-methods in this thesis does not only format URLs after a specific standard, it also removes URLs which can be considered as invalid. There are many reasons why an URL should be considered as invalid. These are the properties of invalid URLs, which are used by the filter:

1. The scheme in neither HTTP nor HTTPS
2. URL has parameters but no question-mark
3. URL contains only scheme
4. The host-name is a reserved IP address
5. The port-number in host-name, is not in unsigned short format
6. There are invalid characters in the URL

If an URL have passed the first check for validity, then it is formatted to standard format, using the following techniques:

1. Retrieve embedded link
2. Convert to lower-case or upper-case

3. Normalize Percent-encoding
4. Anchor removal
5. Remove multiple slashes
6. Always keep URL in global format (including the scheme-information)
7. Always or never display port 80 in URL
8. Remove unnecessary dots in URL-path-part (Path segment normalization)
9. Always have an ampersand for additional query-parameters
10. Sort query-parameters by parameter-name

All of these methods except number 1, 4, 9 and 10 can be found in RFC3986, where number 5, 6 and 7 can be seen as parts of the Scheme-based normalization. [IETF 1]
Method no. 1 is useful for obvious reasons. No. 4 is required since the anchor does not add any useful information in regard of which page the URL points to. No. 9 is also useful, since some times additional query parameters have a question-mark as delimiter instead of an ampersand, this just creates more duplicates.
Finally no. 10 is useful since the order of parameters never matters for a proper web-server, so it should always be the same, which is why parameters should be sorted by name .
For OGC-links there are actually even more methods to use, which will further reduce the number of duplicate URLs.
Here are the filter-methods which can be used if a link is of OGC-type:

1. Remove all OGC-parameters except service and version.
2. Set request parameter to either or capabilities

Having performed these two steps after the 10 steps which may be performed on any URL, the filtering is complete.

Duplication removal by URL (R_1)

The most trivial way to do this is to keep a hash-set of all filtered URLs found, which would avoid duplicate URLs being added. There are however two additional methods one may use for detecting even more duplicates:

1. If two URLs differ only by an initial string of "www" after the scheme-part, consider the latter of them as a duplicate of the first.
2. If two URLs differ only in the value of the version-parameter, consider the URL with the lowest value for version as a duplicate of the other.

This way only URLs pointing to the latest versions will be saved. Not all URLs begin with "www", so we cannot assume that an URL will work if we always add www if not present or vice versa, hence this step should not be a formating step.

OGC-service detection by content

The three last methods for OGC-service detection (D, C_2 and R_2) are using meta-data content. These methods were developed much later than the methods for detection by URL.
For more information about these methods, see section 4.3.3. Using the first three components (C_1, F and R_1), will however detect very many duplicate URLs, while still strictly avoiding false positives (i.e., the probability of an URL being detected as duplicate which actually is not a duplicate, is minimal).

3.5 OGC Statistics

In order to analyze the problem of finding OGC-services, it is useful to gather as much statistics as possible. One problem is however that for statistics to be reliable, one needs to have a base dataset that is as large as possible.
The statistics is based on data saved for pages downloaded and OGC-links. The statistics generated have been of several different types. The three most common types are incremental progress statistics for: unique actual OGC-links found, unique actual listing-pages found and unique actual available services found. These are three different measures on the progress, were the latter is the best metric for the actual final result. The first is however useful also, since some methods, like the boss-agents, depend on this metric for their results. The WIRE crawl-results are however more dependent on the listing-pages metric, as shown in section 4.1.3.
Further analysis of these statistics have given some more insight on which methods are best for finding the OGC-services.
For WIRE, also the crawl-time and amount of other links were saved. For SkyLabs and Google-Meta, statistics is based only on the OGC-links.
The most important statistics based on OGC-links are explained below.

3.5.1 OGC-URL Service types

The OGC-Service type statistics is made to be a tool in how to determine which service-type is related to which site. This would reveal the likelihood of finding a particular OGC-service at a certain site. This information can be used to fabricate the correct type of OGC-URL for any known site.
It is of course possible to try all three types at each site, it will however save some time if the correct type is used on the first attempt. It can also be used for determining the best site to visit, when one needs to access a specific type of OGC-service.
This statistics was generated by parsing all URLs acquired by the discovery methods.

The OGC-service type statistics generation is split on two groupings. One by top level domain (TLD) and one grouped by site-address.

3.5.2 OGC-URL composition

The statistics for URL-composition is useful for finding directory-structures which are related to OGC-services. Directory-names which are commonly found as part of an URL, can be used for meta-searching or for fabrication of OGC-URLs on a site known to host them.

The URL-composition statistics is grouped by URL-part. The URL-part is any part of the name of a requested document, separated by a '/' -character, i.e., it is either a directory or the name of the requested file, including parameters. The URL-parts are sorted by frequency, with the most frequent listed first. Each entry lists also how many documents that are available for each TLD.
This type of statistic, also called frequency analysis, proved to be very useful for finding new words to use in meta-search discovery-processes like Yahoo! BOSS.

3.5.3 Poller program

The measure of the total number of unique OGC-links found, is not the best metric for how successful a method have been. A much more reliable way to measure the result is based on the download of the content pointed to by the URLs. This is true because many URLs found may be dead, or have a low availability, or may point to badly implemented services. To analyse the capabilities document instead of the URLs gives a much more accurate measure on the actual result.

To meet this need of a more accurate result-measure, a poller-program was developed. It is based on the multi-threaded downloader used in the boss-experiment in section 3.1.3. The major difference is that it separates the download on sites so that there will never be more than one connection to each site at the time. Also, the results downloaded are written almost immediately to files, so that it can be aborted after some time, if it should crash and may then be resumed.
For the poller-program several statistics-generation functions were written, in order to analyse the downloaded content. Their implementation was straightforward, and due to their simplicity, the detail of their actual functionality is omitted in this report. Most of these methods have however been included in the poller-analyser module, in the final prototype.

Site responses for OGC-URLs

Site response statistics is useful in order to determine which sites have the most stable URL, i.e., the highest availability. The response times are also useful for determining which sites have the best connection. Sites with high availability and fast connection, could be prioritized in the search for OGC-services. Also the total number of URLs found for each site should be examined, since a site with many services is still useful even if the availability is somewhat low.

Chapter 4

Analysis

This chapter contains the analysis of the result obtained by the methods used in this thesis. For a detailed explanation on the functionality of each method, see the method-chapter.

4.1 OGC-service Discovery

If one assumes that there is little or no knowledge on how to find any OGC-service, but that it is possible to determine whether an URL which have been found is pointing to an OGC-service, then there are practically four different ways to find a OGC-service.
These are:

- Using a known web-page with OGC-services listed
- Using existing search engines for meta-search
- Using a crawler, starting at a set of seed pages
- Using fabricated HTTP-requests for URL:s which are likely to be pointing to OGC-services.

4.1.1 Harvest from listing-pages

There are some pages available which lists a large number of OGC-services. These pages can be harvested by a simple downloader for a quick acquirement of many OGC-service URLs.
Two of these listing pages are Skylab Mobilesystems Ltd.'s sites for WMS-services.
[SkyLabs 01] [SkyLabs 02]
They contain a large amount of links, even if some of them are old and outdated, the links are still valuable for structure-analysis on OGC-service-links, i.e., the structure of the links can be used to determine what a typical OGC-link looks like. Or more specifically, which parts of the links are frequent and which are not.

The result of this harvest was 1351 links to OGC-services.
This is a decent result, an excellent result even if one takes into account that the process

is very fast to execute. Of these links only 91 pointed to live services, which is less than 7 percent of the total amount. This low relative availability indicates that this is not such a good approach after all. Too many of the links found were of low quality or unavailable.

4.1.2 Meta-search

Meta-search is the usage of already existing search-engines for a search instead of searching directly on the web. With meta-search one will get a lot of functionality for free.
One problem with meta-search is that most major search-engine companies will not allow an agent to connect directly to their engines, i.e., they detect robot usage and employ robot catching techniques such as CAPTCHAs. [CAPTCHA]

Google Meta-search

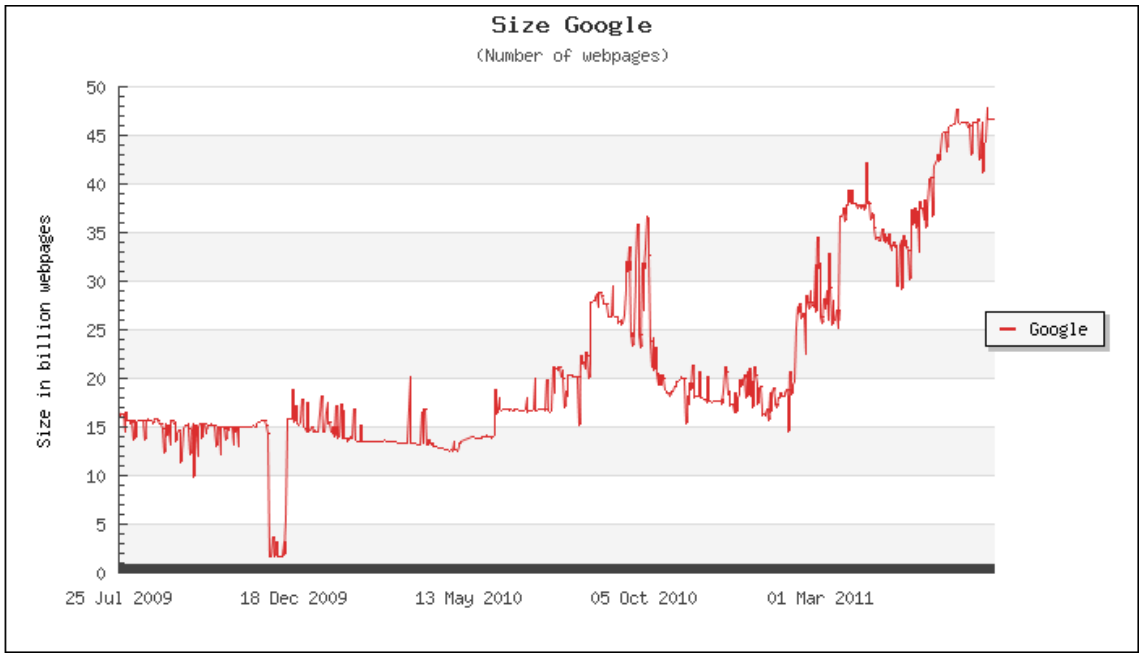

Figure 4.1: Google Search estimated web index size, July 25th

The usage of Google's search engine for meta-search would be preferable, due to its web-coverage. As shown in figure 4.1, its size have been estimated to over 45 billion pages by Maurice de Kunder. [WWWS 01]
The use of Google's search-engine is however complicated.
Firstly, Google's Terms of Service does not allow the usage of any type of external robot for their services. [Google 01]
Secondly, there are no suitable APIs for using their search engine explicitly by an external application. There are APIs like Google Custom Search API, but these APIs will only work for web-pages and not external automated agent-applications.
Since the final objective of this thesis is to develop an external automated agent, these APIs were not examined further.
Thirdly, if one would disregard the terms of service, which we would not recommend,

Google have ways to detect robot usage, and shut down communication.

To see how effective a search engine could be for finding OGC-services, a test was performed on Google's engine. In this test a total of 100 pages were downloaded. These pages were search pages, taken directly from the Google search engine. Each of these had 10 search results. Making it a total of 1000 search results. This lead to the discovery of 686 URL:s for OGC-services. Only 49 of these were duplicates with the already found URLs from the Skylabs-pages. Also 281 of these URLs pointed to unique available services.

The conclusion of this step is, that since manual download is not in anyway useful for an automated agent, the Google search engine cannot be used in such a prototype. The discovery of 281 services from 100 pages however, indicates a high per page effectiveness, and shows that meta-search have a large potential.

Yahoo! Meta-search

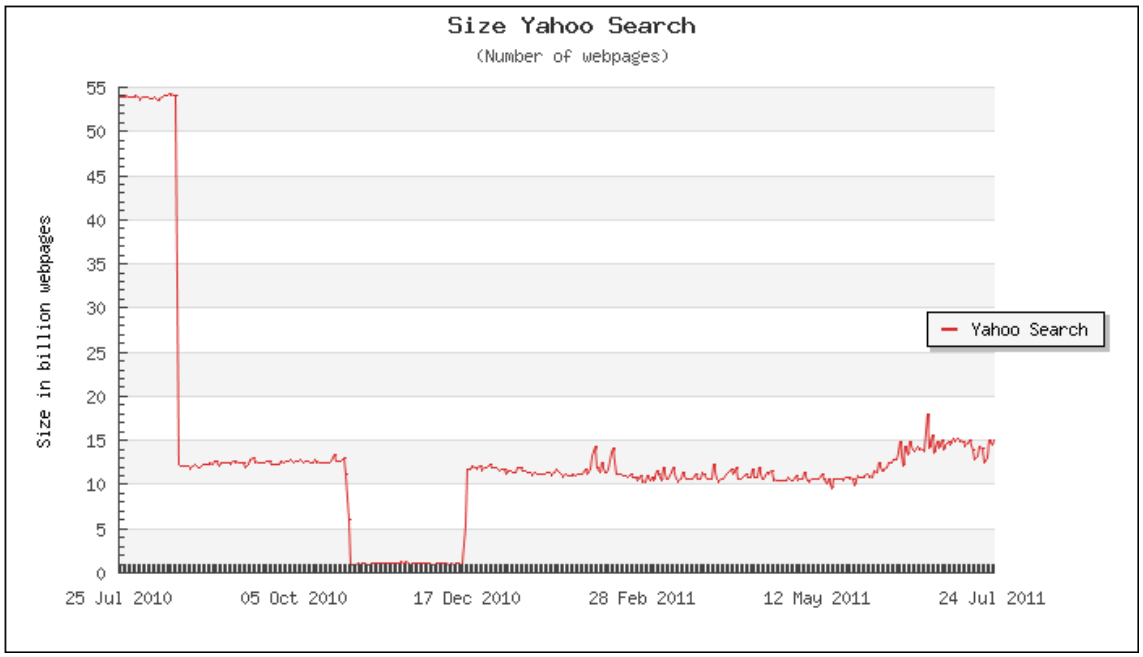

Figure 4.2: Yahoo! Search estimated web index size, July 25th

The Yahoo! Search-engine does not index as large portion of the web as Google. This may seem like a disputed fact, but some estimations have been carried out. For instance De Kunder Internet Media have made estimates of the index-size of different search-engines. [WWWS 01]
According to De Kunder's site, by the end of Juli 2011, Google indexed more than 3 times as much as Yahoo!. [WWWS 01]
Yahoo! have however released an API called Yahoo! BOSS (Build your Own Search Service). [Yahoo 01]
This API allows developers to use the Yahoo! search-engine in their projects, for a small monthly fee. The largest drawback of using the Yahoo! engine compared to Google's is

that Yahoo! have not indexed as large part of the web as Google.
This has not always been the case though. A couple of years ago Yahoo! search actually had a larger index than Google-search. [WWWS 01]
Despite this, BOSS is a really useful tool for finding OGC-links. Connecting to the search engine, using the BOSS API is easy and also the responses given, are fast and well formated. The format for results is either of XML or JSON.

As mentioned in section 3.1.3, two runs of manual testing of search queries were performed for up to 90 queries.

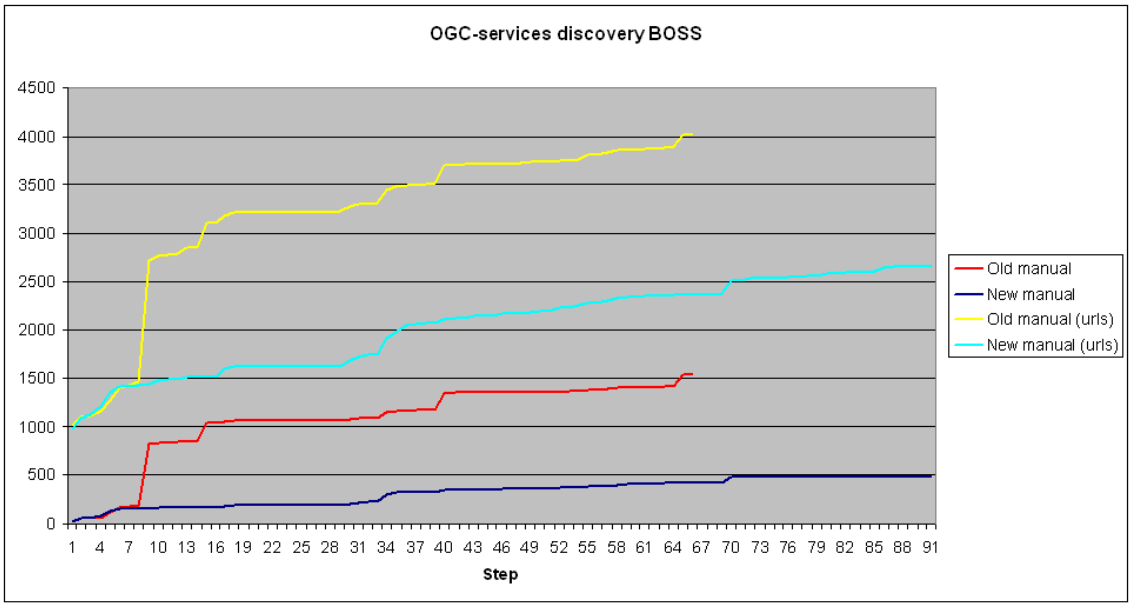

Figure 4.3: OGC-services discovered by BOSS

Figure 4.3 shows the incremental discovery of OGC-links and available services for both of the two runs of the test of the BOSS-method. As can be seen in the graph, there are two flat sections for all four curves. These are between steps 19 to 28 and steps 43 to 50.

The first section can be explained by the fact that the search words used there had very few results for BOSS.

The second section represents the tests performed using the words from the frequency analysis of words in the page-content (W_R). This set of words turned out to be a failure. The failure was partly due to the fact that the Yahoo!-crawler probably disregards many special characters in the content when a page is indexed. Most words in W_R contained special characters.

Using the latest methods for duplication removal, both BOSS test-runs found a total of 1867 live services among a total of 5354 OGC-URLs. This was a significant increase from the previous methods. This method was however given significantly more time and effort than the Google-meta search.

4.1.3 Web-Crawling

Web-crawling may be used as an extension to some other method for OGC-link discovery. There are many already existing open-source web-crawlers available.
The problem about this part is that the type of crawler sought is a focused-crawler. Where the focus is set to find OGC-services only, i.e., the crawler should not download pages unrelated to the OGC-topic, or at least very few of them. Most of the available open-source crawlers are general crawlers.

For a general crawler to be useful in this context, it must in some sense be guided by an external application, i.e., the seed-pages must be carefully selected, non-desirable downloaded content should be deleted and crawling should stop at some point, to restart using other seeds. Of course these steps should be carried out by any crawler system, they are just more important whenever a general crawler is used.

OSBC

One Step Brute-force Crawl (OSBC) is the concept of the simplest possible type of web-crawler. An OSBC-crawler will download all pages that are linked from the seed-pages, without any regard for any other information found on these pages.
It will also proceed only one step in depth. That is, it will not download pages which are linked from the pages that have been downloaded.
In total the OSBC-test gave 1182 OGC-links, of which only 57 were new. OSBC turned out to be not so useful, but not completely useless either. It does suggest that there is at least a little more to find if one is willing to crawl one step or more. At a much later time, it was revealed that OSBC was very ineffective. Only 63 of the 1182 links found pointed to live services. No other discovery process had so low relative availability.

WIRE

Web Information Retrieval Environment (WIRE) is an open-source crawler developed by Carlos Castillo at the University of Chile. [WIRE 01]
WIRE is a crawler of batch-type which is general and can be configured to use both the PageRank and OPIC algorithms for ordering-metric. [PageRank] [Now 01]
Currently, WIRE does only supports the Linux operative system. This is a problem, since the test-framework is written for the C#.NET platform. On the other hand it is very CPU-performance efficient and can be run as several instances on the same machine.
It can be configured in many ways, which may be advantageous in the long run.

For WIRE, as mentioned in section 3.1.5, seeds were generated, using results from the BOSS meta-search test. These seeds where partitioned into sets with seeds of similar quality, which were crawled by WIRE and then parsed.
After the data from the crawls had been parsed, many types of statistics were generated. One of the most interesting for this task, was the statistics for new OGC-list pages per minute.
Since we had saved the time taken for crawling each level, it was a simple task to compute this, after the results from WIRE had been parsed for OGC-links.

An OGC-list page is a page which holds OGC-links. The statistics on list-pages is more useful than the statistics on OGC-links, since some pages may have very many OGC-links. These pages might alter the statistics much in favor for what is actually a very bad seed-group, that just happened to get lucky and find one useful page which holds many OGC-links.

Summing up, we found out four interesting conclusions from the statistics:

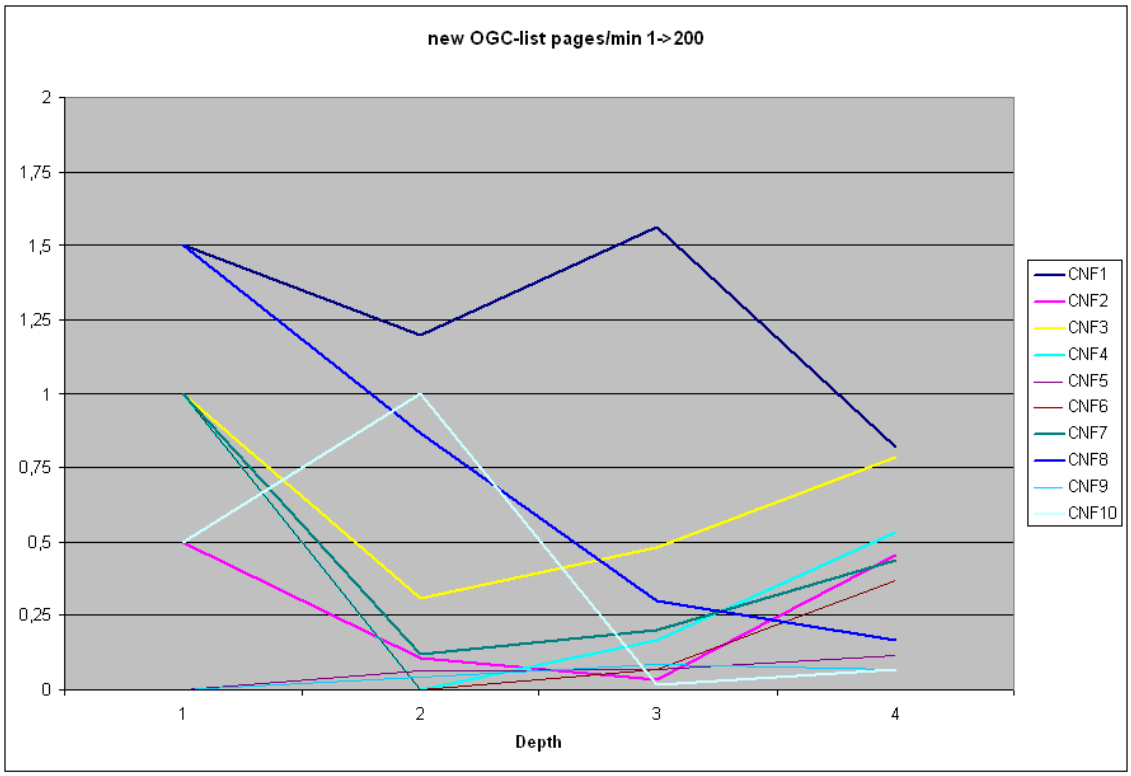

Figure 4.4: OGC-link discovery speed, early groups (CNF)

- Firstly, many curves for the OGC-link discovery speed, as can be seen in Figures: 4.4, 4.6 and 4.7, are behaving somewhat like J-curves, beginning with high values at depth 0, to drop quickly at depth 1, and then increase slowly again.
  This implies that either one should not crawl at all, or one should crawl to at least depth 3. It does seem like since these seed-pages were already good pages for finding OGC-links, one won't find that many direct links from them to other good pages for OGC-links. This helps explain why OSBC did not perform that well.

- Secondly, we found that that the Pearson correlation between the values for new OGC-list-pages $n(d)$ at depth d, and the values for $n(d-1)$ is strong. [Pearson]
  In fact the value for Pearson correlation between list-pages at depths 3 and 2 is $P_{32}^{S} = 0.67$ for the 20 first groups (CNF & CNG), $P_{32}^{L} = 0.85$ for the 40 last (CVX) and 0.82 combined. Between depths 2 and 1 it is 0.75 for CVX and 0.64 for combined. Possibly indicating that smaller groups give larger variations and that correlation increase with crawl-depth.

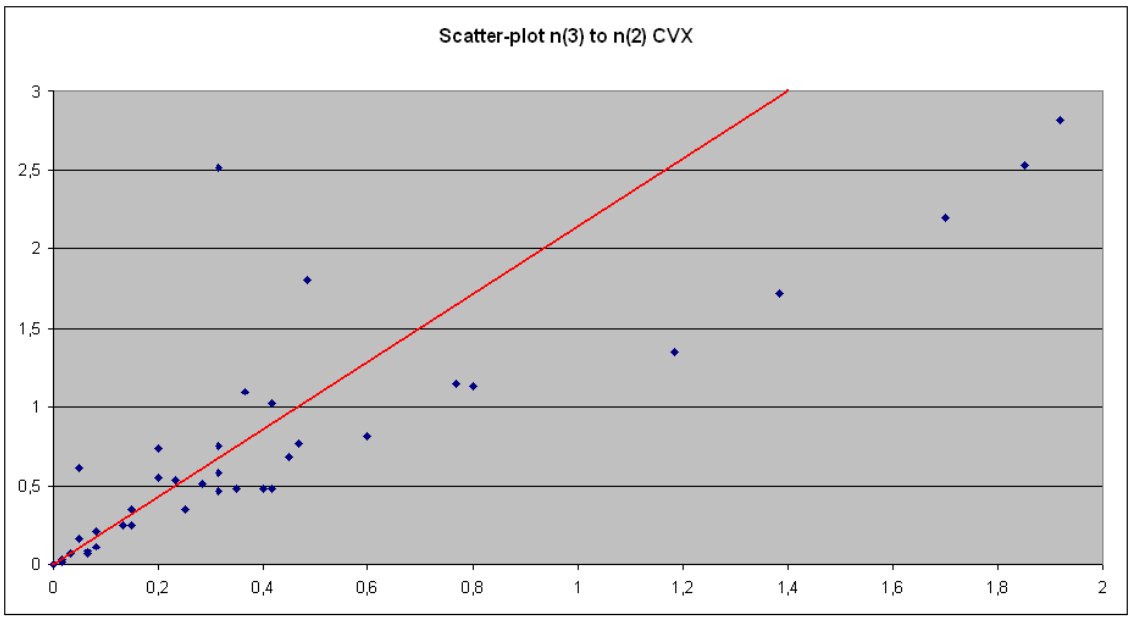

Figure 4.5: CVX points for OGC-link discovery speeds at depths 2 and 3

This actually means that one can roughly estimate how many new OGC-list-pages that will be found at the next crawling depth, simply by knowing how many that was found in this step. The scatter-plot for depths 3 and 2 for CVX (Figure 4.5), reveals that there is a strong correlation P_{32}^{L}. I.e it is possible to adjust a line to the dots from (0; 0) to approximately (1.4; 3) in the graph.

- Thirdly, the performance generally decreased with later crawl-groups, indicating that the rank-value from BOSS, actually is a useful measure for seed-quality.
  A simple comparison between Figure 4.4 and 4.6 illustrates this.

- Lastly, the performance of the first groups in CVX is higher than that of the later groups in CNG, indicating that 20 seeds per group is sub-optimal and that groups of 100 is much better. (Figure 4.7 compared to 4.6)

Three cases were selected for crawl two levels deeper than the rest.
The results of these deeper crawls, are show in figure 4.8. From these high-depth test-cases, there are a few conclusions one can draw, knowing however that they are very uncertain, due to the very low amount of data-instances.
These are that, for every one of the test-cases, the discovery speed goes down again after depth 4. For the best set at depth 3, it actually plummets.
This indicate that it might not be useful to crawl much deeper than to level 4. After that, WIRE seem to find too many unrelated pages that lowers the total discovery-speed too much.

The number of total live services found by WIRE-crawl was 1673.
Among these, 1111 had not previously been found by BOSS (66%).

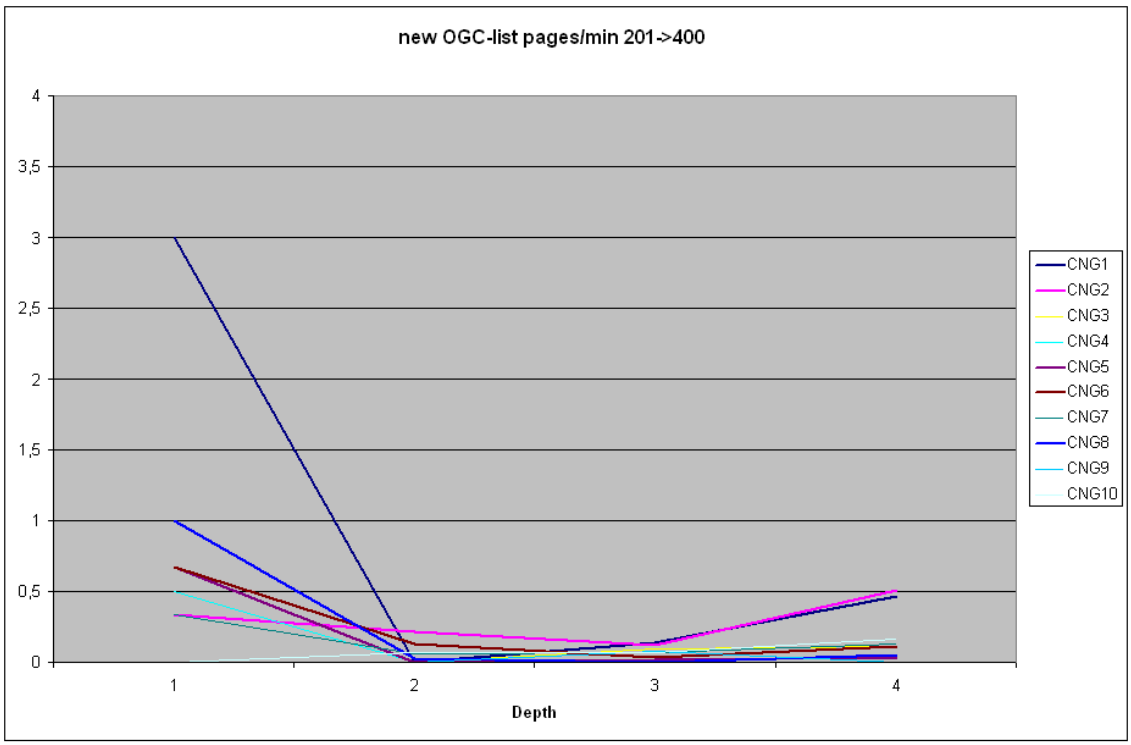

Figure 4.6: OGC-link discovery speed, middle groups (CNG)

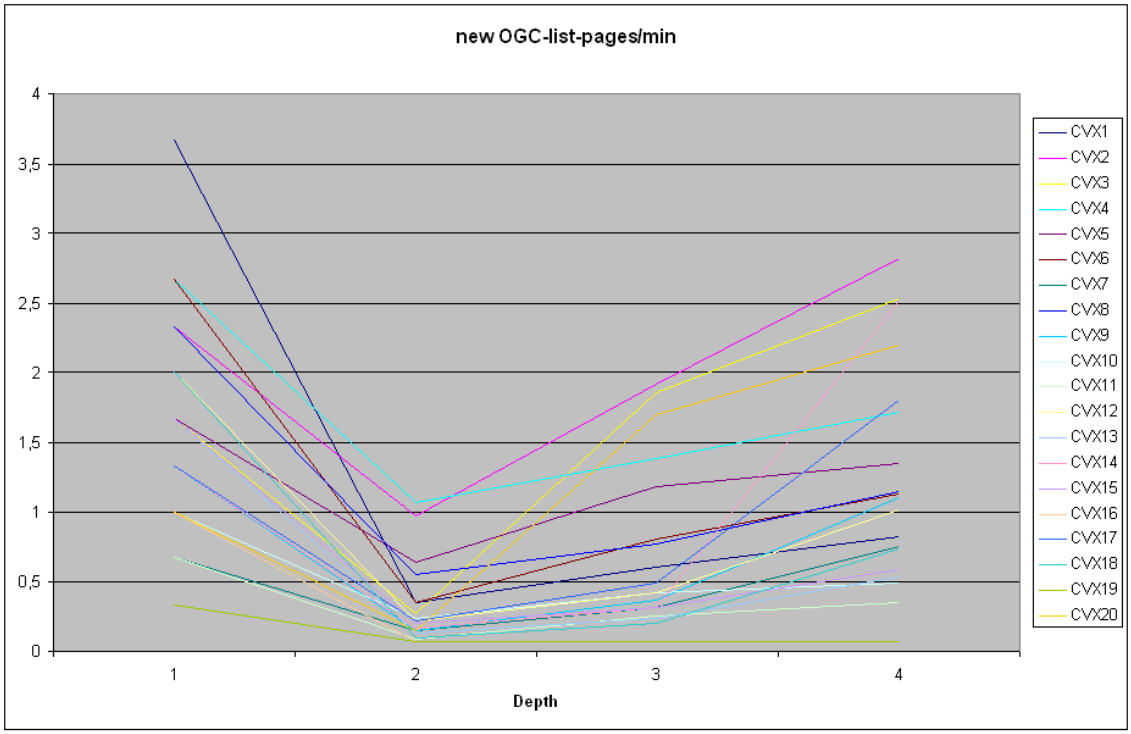

Figure 4.7: OGC-link discovery speed, first half of late groups (CVX)

It is however difficult to estimate how much WIRE-crawl may increase the total number

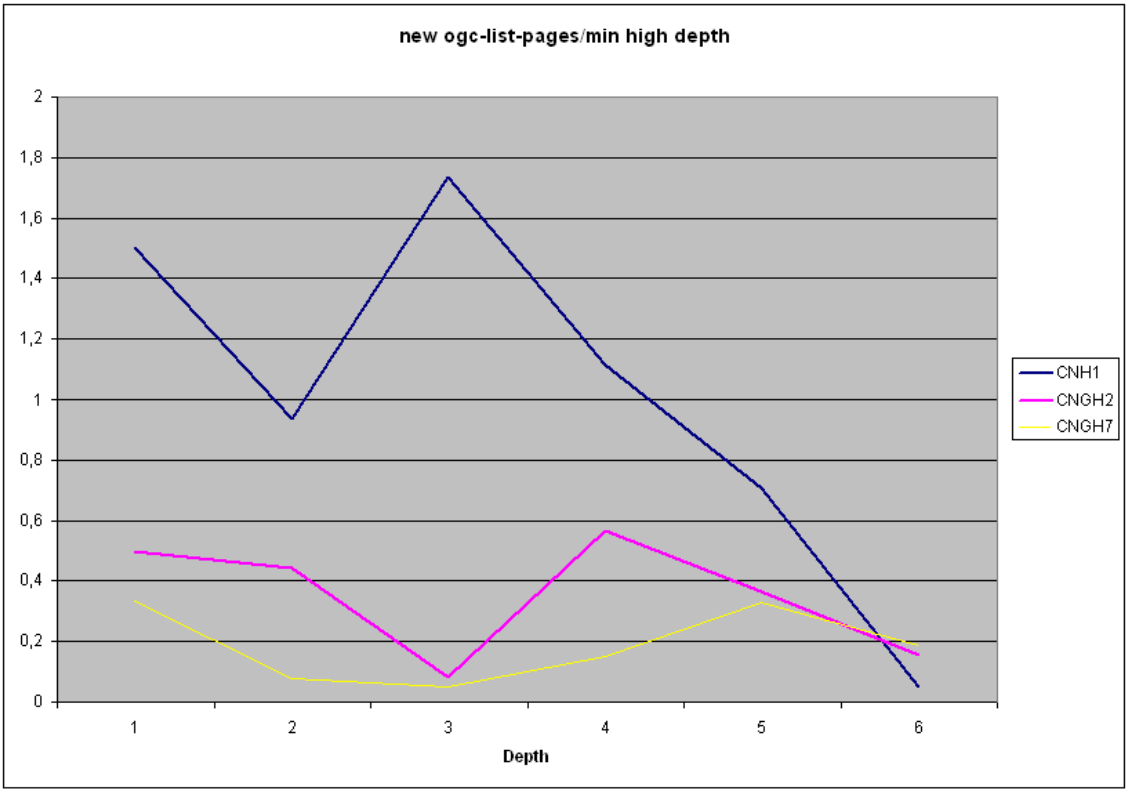

Figure 4.8: OGC-link discovery speed at high depth

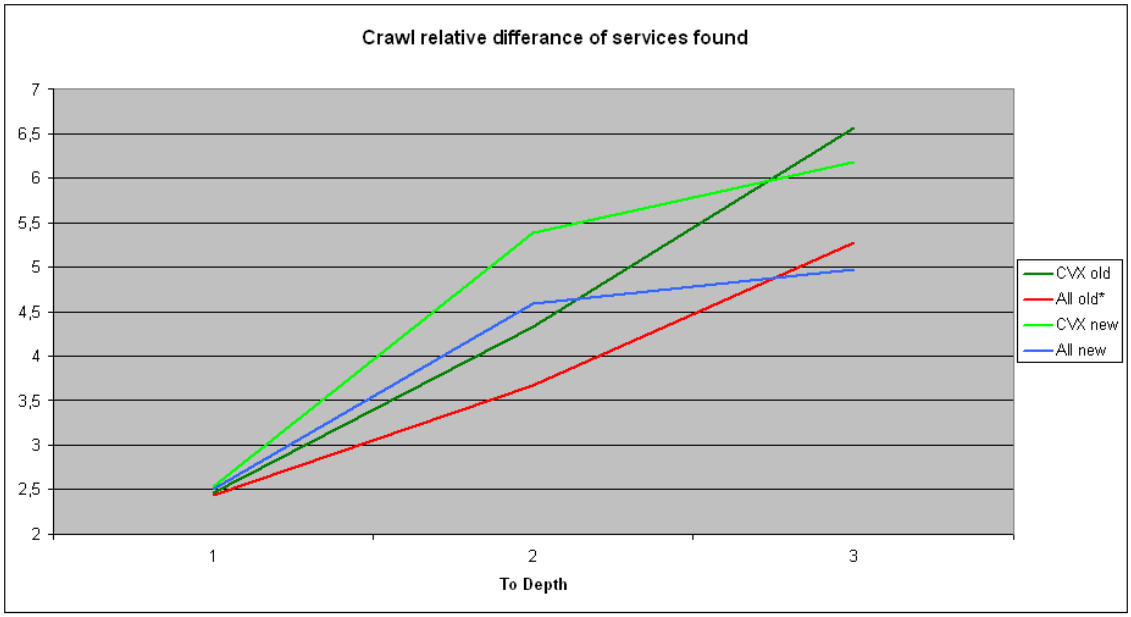

Figure 4.9: Relative difference of services found per crawl depth, compared to depth 0

of services found. One way, is to take as large and diverse set of crawl-groups as possible and compare the number of services found per level. It is however likely that the ratio of additional services found versus the services found at depth 0 is decreasing with large seed-sets. Figure 4.9 shows that for the largest collection of groups the relative gain of

discovered services is about 5.5 times or less at depth 3.

What also is interesting is that when including the CNF and CNG groups, the ratio gets much lower. Then the gain is about 4 times.
The graph is based on the counted number of live services found. This measurement was taken, using the poller, at two difference occasions. The first was taken when the poller was completed, and the second 106 days later, as a reference.
The new measurement shows that near half of the services have disappeared in the 106 days which passed. In fact the half-times measured for the availability (t_a), was between 94 and 146 days. The most stable services resided at depth 2 from the seeds.

The red line, showing the ratios for all of the three groups at the first measurement, is in fact estimated using the other three curves, since statistical data for it was missing.
A decent estimation for the ratio of gained discovery, using seeds with more than 400 available services with WIRE-crawl may be something less than 4 times, maybe 3 times.

This means that WIRE-crawl is an effective way to find more OGC-services.
One problem is however, the time required for crawls. In general, for every additional crawl-level, the time required per level increases. Also the time increases with larger seed-groups. Of course it is possible to run several simultaneous instances of WIRE.

A statistic for the incremental crawl progress per crawl-group at each crawl-level was made. This is displayed in figure 4.10. As can be seen, the CVX-groups perform much better than the others, especially at depth 2. The CNG-groups perform the worst, except at depth 3. This leads to the same conclusion as previously mentioned in this section. Which is that crawl-groups of size 100 perform better than those of size 20.
A statistic for the summary of crawl result per group-type was also made. This is shown in figure 4.11. It shows that the CVX-groups are superior to the CNF and CNG -groups. This is however not completely true, since the CVX groups are 40 to the count, which is 4 times as many as the others.

If one instead computes the number of services discovered per crawl-group, CVX would have similar performance as CNF.
CNF is however better than CNG, except for in the end, where a group in CNG got lucky and found many live services at depth 3. Again these statistics indicate the same as previously stated, that later groups perform worse, except for the CVX-groups, which are larger.

The number of services per page downloaded for each of the crawl groups was calculated as:
$E_{CNF} = \frac{314}{175716} \approx 0.18\%$, $E_{CNG} = \frac{344}{142883} \approx 0.24\%$, $E_{CVX} = \frac{1279}{2165475} \approx 0.059\%$
We also computed the relative number of services hit for WIRE groups at depth 1:
$E^1_{CNF} = \frac{121}{9772} \approx 1.2\%$, $E^1_{CNG} = \frac{31}{18273} \approx 0.16\%$, $E^1_{CVX} = \frac{480}{187684} \approx 0.26\%$
This shows that the hit-ratio was usually much better at the early stage of the crawl.

4.1.4 URL-injection

There was not enough time to test and evaluate this method. Since it anyway requires a site with OGC-links, it can be considered as an extension to the other methods. it cannot

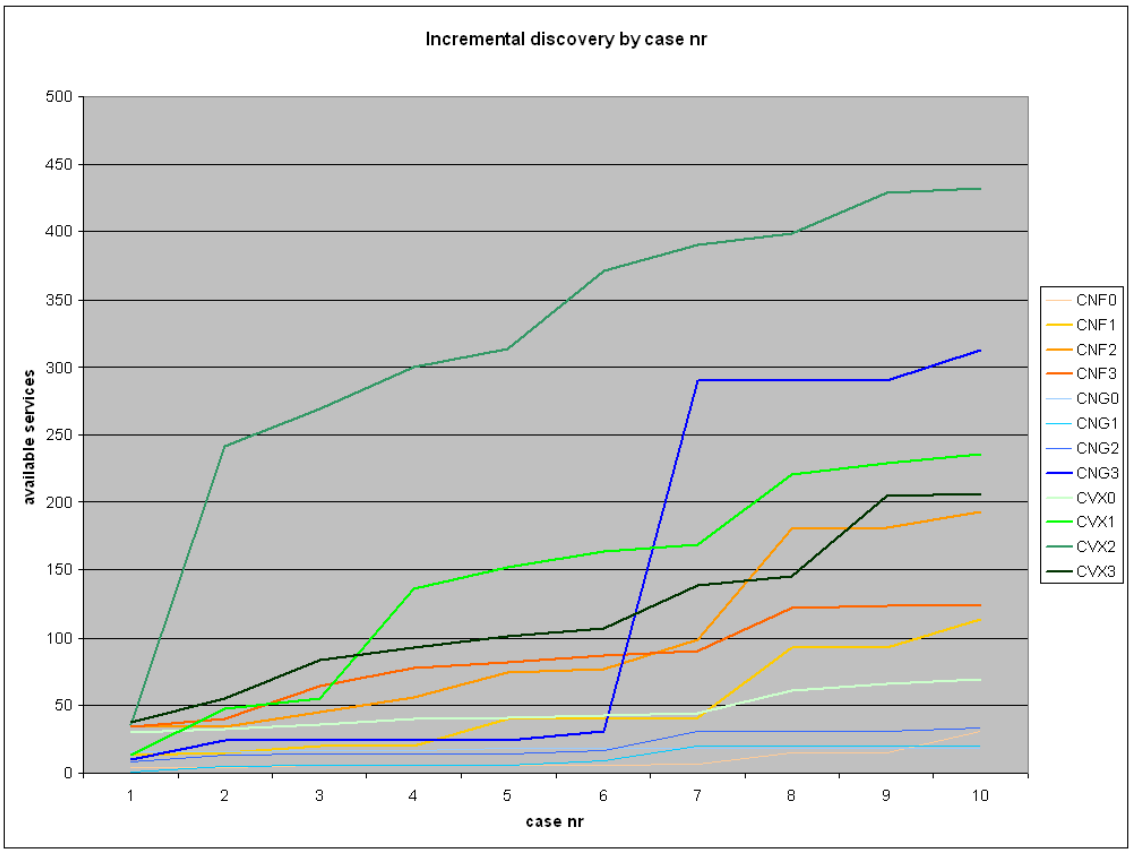

Figure 4.10: Available services found by WIRE, per case

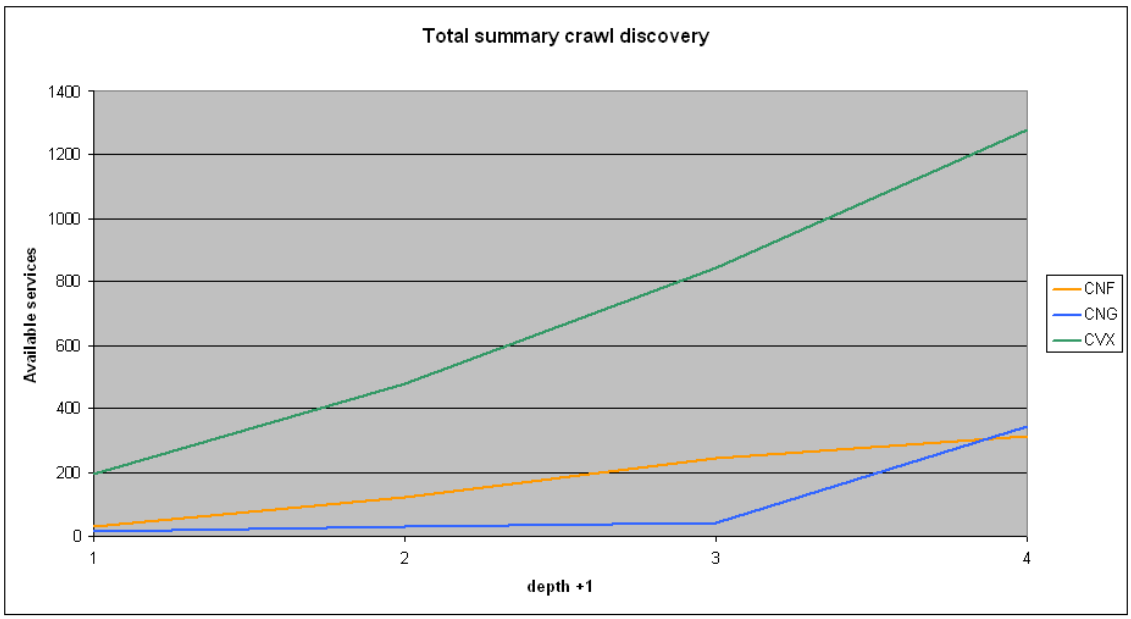

Figure 4.11: Total available services found by WIRE, per group-type

be used alone to discover new OGC-services.

4.2 Automated agent

If one knows how to identify and how to find OGC-services, is there a convenient way to automate a process for finding more OGC-services? An automated agent would be capable of, given a certain initial input of services, discover even more services. This can be done in many ways. The conceptual idea is however that the agent should learn how to find the OGC-services by itself, in an unsupervised manner.

4.2.1 Machine learning

One important aspect of OGC-service discovery is if there is a way to find a recurring pattern in pages which hold OGC-links.
Information which can be used for this include: parts of the URL, abstract text for the URL, the title of the page on which the URL was found. Also one may examine the raw text data on the page to examine whether it is likely that it is a page related to the OGC-topic.
In this study Info similarity is the only machine-learning method tested. A test carried out by N3-labs indicates that this is a decent method for a focused crawler, and may even practically beat Info-spider which is based on neural networks. [N3 01]

4.3 Prototype analysis

The prototype which was to be developed, consists of three type of modules. These are: BOSS-Agent, WIRE and management -modules. In total it was decided that the prototype should be built up by six modules.

4.3.1 BOSS-Agent analysis

The BOSS-Agent was decided to be the first module of the prototype. The performance of the developed BOSS-Agent have been analysed mainly through test-runs and comparison with the manual testing sessions of the BOSS-API.
For more information on the methods of testing for the BOSS-API, see section 3.1.3.
The BOSS-Agent have been tested with many different settings. For simplicity in this analysis, each of these different setup agents, are regarded as different agents, despite the fact that they use the same software.

Manual runs of BOSS

In order to compare how effective the automatic BOSS-agents are, data gathered in the tests of the manual BOSS-meta-search were analysed into statistics.
This data was gathered by the BOSS-module between the 2nd and the 6th of May.

Due to a change in the implementation-standard in BOSS on the 4th of May, we had to change the BOSS-module slightly. After the change, another full run of the manual agent was performed. This is the reason behind the fact that there have been two separate manual runs. To clarify, a test run is considered manual, when the queries to use in every step have been selected in advance of the test, or selected by a human at every step of the test run.

The manual selection we made was based on frequencies-summaries for OGC-links, and did not take title or link-text into account. Of course we used common sense to exclude some weird looking results, something that the automatic agents cannot do. That is an advantage, where not looking at title or link-text is a disadvantage. The hope was that the automatic agents would be capable of performing just as good as the manual runs.

The first manual run was made from the 2nd of May to the 4th of May, and the second was made on the 5th of May to the 6th of May.
The result count was actually worse for the new run. This could be due to some old material being removed by the BOSS-crawler. Even though some listing-pages for OGC-links are unavailable, there may be links that were present on these pages, which still points to available services.

The character of the curves for the two manual runs are very similar with one exception, as can be seen in figure 4.12. The old one has a spike of more than 600 services at step 9, which is missing in the new one. This is most likely due to the deletion of a very large OGC listing-page from the web.

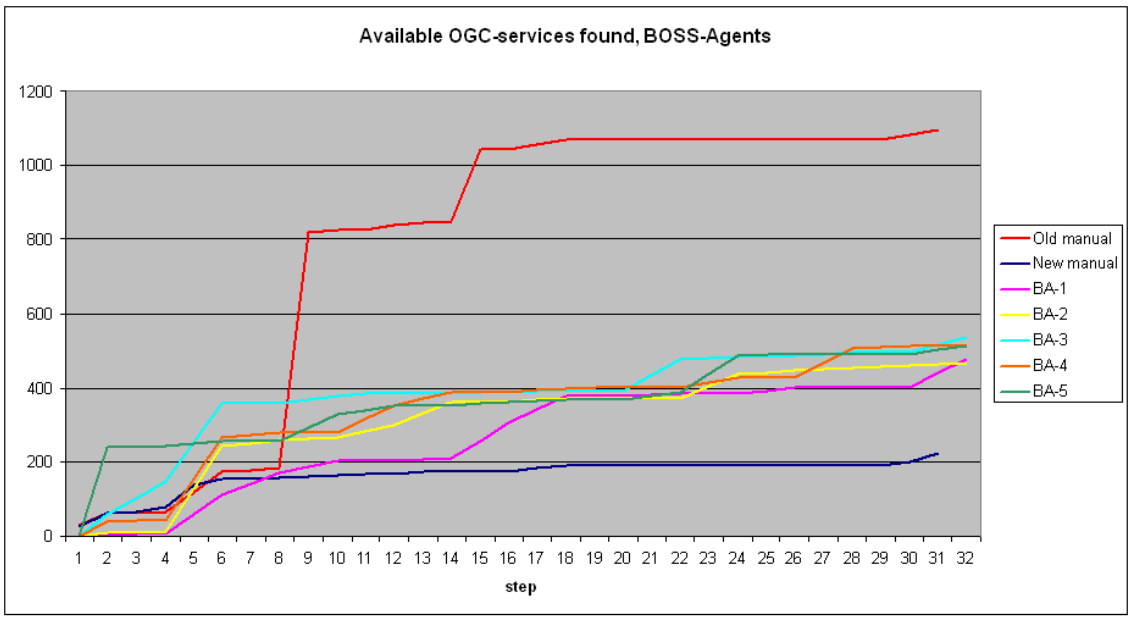

Figure 4.12: OGC-services discovered by automatic agents, compared to manual runs

Prototype tests

For the purpose of testing the performance of the agent, some tests were carried out. These tests were nothing other than test-runs of the agent, with a set maximum number of iterations. An iteration is a Manager-Seeder cycle, where the manager have asked BOSS a set of queries Q_i, and gotten replies, which also have been downloaded, whereafter the seeder have processes the result and selected the new set of queries Q_{i+1}, for the next iteration i+1. The notation of steps-number used in this chapter, refers to the numbered query asked to the BOSS API, e.g., step 6 means the 6th query asked to BOSS. Since the agent is capable of asking many questions in each iteration, it is important to separate this notation from the iteration notation.

Test settings: For each test run of the BOSS-Agent, a different setting was used. Each test-run was given a name with a BA prefix.
Table 4.1 shows the settings used. The selection of the queries in the set of seed-queries (Q_S), is explained in section 3.3.1
The k-value in the table, is the number of queries used per iteration, this does not apply to the initial queries used (Q_0) in the first iteration.

Different seed short tests: First a test was made for determining the effectiveness of different search-words in Q_S.
Each one of the selected seed-queries in Q_S were used alone, and the count of OGC-links in every step was recorded and compared to that of other automatic test-runs and manual runs.
For these tests the k-value was set to two, in order to reduce the testing time. This means that two queries are asked per iteration. The effective number of iterations was 16. As can be seen in figure 4.12, all agents are outperforming the new manual run. But the old

Name	Seed-Queries ($Q_0 \subseteq Q_S$)	k-value	Length (iterations)	Correlation metric used
BA-1	geoserver	2	16	No
BA-2	wmsservlet	2	16	No
BA-3	esrimap	2	16	No
BA-4	wmsconnector	2	16	No
BA-5	request=getcapabilities	2	16	No
BA-12	wmsservlet	2	31	No
BA-15	request=getcapabilities	2	31	No
BA-37	Q_S	1	87	No
BA-42	wmsservlet	1	31	Yes
BA-47	Q_S	1	57	Yes

Table 4.1: Boss-Agent test settings

manual is much better, especially at the end. Where agent no. 1 and 2 are performing slightly worse than the others.

Extended test length: One of the better agents BA-5 and one of the worse agents, BA-2, were also tested for 30 more steps.
The agents are made in such a way that it is possible to abort a run at any iteration and then resume it again later.
So the tests for two agents were resumed till step 61 and compared to the manual results. There was not much point in continuing the tests that much longer, since the old manual test only goes 67 steps.

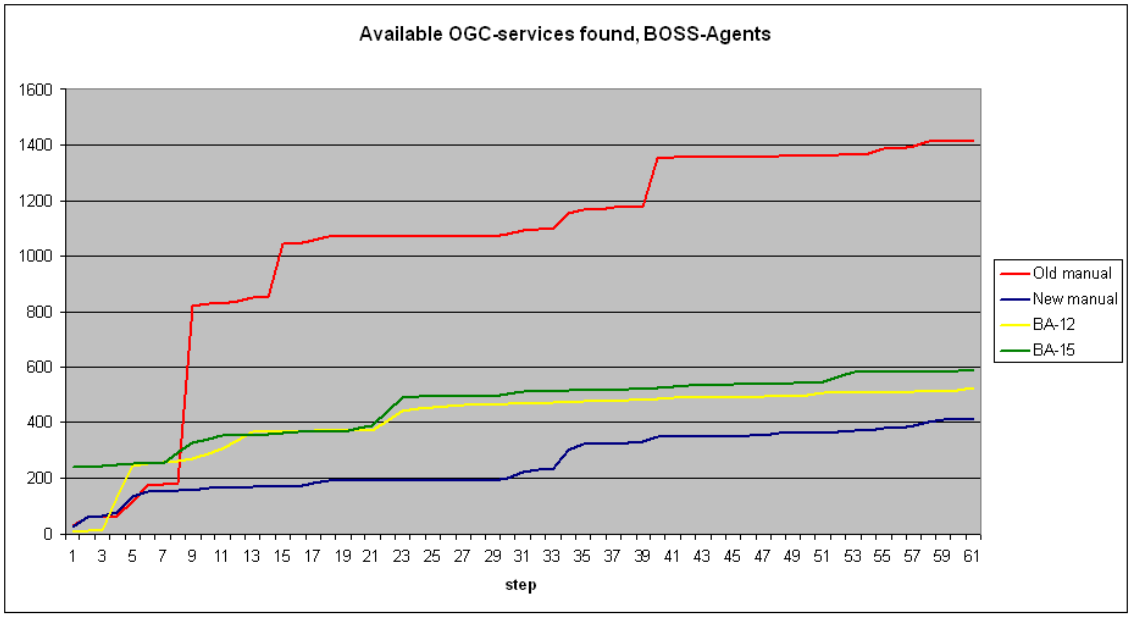

Figure 4.13: OGC-services discovered by BOSS-agents in longer runs

Figure 4.13 shows that both the agents, reach a performance of something slightly better

than the new manual run, in the end.
The better agent, BA-15 stays ahead of BA-12 all the way after step 31, the difference between them does not change much in the 30 additional steps.
The only major difference between the agents is prior to step 5, where BA-15 performs much better. So maybe, the selection of seed, does not matter that much to the final result, as long as the seed is of decent quality.

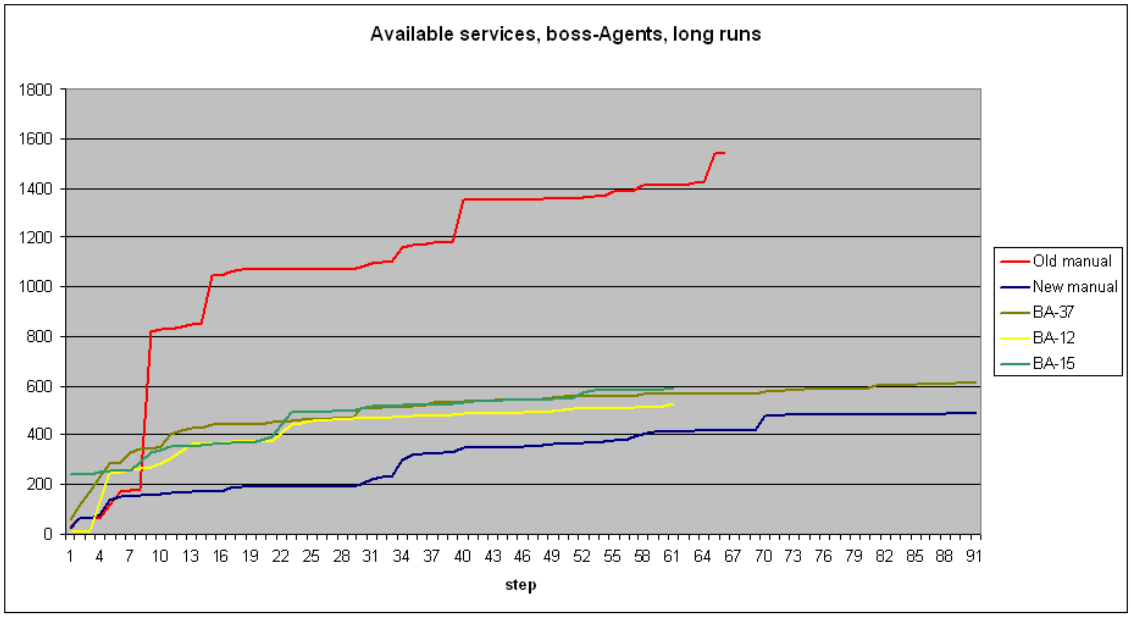

Figure 4.14: OGC-services discovered by combined BOSS-agent, compared to previous best Agents

Test of combined seed agent: As a comparison with previous results, a test was run with an agent, named BA-37, that is using all 5 of the seed-queries. Also this agent used a value for k set to 1, which should lead to better results, assuming that the scoring-algorithm is correct in assigning scores.
This agent was run for 91 steps, which is just as long as the new manual run.

Figure 4.14 shows that the combined agent BA-37, initially outperforms the other agents. It cannot reach the performance of the old manual run however. At step 61, it seems like BA-15 actually performs better than BA-37, something which was unexpected. Since BA-15 was not run for as long as BA-37, one cannot tell for sure which one is better at step 91. Maybe this means that it is better to use a fewer number of seed-queries of good quality, or that the metric for assigning score to queries is not that good.
It could also be that random noise in the download processes have affected the result in favor for a worse agent.
The performance value of an agent should be the average over very many test-runs, in order to reduce noise. In this study, there was neither enough time nor possibility to perform many test-runs of the same agent, considering that the usage of the BOSS API is non-free.

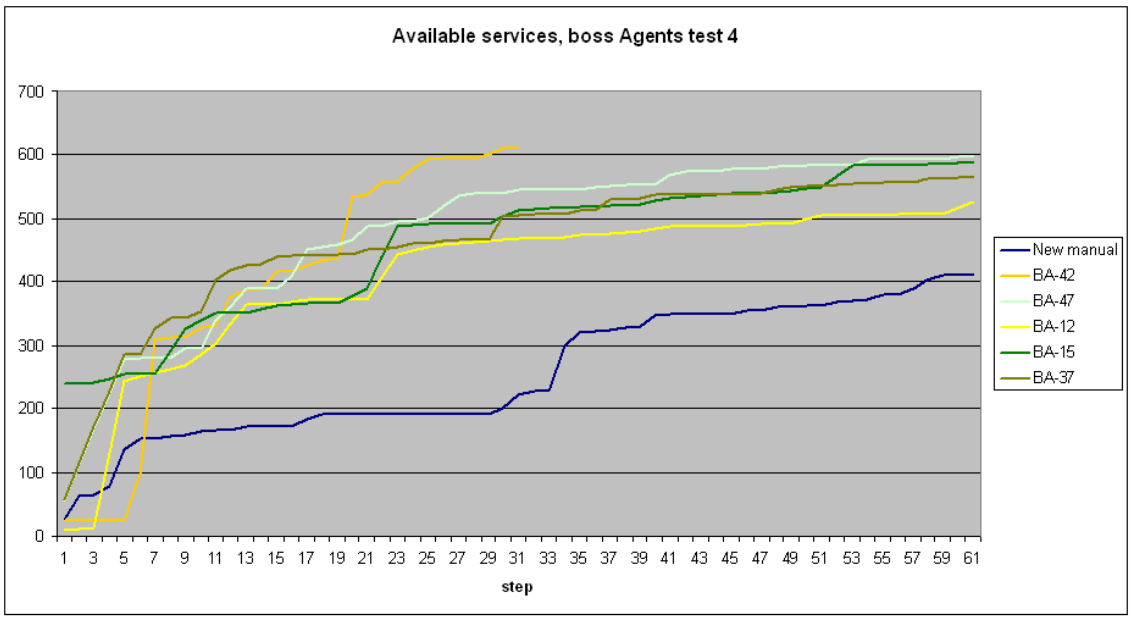

Figure 4.15: OGC-services discovered by correlation-metric BOSS-agents, compared to previous best Agents, old manual run is omitted

Test of agents with correlation metric: After the correlation-calculation had been added to the BOSS-agents, two more test-runs were made. One run was made using the 2nd seed (BA-42) and one using all seeds (BA-47). BA-42 were run for 31 steps, and BA-47 for 61 steps. Both agents asked one query per iteration ($k = 1$).
Figure 4.15 shows that initially BA-37 is still best, but after some 17 steps, the new agents take over. The BA-42 gets a large jump at step 20, and even outperforms BA-47. This jump comes at the query "service=wms getmap". This query was also used by BA-47 at step 21. The increase in found OGC-services at step 21 for agent BA-47, was however moderate. This indicates again, that there is a significant noise in the process of determining which agent performs best. Judging from only a few test instance it may be near impossible to tell which of two agents are best. The best agent should be the one which have the highest expected result from a random test trial. If one disregards the noise, it can be said that BA-47 is almost 6 percent better than BA-37 at step 61, which is a bit of relief, since it means that the correlation-metric improves performance slightly.

Noise in prototype testing

As mentioned before, there must be a significant amount of noise in the process of discovering OGC-services. If a query q is asked to BOSS at time t_x, then if the same query is asked at time t_y, there is no noise if they lead to the discovery of the same set of OGC URLs, i.e., $O(D(B_r(q, t_x))) = O(D(B_r(q, t_y)))$ if there is no noise in function B_r, D and O. $B_r(q, t)$ is the result downloaded from BOSS when asked the query q at time t. D is the downloader-function, which downloads the pages given by BOSS. O is just a selection function, which selects OGC-links from a set of pages given.

There should be no noise in function O, unless there is an implementation error in either the parser or OGC-link identifier modules. The other two functions will always be

subject to noise, since the web is dynamic.

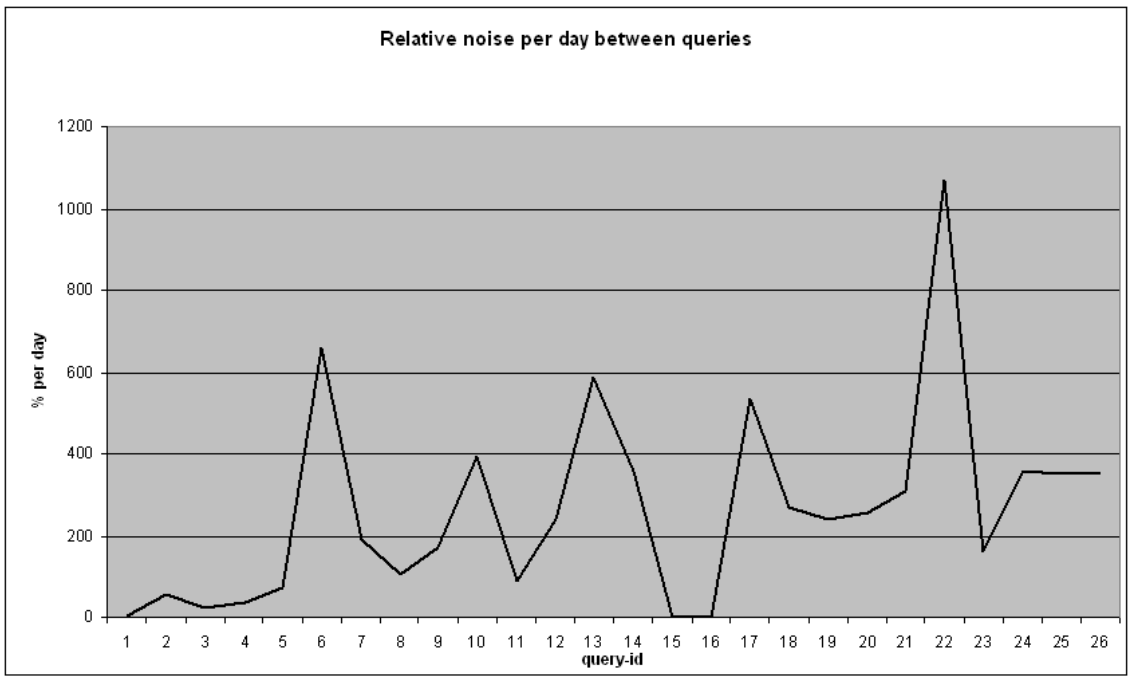

Figure 4.16: Relative noise per day in percent for BOSS-requests

In order to get a measure for the noise in the first function (B_r), we filtered out the queries, which were used by both of agent-test runs BA-42 and BA-47. We then computed the value for the relative noise for each of these queries as $r_n = \frac{|A \cup B| - |A \cap B|}{|A \cup B|}$, where A is the set of results from BOSS for a query for the BA-42 test, and B is the corresponding result for the BA-47 test.

To get a better measure of the noise, we divided the noise with the time in minutes between the two BOSS-connects. This is useful, since over time there will always be a significant noise introduced.

Figure 4.16 shows that there is a large variation in the noise per day between the queries, this means that the noise is not alone linearly dependent on time between tests. It turned out that the queries which gave more noise were typically the queries which gave more results. Although the total number of results per BOSS-query was not saved, other than for queries consisting of single words, the queries with higher noise contained words which had a higher number of results for single search, than the queries with low noise. The highest values for the noise was over 1000% per day, meaning that the entire result would change 10 times per day. On average the half-time of change was 1.6 days. This is about 30 times less than the half-time suggested for the web by Cho. [Cho 01]

Available meta-data summary

A poll was performed on the 3181 URLs found by all of the seven first agents (BA-1 to BA-15). The result was a bit disappointing. Only 17,6% of the URLs found pointed at live services (561 in total). This can probably be explained by the fact that a lot of material

found was located on large listing pages. The material on these pages is usually not so frequently updated, as can be seen in the case for Skylab's listing-pages, leading to a low availability.
The composition among OGC-services was similar to previous results: 78% WMS, 17% WFS, 5% WCS, and less than a half percent of other OGC-services.
In comparison with the manual BOSS-runs it is clear that the old run is superior to the agents, but the new one is actually performing slightly worse. It finds 418 new services after 61 steps, this is about 30% worse than the best automatic agent, which found 590 services.

The BOSS-method is too limited to find very many OGC-links. The manual new runs found 485 out of 2982 of all OGC-services found in this thesis study (16.3%). The complement of some other method, such as WIRE-crawl, could drastically increase the performance at the end. The WIRE-method used in this thesis found 1673 services, and its result is based on BOSS-searches. 562 of the services found by WIRE, had also been found by BOSS. If one takes this into account, plus the fact that about 92% ($\frac{4400}{4800}$) of all sites for seeds were used in the selection, which in turn used 75 % of the pages found by BOSS, the total expected gain from using WIRE would be:
$g_w = \frac{1673-b_w}{b_w} = \frac{1286,625}{386,375} \approx 3.3$, where $b_w = 562 \cdot \frac{4400}{4800} \cdot 0.75 = 386,375$.
This is near 3 times, as was previously estimated in section 4.1.3.

Another interesting fact, is that the shorter runs for the agent had a higher average relative availability than the longer. This is a result which is opposite to the manual runs. The average relative availability for the shorter runs was: $A_{ra} = \sum_{n=1}^{5} \frac{a_n}{t_n} \approx 0.20$
For the longer ones it was about 17%. This means that the best pages are usually found early. Where pages with bad links come later in the process.

Contribution from large listing pages: There are a few listing-pages which gives very many OGC-links.
Which are these, and how are they found?
An analysis of the queries asked by the agents show the following: The queries "request=getcapabilities", "service=wms wms" and "service=wms 4326" are frequently occurring for the steps which give more than 500 OGC-links as a result. By analysing logs from the test-runs, we found 4 sites with more than 500 OGC-links.
These are: "http://www.skylab-mobilesystems.com", "http://acordocoletivo.org", "http://gis.glin.net" and "http://www.microimages.com".
The first one is familiar, download from it was used a separate discovery-method. (see section 3.1.1)

In the tests performed, it seemed like most of the results are achieved by finding a few large listing pages.
Therefore, we explicitly downloaded the OGC-links from the four large listing pages found. The result was a total of 1077 unique links. The relative availability of the OGC-links found on these pages were only 2.7%, which means that they do not contribute significantly to the actual result.

Ideas on improvement

The BOSS-agent developed in this thesis is by no means optimal. The goal for the agent-development was however not set on optimality.
There are many ways, in which it can be improved, in terms of an increase of the the number of available services found per time unit.
Some deeper investigation on the search-results might be useful. Some search-words give very few results and should get a lower score.

One could also poll the OGC-links found and use the result as a complement to the score given and update scores after only live OGC-links instead of all OGC-links. This would alter the score in favor for words that actually find many available OGC-links.

A problem as a whole, with the heuristic currently used, is that it tends to get stuck in a local maximum. Since it uses the information on links found to find more links, it will fail to find OGC-links which look a bit different. Some machine-learning approach, which also takes the result in every step into account, maybe would be preferable. This way over-fitting unto the most common type of links may be avoided.

4.3.2 WIRE-modules analysis

To be developed as modules no. 2, 3 and 4, the WIRE-modules should handle the crawling. The first WIRE-module is responsible for extracting seeds from the BOSS-Agent. The second is to be used for the actual setup of the crawling and communication with WIRE. The last is a parser for results generated by WIRE.

Seed Extraction analysis

The seed-extraction will use a setting of a minimum size for each group and a maximum number of groups.
Each seed-group will have roughly the same size. The seed-extraction module was only analysed for the amount of seeds it generated, due to its simple functionality.
The number of seeds generated by module no. 2, was measured for each agent run, the result can be seen in figure 4.17.
In order to avoid a cluttered graph, only the three latest versions of the agent is displayed, along with average values for all agents (AVG), averages for the three latest (AVG v2) and a linear estimation of the average (AVG LIN). For a run of 31 steps, which is the default number, there will be about 7000 seeds available.
One interesting fact is, that the number of seeds generated by later versions of the agent is generally less, as can be seen if one compares the average values for all agents (AVG), to the average for the three latest agents (AVG v2). Generally the BOSS-Agents generate more seeds than the manual runs of BOSS, this is probably because the BOSS-Agents use more different queries at each iteration, and will thus download a more diverse set of pages.

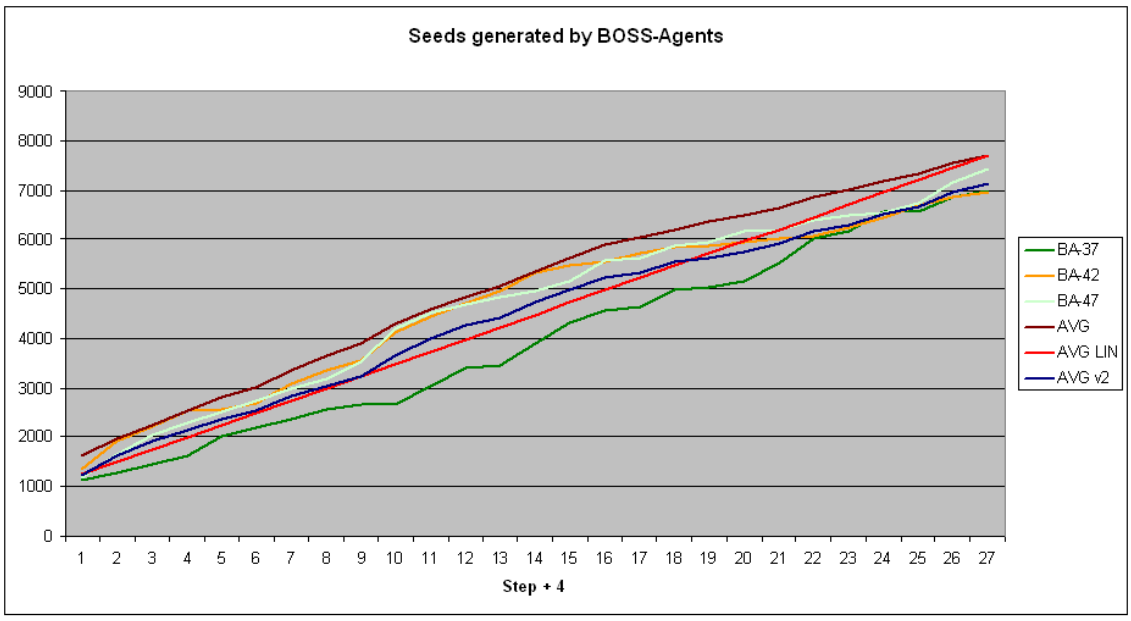

Figure 4.17: Number of WIRE-seeds generated by BOSS-Agents

Analysis of module 3 and 4

Due to the fact that the 3rd module was never finished, and that the 4th requires input from the 3rd, no analysis was made for these modules.

4.3.3 Management-modules analysis

The management of OGC-services, was handled by a poller-program (module no. 5) and an analysis program (module no. 6). The poller was capable of detecting which services that were available. The analysis-program for the poller, analysed data downloaded by the poller to keep track of duplicates.

The poll method

Developed as module 5, or OGC-poller. The developed poll-method have been improved and changed several times during this thesis work. Basically it just responsible of downloading meta-data to OGC-services in a fast and reliable way.
It has been run at three major occasions. The first time was after the first test of OSBC, the second was after all methods had been tested and the last was after the BOSS-agent had been developed.

The first time the poller was run, it reported that $F_o = 40.18\%$ of the currently found links pointed to live OGC-services. For links from the first two discovery-processes, Skylabs and Google Meta, the corresponding fraction was just $F_{sgm} = 22.75\%$.

At the second run, when all discovery-processes had been examined, the alive fraction of OGC-links was $F_w = 40.50\%$
This can be explained if one takes into account that the web is dynamic. It changes constantly, so the most recently acquired links are more likely to be alive than older links. The links found on the Skylabs-pages, had been crawled for, at some earlier time by Skylabs, meaning that many of these links were old already when we acquired them.
Using the information on the total amount of links $O_o = 8778$ and the amount of links from Skylabs and Goggle-meta-search $O_{sgm} = 1987$, it is easy to see that $F_b = 45\%$ of the new links were alive, which is about twice the fraction for the old part.

$$F_b = \frac{F_o \cdot O_o - F_{sgm} \cdot O_{sgm}}{O_o - O_{sgm}} \approx \frac{3527 - 452}{6791} = \frac{3075}{6791} \approx 0.4528$$

The poller also gave a hint about the composition of the different types of OGC-services among the available services.
The composition was found to be about 75% WMS, 21% WFS and 4%, WCS after the first run, and 76% WMS, 19% WFS, 4% WCS, after the WIRE-tests in the second run.
For the Skylabs and Google -pages this was: 80% WMS, 12% WFS, 7% WCS, after the first run. The remaining OGC-services found totaled approximately 0.1% for all these cases.
Since the Skylabs-pages are said to only contain WMS-services, it is understandable that the portion of WMS-services is higher here.
Although it is a little odd that the fraction of WCS is largest for the Skylabs & Google -URLs, the statistical data for this set is smaller than for the other tests, so some deviations are expected.
To summarize it, a likely composition of OGC-services on the web is about 75% WMS, 20% WFS and 5% WCS, making WMS about 3 times as common as the other two services

together.

The poller was finally run a third time, using the duplication detection technique described above.
In this run, the live OGC-fraction was 42.65%, which is comparable with previous results. The total unique count of OGC-links was found out to be 2982, which is 33.97% of all found links. The composition of OGC-services among the unique available was this time: 76% WMS, 19% WFS, 4% WCS and 1% other OGC. Some more focus was put into actually identifying other OGC-services, which explains the higher result for it this time. Otherwise, the composition is similar to previous results.

The improved duplication removal

Developed as a part of module no. 6. The problem of detecting duplicates when only having the URL to a service, is as stated in section 3.4.2, difficult.
When the content is available on the other hand, the duplication detection problem becomes much easier. For any generic data, it is trivial to tell whether two documents are duplicates. A byte by byte comparison will then be sufficient.
This comparison is too strict to find very similar documents that differ on a few white-space-characters, and perhaps should be regarded as duplicates, since these characters does not change the semantical meaning of the document.

In the context of OGC-service identification, it will be possible to find two content-wise different capabilities-documents, that describe the same service. This may be due to different versions of the document, or some other minor difference.

There is a way to detect duplicates among OGC-services, using the capabilities-document. There exists a tag called OnlineResource, inside the specification of each capability.
This tag has an attribute, called "xlink:href", which can be used as an identifier for the service.
Comparing only this identifier between documents will reveal many duplicates. It is however not completely fool-proof, since the attribute is an URL, we are back at the original problem. URLs cannot uniquely identify resources, as stated in section 3.4.2. Checking two URLs instead of one, will however reduce the number of duplicates significantly. The probabilities for detection possibilities are of four kinds (true negatives, false negatives, true positives and false positives) and the following holds:

$$p^x_{tn} + p^x_{fn} + p^x_{tp} + p^x_{fp} = 1 \tag{4.1}$$

Where x is 1 or 2, depending on if it is the first or second check.
p_{fn} is the probability of an URL being detected as a false negative duplicate and having two checks will make the combined probability for false negatives: $p^c_{fn} = p^2_{fn} \cdot (p^1_{fn} + p^1_{tn})$
Where p^1_{tn} is the probability for true negative for the first check, i.e., the chance of an URL actually being unique.

This is true because the first check must give a negative result, for the second to be used.

The probability for detection of services as false positives p_{fp} would then become:
$p^c_{fp} = p^1_{fp} + p^2_{fp} \cdot (p^1_{fn} + p^1_{tn})$
All of this assumes that the two detections are probabilistically independent.
If the probabilities for false positive are the same for both checks, i.e., $p^1_{fp} = p^2_{fp}$, then $p^c_{fp} = p^1_{fp} + p^1_{fp} \cdot (p^1_{fn} + p^1_{tn}) = p^1_{fp} \cdot (1 + p^1_{fn} + p^1_{tn})$

From the formulas, it is clear that the probability for false negatives (p^c_{fn}) will be lower than p^1_{fn} , since the probability for negatives $p_n = p_{fn} + p_{tn}$, is less than or equal to 1. As can be seen in formula 4.1
The probability for false positives (p^c_{fp}) will however be greater, since it is multiplied by one plus the probability for negatives. This is bad, but since the probability for false positives should be very small, it does not matter that much.

When this method using the xlink-attribute in XML was used, it revealed that about 20% of the found live OGC-services were duplicates.
It is of course possible that there are more duplicates. This reveals that the first duplication detection method, which is performed only on links, is quite effective.

4.3.4 Estimated prototype run times

In order to estimate the total run time for the prototype, one must be able to estimate the run-time for each of its modules. The total runtime $t_t(m, i)$, using m machines for crawl, and running the BOSS-agent to step i can be formulated as:
$t_t(m, i) = t_a(i) + t_e(i) + t_c(m, i) + t_p(i) + t_o(i) + t_n(i)$
Where each component is:

- $t_a(i)$: The time for the BOSS-agent after i steps.
- $t_e(i)$: The time for the seed-extraction for WIRE.
- $t_c(m, i)$: The total crawl time for the seeds using WIRE.
- $t_p(i)$: The total parse time of the crawling results from WIRE.
- $t_o(i)$: The total time for the poller-module.
- $t_n(i)$: The total time for the poller-analyser module.

The estimation of the BOSS-agent run time ($t_a(i)$) is simple, since it has been run on several occasions, the times for the test-runs can be used as an estimate. The times are near linearly proportional to the step, with an approximate time for each step of 155 seconds.

The extraction time ($t_e(i)$) is very short and is set to one minute for simplicity of the time-analysis.

The crawl time ($t_c(m,i)$) is somewhat complicated to estimate, since it is depending on how many machines that are used (m) and if dynamical selection of crawl-depth is used. Dynamical crawl depth selection can be used if the result from a crawl at a level is parsed and then the result is analysed for how many OGC-listing-pages that was found during the crawl of the level. Then if the count exceeds a threshold value, the crawl will continue to next level, else it will halt. The metric of the number of OGC-listing-pages found at a level is most likely useful for determining how many listing-pages that will be found at the next level, as indicated by the statistics in section 4.1.3.

The actual speed-benefit for dynamical crawl-depth selection depends on settings which need a lot of experiments to adjust, which means that it is difficult to estimate the actual expected performance increase.
One can however calculate the most likely limits for the performance gain. In worst case, all groups are of same seed-quality and then there will be no benefit at all using dynamical depth selection. (all groups will be crawled to the default depth of 3). In best case, one group is of great quality and all the other groups are of bad quality, meaning that the bad groups will be crawled to depth 2 and the great one to depth 4. This assumes that the dynamical depth is limited to the range $[2,4]$.
The average crawl-times measured were 146.05 and 254 minutes for CVX-crawls at depths 2 and 3 ($t_2$and t_3).
The crawl time for depth 4 (t_4), was estimated to 374 minutes. Using the measure of average gain of acquired additional services at depth 4 vs depth 3, as 23 %. the following formula describes the best-case effectiveness change using dynamical crawl-depth selection:
$p_d(g) = \frac{1.23}{\frac{t_2\cdot(g-1)+t_4}{t_3\cdot g}} = \frac{1.23\cdot g}{g\cdot\frac{t_2}{t_3}+\frac{t_4-t_2}{t_3}} = \frac{1.23\cdot g}{g\cdot 0.575+0.897}$ where g is the total number of crawl-groups.
The limit, when the number of groups goes to infinity is:

$$\lim_{g\to\infty} p_d(g) = \lim_{g\to\infty} \frac{1.23}{0.575+\frac{0.879}{g}} = \frac{1.23}{0.575} \approx 2.14$$

The crawl-times are as the function-definition indicates, also dependent on the number of machines used (m) and the steps used for the BOSS-agent (i).
Using two machines for crawl, instead of one may reduce time required with up to two times. Also running the BOSS-agent twice as many steps may increase the time with up to two times.
A simplified formula can be specified as: $t_c(m,i) = c\cdot\frac{n(i)}{m}$, where $n(i)$ is the function of the crawl-group count and c is a constant that is dependent on many other factors.

The parse time ($t_p(i)$) depends on whether the results from crawls are parsed by the crawl-machines after each depth of the crawl, or if all results are parsed by a single machine when the crawls are finished.
The latter option would require significantly more time. The parse time for it can be formulated as: $t_p(i) = p\cdot n(i)$, where p is the time required per crawl-group.
If parsing is done by crawl-machines simultaneous with crawl, the total parse time required would be: $t_p^s(i) = 4\cdot\frac{p}{2} = 2\cdot p)$
This is true, since only the last level for four groups would need to be parsed non-simultaneous to crawl for each crawl-machine. The time required for the last level is

assumed to be half of the total parse-time for the group, hence the division by 2 in the formula.
It is assumed that there is at least one crawl-group, in queue for each machine, otherwise the parse-time would just be: $p \cdot \frac{n(i)}{m}$.
In other words, each machine has at least 5 groups to crawl ($n(i) \geq 5 \cdot m$). This is true for all cases examined in this analysis.
The parse time per crawl-group to depth 3 (p) have been measured to 20 minutes, this is the value used in this analysis.

The total time for the poller-module ($t_o(i)$) have been measured to be 20 minutes for 2635 found services. It is assumed that the time is linearly proportional to the number of services found in this analysis.
So the time for the poller is: $t_o(i) = 20 \cdot \frac{f_s(i)}{2635} = \frac{f_s(i)}{131.75}$, where $f_s(i)$ is the number of services found, when i steps are used in the BOSS-agent.

The total time for the poller-analyser is estimated in the same way as for the poller, and is: $t_n(i) = 30 \cdot \frac{f_s(i)}{2635} = \frac{f_s(i)}{54.5}$ minutes

Chapter 5

Results

This thesis is about the search, evaluation and development of discovery and management methods for OGC-services on the web, as stated in section 1.

5.1 Discovery methods

There are some base types of methods for discovering OGC-services. In this thesis, one simple harvest method, two meta-search methods and two crawling methods, are tested.

5.1.1 Found methods

The methods found and tested are:

- Harvest from Skylab's pages
- Manual Google meta-search
- Auto Yahoo! meta-search
- One Step Brute-force Crawling (OSBC) web-crawl
- WIRE crawl

The two last methods, used results found by the API to Yahoo!'s search engine called BOSS. Hence their result is dependent on the result of the Yahoo! meta-search method. This was practically unavoidable, since the crawl-methods required a large amount of varying pages related to OGC. The best way to find many of these page was at the moment by using the BOSS-API.

5.1.2 Discovery method evaluation

The evaluation of the discovery methods are partly based on the two management-methods developed. This evaluation assumes that the management-methods are fail-proof. Something which we cannot guarantee, which is why the evaluation results should only be seen as guidelines, and not as absolute truths.

Even if the the management methods would be free of bugs and errors, they would only give an answer of the effectiveness at the current time. This is because the web is

not static, i.e., a method which is superior today, may be inferior tomorrow. How the web and its tools develops is impossible to predict. In any way, this section describes the results of each method, calculated by methods developed during this thesis work.

Skylabs results

As mentioned in section 4.1.1, this method is very limited and usually gives poor results, most probably due to outdated links listed on the pages. Since the Skylabs-pages were the only major OGC-listing pages found in this master-thesis work, the result from listing-pages is the same as the result from the skylabs-pages. And with only 91 live services found, this method has to be considered as poor. The performance per page was however very high, since 45,5 services per page were found. (There are two skylabs-pages)

Google meta-search results

Described in more detail in section 4.1.2. This method could not be automated, due to terms of use conditions. [Google 01]
The result was 686 URLs, of which 281 were pointing to live services, for 1000 search results. This indicates a decent performance anyhow. However, since the methods for discovery should be able to automatize, this method have to be considered as a poor choice.

BOSS results

This method is analyzed in section 4.1.2. Compared to the Google meta-search method, this method can be automatized, which makes it the only decent choice of meta-search method to be tested in this master thesis. Since the size of the Yahoo! index was about one third of the Google-index, it was thought to be good enough and no other search engines index were tested with meta-search.
Comparing the results, to the Google search engine is difficult, since the test for Google only used inURL-search. The Yahoo! test used only standard search. To compare in a somewhat fair way, one could look at the results given for BOSS using the same search word as was used for the Google-test. This search word was "request=getcapabilities". For BOSS it gave 797 URLs and 240 live services for 688 results. This means that this method gave about 35% live results compared to 28% for Google. It did however include fewer results, and each result in BOSS could give more than one live service.

OSBC results

As mentioned in section 4.1.3, this method had a poor performance. Only 63 live services were found using it. These services were found among 1182 URLs found after 368 seed-pages had been crawled one step. The total number of downloaded pages were 12026. The effectiveness of this approach (live services found per page downloaded) was very low. $E_{OSBC} = \frac{63}{12026-368} = \frac{63}{11658} \approx 0.0054 = 0.54\%$
Not even a percent. This meant that there is a need for a much better crawl-method.

WIRE results

The results from WIRE are described in detail in section 4.1.3. In total 1673 live services were found, of which 1111 were not previously found by using the BOSS-API.
As a comparison with the OSBC method, the WIRE method also had a very low number of live services found per page downloaded.
In section 4.1.3, the effectiveness for each of the crawl groups for depth 3, was computed. This was at most 0.34 %, which is in fact even worse than the OSBC results. This statistics is a bit misleading though. This is because, the crawl-groups in WIRE are crawled to depth 3, where OSBC is only crawled to depth 1.
At higher depths it is understandable that the hit-percentage gets lower for a general crawler like WIRE.

For all groups except CNG, the relative number of services found is much higher at depth 1. Also for the best seeded crawl-group CNF, the relative number of services found is more than twice of the OSBC-test. (1.2 % vs. 0.54 %)
This means that WIRE is not much better than OSBC in terms of services found per page. WIRE does however have a significantly higher download speed than OSBC.
WIRE is however even for crawls to depth 1, a better choice, since it is more bug free, polite and configurable than my own implemented OSBC-crawler.
As expected however, using a general crawler like WIRE, is not much better than brute-force crawl.

5.1.3 Developed discovery methods

WIRE and BOSS were the two of the methods tested, which gave a much better result than all the others, which lead to the idea of the agent for discovery to be developed.

The BOSS-method

The boss-method was developed as an automated engine in module 1, for the prototype.

The WIRE-method

The WIRE-method was planned to be developed as three modules, numbered 2, 3 and 4. But due to lack of time, the third module was never developed.
Module number 2, was developed as an extract from testing methods. It is capable of generating seeds and settings for the WIRE-crawler using results from the first module.
Module number 4, was developed also as an extract from testing methods. It can parse content downloaded by WIRE, for OGC-links.

5.2 Management methods

The problem of managing a database of OGC-services revolves around several issues. Most of which were not examined in this thesis work. There were mainly two methods for management that were tested. Detecting online services and detecting duplicate services.

5.2.1 Found management methods

The methods for management arose from the need to check the actual real effectiveness of the discovery-methods. Hence there were no active search for management methods in this thesis work. The focus was put on the discovery methods.

5.2.2 Management method evaluation

Not much was done in the area of evaluation of these methods. The evaluation of these methods are problematic, since there is not that we know of any statistic available for the expected fraction of live OGC-services or fraction of duplicate services.
All that has been done, in this area for this thesis work, is testing and reasoning that the results given seems to be valid, or near valid. Again due to the dynamic nature of the web, this is a problematic issue.

5.2.3 Developed management methods

Both of the used management-methods were also developed as modules 5 and 6 for the final prototype.
The management-modules were used to evaluate the discovery-methods.
Using the links to all found live OGC-services, the percentage of all services found, was computed for all discovery-processes.
The results was:

- BOSS: 62.61%
- WIRE: 56.10%
- Google-Meta: 9.42%
- Skylabs: 3.05%
- OSBC: 2.11%

All the live services found by Skylabs and Google Meta-method was also found by the BOSS-method. Meaning that the BOSS and WIRE -methods were far superior to the others, and combined they found 99.87% of all live services, which is almost all.

5.3 Left to implement

The third module, also known as the Wire-Crawl-System-Starter module, is left to implement. This module is to be responsible for the WIRE-crawl-management.
That is it is supposed to setup the machines for WIRE-crawl, start the crawl and then wait and forward the results from the crawl to the machine running the module-handler. See section B.5, for a detailed explanation of how this module is supposed to be implemented.

Chapter 6

Conclusion

There are many conclusions to draw from the results of this master thesis. The most important of them are regarding the effectiveness of the tested methods for OGC-service discovery and the expected performance of the agent developed.

6.1 Effectiveness of different discovery processes

There are many ways in which on can measure the effectiveness of service discovery. One decent method is measurement of the number of unique live services discovered per time unit, i.e., live service discovery speed.
It can be measured as: $V_d = \frac{S_L}{t}$ The the discovery-velocity (V_d) is the number of new live services (S_L) discovered per second (t).
This speed is initially high for most methods, and then gets lower, this is usually because some services are more common than others and will frequently reappear in the searches. This means that these services are usually found early, and the amount of new services found keeps decreasing.

6.1.1 Effectiveness of list-page usage

For list-page usage the effectiveness value is usually very high, this is because there are very many already unique services at a single page.
The downside is usually that the relative availability of the services found here is usually low, so the total amount of live services found this way has been proved to be very small. Also most OGC-services found are not reachable via large listing-pages. In other words all too few OGC-services were found using this method, to say that it is a reliable method.

6.1.2 Effectiveness of meta-search

A much more reliable method than usage of listing-pages is the usage of search engines for meta-search. Meta-search have been proved to be really effective. It has given a relative number of found live services of about 30% for the first 1000 pages.
After a measurement of the time in between the saved boss-results had been done, using the modification time-stamp for files related, the actual time-based effectiveness of BOSS meta-search was computed and plotted in figure 6.1.

As can be seen in the figure, the old manual BOSS-search performs best in the end with a speed of about 0.15 new services per second.

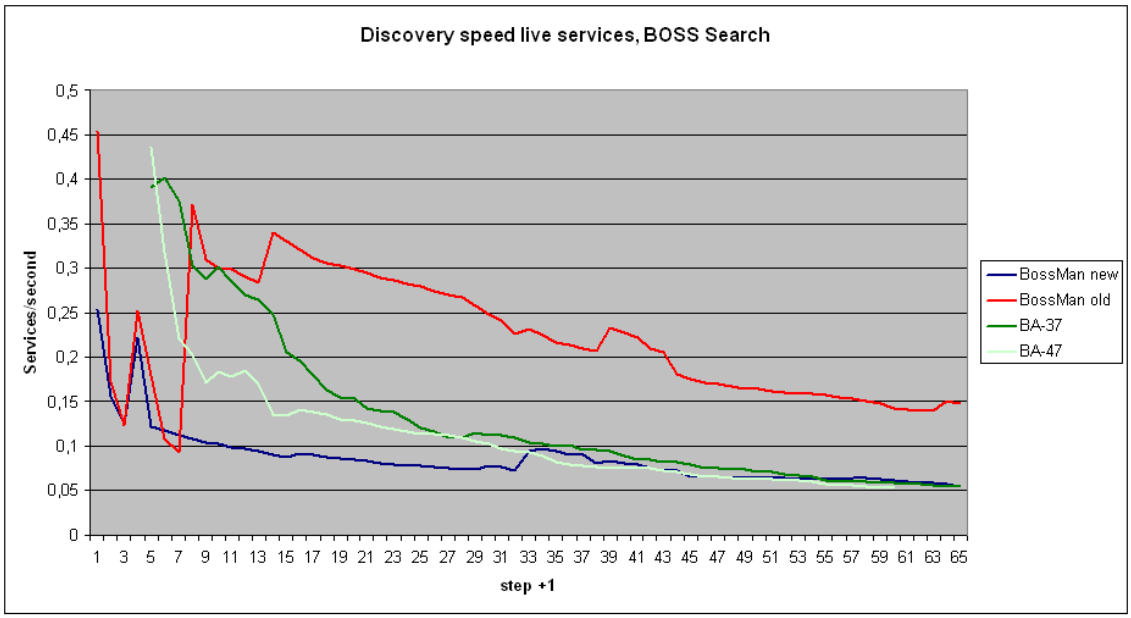

Figure 6.1: Discovery speed for BOSS-search

The BOSS-agents have practically the same speed at the end as the new manual BOSS-search. This speed is about 0.06 new live services per second.

One interesting fact is that the improved boss-agent BA-47 has lower effectiveness as the older BOSS agent BA-37. This is because even though the new BOSS-agent finds more services than the old, it takes extra time to do so, hence, its effectiveness is actually somewhat lower. In fact the second version of the boss-agent finds about 6 % more OGC-services than the old, but at a cost of 17% more time.
We would still recommend using the latest version of the BOSS-agent, unless there is very limited time available for the search, since it actually finds more services.
In the beginning, the BOSS-agents outperform the manual search, but then its performance levels with the manual search.

6.1.3 Effectiveness of general crawling

General crawling can be used as en extension to some other method for OGC-discovery, such as meta-search, since it requires starting pages as seeds.
In this way, a crawler can be used to acquire more material from a search engine, which is not directly reachable by its index, but which is available from links to indexed pages.

Only the WIRE crawler was examined for effectiveness, due to time constraints. For the latest examined crawl-groups-types CNG and CVX, time-data was saved for each crawl level, which enabled the construction of effectiveness statistics at each crawl-level for these groups. Figure 6.2 shows these statistics.

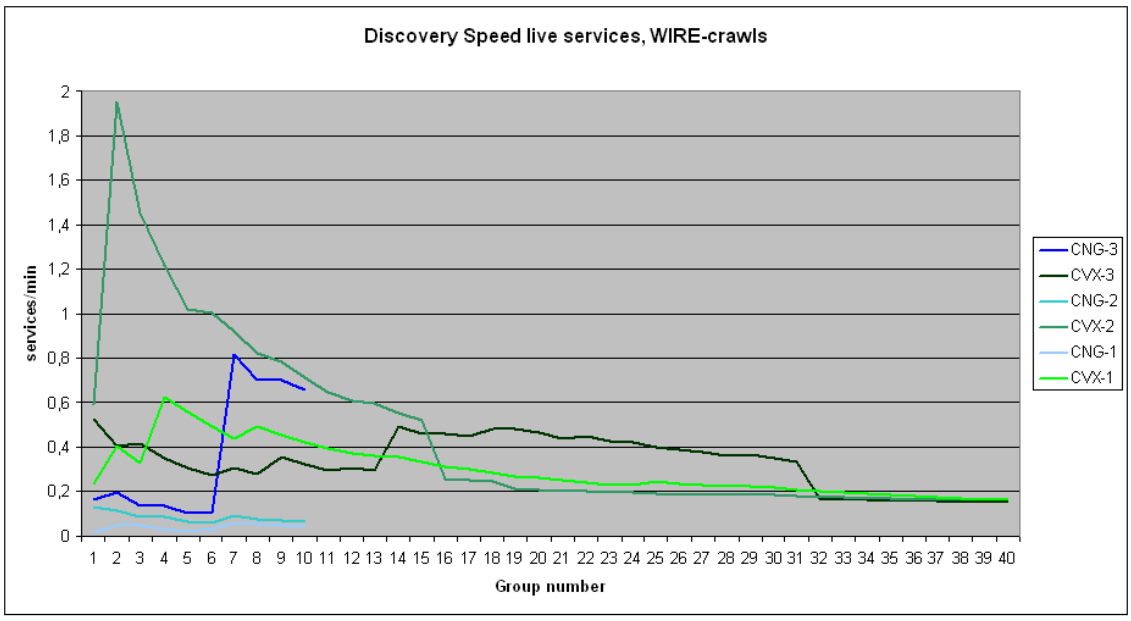

Figure 6.2: Discovery speed for WIRE-crawls

For the CVX-groups, the effectiveness goes to about 0.2 new live services per minute, in the end. This is about 20 times slower than the meta-search speeds for BOSS-Agents. The crawling for the groups can however be run as parallel processes. Something which is not possible for the BOSS-agents, since the result from a search for the agents, is used to determine which query to use in next search.
If one assumes that 10 crawl-groups are used and all are run in parallel, than this part is actually only 2 times slower than the meta-search, which is not so bad.

For the CNG-groups, the effectiveness is as low as 0.05 for the first levels. For level 3, one group found very many new OGC-links, so the result deviates a lot from the other depths, and reaches about 0.7.

6.2 Prototype performance

The developed prototype for OGC-service discovery and management, consists of 6 modules. The first one handles automatic meta-search and its effectiveness can be compared to the statistics for the BA-47 agent. Even though it can be configured to work as any of the agents mentioned in this report, BA-47 gives the best result, so it is regarded as the default configuration.
The modules 2 to 4 handles additional crawling using the WIRE-crawler. Of these the third module is not yet written.
The last two modules numbered 5 and 6, handles management of the discovered services, and evaluation of the search-results.

6.2.1 Estimated effectiveness of prototype

For the planned prototype, it is possible to estimate the effectiveness and performance. Since the prototype is not yet finished, there can only be estimations of the performance. These estimations are based on the observed results of the testing sessions for the finished modules.
Since the missing third module can be implemented in several different ways, there are more than one estimated performance for the prototype.
There are at least three features which significantly affects the performance of the third module. These are:

- Distribution of WIRE-crawl onto multiple machines
- Dynamical selection of crawl-depth for each group
- Parsing of completed groups, simultaneously as others are crawled.

Another setting which affects the performance is for how long the first module is run. If it is run for a long time, there will be much more seed-pages to use for the WIRE-crawl. In fact the number of seeds generated by the BOSS-agents is near linear to the number of steps it is run.

After 31 steps, the best BOSS-Agent found 545 services.
It is of course difficult to estimate how many more services that will be found when crawling, compared to how many one had found prior to crawl. In this analysis the total number of services found when using WIRE, for all sessions was compared to the number of services found at the first level of the crawl. It was found that approximately 3.6 times more services were found after crawls to depth 3 were completed.
For simplicity this value will be used to estimate the number of services found after crawl of the BOSS-seeds. Since this analysis is only a crude estimate, the actual factor could be much better, since there are more seeds available, or worse for the same reason. Earlier crude estimations of this value in section 4.1.3, indicate that it should be somewhere between 2 and 4 times, so maybe it is set to high.
Using this factor, the prototype with default settings would find:
$545 \cdot (3.6 + 1) = 2507$ services. This would be a very good result, and perhaps a slight overestimation, since the total amount of live services found during all of the tests performed for this thesis work was 2982.

The total times calculated for some different possible prototype runs, are shown in figures 6.3 to 6.6. The run-times are displayed for one to four machines, in hours. For a detailed explanation on how the run times were calculated for all the modules, see section 4.3.4.
On x-axis, each number corresponds to the BOSS-agent step (i) of, one plus ten times the number of x-axis. ($i(x) = 10 \cdot x + 1$)
The effectiveness-values are displayed as dotted lines, with the unit of services discovered per hour for the y-axis.
The dynamical statistics are computed using the average gain-factor of the two extreme

cases, examined in section 4.3.4. ($d_{avg} = \frac{1+2.14}{2} = 1.57$)
This is a crude estimation, but there was not enough available statistical data, to compute better values for the expected gain.

Since the increase of efficiency of using two machines compared to one is near two times, we would recommend this. Even though the effectiveness gets lower with more steps used for the BOSS-agent, we would recommend running the agent 31 steps, since the loss is not that much.
The implementation of both simultaneous parsing and dynamical depth-selection is a must. Dynamical versus statical depth selection increases effectiveness with circa 52 %, and parsing simultaneous increases effectiveness by 66 %.

In the end these settings would lead to the discovery of about 3000 services in 33 hours. This would correspond to an effectiveness of 93 services per hour. (0.026 services per second) This is more than two times slower than the discovery speed of the BOSS-agent, but it may be worth it, since about 5 times more services may be found using the full prototype, than the BOSS-agent alone.

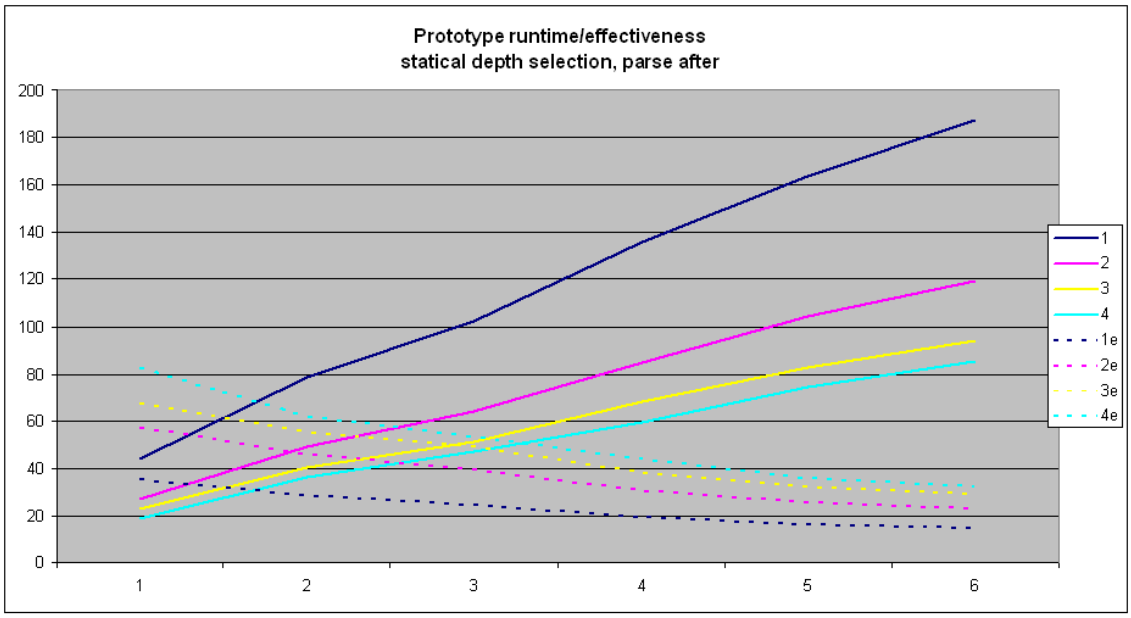

Figure 6.3: Estimated prototype run times and effectiveness, statical depth selection, parse after crawl

What have not been examined, is how the performance would be if the WIRE-crawling part was replaced with a focused crawler. This could probably greatly improve the performance, since the related pages would be found much earlier. This could potentially increase the effectiveness by 3 times, since accuracy for focused crawler selection-metric compared to standard selection-metrics such as PageRank, have been shown to be about 3 times greater. [N3 01]

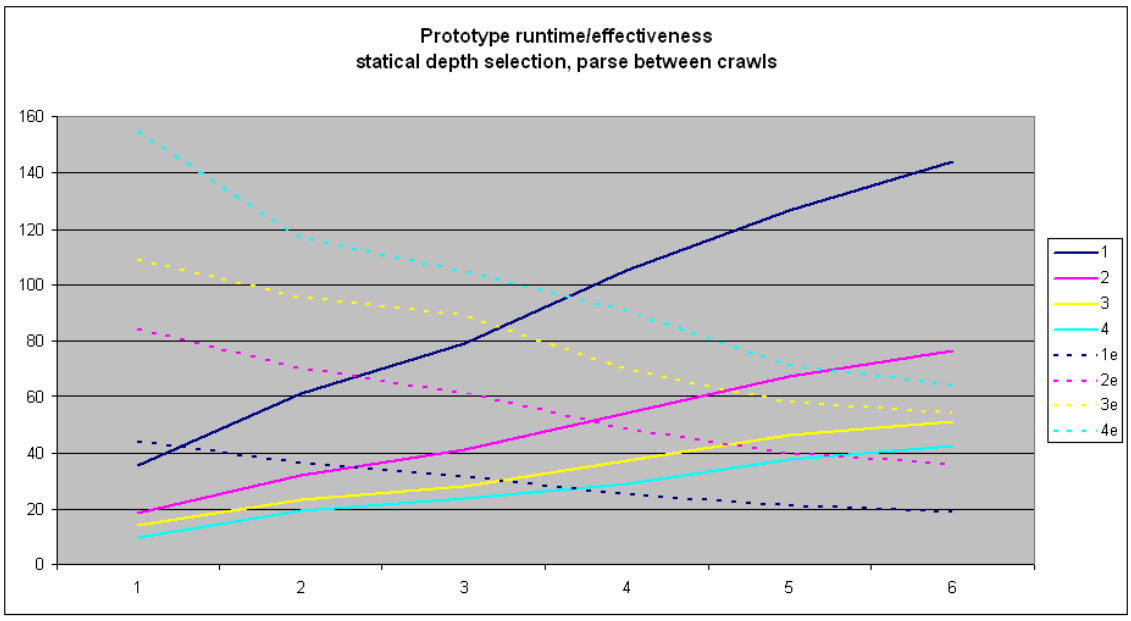

Figure 6.4: Estimated prototype run times and effectiveness, statical depth selection, parse simultaneous with crawl

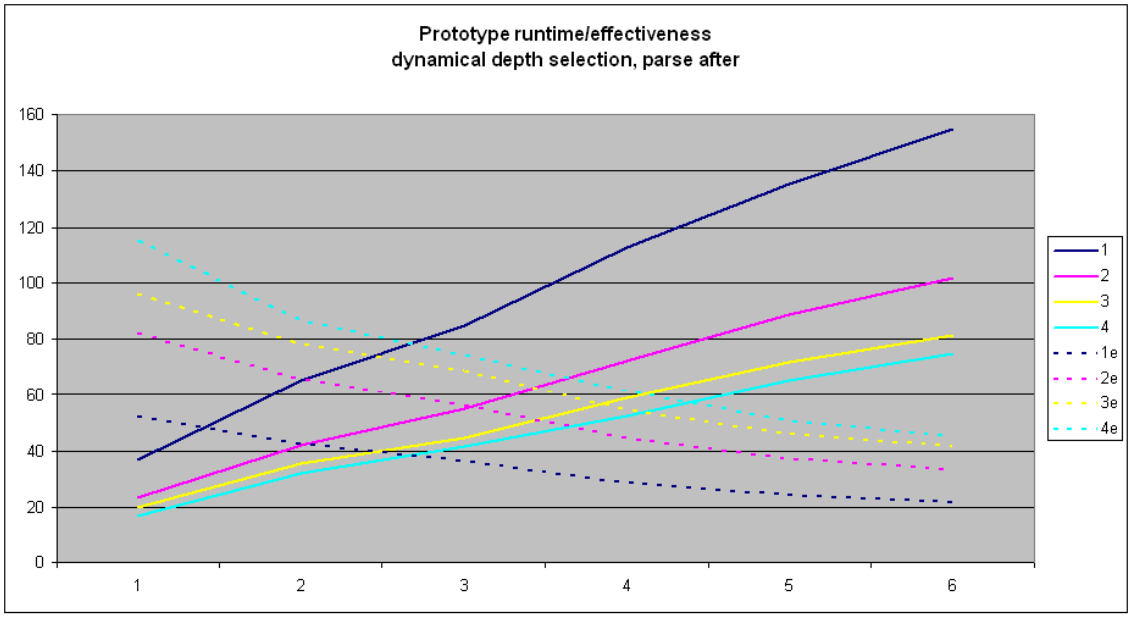

Figure 6.5: Estimated prototype run times and effectiveness, dynamical depth selection, parse after crawl

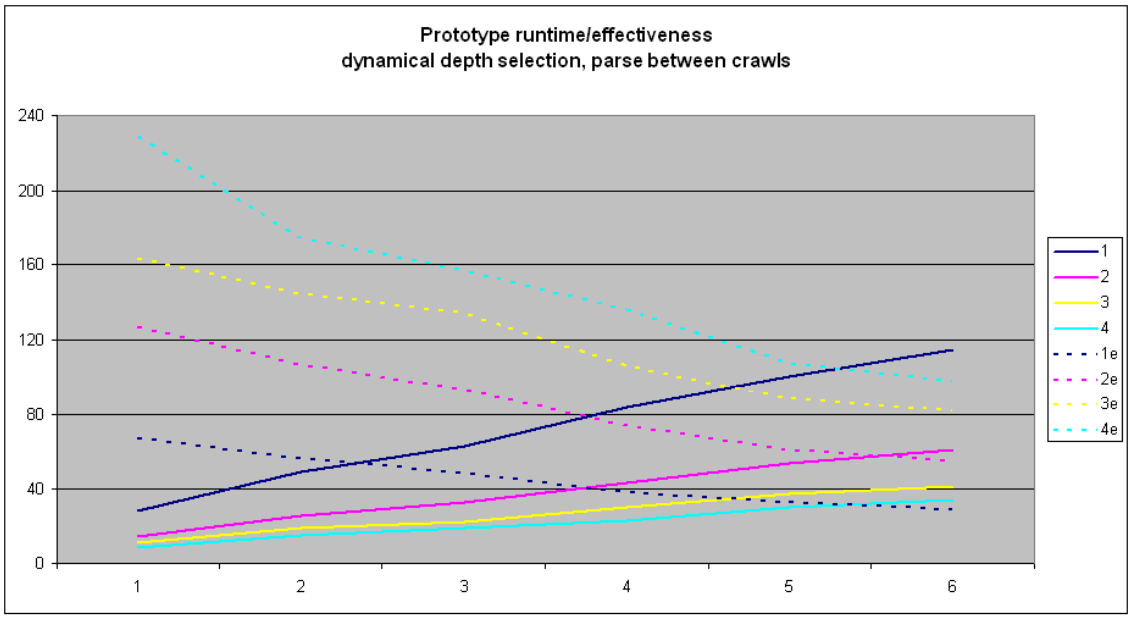

Figure 6.6: Estimated prototype run times and effectiveness, dynamical depth selection, parse simultaneous with crawl

Appendix A

User documentation

A.1 Introduction

The application developed during this thesis work, called OGCSearchSystem (OSS), consists of six separate modules and a manager-program for using them together.
The purpose of the application is automated search for OGC-services, and evaluation of the search-results. Every module of OSS is implemented as a standalone executable, and may be executed on their own. It is however recommended to use the module manager.

A.2 Tutorial

The module-manager has a simple forms-interface, which is used for selecting which modules to be run. The modules are listed in their run order from top to bottom: The check-box next to the module is to be checked if the module should be run.

To the right are two shortcut settings for three typical runs, boss-agent only, wire only and full run. The boss-agent only run is currently the only fully implemented setting. It consists of a run of modules 1,5 and 6. Clicking the run button will start the execution of the modules in order top to bottom.

There are two check-boxes which control the configuration of the module-handler. These are named "Configuration test" and "Enforce module compatibility". The first one is initially checked. Having it checked will result in a test of the configuration of each module selected to be run, instead of a regular run of the modules.
It is recommended that a full configuration test is run the first time the module-handler is used, since this will generate the default config-files without starting the modules.

The other check-box enforces module compatibility. This means that the module-handler will change the configuration-settings for the modules selected to run, so that they will be compatible with each other.

Since the modules are run sequentially, the settings of the first module is used as a basis, which will affect the settings of later modules, so that they will work together. For instance if module 5 and 6 are to be run, the setting for the path to the downloaded files by the poller (Module 5) will be set in module 6.

There is also a status-box at the bottom of the forms interface. It will display the current status of the module handler.
A Ready status indicate that the module handler may be started by clicking the run-button. Any other status will show if there is a configuration-test or an ordinary run of a module in progress, along with the name of the module currently used.

The output for the modules are saved in text-files with their corresponding name. For instance output_BossAgent.txt is the name of the output-file for the first module, named Boss-Agent.

A.2.1 Modules

There are 6 modules in total, which all are needed for a full run of the OGC-search-system. These are the modules:

- Boss-Agent: The Yahoo! BOSS agent. Uses automated meta search for finding OGC-URLs and OGC-listing pages.
- WireSeedExtractor: This module extracts seeds from the BOSS Agent to be used for crawling in WIRE.
- WireCrawlSystemStarter: This module is not yet implemented, but should be a communication-module for transferring the seeds to a WIRE-enabled host, start WIRE and then wait and transfer the result back.
- WireResultsParser: A module for parsing results from WIRE-crawls for OGC-links.
- OGCPoller: This module polls OGC-links listed in a file. It connects to the URLs and attempts to download the documents that the URLs point to.
- OGCPollerAnalyser: This module analyses the results from the OGCPoller-module and outputs a set of XML-documents without any duplicates or errors. It also return lists of summaries of the polling result.

A.2.2 Results output

Results for each module should be saved in the folder belonging to that module. Every module has its own folder. This can however sometimes be changed in configurations.

The default settings for the boss-Agent will output the results in folder:
Boss-Agent\bin\Debug\kb_r51\ogc.
For poller (Module 5) results are in folder: OGCPoller\bin\Debug , For Poller-Analyser (Module 6) it is in: OGCPollerAnalyser\bin\Debug.

A.3 User guide

For a more advanced run, there is a possibility to change the settings for each module. Every module has a default setting, which is generated if there is no config-file present.

If a config-file is present, the module will attempt to load these settings and use them instead of the default settings.

A.3.1 Configuration of non default settings

All modules has a configuration-file named config.txt. Also the module-handler has a configuration-file. There are not that many options for configuration of the module-handler, and the default values are recommended. For the other modules there are more settings which may be edited. For a list of all the configuration options and an explanation of them see section A.4.1

A.4 Reference Manual

A.4.1 Configuration Options

Here is a list of all the different configuration options with a brief explanation. All of the configuration-settings are of the format key : value. Where the delimiter is a colon-character surrounded by two white-spaces.

OGCSearchSystem

- consoleBufferSizeChars : This is the size of the text-buffer in characters, used by the module handler. This text-buffer stores the text-output from the standard output of the currently running module. The default size is 2048 characters.
- consoleUpdateIntervalMS : This is the interval time for the console update in milliseconds. A console update is a buffer read and a save to file of the text-output. Default value is 1000. This with the default value for the buffer-size will lead to a maximum speed of 2048 characters per second, which should be enough for most modules.
- writeConsoleToFile : This boolean should be set if the text output should be saved to file. It is recommended to save the text-output, so the default value is True.
- chainModuleConfiguration : This boolean sets the initial value of the check-box for enforce module compatibility. Default is False.
- configurationTest : This boolean sets the initial value for the Configuration test check-box. Default is True.
- verbose : Verbose text output mode. Default is False.
- m1Enabled : Is module no. 1 enabled. (Boss-Agent)
  Default is True.
- m2Enabled : Is module no. 2 enabled. (Wire-Seed-Extractor)
  Default is True.
- m3Enabled : Is module no. 3 enabled. (Wire-Crawl-System-Starter)
  Default is False

- m4Enabled : Is module no. 4 enabled. (Wire-Result-Parser)
  Default is False
- m5Enabled : Is module no. 5 enabled. (OGC-Poller)
  Default is True
- m6Enabled : Is module no. 6 enabled. (OGC-Poller-Analyser)
  Default is True.

Boss-Agent

- k : The number of queries to use before recomputing the knowledge-base.
- fileMode : Enable this if you wish to load boss-query results from file. This is not recommended, unless you are using results from a very similar run again.
- kbRelPath : The relative path to the knowledge-base for the boss-agent.
- stopAtIndex : Iteration index after which, the boss-agent is complete. A zero means that only the initial static queries will be used.
- continue : Set this to true if you wish to continue a boss-agent run, which has been stopped at a previously lower set stopAtIndex value.
- compactMode : Compact mode means that the hash-tables for URLs used are stored only by their hash-values. This is recommended, since it saves a lot of space and some time.
- useCorMetric : If set to True the invert correlation metric will be used, this metric generally leads to better results faster, hence it is recommended to be set to True.
- seedMask : The seed mask is a binary mask that determines which queries in the "initQueries.txt" file that should be used. The mask works as follows, a one bit means that the query should be selected, zero excluded. Each bit in the mask corresponds to a row in the query-file. The matching is least significant bit to first row. For instance a number $16 = (10000)_{base2}$ means that the fifth row only in the query-file will be used.
- idNr : This is the id-number that is associated with the run of the boss-agent. Changing the id-number will result in a different storage-location for the knowledge-base, when the run is completed.
- hoursManagerRefreshPages : This time which is specified in hours, describe when the pages previously downloaded are allowed to be downloaded again. Normally the manager avoids repetitive download of pages with the same URL. The default value of 150 here is equal to 6.25 days. In practice this is only useful when an agent that was run some time ago is resumed.
- parserBatchSizePages : The amount of pages per batch for the parser. The manager will download and parse all pages in batches, to avoid running out of memory. The

default value of 200 pages, will typically lead to a memory usage of less than 160 MB for this part, and is recommended.

- parserThreadCount : The amount of threads to use for the parsing. For a multi-core CPU, this should be set to a value of at least the same as the number of cores, if maximum performance is required. The default value of 3 is set for optimal performance on a dual-core machine.

- downloadBOSSPageLimit : This is the limit of the number of pages that may be downloaded from boss per query. This limit exists for two reasons. Firstly using BOSS costs money, so setting this to a low value makes the agent cheaper to run. Secondly, download of pages takes time, so it is also a performance issue. Setting the value too low will however lead to poor performance, since only highly ranked pages then will be downloaded. The default value is set to 2000, which corresponds to 40 BOSS-requests and a per query cost of 0.032 USD.

- recomputeICorMaxRounds : This is the limit of iterations that may pass in inverse correlation-mode before the correlation-values should be recomputed. Since the score will become more accurate on a re-computation, it is recommended that this is done some times during a BOSS-Agent run. A re-computation will take some time to perform however, especially at later steps, so it saves performance to set it to a high value. The default value of 10 is a decent trade-off between accuracy and performance.

- excludeUrlPICorLimit : An inverse correlation score lower than this value will lead to that the corresponding query word will be excluded. This is a performance-optimization limit. Setting this value low will lead to worse performance and a more accurate score, and vice versa. The default value is 0.01.

- refreshHoursResultCount : This is the duration of time for when a result count for a query-word is to be updated. Sine the database for the Yahoo! search engine is frequently updated, this value should not be set that high. The default value of 90 is less than 4 days. Again this is a matter of performance and accuracy trade-off. Lesser value means lower performance but higher accuracy. Setting this value right is important, since the result-count-list is reused by later agents.

- combinedQueryGroupSizeLimit : This is the maximum result count for a query word to be used in single mode. I.e if BOSS returns a count less than this value, the query word will not be combined with other query words.

- singleCombinedQueryScoreAdj : This is the score penalty for combined queries. A value of 1 means that combined queries will get the average score of their two single parts. A value higher than 1 prioritizes single queries, a value lower than 1 prioritizes combined queries. A value slightly higher than 1 is recommended, since single word queries are usually better at finding new OGC-services. Hence the default value is set to 2.

- resultUpdateScoreLowThreshold : This is the minimum score for a query that should be used in BOSS. If there are no queries left with a value greater than this, the agent

will stop. The reason behind this limit is that queries which give bad results should not be used in BOSS, since this would just cost money.

- icorMaxContributionCombinedQueryFrac : This is maximum contribution that an inverse correlation value may have on a combined query. The default value is set to 0.75, which means that the score for combined queries at most can be lowered by 4 times ($\frac{1}{1-0.75} = \frac{1}{0.25} = 4$)
- bossFetchThreadCount : The number of threads to use for the MTD in boss-fetch, this is equal to the number of simultaneous queries asked to BOSS. This value should not be set too high, since it could stress the BOSS API, and not too low either, since that would give bad performance. Default is 10.
- bossFetchTimeoutSeconds : The timeout in seconds for connections to the BOSS API. Default is set to 30. There should be no timeouts under normal circumstances, this is just a safety precaution.
- bossOAuthKey : This is the OpenAuth public key to be used for BOSS. When a BOSS-account is created for an application this key is generated. It should be entered in this config-file. The default key is connected to the BOSS-account that was used during the thesis work period.
- bossOAuthSecret : Same as for bossOAuthKey, except that this key is private, and should be kept secret.
- mtdThreadCount : The thread-count for the MTD to be used during download of pages the BOSS-queries have pointed out. This value should be set as high as possible, but not to high. 50 is default.
- mtdTimeoutSeconds : The timeout for TCP-sockets in the MTD. Default is 30 seconds.
- mtdMaxRedirectJumps : The maximum number of HTTP redirects to follow. Default is set to 4.
- mtdPageSizeLimitKB : The page size limit in kilobytes for the MTD. An attempt to download a page larger than the limit this will usually lead to only the first bytes with the size of the limit being download. A value of -1 here will lead to no limit being used. Default is 400 KB.
- scoreHalfTimeHours : The score for a query is depending on the time since last time it was used, making it theoretically possible for the agent to ask the same query twice, if enough time has passed since the query was used the first time. The default is set to 1200 hours, or 50 days. This might be a little high. It have however been measure as the typical halftime of the web.[Cho 01].
- scorekUrl : This is the weight setting for score-computation using the parts of the URL only.
- scorekTitle : Weight setting for the title words. In the default settings, this is set higher than the URL- and abstract-parts, since there is less of this material available.

- scorekAbstract : Weight setting for the abstract of URLs. The abstract in this context is the link text for the URL. The sum of the three weights for score computation should be equal to one. The default setting is 0.25, 0.5 and 0.25, in top to bottom order.
- relPathPollerAnalyser : The Boss-Agent needs the relative path to the PollerAnalyser. The default setting of ..\..\..\OGCPollerAnalyser\bin\Debug should not be changed unless the poller-analyser or the boss-agent is moved.

Wire-seed-extractor

- bossAgentKBRelativePath : This is the base-part of the relative path to the knowledge-base of the boss-agent. I does not usually need to be changed, unless the name of the knowledge-base is changed in the boss-agent settings, to anything other than KB.
- bossAgentID : This should be changed to the correct value for every different boss-agent run. The value should be equal to the idNr setting of the boss-agent.
- minimumGroupSize : This is the minimum size of a seed set specified. Default is 100.
- maxNumberOfGroups : The maximum number of crawl-groups. Default is 20.
- wireIdString : The id-string for the all crawl-groups generated in the run.
- SA1RandomSelection : A boolean for determining whether to use random selection of URLs from ranked hosts when selecting which URLs to use. If False, the shortest URLs will be used. Default is False.

Wire-crawl-system-starter

- message : By default a message saying that this module is not yet implemented.

Wire-results-parser

- wireReposRelPath : The relative path to the wire-repository. I.e the path to the folder which contains the results from a wire-crawl.
- wireAnalysisRelPath : The relative path to the folder for the analysis result. This is the folder where the results from the parsing will end up.
- wireBaseId : The base id for the wire-crawl to analyse. All files associated with this wire-id will be parsed and analysed.
- startNumber : The group number at which to start the parsing. Default is 1, which should be the first group.
- endNumber : The group number at which to end the parsing. Default is 10.

- processNumber : The process-number. This is the number of the process-queue used, when several wire-instances have been run simultaneously on the same computer. The default value of -1, means that wire was used in single instance mode.
- threadCountParser : The number of threads to use for the parser.
- maxDepth : The maximum depth to analyse. This can speed up the parsing process, in case it is set to a value of less than the actual crawled depth. The default value is 3.
- verifyPageCount : A boolean determining if the count of pages downloaded should be verified. Default is True.
- excludeRobotsPagesInCount : This excludes all robots.txt files from the page count. Default is True.
- makeDocNameMap : Setting this boolean to True enables generation of a document-name map that is saved to disk. I.e document-id and URL pairs are saved in a list. This is not really that useful information unless one is debugging this module. Default is therefore set to False.
- summaryFilename : This is the summary file-name, which will be used as a prefix for the summary-files generated by this module.
- makeTimeAnalysis : This is a boolean for setting for whether a time-analysis should be made or not. The time analysis is actually a simple summary of the crawl-times at different steps for each crawl-group.
- formatForExcel : A boolean determining if the output should be formated for excel. Default is True.
- excelOutputToFile : Setting this parameter to true will give files for each of the excel-results. False is default.
- makeSummaryAnalysis : Setting this boolean to true will lead to a generation of a summary analysis. This summary is a summary of all URLs found by the crawls. The summary is made incremental per crawl-depth. False is default.
- computeDifferanceSummaryAnalysis : A boolean setting for whether the difference value should be used instead of the summary value between two levels, in incremental summary analysis. This setting will affect nothing unless makeSummaryAnalysis is set to True. Default is False.

OGC-poller

- urlsFileName : This is the relative path to the file containing the URLs to poll. The default value is simply inputUrls.txt.
- latestVersionOnly : Setting this to true will only keep the OGC-URLs to getcapabilities-requests which have the highest value for the version parameter. Default is False.

- doExcludeUrls : Set this to enable exclusion of some URLs from the input. Default is False.
- downloadRelPath : This is the relative path to the download folder. If it does not exist it will be created. If it does exist, files in it will be overwritten. Default is XMLs.
- siteUrlFormatInput : A boolean setting for if the input URLs are in site-URL format. Site-URL is a format which not only holds an URL, bu also another URL which points to the site where the first URL was found. Default is False.
- saveMetadateToFile : A boolean setting for if the meta-data-content should be saved to file. This is required for the poller-analyser-module, since it analyses this data. Default is True.
- verbose : Verbose output. Default is False.
- spiderCount : The amount of independent download-queues with their own threads. Default is 100.
- restartFrequency : The restart-frequency. The higher the more often spiders will restart. Some spiders will restart when $\frac{C}{F}$ spiders are finished. For instance using the default values, there will be a restart once $\frac{100}{5} = 20$ spiders are done. The restart will concern at least 20 spiders. Default value is 5.
- excludeUrlsFilename : The name of the URL-file containing URLs which should be excluded from poll. The use of this function requires that the doExcludeUrls parameter is set to True. Default value is excludeUrls.txt.
- milliSecDownloadCheckInterval : The spider (download-queue-handler) manager will make checks on all spiders with a certain interval. This is set to 10000 milliseconds by default.
- spiderTimoutSeconds : The timeout for download finishing for spiders. This should be set to higher value than then downloaderTimeoutSeconds-setting, else both will be set to the lower value. The default is 40.
- maxRedirects : The maximum number of redirect jump starting with a specified URL in the queue. Set to 4 by default.
- downloaderTimeoutSeconds : The timeout for TCP-socket connections in seconds for spiders. Is set to 30 by default.
- KBpageSizeLimitDownload : The size limit for pages downloaded in kilobytes. This will not be enforced by all servers, so larger pages may be downloaded anyway. The default value is 400.

OGC-poller-analyser

- filenameCaseList : The name of the case-list file. This file contains the list of all cases that should be analysed. This is either all wire or all boss -cases. The default name is auto_case_list.txt.
- pollerDBRelPath : This is the relative path to the download folder from the OGC-poller. Is default set to ..\..\..\OGCPoller\bin\Debug\xmls
- processXmls : A boolean for determining whether XMLs should be processed or not. This should only be set to false if the poller-analyser already have been run once before on the same poller-database. Default is True.
- xmlProcessedRelPath : This is the path to the processed XML-files. Default value is xmlp.
- baseStatisticsFilename : This is the filename of the base-statistics file. Default value is baseStatsPoll.txt.
- moveduplicates : This is a boolean determining whether duplicates should be treated as error-files and moved to the errors-folder. Default is True.
- useDuplicatesList : This setting enables the use of the duplicate-list, which will lead to more accurate results in the statistics. It is recommended to have this set to True. Default is True.
- bossAgentCases : A boolean for setting to determine if the case-list contains boss-agent cases or wire cases. Default is False.
- strictAvailabilltyCounting : Setting this parameter to true will make the availability counting strict, which means that duplicates are not counted. Default value is set to true.
- wireMaxDepthForStatistics : This is the maximum depth to use for wire-cases. Default is 3.
- makeHashUrlMap : This boolean enables the construction of hash-to URL maps. Since all XML-files are saved using the Md5-hash of their URLs, there is a possibility to make a hash-URL map so that one may lookup which URL belongs to which downloaded document. The default value is True.
- simpleMode : In simple-mode the URL-files used are fetched directly from a files, instead of going through multiple files for boss-agent or wire -cases. The default value for this is false.
- simpleUrlFile : This is the path to the URL-file list to use in simple mode. The default value is: ..\..\..\OGCPoller\bin\Debug\inputUrls.txt
- verboseMode : Verbose text output. Default is False.

A.5 Installation

There is an installation program available. To install OSS, compile all content of the src-folder using Microsoft Visual Studio 2010 or similar CASE-software. Then run the Installer application from command-prompt and enter the installation path as first argument and the path to the project (path to src-folder) as second argument. Do not change the structure of the files or folders in OSS, else the modules may stop working. Instead of compiling, one may use the files in the bin-folder directly. It should contain the latest version of the project.

Appendix B

System documentation

Requirements document

B.1 Requirements

Since the focus of the thesis-work is not on software-development, the product developed is to be considered as a simple prototype.
Furthermore, it was an optional goal in the time-plan to develop a prototype. Because of this, the requirements-specification is not that important and is quite simple.

B.1.1 Introduction

The system to be developed shall be a simple prototype. It should be a type of automated agent, which purpose is to find as many available OGC-services as possible within a certain time-frame.

B.1.2 Purposes

The purpose of the prototype is to demonstrate the possibility of automated discovery of new OGC-services. This means that the prototype should be thought of as a proof of concept application, without any guarantee for fail-proof execution. I.e the system may crash, hang up or sometimes deliver slightly erroneous results.

B.1.3 System Overview

Once started, the prototype system shall be capable of running by itself, without the need for any external input. The output should be a list of OGC-service-URLs.

B.1.4 Product Functions

The main function given by the prototype is, given a set of start queries for OGC-services to a search-engine, the output should be a list of fairly unique get-capabilities-request-URLs to OGC-service.
This output should be generated in increments, and thus there will be some result given even if the prototype run is aborted before it is finished.

The run time should be specified prior to launch of the agent, to prevent it from running forever.

B.1.5 Non functional requirements

A list of discovered OGC-links should be saved to file. This list of links discovered should be relatively free from duplicates, where at least half of the discovered links should be non-duplicates.
The security of the system should be ensured through proper use of the openAuth-standard for verification.
The performance of the system should be better than the performance of a non-automated version of this task, where a person connects to a search-engine and downloads links manually. This performance is however hard to estimate. Since no estimations on manual discovery-speed using a search engine have been done, this requirement is not that strict.

Sys. spec

B.2 System specification

The system consists of 9 different modules. The modules are of three types: Runtime libraries, Process modules and the module-handler executable.

There are two runtime libraries, HTMLparserLibDotNet20.dll and OSSLibstd.dll. The first one is from an open-source-project called Majestic12Parser [M12] and the second was developed during this master thesis work. Both are used by all of the six process modules.

The module-handler is capable of executing the six-process modules in the right order, using the proper configuration for them.

All of the process-modules may be used by themselves, but if more than one should be used, the module-handler is capable of a controlled execution of the modules needed, in the right order.

B.2.1 Input files

There are a number of files which control the input for the modules. These files are mostly standard configuration-files with key-value pairs.

Configuration-files are found at paths:

- OGC-Search-System (module handler): OGCSearchSystem\bin\Debug\config.txt
- Boss-Agent: BossAgent\bin\Debug\config.txt
- Wire-Seed-Extractor: WireSeedExtractor\bin\Debug\config.txt
- Wire-Crawl-System-Starter: WireCrawlSystemStarter\bin\Debug\config.txt
- Wire-Results-Parser: WireResultsParser\bin\Debug\config.txt
- OGC-Poller: OGCPoller\bin\Debug\config.txt

- OGC-Poller-Analyser: OGCPollerAnalyser\bin\Debug\config.txt

Other input files:

- Boss-Agent initial queries:
  BossAgent\bin\Debug\initQueries.txt
- Default OGC-Poller-Analyser bosswire case-file:
  OGCPollerAnalyser\bin\Debug\auto_case_list.txt
- Default OGC-Poller URL-input-file:
  OGCPoller\bin\Debug\inputUrls.txt

B.2.2 Output files

The results from all modules are written to specific output-files. Their path and name usually depends on the settings of the module's configuration-file.
Below are the default names and paths to certain results-files:

- Boss-Agent results:
  - Found OGC-URLs, BOSS-results, OGC-URL count per page:
    BossAgent\bin\Debug\kb_r51\ogc*
  - Result count for single queries:
    BossAgent\bin\Debug\kb_r51\res_dt.txt
  - Downloaded pages with date for download:
    BossAgent\bin\Debug\kb_r51\dlPages-dt_Manager.txt
  - Downloaded queries with numbering for each step:
    BossAgent\bin\Debug\kb_r51\dlQueries_Manager.txt
  - OGC-URLs found, non-formated:
    BossAgent\bin\Debug\kb_r51\ogc_urls.txt
  - Count of OGC-URLs found per step:
    BossAgent\bin\Debug\kb_r51\ogc_urls_gc-req_cnt.txt
- Wire-Seed-Extractor results:
  - Concatenated seed-set-file:
    WireSeedExtractor\bin\Debug\baSeedSet.xml
  - Wire-seed folders:
    WireSeedExtractor\bin\Debug\cxf*

B.3 Detailed system specification

Since the focus of this thesis was not set on the prototype development, this part is not as rich of content as it could be.

B.3.1 Module functionality

This is the main functionality of each module that is a part of the OGC-Search-System:

1. Boss-Agent
    - Automatic selection of queries for BOSS, based on result downloaded and initial queries used.
    - Downloading BOSS-results in XML-format
    - Download of pages listed in BOSS-result
    - Parsing of OGC-links on downloaded pages
    - Tracking of which OGC-links were found at which step
    - Tracking of number of OGC-links found per page
2. Wire-Seed-Extractor
    - Extract of best ranked seeds to use from downloaded pages
    - Partitioning of seeds on crawl-groups
    - Selection of top-level-domains to use in each crawl.
3. Wire-Crawl-System-Starter
    - Not implemented in this thesis work.
4. Wire-Results-Parser
    - Extraction of URLs for OGC-links and their parent-links from downloaded HTML-data
    - Counting of downloaded pages at each crawl depth
    - Counting of OGC-links found at each crawl depth
    - Counting of OGC-links found at each crawl-group
    - Counting of OGC-listing-pages found at each depth
5. OGC-Poller
    - Parallel download of given set of OGC-links
    - Crude initial splitting of data downloaded on OGC-service-type, based on the OGC-link
6. OGC-Poller-Analyser
    - Conversion of raw HTTP-data to XML-data.
    - Grouping of OGC-data on service-type, based on the data-content
    - Fixing of slight XML-errors, like unescaped special characters.
    - Removal of non-parse-able XML-data by xpath, and grouping by error type.
    - Removal of duplicate downloaded services, and keeping statistics of duplicates.

- Construction of URL to filename index-file.
- Listing of all files moved from their original grouping by OGC-Poller, including error files
- Counting of available OGC-services found at each step for the BOSS-Agent or the WIRE-crawler

B.3.2 Data-structures

The data-structures used are hash-maps, tree-maps, linked lists, array lists, arrays and hash-sets from the .Net standard library.

B.3.3 Format

The formats for files used by the modules in this project is ASCII, binary or XML. The file extension reveals the type. I.e ".txt" is ASCII, ".bin" is binary and ".xml" is ASCII-encoded XML.

B.3.4 Algorithms

Some algorithms are far from straightforward and needs a better explanation. These algorithms are listed here, along with their explanation.

S_{A1}: Algorithm for generating the n best seeds for WIRE

Having the base set of seed-URLs S_b downloaded from BOSS, saved with their ranking value r_k for every URL k. The algorithm works like follows: Part 1:
For each URL u in S_b:

1. Extract the host h_u for u
2. Add the score for u , $s_u = 1/r_u$ to the value in the map $m_s[h_u]$
   If there is no value for $m_s[h_u]$, set it to s_u
3. Add u to the set $m_h[h_u]$

Part 2:
Select the n hosts with the highest values in m_s as H For each host h_i in H:

1. Select the URL u in $m_h[h_u]$ which has the lowest path-length.
2. Add u to set U

Return U.

This algorithm starts by grouping the URLs by hosts, this is a performance choice. WIRE only makes one connection at the time per host, so it does not make that much sense to have more than one seed belonging to the same host.
A host with many URLs found by BOSS should be an important one, so with this algorithm, the score for a host will increase with the score of each of the URLs found on it.

Causing these hosts to get a larger score than hosts with few URLs.
The URL with lowest path-length is selected in the end. This is just a choice made for making the algorithm deterministic, another option is to select randomly among the links for a chosen host. WIRE was actually configured to also add the root-page to every seed as a link to follow, so random selection might have been a performance-wise better choice in this case.

S_{Aq}: Algorithm for generating subset of queries for BOSS re-fetch

The S_{Aq} algorithm consists of a probabilistic algorithm S_{Aqp}, which was is set to run 300 times at default, and then the run with fewest queries is selected by S_{Aq}. In every run of S_{Aqp}, elements are examined in a random order, which is reshuffled by S_{Aq} once every 30th run by default. Using default settings, a total of 10 shuffles are thus used.
In S_{Aqp}, for every element (URL), it results in the selection of one set, which covers the element, if it is not already covered. The probability of the selection of a set, is proportional to the size of the set.
This algorithm can be considered as a greedy probabilistic algorithm, since it follows a simple rule. Its performance has been proved to be sufficient for the task, after several tests have been performed.

B.4 Methods Reference

Each module contains numerous methods. Because of this, only the non-internal methods of the dll-library OSSLibStd, which is used by all the modules, is being described in this chapter. This documentation have been generated using doxygen and consists of 134 pages. It is available at: http:// ?.?.?

B.5 Left to Implement

One module is left to implement, this is the third module, called Wire-Crawl-System-Starter. It is supposed to be a communication and monitoring module.
Its purpose is to move the extracted seeds from module 2 to one or more WIRE-enabled computer(s) and start the crawling. Then it should monitor the crawl, and determine whether or not to continue the crawl at each finished level, based on the result at that level.
When a crawl-group is finished the module should transfer the result back to the computer running OSS, so that the result may be parsed by module 4. This way the parsing of results will not take that much extra time.

Appendix C

Development documentation

C.1 Source code

Source code is available at: ?

C.2 Log

Development log is available at:

C.3 Time plan

The initial time plan is as follows: The total time of 20 weeks is divided evenly on the two parts of research and development. That means 10 weeks of initial research followed by 10 weeks of agent development. The latter part may start earlier or later, depending on how the research progresses.
Time for planning and report writing is included in the time specified for the other parts of the thesis-work. I estimate this to be approximately 20% of all time, that is in total 4 weeks of documentation of the work. Furthermore, the time-plan for the research part is set in this way: The first 2 to 4 weeks will focus on the identification of mainly WMS-services. The detection of WFS- and WCS- services will probably be carried out in a similar way, so it will suffice to focus on WMS initially.
The later 6 to 8 weeks of the first part will be spent upon research of web-crawler technology and testing of various algorithms utilizing this technology.

The time-plan for the second part of the work is more difficult to set up, and it should follow from the research-step, which parts of the agent are more time-demanding to develop.
Hence an initial time-plan for the second part will be somewhat inaccurate. The time-plan for the second part does not exclude the usage of open-source code, which could shorten the development time significantly. This is also a matter for the research-step to discover.

Since it is not yet possible to determine what type of agent that will be developed in the second step. I assume that the design of the agent will be similar to that of a common web-crawler, since this is what seems most likely at the moment. Therefore, I build the time-planning for the agent development using the structure of a web-crawler.

The two most integral parts of a web-crawler is the Scheduler and the Multi-Threaded down-loader (MTD). The scheduler determines in which order URL:s will be visited, using some kind of ranking algorithm. The Multi-threaded down-loader, downloads all the pages and resources given by the scheduler, and stores new URL:s for the scheduler in a queue.
In this preliminary time-plan, 5 weeks are set to be spent on work with the scheduler and 4 weeks are assigned for the MTD. The remaining one week is to be spent on testing and refinement of the prototype.

For the Scheduler:

1. Generation of initial URL:s, perhaps by using meta-search URL:s for search engines like Google or yahoo!-search: 1 week.

2. Development of Pagerank or similar algorithm, for determining the selection policy: 2 weeks.

3. Development of politeness policy, i.e. avoiding overloading servers: 1 week.

4. Development of revisit policy, i.e. setting a suitable revisit-frequency for each page or resource, when searching for updates: 1 week.

For the MT-down-loader:

1. Identification of resource URL:s for VMS/WFS/WCS -services as well as page URL:s such as HTM,HTML, jsp, asp -pages etc. : 2 weeks.

2. Multi-threaded download of resources into database and pages into queue: 2 weeks.

Note that the scheduler must be developed in conjunction with the MTD, else it will be difficult to test either of them. This means that only the total required time has been specified in this document, not the order of development.
For each of the two modules of the agent, the order of development is as listed from top to bottom. This is of not a strict order and it may be subject to change later.

References

[Cho 01] Cho, Junghoo.: *Crawling the Web: Discovery and Maintenance of a Large-Scale Web Data*, Ph.D. dissertation, Department of Computer Science, Stanford University, November 2001, http://oak.cs.ucla.edu/~cho/papers/cho-thesis.pdf

[N3 01] Filippo Menczer, Gautam Pant, Padmini Srinivasan: *Evaluating Topic-Driven Web Crawlers*, Chapter in SIGIR 2001 pp. 241-249, School of Library and Information Science, The University of Iowa, September 2001, http://www.n3labs.com/pdf/focusreview.pdf

[Car 01] Carlos Castillo: *Effective Web Crawling*, Ph.D. dissertation, Department of Computer Science, University of Chile, November 2004, http://www.chato.cl/papers/crawling_thesis/effective_web_crawling.pdf

[Now 01] Cristopher Olston, Mark Najork: *Web Crawling*, Survey extract of Foundations and Trends in Information Retrieval vol. 4. no. 3, pp. 175-246, NOW Publishers, 2010, http://research.microsoft.com/pubs/121136/1500000017.pdf

[WMS 1.3] Open Geospatial Consortium Inc.:
OpenGIS Web Map Server Implementation Specification,
version 1.3.0, Standards specification, OGC, Mars 2006, http://portal.opengeospatial.org/files/?artifact_id=14416

[WMS 1.0] Open Geospatial Consortium Inc.:
OpenGIS Web Map Server Implementation Specification,
version 1.0.0, Standards specification, OGC, April 2000, http://portal.opengeospatial.org/files/?artifact_id=7196

[WCS 2.0] Open Geospatial Consortium Inc.:
OGC WCS 2.0 Interface Standard - Core,
version 2.0.0, Standards specification, OGC, October 2010, http://portal.opengeospatial.org/files/?artifact_id=41437

[WCS 1.0] Open Geospatial Consortium Inc.:
Web Coverage Service (WCS) ,
version 1.0.0, Standards specification, OGC, August 2003, http://portal.opengeospatial.org/files/?artifact_id=3837

[WFS 2.0] Open Geospatial Consortium Inc.:
OpenGIS Web Feature Service 2.0 Interface Standard ,

version 2.0.0, Standards specification, OGC, November 2010, http://portal.opengeospatial.org/files/?artifact_id=39967

[WFS 1.0] Open Geospatial Consortium Inc.:
Web Feature Service Implementation Specification ,
version 1.0.0, Standards specification, OGC, May 2002, http://portal.opengeospatial.org/files/?artifact_id=7176

[OGC 1] Open Geospatial Consortium Inc.:
Web Map Service, Downloaded July 27th, Web page, OGC, July 2011, http://www.opengeospatial.org/standards/wms

[OGC 2] Open Geospatial Consortium Inc.:
FAQs - OGC's Purpose and Structure, Downloaded August 2th, OGC, June 2010, http://www.opengeospatial.org/ogc/faq

[MapServer] MapServer
MapServer - open source web mapping, Downloaded November 28th, Web page, Stephen Lime, July 2011, http://mapserver.org/

[Wiki 01] Wikipedia Foundation:
Geospatial analysis , Downloaded April 19th, Collaborative encyclopedia, Wikipedia, April 2011, http://en.wikipedia.org/wiki/Geospatial

[Wiki 02] Wikipedia Foundation:
Open Geospatial Consortium , Downloaded April 19th, Collaborative encyclopedia, Wikipedia, April 2011, http://en.wikipedia.org/wiki/Open_Geospatial_Consortium

[Wiki 03] Wikipedia Foundation:
Geographic information system , Downloaded April 19th, Collaborative encyclopedia, Wikipedia, April 2011, http://en.wikipedia.org/wiki/Geographic_information_system

[Wiki 04] Wikipedia Foundation:
Web Map Service , Downloaded August 2th, Collaborative encyclopedia, Wikipedia, July 2011, http://en.wikipedia.org/wiki/Web_Map_Service

[Set Cover] Wikipedia Foundation:
Set Cover Problem , Downloaded October 14th, Collaborative encyclopedia, Wikipedia, August 2011, http://en.wikipedia.org/wiki/Set_cover_problem

[CAPTCHA] Wikipedia Foundation:
CAPTCHA , Downloaded December 13th, Collaborative encyclopedia, Wikipedia, December 2011, http://en.wikipedia.org/wiki/CAPTCHA

[Pearson] Wikipedia Foundation:
Pearson product-moment correlation coefficient , Downloaded December 13th, Collaborative encyclopedia, Wikipedia,

November 2011, http://en.wikipedia.org/wiki/Pearson_product-moment_correlation_coefficient

[PageRank] Wikipedia Foundation:
PageRank , Downloaded December 13th, Collaborative encyclopedia, Wikipedia, December 2011, http://en.wikipedia.org/wiki/PageRank

[IETF 1] The Internet Engineering Task Force:
Uniform Resource Identifier (URI): Generic Syntax, Downloaded April 19th, RFC, IETF, January 2005, http://www.ietf.org/rfc/rfc3986.txt

[SkyLabs 01] SkyLAB Mobilesystems Ltd.:
OGC WMS Server List , Downloaded April 7th, Web-page, Skylabs, April 2011, http://www.skylab-mobilesystems.com/en/wms_serverlist.html

[SkyLabs 02] SkyLAB Mobilesystems Ltd.:
OGC WMS Server List , Downloaded April 8th, Web-page, Skylabs, April 2011, http://www.ogc-services.net

[Google 01] Google Inc.:
Google Terms of Service , Downloaded July 25th, Web-page, Google, July 2011, http://www.google.com/accounts/TOS

[WMS Mining 02] Chris Tweedie: *WMS Service Mining*, Downloaded April 7th, Blog, Chris Tweedie, Jan 2006, http://blog.webmapper.com.au/2006/01/19/wms-service-mining

[Yahoo 01] Yahoo! Inc.:
Yahoo! Search BOSS , Downloaded July 27th, Web-page, Yahoo!, July 2011, http://developer.yahoo.com/search/boss

[Yahoo 02] Yahoo! Inc.:
Yahoo! Search BOSS - Pricing, Downloaded July 27th, Web-page, Yahoo!, April 2011, http://info.yahoo.com/legal/us/yahoo/search/bosspricing/details.html

[WWWS 01] Maurice de Kunder:
The size of the World Wide Web, Downloaded July 25th, Web-page, De Kunder Internet Media, July 2011, http://www.worldwidewebsize.com

[WIRE 01] Carlos Castillo:
WIRE - Web Information Retrieval Environment, Downloaded July 27th, Web-page, Department of Computer Sciences - University of Chile, May 2011, http://www.cwr.cl/projects/WIRE

[M12] Alex Chudnovsky:
Majestic-12 : Projects : C# HTML parser (.NET), Downloaded June 7th, Web-page, Majestic-12 Ltd, June 2011, http://www.majestic12.co.uk/projects/html_parser.php

[OAuth] Google Code:
oauth - Revision 1263: /code/csharp, Downloaded April 14th, Web-page, Google inc., October 2011, http://oauth.googlecode.com/svn/code/csharp/