A domain-independent architecture for efficient information retrieval on the World Wide Web

Master Thesis (Diploma)
Faculty of Technology
University of Bielefeld

presented by
Marc Langheinrich[1]

Advisors

Prof. Dr. Ipke Wachsmuth[2]
Faculty of Technology, AG WBS, University of Bielefeld
Universitaetsstr. 25, 33615 Bielefeld, Germany

Prof. Oren Etzioni[3]
Department of Computer Science, University of Washington
Box 352350, Seattle, WA 98195, USA

[1]marclang@cs.washington.edu
[2]ipke@techfak.uni-bielefeld.de
[3]etzioni@cs.washington.edu

Ich versichere, daß ich die vorliegende Arbeit eigenständig angefertigt habe und mich keiner anderen als der ausdrücklich angegebenen Hilfsmittel und Quellen bedient habe.

Bielefeld, den 29. August 1997

University of Bielefeld

Abstract

A domain-independent architecture for efficient information retrieval on the World Wide Web

by Marc Langheinrich

Advisors

Professor Ipke Wachsmuth,
University of Bielefeld, Faculty of Technology

Professor Oren Etzioni,
University of Washington, Computer Science & Engineering

The World Wide Web's rapid growth in recent years has provided a wealth of online information. Including already more than hundred million documents, finding a particular page has become a daunting task of battling the Web's "information overload".

Most popular methods of finding information on the Web are known for being either notoriously imprecise or often incomplete: simple searches can easily return hundreds or thousands of irrelevant pages, while others might fail to include even a single relevant one.

This thesis presents a novel architecture called "Dynamic Reference Sifting", which attempts to combine the comprehensiveness of Web indices, such as AltaVista or HotBot, with the accuracy of Web directories, such as Yahoo!. Dynamic Reference Sifting uses the output of general purpose search services, combined with additional, orthogonal information sources; domain specific heuristics; and a flexible categorization scheme to filter out all but the single correct page.

Our experiments show that for certain types of pages, this approach can provide nearly twice the accuracy and at least the same coverage as any existing service. We have implemented a prototype called "Ahoy! The Homepage Finder", which demonstrates the feasibility of our approach. Ahoy! is publicly accessible on the Web, and has served more than 500,000 queries since it was fielded in May 1996.

In order to demonstrate the domain independence and generality of our architecture, we will also present two simple prototypes using Dynamic Reference Sifting in the domains of academic papers and jokes. Both systems were developed and implemented in less than ten days, but prove highly successful in our initial experiments.

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Thesis . . . 2
- 1.3 Methodology & Contributions . . . 3
- 1.4 Related Work . . . 4
- 1.5 Thesis Overview . . . 7
- 1.6 Acknowledgements . . . 8

2 Softbots and the Information Food Chain **9**
- 2.1 Agents, Softbots and the Internet . . . 9
  - 2.1.1 The Intelligent Agent paradigm . . . 9
  - 2.1.2 Rodney, the Internet Softbot . . . 10
- 2.2 Finding Information on the Web . . . 12
  - 2.2.1 Introduction . . . 13
  - 2.2.2 Web Directories . . . 14
  - 2.2.3 Web Indices . . . 15
  - 2.2.4 Manual Search . . . 18
  - 2.2.5 Web Agents . . . 19

3 Dynamic Reference Sifters **23**
- 3.1 Introduction . . . 23
  - 3.1.1 Design Goals . . . 24
  - 3.1.2 DRS Set Characteristics . . . 25
- 3.2 DRS Architecture . . . 27
  - 3.2.1 Information sources . . . 30
  - 3.2.2 Filtering . . . 31
  - 3.2.3 Learning . . . 34
  - 3.2.4 Sample domain implementations . . . 38
- 3.3 Summary & Discussion . . . 38

4 Ahoy! The Homepage Finder **43**
- 4.1 Finding personal homepages on the Web . . . 43
  - 4.1.1 Domain description . . . 43
  - 4.1.2 Current Methods . . . 44

4.1.3 Using Dynamic Reference Sifters . . . 45
4.2 The Ahoy! Web Service . . . 48
4.2.1 History and Overview . . . 48
4.2.2 Ahoy! Input: Searches and sessions . . . 52
4.2.3 Information Sources . . . 54
4.2.4 Filtering: Heuristic Analysis & Classification . . . 55
4.2.5 Ahoy! Output: The Display Manager . . . 59
4.2.6 Learning . . . 61
4.2.7 Miscellaneous . . . 64
4.3 Evaluating Ahoy . . . 65
4.3.1 IR Performance Measures . . . 66
4.3.2 Experimental Setup . . . 70
4.3.3 Results . . . 73
4.3.4 Discussion . . . 74
4.3.5 The effect of Hypothesis Statistics . . . 77

5 Beyond Ahoy: Using DRS in other domains 81
5.1 Introduction . . . 81
5.2 Academic Papers . . . 81
5.2.1 Description of the Paper Domain . . . 82
5.2.2 Paper Mate: A Case study . . . 83
5.2.3 Experiments . . . 87
5.3 Jokes . . . 92
5.3.1 Description of the Joke domain . . . 92
5.3.2 Joker!: A Case study . . . 94
5.3.3 Experiments . . . 96
5.4 Summary & Conclusions . . . 97

6 Conclusions and Future Work 99
6.1 Summary & Conclusions . . . 99
6.2 Future Work . . . 100
6.2.1 Extending the functionality of existing DRS-Systems . . . 100
6.2.2 Extending the generic DRS framework . . . 101

Bibliography 103

A Glossary 113

B The Ahoy! System: Maintenance & Troubleshooting 121
B.1 Implementation Details . . . 121
B.1.1 Main scripts: searching, guessing and following . . . 121
B.1.2 Managing multiple server . . . 122
B.2 File List . . . 122
B.3 On-line documentation . . . 125

B.3.1 General Information 125
B.3.2 Ahoy! Directory Guide 126
B.3.3 The Ahoy! machine cluster 131
B.3.4 How to make changes to the source code 132
B.3.5 Web Statistics using htstats 134
B.3.6 MetaCrawler Interface 136
B.3.7 List of Modules 138
B.3.8 Ahoy! Server Troubleshooting 141
B.3.9 Howto: Count hypotheses 147
B.3.10 Howto: Maintain the Institutional DB 147
B.3.11 Howto: Block specific sites from using Ahoy! 149
B.3.12 Howto: install an edit version of Ahoy! 152
B.3.13 Howto: install Ahoy! in a new directory 154
B.3.14 Howto: Add a new machine to the Ahoy! cluster 157
B.3.15 Howto: Install required third-party programs for Ahoy! 159
B.3.16 Howto: fix Ahoy! bugs 160
B.4 Embedded Source Code Documentation 163

List of Figures

2.1 WWW Hosts Growth 13
2.2 Generic Web Index Architecture 16
2.3 The Information Food Chain 21
2.4 The Softbot Family Tree 22

3.1 DRS Control Flow 29
3.2 DRS Table 33
3.3 URL Extractor 36
3.4 URL Generator 37

4.1 Ahoy! Search Form 45
4.2 Ahoy! Status Report 46
4.3 Ahoy! Results 47
4.4 Ahoy! Module Overview 51
4.5 Ahoy! Cross Filtering 57
4.6 Ahoy! Table 61
4.7 Ahoy! Zones 62
4.8 Precision-Recall Graph 68
4.9 Precision-Recall Graph in DRS-domain 69
4.10 Performance Comparison 72
4.11 Restricting Ahoy!: *precision′* Performance Comparison 75
4.12 Restricting Ahoy!: *recall′* Performance Comparison 76

5.1 Paper Mate Search Form 84
5.2 Paper Mate Results Screen 85
5.3 Paper Mate Base Reference Set Queries 85
5.4 Paper Mate Control Flow 86
5.5 Paper Mate Experimental Results 90
5.6 Joker! Search Form 93
5.7 Joker! Results Screen 94
5.8 Joker! Control Flow 95

B.1 Embedded Documentation Sample 164

List of Tables

2.1 Characteristics of Information Agents 11
2.2 Top Level Hierarchies of the Yahoo! Web Directory 15
2.3 Web Indices Index Size 17
2.4 Popular Web Search Methods 20

3.1 DRS Design Goals 24
3.2 DRS Set Characteristics 28
3.3 Implementation Samples 39
3.4 Implementation Samples (continued) 40

4.1 Initial Architectural Approaches 49
4.2 Ahoy! Modules by Category 52
4.3 Contents of Ahoy! Session Directory 53
4.4 Ahoy! Analyzer: Ownership Codes 56
4.5 Ahoy! Analyzer: Page Locations 58
4.6 Ahoy! Analyzer: Page types 58
4.7 Ahoy! Zones 60
4.8 Ahoy! Placeholders 63
4.9 Qualitative Performance Comparison 67
4.10 Average Rank of Targets 71
4.11 Cumulative Results 74
4.12 Successful Hypothesis Application by Site 78
4.13 Sample Searches 79

5.1 Paper Mate Results per Article 89
5.2 Paper Mate Results of Follow-up Experiment 91

1 Introduction

1.1 Motivation: Searching the Information Superhighway

The World Wide Web is growing at an exponential rate. Every day a few hundred new servers are hooked up to the Internet, each providing a wealth of information, ranging from merely a dozen to up to a hundred pages. Various search tools currently fielded on the Web are indispensable for finding information about general topics on these servers.

Web directories like Yahoo! or Lycos' A2Z constitute a convenient way to search the vast amount of topics covered by the Web's online information. They provide a great number of links categorized into topics and subtopics that allow the user to focus her search on only a subset of the available pages (*e.g.*, search only pages describing Internet providers in Washington State), thus being able to achieve both high accuracy and coverage in certain well defined categories. However, Web-Directories have a major drawback when it comes to coverage: in order to provide this methodological (typed) approach, all pages that can be found within these directories need to be registered and (manually) categorized first. Consequently, they lack sufficient coverage in all but a fair number of small areas, by only providing the subset of pages that have been registered with their service. With more than 100 million pages currently on the Web,[1] even the largest directories like Yahoo! or Lycos' A2Z already feature only a fraction of all available pages,[2] forcing the user to query automatically generated indices such as AltaVista or HotBot.

These so called *Web indices* typically allow boolean keyword searches that work regardless of the page or topic type the user is searching for, usually providing the most up-to-date information available. However, for any given query, the user is typically confronted with a huge number of references (hits) that the search engines deem to be of relevance – a number that, especially for simple queries, can easily reach hundreds, if not thousands. This imprecision can be described as "information overload" [BM85], where the sheer amount of references returned makes it hard to find the right answer among them. Often the user would be forced to manually look

[1] as of July 1997, the Lycos search service features over 100 million pages. However, industry estimates predict the real number of available documents on the Web to be close to half a billion pages. [where, where did I find this?]

[2] [Yan97] reports about one million pages for the Internet's largest directory, Yahoo!

at every reference until the desired answer can be found – an approach that fewer and fewer users are willing to take.

But even such a laborious approach might not work. [SE95] demonstrates that automatically generated indices are not completely comprehensive for three reasons: First, each index has its own strategy for selecting which pages to include and which to ignore. Recent studies by [Mel97] show great discrepancies between services indexing pages at the same site, sometimes featuring less than half the number of indexed documents. Second, some time passes before recently minted pages are pointed to and subsequently indexed, ranging from bi-weekly [Hot97] to monthly [Mon97] updates. Third, as the Web continues to grow, automatic indexers begin to reach their resource limitations.

In summary, users looking for specific information face a daunting task: Either they try using Web Directories, which feature reasonable accuracy at the expense of coverage in most areas, or they must manually search through hundreds of "relevant" hits returned from a standard Web Index. However, even after looking through *all* returned references of a Web Index and failing to find the desired information, they can not be sure that the desired information is indeed not on the Web – it might simply not be indexed by the particular service they chose, or they might have overlooked the single relevant reference among hundreds of irrelevant ones.

1.2 Thesis

Some users have already accepted these limitations as inherent to the Web: When searching for a specific page, they expect to manually browse through multiple answer pages, or even issue multiple queries, both at the same site (for refining the query) and at different sites (for increased coverage).

In this work, we propose a novel information retrieval architecture that is designed to address the above mentioned problems with standard search services. We call this architecture *Dynamic Reference Sifting* (DRS). It contains the following key elements:

1. **Reference source:** A comprehensive source of candidate references (e.g., a Web index such as AltaVista).

2. **Cross filter:** A component that filters candidate references based on information from a second, orthogonal information source (e.g., a database of e-mail addresses).

3. **Heuristic-based Filter:** A component that increases precision by analyzing the candidates' textual content using domain-specific heuristics.

4. **Buckets:** A component that categorizes candidate references into ranked and labeled buckets of matches and near misses.

5. **URL Generator:** A component that synthesizes candidate URLs when steps 1 through 4 fail to yield viable candidates.

6. **URL Pattern Extractor:** A mechanism for learning about the general patterns found in URLs based on previous, successful searches. The patterns are used by the URL Generator.

Using its *reference source*, a DRS-System first obtains a large number of possibly relevant references to the user's search request. Then, using a combination of *cross filters* and *heuristic-based filters*, it sorts all available references into a large number of ranked and labeled *buckets*. If one of the buckets that indicate a success has been filled, a *URL Pattern Extractor* extracts general descriptions of these relevant references. In case only suboptimal buckets could be filled, a *URL Generator* can then be used to synthesize candidate URLs from these patterns in order to search directly for the relevant pages.

This thesis attempts to support the hypothesis that for a certain class of pages, such a DRS-System can provide both high accuracy *and* large coverage, offering quality information from the Web. Members of this class share the following common characteristics, as first described in [SLE97]:

- **Availability:** Many, but not necessarily all, member pages are accessible via traditional search services.

- **Focused attention:** During a given search, the user is interested in a tightly bound subset of pages, typically a single one.

- **Strong cohesion:** Each page is easily identifiable as belonging to the class.

- **Large cardinality:** The number of members prevents manual indexing.

- **Widely dispersed:** No centralized site, like a central repository, allows easy access to the pages.

Current Web-Search methods (as described above) often provide no satisfying way of searching for members of these sets: either the returned references are outdated, incomprehensive, or imprecise. Specific tools based on the DRS architecture, taking the characteristics of these sets into account, can overcome the limitations imposed by current search methods and services and provide an effective retrieval mechanism in these domains.

1.3 Methodology & Contributions

In order to support the above hypothesis, this thesis describes a fully implemented system, called Ahoy! The Homepage Finder, that offers the aforementioned enhancements for a specific domain, the Personal Homepage domain. As with other class of

pages, current search methods have shown to be inadequate for this retrieval task (see section 4.3).

To further demonstrate the general usefulness of our approach, we continue to describe two simpler implementations in additional domains, Ahoy! Paper Mate in the Academic Paper domain and Joker! in the domain of on-line jokes. Both domains exhibit similar characteristics in terms of cardinality and distribution, which make it hard to find member pages with standard search services.

Specifically, our work contributes to the field of Information Retrieval on the World Wide Web by:

- *Demonstrating the feasibility of providing high accuracy without sacrificing coverage*, even in highly unstructured domains as the World Wide Web. The Ahoy! system achieves remarkable performance when compared to standard search services, showing more than twice the accuracy of its closest competitor, while still offering the largest coverage.

- *Developing a domain-independent architecture that works in a variety of domains*, enabling the rapid prototyping of simple, yet powerful systems. Using the generic DRS architecture, we are able to quickly construct two simple prototypes in different domains, both offering high accuracy combined with high coverage on the available test sets.

- *Proposing a novel method of resource location on the Web*, by learning about the structure underlying a specific domain. In one experiment, Ahoy! is able to find nine percent more references than its closest competitor by using its URL extraction and generation methods.

- *Examining effects of simple machine learning techniques on retrieval performance*. We compare three different strategies for resource ordering in Ahoy!'s URL learning module, and show how keeping simple statistics can greatly improve performance.

1.4 Related Work

With the advent of the World Wide Web as both a testbed for new technology and the focus of consumer interest, a large number of research projects are dealing with, or are closely related to, the task of information retrieval on the Web. The DRS framework proposed here combines a number of current research areas – Searching, Information Integration, Learning, and Web Agents – but offers an approach that tries both to be feasible using current technology, and to be truly useful for the everyday user. As [KSC94] puts it: one of the most difficult aspects of agent design.

Searching

With the ever increasing size of their databases, standard Web indices like AltaVista [DEC95a], HotBot [Hot96] and Infoseek [Inf96] have begun to offer advanced query syntax that can help increase accuracy. When looking for people, AltaVista's NEARness operator, HotBot's "Person search" or Infoseek Ultra's [Inf97] automated name recognition feature can be used to greatly improve performance of such indices. However, as our experimental results described in section 4.3 indicate, DRS-Systems can exceed both the accuracy and coverage of these services in certain categories such as personal homepages.

More efficient techniques to create and distribute such indices have been explored by the Harvest Project [BDH+94], but improved accuracy or coverage have not yet been demonstrated. The Harvest techniques have been used to build a demonstration search service, the WWW Home Pages Harvest Broker [BDH+95]. The broker indexes only 110,000 homepages, few of which are personal homepages.[3]

Directories like Yahoo! [Yah95] or Lycos' A2Z [Lyc96] can not keep up with growing number of homepages, as our experiments demonstrate. Similar performance is exhibited by other manually-created indices that are specialized on listing homepages, including "People Pages," [Hoy95] "Net Citizens," [Tra97] "Housernet," [Dei95] and many other sites [Dec95b].

The WebFind system [ME96] is comparable to our DRS implementation in the academic paper domain, Ahoy! Paper Mate. However, although similar in its scope, it uses a very different, sequential approach, which proves much less efficient than the DRS application. FAQ-Finder [HBML95] also works on a specialized domain very much like a DRS-System, but uses a local repository of carefully selected documents instead of the dynamic downloading in DRS-Systems.

In recent years clustering techniques have emerged as a way to cope with the increasing number of "relevant" references returned by standard search engines. [CSO96], [CKPT92], and [ZEMK97] have demonstrated systems using clustering techniques for Web retrieval. These techniques are suited for browsing large collections of documents, sometimes allowing the user to incrementally *refine* her query, as she is able to more precisely express what she is looking for. AltaVista's Live-Topics is an example for such a query refinement cycle. [CSO96] reports that user get confused when presented with clustered maps of search results, suggesting that user interface issues still prevent widespread use.

Information Integration & Extraction

[LRO96] and [KLSS95] describe systems that integrate information from multiple reference sources and use AI planning techniques to solve information gathering tasks. However, their work focuses on field-oriented information sources (in contrast to full

[3] Other types of homepages include *business* homepages, *project* homepages or *university* homepages.

text sources like Web indices, which offer less structure) and assumes a successful information extraction prior to the integration. [ME96] also examines aspects of information integration and describes the *field matching algorithm* for matching institutional name variations.

These systems, like the DRS-Systems presented in this work, require hand-coded models of the information sources used, requiring substantial effort to add new or update existing sources. The ShopBot system [DEW97] and recent work on "wrapper-induction" [KWD97] try learning to *extract* this information automatically, while the ILA system, described in [PDEW97], learns to *translate* this information into its own internal concepts.

The heuristics used in a DRS-System could in theory be used to classify pages, while piggybacking a spider that indexes pages for a standard search service. However, a lot of the accuracy of a DRS-System depends on the availability of additional, orthogonal information sources. Using only heuristics might not be as effective when trying to identify the correct page type. A system developed by [CFM+97] tries to automatically classify pages into arbitrary categories by using hierarchical knowledge base, but has yet to be demonstrated to operate on real Web data.

[BS92] surveys a number of highly specialized name matching applications, most of them commercial in nature. The Ahoy! prototype described here uses its own, rather sophisticated, but moderately large, name matching scheme for locating a person's name on a reference. We can use this rather simple approach since Ahoy! can use additional, orthogonal information to disambiguate between names it would otherwise fail to distinguish.

Learning

Many systems learn user preferences, either for Web Pages [PMB96], [BS95], [AFJM95], or for Net News [Lan94], [PS96]. These systems differ from the DRS-Systems described in our work, since DRS-Systems learn about the *Web* rather than *single users*. Systems that learn about users typically need a considerable number of *training examples*[4] before being able to exhibit any reasonable performance. By relying on domain specific heuristics, DRS-System can perform well even without learned knowledge. In effect, a user can immediately begin to effectively use the system, without having suffer through series of bad performance while bootstrapping its knowledge base.[5]

Web Agents

The DRS-System described here are closely related to the growing family of Web agents called "softbots,"[6] first described by [EW94], featuring systems such as Rodney, the Internet Softbot; the MetaCrawler [SE95], a meta search service; ILA [PDEW97],

[4] Usually user queries that provide explicit (i.e. user solicited) or implicit (i.e. observed) feedback.
[5] See section 3.2.3 for a description of learning mechanisms in DRS-Systems.
[6] See section 2.1.2.

an agent for information understanding; and the ShopBot [DEW97], a comparison shopping agent for the World Wide Web.[7].

[WE94], [Eic94], and [Kos95a] describe ethical issues of Web agents, which become increasingly important as more and more softbots, cancelbots, spiders and worms roam freely around the Web. The Ahoy! system honors the robot protocol [Kos95b], and uses statistics to reduce Web traffic when applying learned patterns on the Web.[8]

1.5 Thesis Overview

Chapter Two will begin by introducing the concept of an intelligent agent, and how one can be used to assist humans in performing everyday tasks. Our research stems from two specific kinds of agents: "information agents" and "softbots". Softbots are capable of interacting with software tools on a human's behalf, while information agents help the user to locate information in on-line databases. We will define what an information agent is, and present the idea of softbots in more detail. Then, we will bring the two concepts together by describing current limitations of locating information on the Web, and showing how softbots can be used to find information on it.

Chapter Three will describe our *Dynamic Reference Sifting* (DRS) architecture, and how it can be used to create a softbot for efficient information retrieval on the World Wide Web. We will summarize the design goals and give a number of example for possible domains for DRS-Systems. We will outline the key elements of a DRS-System– information sources, filtering and learning – and conclude with a brief summary and discussion of our framework.

Chapter Four will contain a detailed account of a case study using DRS in a specific domain. We will describe Ahoy! The Homepage Finder, a fully implemented and publicly fielded DRS-System in the Personal Homepage domain, by starting out with a brief description of personal homepages, followed by design criteria and historic development. We will describe the system and its modules in an overview, and then examine how each of the general architectural features – information sources, filtering and learning – are implemented in Ahoy!. We will end the chapter by describing our experiments with Ahoy!, which examine its accuracy and comprehensiveness compared to traditional Web search services. In a separate experiment, we compare the effectiveness of different methods to support the domain-specific learning in Ahoy!.

Chapter Five shows how the general DRS framework can be applied to other domains. We describe two additional DRS prototypes: Ahoy! Paper Mate in the Academic Paper and Joker! in the On-Line Jokes domain. For each prototype,

[7] See softbot family tree in figure 2.4 on page 22

[8] i.e., to keep the number of false tries minimal.

we will briefly describe its domain features, and how various parts of the architecture take these into account. We will also report our initial findings on some very brief, but suggestive, experiments conducted with both prototypes.

Chapter Six finally contains our conclusions and directions for future work, followed by an appendix containing a small glossary and some notes on the fielded Ahoy! system.

1.6 Acknowledgements

This work would not have been possible without the support of my advisors, Professor Ipke Wachsmuth at the University of Bielefeld and Professor Oren Etzioni at the University of Washington. Professor Wachsmuth made it possible for me to continue my work on DRS by accepting it as my diploma thesis, while Professor Etzioni generously provided office space and equipment, as well as offering valuable insights and suggestions during our weekly meetings.

Jonathan Shakes, who not only initiated the Ahoy! project but also christened it perfectly, remained a constant source of help during the development of Ahoy! and the other DRS-Systems, even after graduating this spring.

Britta Lenzmann gave valuable advice on how to cope with the frustrations of writing a thesis and constantly reminded me not to "sweat the small stuff".

Erik Selberg provided the red carpet, back-door connection to the MetaCrawler and spent countless hours discussing maintenance and performance aspects of the system, installing hardware or troubleshooting the servers.

I thank Tessa Lau,[9] Jonathan Shakes and Britta Lenzmann for lots of helpful comments on earlier drafts of this thesis, especially the first chapters.

[9] Although Tessa constantly tried to convince me of the many virtues of Python, I chose to implement Ahoy! in Perl, thus giving way to the countless number of bugs still hidden deep inside its source code.

2 Softbots and the Information Food Chain

In this chapter we examine what an intelligent agent is and describe the types of agents that form the basis of our research on DRS: *information agents* and a specific type of *software agent*, called "Softbots".

We will look at how softbots can facilitate the use of complex and inaccessible systems like UNIX and the Internet, and examine why it makes sense to use them to create "information carnivores" on the Internet's most popular service, the World Wide Web. In the course of this investigation, we will examine existing Web information services like AltaVista and Yahoo!, which we call "information herbivores", and show how they are suited to cope with today's wealth of on-line information.

2.1 Agents, Softbots and the Internet

2.1.1 The Intelligent Agent paradigm

[Joh97] offers a straightforward definition of an agent, which proves sufficient for giving an idea about their nature and environments:

> Artificial agents are computational systems that inhabit dynamic, unpredictable environments. They interpret sensor data that reflects events in the environment and execute motor commands that produce effects in the environment.

Agents can take many different physical forms, depending in part on the nature of their environments, part on the nature of their goal. For example, agents inhabiting the physical world typically are *robots*. The types of agents we are interested in are called *information agents*. Information agents help the user to locate information within huge database systems. Visionaries such as Alan Kay have seen information agents as the key to manage todays wealth of information available through on-line databases and the World Wide Web: "A retrieval 'tool' won't do because no one wants to spend hours looking though hundreds of networks with trillions of potentially useful items. This is a job for intelligent background processes that can successfully clone their users' goals and carry them out." [Kay90, page 203] Information agents allow the user to *abstract* these complex tasks involving hundreds if not thousands

of operations, sparing him from the need of direct manipulation – the pervasive interaction style for today's computer where users click, drag, and drop icons.[1]

Webster's English language dictionary [web95] defines an agent as "a person or business authorized to act on another's behalf." In that respect, these *information agents* can be seen as artificial counterparts to travel agents, insurance agents, or real-estate agents. They engage upon a client's specific request and try to satisfy his or her need by providing a small amount of information pulled from a much larger pool of available references. The agents goal is to enable its client to focus on the references relevant to his or her request while shielding the client from the noise of unwanted information.

Table 2.1 enumerates a list of characteristics that have been proposed as desirable agent qualities in the field of information agents, such as *Autonomous*, *Communicative*, *Mobile* or *Adaptive* [EW95]. While no single agent currently has all these properties, many prototype agents embody a substantial subset of these. Our own work has focused mainly on *goal oriented*, *collaborative* and *flexible* systems, while investigating aspects of *temporal continuity* through machine learning. Although there is little agreement about the importance of the different properties, most agree that these characteristics are what differentiate agents from simple programs [EW95, FG97].[2]

We will return to some of these properties during the course of defining our DRS architecture, but for now it is sufficient to note the basic idea of an *information agent* as a *goal-oriented*, *collaborative* and *flexible* "computational system" which retrieves on-line information in a "dynamic, unpredictable environment".

2.1.2 Rodney, the Internet Softbot

The term "softbot" is short for "software robot", and was first introduced by [ES92]. Softbots are Intelligent agents that use software tools and services on a person's behalf, in many cases relying on the same tools and utilities available to human computer users – tools for sending mail, printing files, querying databases, etc.

While *information agents* were characterized by their *task* – finding on-line information – *softbots* are described by their *means*. Instead of having the physical world as their environment, as robots do, a softbot's sensors and effectors use "Software" that was written for humans: Unix system commands like `ls` or `cp`, Internet software like `ftp` or `finger`, or even Web forms at on-line stores.

Using a softbot, the user can be shielded from the plethora of Internet and Unix services, allowing her to delegate tasks like *monitoring* (*e.g.*, disk utilization, user activity) *manipulating objects* (*e.g.*, compiling source code, converting documents, accessing remote databases) and *constraint enforcement* (*e.g.*, making sure that all files in a shared directory are group writable) to the automated assistant.

[1] Although this form of manipulation is appropriate when dealing with a small number of items, such as a couple of harddisks and a printer, it makes operations on larger scales, such as querying thousands of information sources dealing with millions of objects, cumbersome to handle.

[2] See [Pet96] for a discussion of *intelligence* in existing agent architectures.

1. **Autonomous:** an agent is able to take initiative and exercise a non-trivial degree of control over its own actions:
   (a) **Goal-oriented:** an agent accepts high-level requests indicating what a human wants and is responsible for deciding how and where to satisfy the request.
   (b) **Collaborative:** an agent does not blindly obey commands, but has the ability to modify requests, ask clarification questions, or even refuse to satisfy certain requests.
   (c) **Flexible:** the agent's actions are not *scripted*; it is able to dynamically choose which actions to invoke, and in what sequence, in response to the state of its external environment.
   (d) **Self-starting:** unlike standard programs which are directly invoked by the user, an agent can sense changes to its environment and decide when to act.
2. **Temporal continuity:** an agent is a continuously running process, not a "one-shot" computation that maps a single input to a single output, then terminates.
3. **Communicative:** the agent is able to engage in complex communication with other agents, including people, in order to obtain information or enlist their help in accomplishing its goals.
4. **Adaptive:** the agent automatically customizes itself to the preferences of its user based on previous experience. The agent also automatically adapts to changes in its environment.
5. **Mobile:** an agent is able to transport itself from one machine to another and across different system architectures and platforms.
6. **Character:** an agent has a well-defined, believable "personality" and emotional state.

Table 2.1: **Characteristics of Information Agents**, after [EW95]. While no single agent currently has all these properties, many prototype agents embody a substantial subset of these. The work described here focuses mainly on *goal oriented*, *collaborative* and *flexible* systems, while investigating aspects of *temporal continuity* through machine learning.

According to [EW94, Wel97] a softbot embodies the following ideas, loosely a subset of the agent characteristics given in table 2.1:

- **Goal oriented:** a request indicates *what* the human wants. The softbot is responsible for deciding *how* and *when* to satisfy the request, then perform the actions without supervision.
- **Collaborative:** a request is not a complete and correct specification of the human's goal, but a *clue* or a *hint* that the softbot attempts to decipher and then satisfy. Instead of issuing a passive error message in response to incorrect or incomplete specifications, a softbot should *collaborate* with the user in order to build a reasonable request.
- **Balanced (Communicative):** the softbot has to balance the cost of finding information on its own, against the nuisance value of pestering the human with questions
- **Integrated:** the softbot provides a single, expressive, and uniform interface to a wide variety of services and utilities.

Much of the early work on softbots has been focused on the "Internet Softbot", also known as Rodney [ES92, EW94]. Rodney uses a Unix shell and the Internet to interact with a wide range of local and remote resources.

The core of Rodney's functionality and added value to the user is its sophisticated planning technology. Given a users request, the planner dynamically chooses the appropriate software tools from a list of declarative representations, and chains together a particular sequence of requests and actions that fulfill this request. At runtime, Rodney is able to fluidly backtrack from one tool to another in response to transient system conditions (i.e. the finger gateway is down) [EW94].

Although softbots can be used for a wide variety of tasks, the focus of this work will be on how to use them to create the *information agents* described in section 2.1.1. The following will examine current means of accessing the World Wide Web, and show how softbots can be used to improve current methods of locating information. We will see that standard tools like Web indices and directories provide a powerful way to access the Web, but that using them directly can be time consuming and often frustrating. Delegating these tasks to a softbot allows the user to focus on the high level goals only, without having to know the underlying mechanisms.

2.2 Finding Information on the Web: Indices, Directories and Browsing

This section summarizes the different means for a user to find specific information on the Web. It will contrast three popular methods, Web indices, Web directories, and manual search, and try to evaluate them with respect to their accuracy and

comprehensiveness. It will then give an example of how a particular softbot, called the MetaCrawler, can provide a powerful new service based on existing Web indices.

2.2.1 Introduction

When Tim Berners Lee 'invented' the World Wide Web in 1990, the number of initial Web servers was less than a dozen [Cai93]. For the first years, this number stayed fairly constant, making Web navigation not much of an issue. Then, in 1993, the Web began its relentless expansion into the real world: software for creating Web pages was making its way into more institutions; more places had Internet-connected graphics terminals so that Web sites could feature pictures as well as text; the Web browser Mosaic started appearing everywhere; and individuals – rather than institutions – began putting up content pages on the Web.

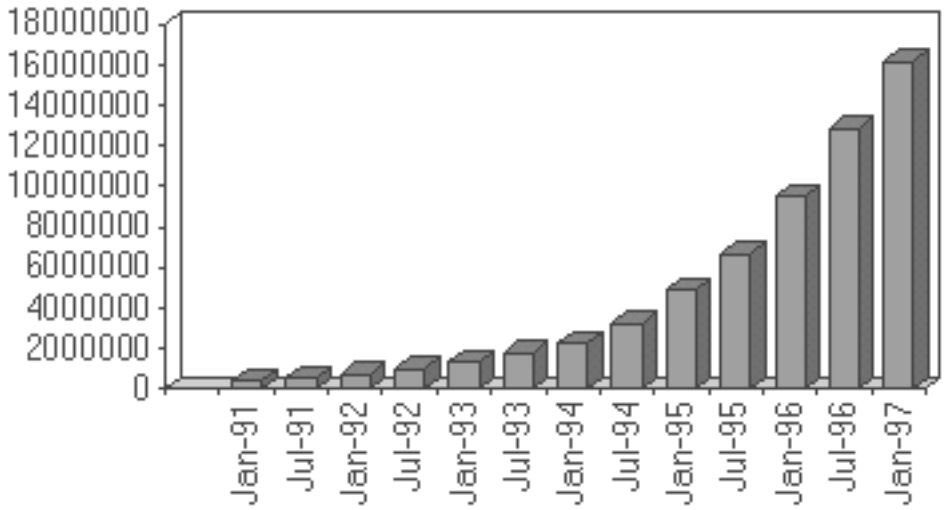


Figure 2.1: **WWW Hosts Growth**, as estimated by [Lot97]. The chart lists the number of hosts serving Web documents over a period from January 1991 to January 1997. For the first years, the number of available WWW hosts stayed fairly constant, making Web navigation not much of an issue. Today, the World Wide Web already features more than 16 million host machines, doubling every year in size.

Today, the World Wide Web is a huge network of interconnected computers, already featuring more than 16 million host machines, doubling every year in size. Figure 2.1 shows the exponential growth of Web hosts in recent years, as estimated by [Lot97]. According to visionaires such as Ambuj Goyal, Vice President of Systems and Software at IBM Research, everything from our office computer to the coffee machine at home, even individual light switches will eventually be 'online',[3] ready

[3]See [Ven96] for a discussion of the future of information technology in the home.

to provide an unimaginable wealth of information [Goy97]. Today, the Web already offers individuals the ability to track shipments, get movie listings, or order almost anything from books to videos to dishwashers. However, with more than 100 million documents out there,[4] looking for specific information is no easy task.

Today, users usually follow three popular methods – or a combination of them – to locate on-line information: *Web Directories* allow a focused search within a certain topic; *Web Indices* allow keyword searches across a large subset of all available Web documents; *Manual Browsing* accounts for searches where a lot of background knowledge exists, allowing the user to narrow the search to a specific region of the Web. But each method has its drawbacks, and we are going to look at each of them in more detail in the following sections.

2.2.2 Web Directories

A Web directory is basically like the Web's Yellow Pages: Instead of offering telephone numbers, a Web directory sorts *Web Links* into a small number of top-level categories, such as "Entertainment", "Politics", or "Business", offering the user a focused search within the limits of a certain category.

One of the first and by far most popular of these services is the "Yahoo! Internet Directory", founded in 1994 by two Stanford graduate students. Starting out as a humble little Web site called "Jerry's Guide to the World Wide Web",[5] it soon attracted thousands of hits per day, for it allowed users for the first time to access a large number of Web resources sorted into an ever expanding number of predefined categories, as shown in table 2.2.

Today, Yahoo! features over 100 thousand categories, containing more than one million different entries from over 400 thousand unique Web sites [Yan97].[6]

The advantage of using a human staff to manually index a large number of Web references is two-fold: First, users with only a vague idea of what they are looking for can easily *browse* the hierarchy tree, gradually narrowing down their field of interest until they "stumble" over the desired entry. Second, when using keywords to search the whole Index for a given topic, a Web Directory classifies the returned references into categories, which allow a fast elimination of unwanted information.

However, directories usually feature only a fraction of the pages indexed by the automated spiders and robots used by Web indices: While Lycos features more than 100 million pages, Yahoo! has only about one percent of this, one million pages, in its database.[7] A human editorial staff simply cannot keep up with the ever increasing

[4] As of May '97 Lycos Pro claims an index of 100 million pages. This figure serves as a lower bound on the total number of existing pages. See `http://www.lycos.com/press/pro3.html`.

[5] After one of the founder, then graduate student Jerry Yang.

[6] The ratio of entries per category is much higher than ten, however, due to the large number of cross-linked topics (i.e. topics that contain the same entries, but are at different points in the hierarchy).

[7] Counted in July 1997, Yahoo! featured exactly 1,059,641 entries. This number was semi-automatically obtained by adding up the numbers given in brackets next to the top level entries.

- **Arts and Humanities**
  Architecture, Photography, Literature ...
- **Business and Economy**
  Companies, Investing, Employment ...
- **Computers and Internet**
  Internet, WWW, Software, Multimedia ...
- **Education**
  Universities, K-12, College Entrance ...
- **Entertainment**
  Cool Links, Movies, Music, Humor ...
- **Government**
  Military, Politics, Law, Taxes ...
- **Health**
  Medicine, Drugs, Diseases, Fitness ...
- **News and Media**
  Current Events, Magazines, TV, Newspapers ...
- **Recreation and Sports**
  Sports, Games, Travel, Autos, Outdoors ...
- **Reference**
  Libraries, Dictionaries, Phone Numbers ...
- **Regional**
  Countries, Regions, U.S. States ...
- **Science**
  CS, Biology, Astronomy, Engineering ...
- **Social Science**
  Anthropology, Sociology, Economics ...
- **Society and Culture**
  People, Environment, Religion ...

Table 2.2: **Top Level Hierarchies of the Yahoo! Web Directory.** The 14 top categories lead to over one million entries in over 100 thousand categories.

growth of the Web – as the Web grows larger and larger, the gap is more likely to grow even wider.

And as good as this concept of "less is more" works for popular topics,[8] it is bad for single pieces of information. With only about 1% of the Web covered,[9] Web directories often fail to find even a single reference in their database when dealing with a fairly specific query such as a person's name or the title of an academic paper.

2.2.3 Web Indices

Another popular way of searching the Web is using Web indices. In contrast to Web directories, indices use automated processes to create their searchable index, making it possible to handle the large number of documents on the Web much faster than an editorial staff of Web directories.

Figure 2.2 shows the general architecture of a standard Web index. An automated process, often called spider, traverses the Web by following every hyperlink on an HTML page, indexing all words occurring on the page in a central database, and

[8] where a couple of hundred references are much easier to sift through than a couple of thousand.
[9] assuming Lycos' 100 million indexed pages as a lower bound for the real size of the Web.

then going on to the next link, and the next link, and so forth.[10]

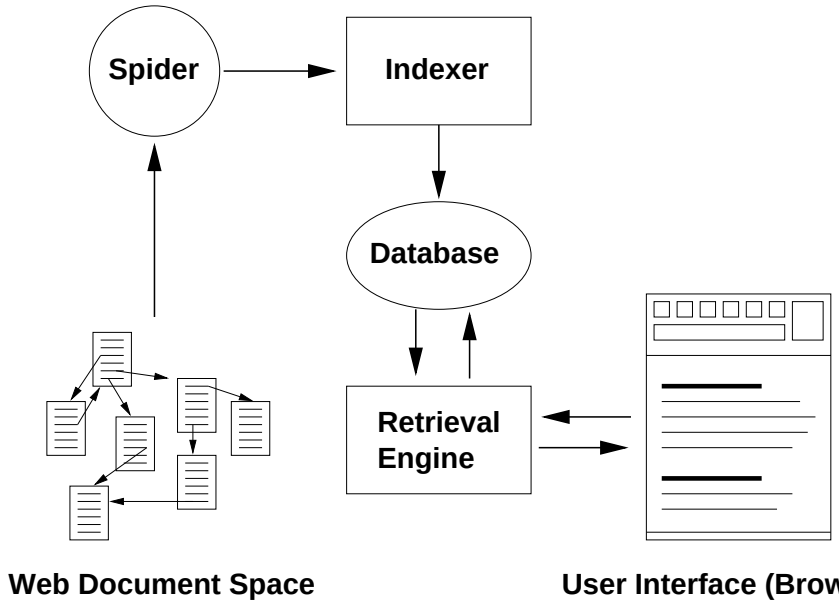


Figure 2.2: **Generic Web Index Architecture.** An automated process (spider) traverses the Web and indexes the encountered documents in a central database. Using a Web browser, this database can then be queried to obtain references to documents that are possibly relevant to the specific set of keywords given.

To prepare a document for indexing, a lexical analyzer breaks it down into a stream of words that includes tokens from both the title and the body of the document. The words are run through a "stop list" to prevent common words from being indexed, and are usually weighted by their frequency with which they appear in the document [Pin94]. The database then stores an *inverted index* of all encountered words to make queries fast: looking up a word produces a list of pointers to documents that contain that word. More complex queries can be handled by combining the document lists for several words with conventional set operations.

Once such a spider has indexed a sizable portion of the Web in the database, users are then able to query for pages containing certain keywords they're interested in. The documents[11] containing some or all of those words are retrieved from the index and presented to the user as an ordered list sorted by relevance.[12]

Today, these indices are one of the most popular way of searching the Web for

[10] Another way to think of it is that the Web is a large directed graph and that the spider is simply exploring the graph using a graph traversal algorithm.

[11] A document in a Web index is usually represented by its *title*, its Web address in form of a *Uniform Resource Locator* (URL), and possibly a small excerpt (*snippet*) of the full text of the original document.

[12] Relevance is usually determined according to the *vector-space model* [Sal89], although as with most other aspects of (commercial) Web indices, this information is often proprietary.

information. However, even though they feature databases containing hundreds of millions of pages, search indices suffer from two important drawbacks:

1. **Large Answer Sets:** Almost every imaginable English word and name now has countless occurrences throughout the Web. This makes it all too easy for index searches to be comprehensive beyond the point of usefulness. An average search can easily produce a few hundred, if not thousand results, increased by the fact that many default retrieval semantics further enlarge these sets by including pages that feature only *some* of the keywords. Most users are not likely to look through more than the top ten references returned, as studies by the OpenText corporation [Win96] point out. Even if a user does invest the time to look through most, if not all, returned references, she could end up spending a significant amount of her time without finding anything that would justify the extra effort.

Service	URLs claimed[13]
AltaVista	31 million
Excite	50^+ million
HotBot	54 million
Infoseek	50^+ million
Lycos Pro	100 million

Table 2.3: **Web Indices Index Size.** Popular Web indices feature between 30 and 100 million documents in their databases. Assuming that the huge size of the Lycos Pro catalog subsumes those of its competitors is premature, however. Searching for the authors name using the "small" AltaVista catalog results in 41 documents; the three times as large Lycos Pro catalog finds only 3 (as of July 1997).

2. **Lack of Completeness:** Even though Web indices have a vast coverage of millions and millions of pages, even the largest indices are far from being complete for three reasons.

   Studies by Selberg [SE95] have shown that the Web evolves much faster than those centralized indices can keep up with: It was found that no single service chosen from a list of five of the most popular search engines would cover more than 30% of all returned references, when combining the output of all sources into one collated list.

   A similar problem arises from the "partial indexing" practiced by many services: In order to keep index time short and the database small, only a certain number of pages at a each site are actually stored in the central index [Pik97, Mon97]. Not surprisingly, the specific set of pages indexed at a particular site varies

[13] See `http://altavista.digital.com/`, `http://www.excite.com/Info/features.html`, `http://www.infoseek.com/doc?pg=comparison.html`, `http://www.lycos.com/press/pro.html` and [Hot97].

widely from service to service. Assuming, for example, that the huge size of the Lycos Pro catalog subsumes those of its competitors is premature. Searching for the authors name using the "small" AltaVista catalog results in 41 documents;[14] the three times as large Lycos Pro catalog finds only 3 (as of July 1997).

Finally, pages that are not yet 'interwoven' into the mesh of hyperlinks pointing to pages containing hyperlinks that will point to other pages, will *never* be reached by *any* spider. Only when a page that is repeatedly indexed by such an automated indexer finally creates a link to this new page, and with this including it into the 'web' of interconnected hypertext documents, these documents have a chance to get indexed by such a service.

The first drawback, answer sets that contain hundreds and thousands of "relevant" documents, can be remedied by using the sophisticated query language offered by many services, as well as refining the query sub-sequentially until a reasonably small set can then be manually searched. However, only few users are willing to invest the extra time necessary to learn these specialized language constructs,[15] which are then for most parts only usable at one particular service, while being non-existent at another one or having a different meaning at a third one.

Because of the second drawback, the lack of completeness, the user might even have to repeat this process of query refinement and translation into specialized query syntax multiple times, using different services to increase the overall completeness of her search. Although this time-consuming endeavor might make it possible to find the desired information in the end, a casual user is likely to give up much earlier in the process.

2.2.4 Manual Search

When Vannevar Bush's article "As we may think" described in 1945 what would be known today as the first hypertext system [Bus45], his idea was to let the user 'browse' for information by simply clicking on words or concept she was interested in, just following her train of thought. From a specific "entry point" – the users homepage, her departmental or institutional homepage, or even the default page of her browser – a user then would find information by simply following available hyperlinks, until her journey would eventually take her to the desired document.

For example, when looking for Oren Etzioni's latest paper, a user might start with link from her own homepage that links to her personal repository of Web resources. From there, she would follow a link to her school's similar repository, which she knows contains links to a list of higher academic institutions worldwide. Once there, she would quickly locate North American universities, and within this category a pointer to the University of Washington. The university homepage in turn contains

[14] Using `+marc +langheinrich` as the query term.

[15] The Lycos Pro search service features more than x operators, together with the ability to specify a relevance function yourself.

links to all departmental homepages, so she would follow a link to the Department of Computer Science, and from there on a link to a list of faculty. She would locate Professor Etzioni's homepage under "E", and finally find a pointer to his recent paper off his homepage.

Of course, there is a considerable amount of background knowledge necessary to succeed in such a manual search. In order to find a specific document, one must have a fairly clear idea about which part of the Web this page might be located in. For example, an academic paper as in the example above might be linked to one or more of its authors homepages, which might in turn be accessible from their departmental homepage. Although one could expect to know at least one author of the paper, knowing the authors institution is already less likely. Or maybe the institution decided to keep all publications in a central repository, to which the author forgot to link from his homepage. In this case, a manual search has to backtrack and try to find links to this repository somewhere higher up in the existing hierarchy.

It is easy to see that such an approach might result in several minutes of intensive search. If the user stays concentrated enough without letting herself being distracted by the wealth of links encountered during such a search, there might be a good chance that the wanted page is found in the end. However, in case the user was unable to locate the desired page, precious time has been wasted without any results at all.

2.2.5 Web Agents

As the last three sections showed, current methods for finding information on the Web are far from perfect: either they return too much or too little, or they take too long. Table 2.4 sums up the advantages and weaknesses of each approach. This section describes how we can use *intelligent agents* to directly target the weaknesses described above. A softbot, using all those mechanisms as its tools, can perform tasks that would be extremely tiresome or time consuming for a user.

[Etz96] keyed the term "information food chain" (see figure 2.3): the maze of pages and hyperlinks that compromise the Web is the very bottom of the chain. Web indices (such as WebCrawler or AltaVista) and Web directories (such as Yahoo! or Lycos' A2Z) are *information herbivores*, grazing on those Web pages and regurgirating them as searchable indices. Web agents represent *information carnivores* in this framework – intelligently hunting and feating on the Web's herbivores.

But in order to be acceptable, any Web agent has to meet the high standards set by the Web Community:

- **Robustness**: Web users expect a working system, 24 hours a day, 7 days a week. Tools that are most of the time off-line, will soon be removed from a users personal bookmark files.

- **Speed & Reactivity**: Virtually all Web based information systems begin transmitting information within seconds – answers that take minutes will most likely never been read.

Search Method	Advantages & Disadvantages
Web Directories	+ Small index keeps results manageable. + Categories allow for direct browsing. + Restricted search within category. + Search results grouped by category. − Index size too small for many categories, since it typically contains only about 1% of the available documents on the Web.
Web Indices	+ Large coverage, find more pages. − Large index size often leads to huge results set. − Default retrieval semantics often increase result set by including pages that feature only some of the keywords. − Some services indices drop a certain percentage of pages per site. − Web grows and changes too fast.
Manual Search	+ Sophisticated, manages to find otherwise unfindable pages by using large amount of background knowledge. − Time consuming, failures are expensive. − Only works when sufficient background knowledge available.

Table 2.4: **Popular Web Search Methods.** This table summarizes the advantages and disadvantages of the various search methods that are currently used to find information on the Web.

- **Added Value**: The system has to result in a tangible benefit for the user, otherwise there is little incentive to use the new service.

General planning softbots like Rodney are not yet ready for these rigorous demands of public on-line use. *[More here about why not: sophisticated planning software, often implemented in LISP, makes process memory intensive, brittle and time consuming]* Instead, research at the University of Washington focuses on what [Etz96] calls the *useful first* paradigm: "Instead of starting with grand ideas about intelligence and issuing a promissory note that they will eventually yield useful intelligent agents, we take opposite tack; we begin with useful softbots deployed on the Web, and issue promissory note that they will evolve into more intelligent agents."

This useful-first, bottom-up design follows a similar approach taken by Brooks and others for building complete agents and testing them in 'real world' environments. Focusing on emergent behavior instead of carefully crafted intelligence, they try to avoid any simplification in the agents environment, since "it is very easy to accidentally build a submodule of the systems which happens to rely on some of those simplified properties... the disease spreads and the complete system depends in a subtle way on the simplified world." [Bro91]

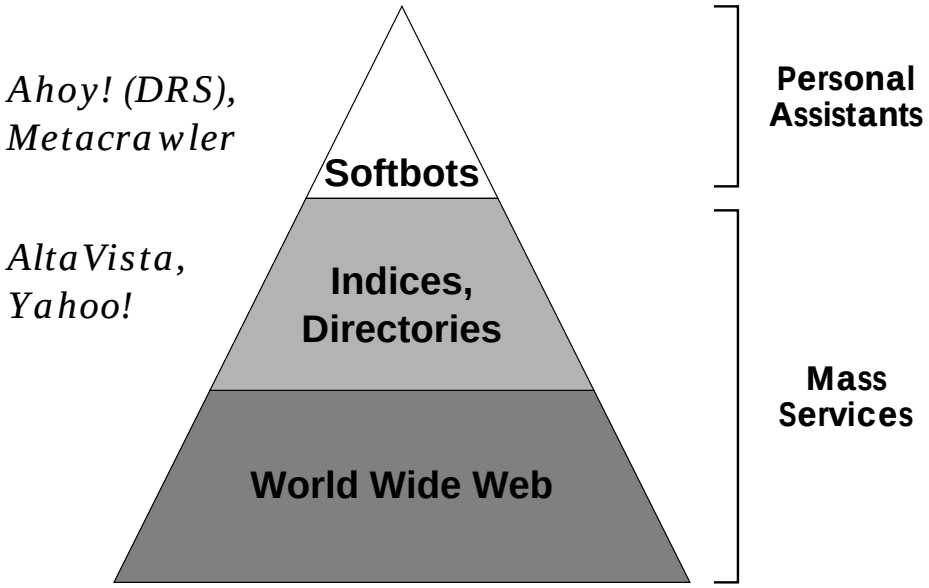


Figure 2.3: **The Information Food Chain.** *Information herbivores* like Yahoo! and AltaVista graze on Web pages and regurgirate them as searchable indices, on which *Information carnivores* like the MetaCrawler and Ahoy! hunt and feast [Etz96].

Also, if restricting the domains of intelligent agents to toy examples for the sake of building truly "intelligent" systems, "the whole reason d'être for software agents is lost" [KSC94], since agents must provide solutions to real problems that are important to real users.

Addressing those "real-life" problems, the Softbot Group at the University of Washington developed and deployed a number of Web agents, as pictured in figure 2.4. Ahoy!, Paper Mate and Joker! are the focus of this thesis. The ShopBot agent is described in more detail in [DEW97]. The MetaCrawler is the enabling technology for DRS softbots, which are perched above it in the information food chain depicted in figure 2.3 .[16]

The MetaCrawler softbot provides a unified interface for Web document searching. It queries five of the most popular Web indices in parallel, eliminating the need for users to try and retry queries across different services.[17] Also, by providing a single interface, MetaCrawler unifies differences in query capabilities of each service: If a user is looking for a specific phrase, MetaCrawler will use phrase searching for those services that support it, while downloading and using its own phrase search for those references returned by search services without this option. A more detailed description of the MetaCrawler softbot can be found in [SE95].

Each of these softbots uses multiple Web tools or service on a person's behalf,

[16]The arrows used in the softbot family tree in figure 2.4 represent a different, *chronological* order.
[17]See problem of completeness mentioned in the last section.

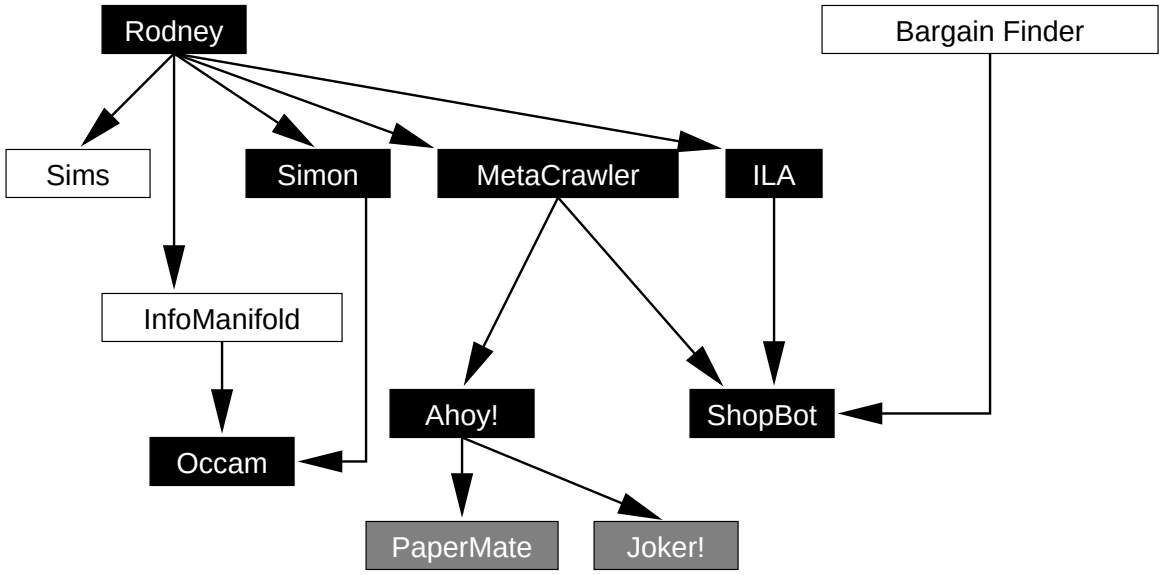


Figure 2.4: **The Softbot Family Tree.** The black boxes represent softbots developed at the University of Washington. MetaCrawler, Ahoy! and ShopBot have been deployed on the Web, while Paper Mate and Joker! exist as experimental prototypes. See [Etz96]

enforcing a powerful abstraction: a person is able to state *what* they want, the softbot is responsible for deciding *which* Web service to invoke in response and *how* to do so. By bringing together the concepts of *information agents* and *software robots*, these simple, yet powerful systems enable the user to delegate the laborious footwork of searching the Web to an intelligent background task, sparing him from the need of directly manipulating tens and hundreds of information sources.

3 Dynamic Reference Sifters

In this chapter, we will present the concept of a *Dynamic Reference Sifter* (DRS-System), which is just another member of the growing number of softbots developed at the University of Washington, and fills the role of an *information carnivore* in our food chain metaphor.

After some historical background we will enumerate our design goals and describe the domains in which DRS-Systems are applicable. We then proceed to list the key elements of the DRS architecture in more detail, and conclude with a brief summary and discussion of our proposed framework.

3.1 Introduction

The Homepage Finder project, and with it our work on Dynamic Reference Sifters, started out following a remark from the University of Washington CS department chair Ed Lazowska, who suggested building a softbot that could find a persons homepage. The generic DRS architecture is a direct result from our work on this challenge. After trying a number of possible approaches to locating individuals homepages with high accuracy (described in more detail in section 4.2.1), the DRS framework soon emerged as the most useful and robust scheme. Although geared towards finding a specific type of page at first, the general mechanisms of this architecture, along with a number of suitable domains (see section 3.2.4), suggested that DRS-System would perform equally well at other tasks. Our results in chapter 5 seem to confirm this.

An important aspect of our work were our efforts to create not only an experimental prototype used by a few number of people within our own research group, but to provide a service that could be useful to a large number of people on the Web. Fielding public services on the Web introduces a large number of challenges, as outlined in the last section of our previous chapter, which seem tangential to the actual research itself. However, as Etzioni puts it in [Etz96], "we need to recognize that intelligent agents are ninety-nine percent computer science and one percent AI".

Taking speed and reliability of our prototype into account not only allows us to respond to the common stereotype "if it works, it ain't AI", but also offers an immediate benefit to the research itself: Instead of conducting experiments with a handful of sample users, we can accumulate data from hundreds of users who use the fielded service each day.

Primary design goals (added value)

1. **High coverage:** In case the desired page can be found by any conventional search engine, we want our DRS application to find it as well.
2. **High accuracy:** We only want truly relevant references displayed to the user, so that the usual hundreds of "relevant" references found by traditional search engines are reduced to less than a handful, preferably a single answer.
3. **Graceful degradation:** In case that no answer could be found, the user should be left with the best information available to the system at this point.

Secondary design goals

4. **Robustness:** The service should be available 24 hours a day, 7 hours a week.
5. **High speed:** It should provide answers within seconds, or at least comparable to conventional search engines.
6. **Reactivity:** The user should be continuously informed about the search progress.

Table 3.1: **DRS Design Goals.** Following the desiderata for Web agents described on page 19, DRS-Systems need to be *fast*, *robust* and *reactive*, as well as providing the user with enough *added value* to make its use worthwhile.

3.1.1 Design Goals

Our ideas for designing a new search service architecture fall into two categories, which are summarized in table 3.1. The primary goal was to provide the user with a real *added value*, by building an information agent that would combine both *large coverage* and *high accuracy*. In case of a failed search, we wanted it to exhibit a *graceful degradation* in performance: Instead of a plain "Nothing found" message, the agent should try to provide as much valuable information as possible. Secondary design choices came more as pragmatic Web challenges: In order for any service on the Web to be attractive, the system had to provide answers in a *reasonable amount of time*, prove *robust* enough for a large number of users, and have the notion of *reactivity* that would keep the user continuously informed about the search process.

3.1.2 DRS Set Characteristics

Contrasted to Softbots in general, Dynamic Reference Sifters are very focused. Compared to the original "Internet Softbot", Rodney, DRS-Systems are only good at one specific task. Whereas Rodney tried to satisfy a wide variety of complex requests such as "Send the budget memos to Mitchell at CMU" or "Move all TEX Sources into the archive, unless they haven't changed, and delete the remaining files", DRS-Systems focus on requests of the form "Find me X", where X could be "the homepage of professor Weld", "the paper by Pettie Maes entitled Learning interface agents" or "that joke with the destroyer and the lighthouse".

The important difference is that the area of expertise of a DRS-System is restricted to a single, well defined topic, such as homepage, academic papers, or jokes. Such a specialized service does not have the expressive power of a full featured Internet Softbot, but it allows us to build our intelligent agents *bottom-up* instead of *top-down*: beginning with a fairly simple, but nevertheless useful agent that can gradually increase in complexity, as more and more sophisticated goals are added — instead of starting with a ambitious framework of intelligence, which has to be massively re-engineered to make it usable.[1]. As [KSC94] points out, one of the most difficult tasks in agent design is indeed "to define specific tasks that are both feasible using current technology, and are truly useful to the user."

In order to achieve this feasibility, we not only have to limit our softbot to the single "Find me X" task, but also more precisely define what exactly we will allow for X. 5 constraints characterize the class of pages that a DRS-System will be able to find:

1. **Availability:** The desired information has to be available on the Web. Clearly, our softbot can not provide an answer to questions concerning non-Web objects, like "Find me *my car keys*."[2] In addition, we demand that most member pages, but not necessarily all of them, are accessible via traditional search services. Although it would in theory be possible create a local index database in very much the same way we described it for Web indices in section 2.2.3, the resource demands would be well beyond the scope of a student project.

2. **Focused attention:** The DRS-System must know what the user is looking for. Although some systems, like the Syskill & Webert system [PMB96] or Mitchell's WebWatcher [AFJM95], are trying to answer the request "Find me *pages I am interested in*," the methods for describing these concepts[3] are not yet precise enough to account for the high precision we demand. If the user herself does not have a clear idea what she is looking for, the hierarchical browsing

[1] see [Bra92] for an account of the massive re-engineering necessary to transform an "intelligent first" knowledge representation system into a usable one

[2] Although this might be a perfectly reasonable request for a physical agent, like a robot, or even a softbot, once car keys would have their own Internet address :).

[3] Explicit and /or implicit user feedback leads for example to a vector space model.

capabilities of Web directories are much better suited than any form of user solicited search.[4]

3. **Strong cohesion:** It has to be possible to identify the desired features on member pages. A question like "Find me *homepages of people with the same birthday as mine*" might be unambiguously stated, however, only few persons might actually include information concerning their birthday on their personal homepages.[5]

4. **Large cardinality:** Sets that are small and fairly static, *e.g.*, the list of British Universities, can easily be enumerated in manually created lists or directories, rendering the use of a specialized softbot unnecessary.

5. **Widely dispersed:** Sets that are distributed from a central repository, such as tax documents or white house press releases, can be comprehensively obtain at the originating site, eliminating the need for a specialized search service.

Although the above constraints (see summary in table 3.2) sound overly restrictive, they still leave a large enough number of possible domains. Examples include:

- **Transportation schedules:** More and more regional and national transportation providers have begun to offer their current timetables on the Web. Not only are changes in the schedule much cheaper to update on-line than reprinting large amounts of paper handouts, but having a Web presence is often seen as a relatively cheap way of offering a better quality of service to an ever growing number of Web-literate customers.

- **Personal homepages:** Personal homepages often contain important contact information or publications, as well as background information and general interests of people. This is a highly dynamic and large list, and even though many individuals invest extra efforts to register their page with a whitepage or directory service, only a fraction of today's available homepages are probably available in these manually generated lists. By soliciting extra information like the affiliation or even the email address of the person one is looking for, the DRS-System can achieve extra accuracy.

- **Business homepages:** The number of business homepages begin to expand rapidly, as more and more companies establish an online presence. Although companies do have the incentive to advertise their Web presence and register it with the most number of possible search services, the sheer number of registrations make it hard for directories to keep up with this ever changing list. In addition, a user searching for a company with a common title is most likely to

[4] By traversing down the hierarchy tree, a user can gradually define the topic of her search simply be following the existing category that seems to best describe her information needs.

[5] Although additional data sources like 'the world birthday net' might add this missing piece of information.

end up with a large number of 'relevant' entries even with directories. However, if someone is looking for a number of companies in a particular kind of area, a directory listing would be more appropriate.[6]

- **Academic papers:** The initial idea of the World Wide Web was the exchange of scientific work and results, and more and more researchers and publishers are putting their work online. Ideally, all writing would be done in HTML or its superclass SGML,[7] so that articles could be indexed like all other pages (with the usual limitations). However, the lack of layout control and support of formulas in the early HTML drafts still forces a large number of authors to use Postscript or PDF formats for their work, which, even though accessible through the Web, currently prevent indexing by standard search engines.

- **Product reviews:** A large number of independent magazines, as well as manufacturers, provide detailed description of products on-line. Although most sites have somewhat adequate search interfaces within their own site, it is not trivial to locate the correct magazine or manufacturer for each product of interest. A DRS-System could help to quickly locate a product description (i.e. the features of a recent notebook) or review (i.e. of a new video camera).

- **Jokes:** As the Internet continues to become a global communication medium, more and more people begin to exchange community oriented information such as recipes, political views, or even jokes. Many individuals post their favorite jokes next to their latest cookie recipes and picture collection off their personal homepage. Using a DRS-System, forgetting the punch line of a joke is not an issue anymore.

After we have discussed the key elements of DRS-System in the next section, we will return to these domains and briefly describe each element in its domain specific implementation, to give an idea about the actual use of DRS in a search application. Chapter 4 will then describe our fielded prototype Ahoy! The Homepage Finder in greater detail, while chapter 5 will report our initial results with two further prototypes in the academic paper and on-line jokes domain.

3.2 DRS Architecture

Now that we have outlined the design goals for our DRS-System, we will enumerate the key elements a DRS architecture uses to achieve this. Some of the following elements will already be familiar to the reader, as they directly relate to our desiderata put forth in the last section, while some introduce new concepts that will be discussed in more detail in the following paragraphs.

[6] According to our requirement of *focused attention*, this would fail to constitute a tightly bound search for only a few references.

[7] HTML and SGML are ...

1. **Availability:** Many, but not necessarily all, of their members are available using a more traditional reference source (*e.g.*, keyword queries to Web Indices like AltaVista).
2. **Focused attention:** During a given search, a user is interested in very few, and often only one, members of the class, and the user can pose a query specific enough to exclude other members.
3. **Strong cohesion:** Their members are easily identifiable as belonging to the class.
4. **Large cardinality:** They are too large or too dynamic to be exhaustively indexed by hand.
5. **Widely dispersed:** No central repository exists where all or most members can be found.

Table 3.2: **DRS Set Characteristics.** DRS-Systems are by no means appropriate for all Web searches. They work best for classes of pages with the above characteristics.

The basic idea of DRS is that of a sophisticated filter: using the output of general purpose search services, combined with additional, orthogonal information sources; domain specific heuristics; and a flexible categorization scheme, a DRS-System filters out all but a few, highly relevant pages.

Instead providing a single service that achieves a moderate performance in a wide variety of domains, a single DRS-System focuses on one domain only, where its custom-tailored elements, taken from the generic DRS architecture, provide it with enough domain knowledge to achieve our ambitious design goals.

A DRS-System usually contains a number of the following key elements, working together as depicted in figure 3.1:

1. **Reference source:** A comprehensive source of *candidate references*. (*e.g.*, a Web index like AltaVista, or even better, a comprehensive Meta-Search service like the MetaCrawler)
2. **Cross filter:** A component that filters candidate references based on information from a second, orthogonal source. (*e.g.*, a database of e-mail addresses)
3. **Heuristic-based filter:** A component that increases accuracy by analyzing the candidates' textual content using domain-specific heuristics.
4. **Buckets:** A component that categorizes candidate references into ranked and labeled *buckets* of matches and near misses.
5. **URL Generator:** A component that synthesizes candidate *Uniform Resource Locators* (URLs) [BL89] when step 1 through 4 fail to yield viable candidates.

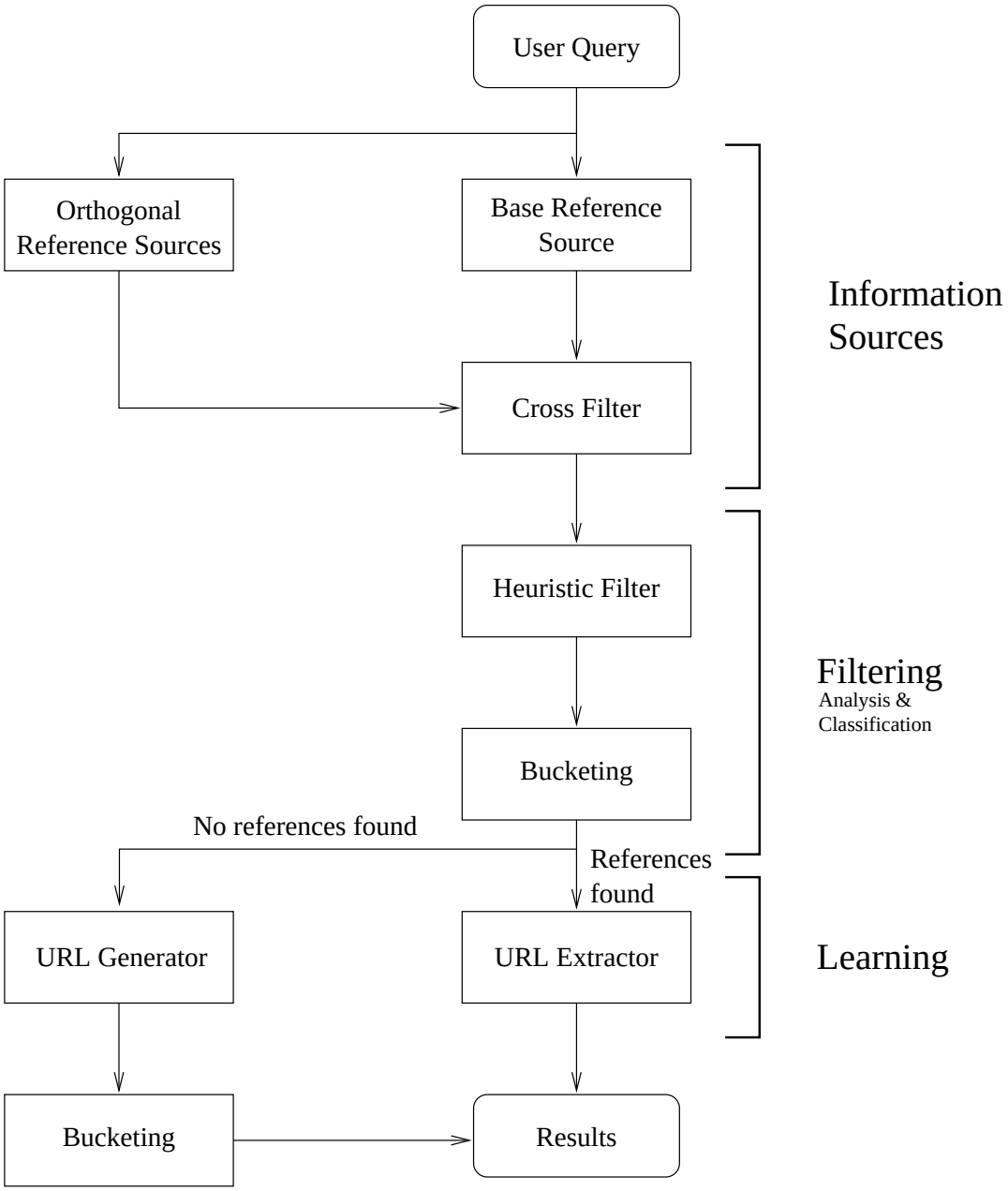


Figure 3.1: **DRS Control Flow.** After sending the user's query to its information sources, the returned references are analyzed and sorted into buckets. In case a relevant reference was found, its general pattern is extracted. Otherwise a list of possible URLs is generated and tried directly.

6. **URL Pattern Extractor:** A mechanism for learning about the general patterns found in URLs based on previous, successful searches. The patterns are used by the URL Generator.

The following paragraphs examine each of these elements in more detail. The six elements are grouped into three categories: *information sources*, *filtering*, and *learning*. A DRS-System for any given domain does not necessarily have to contain all of these elements – some could feature only a single element from each group, or even choose to skip one category completely. Although the overall framework of DRS is domain independent, actual implementations could vary their use of each element according to the problem at hand.

3.2.1 Information sources

Information sources are the foundation for any DRS-System. Here is where all information comes from that is eventually presented to the user. Information Sources come in two flavors: *base reference sources* and *orthogonal reference sources*.

Base reference sources form the set of documents that eventually contain the answer that we are looking for. Since comprehensiveness is important here, general Web indices provide usually the most coverage, or, even better, using a meta search service that pools the answers from a number of indices. In theory, though, any information source on the Web could potentially be used as a base reference source, as long as it contains the answer the user is looking for.

DRS-Systems use these base reference sources in a very broad way. In order to achieve high recall, queries to those underlying information sources are posed as general as possible, to make sure that the desired information is found among the returned references. Once the base reference sources have returned all available references, a DRS-System then relies on it's sophisticated filtering and bucketing mechanisms to find the few *relevant* references.

Orthogonal reference sources can be used in two different ways. Intuitively, they can be used to acquire additional, so called *orthogonal* information related to the specific information the user is looking for. For example, when looking for an individuals' homepage, querying an email-whitepage directory at the same time could produce the email address of the person, which, in case no homepage could be found, might still prove useful to the user.

Additionally, orthogonal reference sources can be directly used for filtering. Any additional sources, as long as they share common fields with our base reference set, can be combined to form a set of higher precision, based on their intersection. For example, during a search for an individuals' homepage, an institutional database could provide a list of possible *hostnames* for that persons institution. Later, a DRS-System could then compare those with the list of homepage references found using the base reference source — in case of an identical hostname, we could assume that we found a reference at the right institution.

As we move on to implementational details, section 4.2.3 contains more information about how to use these sources in detail. The next section will explain the basic idea of filtering in a DRS-System.

3.2.2 Filtering: Feature Analysis, Classification & Selection

Although the last section already mentioned filtering by using orthogonal information sources, the following section will explain the idea of filtering in more detail, and how heuristics together with a flexible bucketing algorithm can help DRS-Systems to provide dynamic responses that always give the best possible answer, even in case of a failure.

The process of filtering in a DRS-System is a three step approach. First, each reference is analyzed according to certain *domain dependent features* relevant to the users query. Second, each reference is categorized into several pre-defined *buckets*, which are labeled according to their respective analyzed content.[8] Finally, a *selection mechanism* locates the "best" available (i.e. filled) bucket which can then be displayed to the user as the result of her search.

Assessing Relevance: Feature Analysis

During the analysis, a DRS-System examines certain parts of the base references, most commonly the *title*, *URL*, and the *snippet* (the short piece of document excerpt common with most Web indices, usually the first 25 words of the text) of the reference and categorize them along the different features deemed relevant for that particular domain given the current query. For example, relevant features for a DRS-System in the personal homepage domain could include the *owner* of the reference (i.e. whether firstname or lastname of the person appears in the title), its *location* (whether it is in the country the user specified, or at the institution that is suppose to employ the person) and its *type* (i.e. if its title contains the word "homepage," which would be favorable, or "publications," which would not).

Additionally available information from orthogonal references, together with the query parameters given by the user, can be used to provide the analyzer with values to compare each document with. For example, when looking for a specific academic paper, an on-line database for scientific abstracts could provide the system with the institutional affiliation of the authors, thus providing a list of domains the URL of a reference should contain.

This feature analysis can be seen as a mapping from query parameter Q, orthogonal information O and document parts P to domain specific document features F:

$$Q \times O \times P \quad \xrightarrow{\alpha} \quad F$$

[8] This process might be repeated in the case when additional downloads are necessary, so that a first analysis can find the most promising candidates to download, while a second analysis then finds the actual relevant pages among the downloaded documents.

Specifically, given a query Q and its parameter $q_1, \ldots q_k$ with values $w_{i1}, \ldots w_{in_i} \in W_i;\ i = 1 \ldots k$; a set O of orthogonal sources $o_1, \ldots o_l$ and their results $r_{i1}, \ldots r_{in_i} \in R_i;\ i = 1 \ldots l$; a given page P with title $t \in T$, URL $u \in U$, snippet $s \in S$ and full-text $d_j \in D$; and a set F of domain specific features $f_1, \ldots f_m$ with possible values $v_{i1}, \ldots v_{in_i} \in V_m;\ i = 1 \ldots m$, a feature analysis function α defines the following relation:

$$(W_1 \times \ldots W_k) \times (R_1 \times \ldots \times R_l) \times (T \times U \times S \times D) \quad \xrightarrow{\alpha} \quad (V_1 \times \ldots \times V_n)$$

Since the full text d of a document is not directly available from the base reference sources, a DRS-System attempts to use only the title t, URL u and eventually snippet s of each page.[9] However, in certain situations or domains where it is necessary to examine the full text d of the document, the DRS-System has to download a page to assert this unknown value – a time and bandwidth consuming process. By conducting a partial analysis using the directly available document parts t, u and s only, a DRS-System can significantly reduce the number of documents that are downloaded. When proceeding in the order suggested from such a partial analysis, the DRS-System has a much higher chance of finding the desired information on the first few requested pages.

Sorting it out: Document Classification

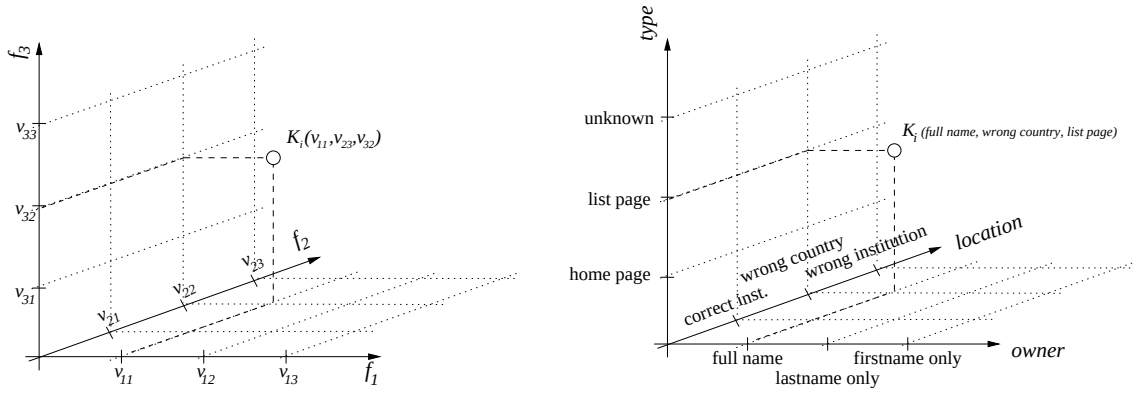
After all feature values have been computed by the analysis function, each reference is sorted into labeled containers, called *buckets*. These buckets allow a DRS-System to sort out pages with different features during the "sifting" through the comprehensive set of reference sources. Each bucket is labeled with a distinctive combination of values of the analyzed features, and once all references have been sorted into the appropriate buckets, a DRS-System can easily identify the bucket containing the most promising pages — it is the one with the best combination of values from the feature analysis.

The number of buckets needed to categorize all elements of any search depend on the number of features and their possible values. We assume only discrete values – any continuous values would need to be expressed in discrete ranges. Having identified m relevant features $f_1, \ldots f_m$ with possible values $v_{i1}, \ldots v_{in_m} \in V_m;\ i = 1 \ldots m$, the total number of buckets is

$$b = \prod_{i=1}^{m} n_m$$

The easiest way to arrange these is naturally in an m-dimensional space, featuring n_i distinct values on each axis i. An example is shown in figure 3.2. Such an arrangement is called a *DRS-Table*.

[9] Not all Web indices provide snippets. Nor do references featured in those services that provide snippets always include one.



(a) Generic Format

(b) Personal Homepage Domain

Figure 3.2: **DRS Table.** Each axis represents one feature as reported by the analysis function. The example on the right side shows part of a possible configuration in the personal homepage domain, which is described in more depth in chapter 4, including one particular bucket K_i that would contain references featuring the *full name* of the person on a *list page* in the *wrong country*.

Choosing Results: Bucket Selection

When arranging all values within each feature (i.e. along each axis) in descending order of relevance to the query, the origin of such an n-dimensional space will contain the most relevant bucket, with more and more irrelevant buckets the further away from the origin.

However, not all features carry the same weight. Some features only carry a preference (i.e. in the individuals' homepage domain, of preferring pages with the title "homepage" over those with only the name as the title), while others are of more importance to the query (like the specified country: when looking for someone's homepage with the name "John Smith" in Great Britain, the user explicitly stated that she is not interested in homepages of other John Smith, say, in Denmark).[10]

In order to equalize over these different scales, an additional layer of so called *zones* provides a high level abstraction for the rather detailed analytical results expressed in each individual bucket. These (domain dependent) zones account for the differences between mere cosmetic features and more essential ones. Zones can then be labeled in a more abstract way, like an unambiguous "Success – found the page you were looking for", a cautious "Partial Success – found a page at the right institution, but doesn't appear to be a personal homepage", or an explicit "Failure – could only find pages in a different country".[11]

[10] Although she might still be interested in out-of-country references: If this is the best thing the DRS-System could find, it could be possible that John Smith recently moved to Denmark.

[11] All examples taken from the homepage domain.

Once all available references are sorted into the appropriate buckets, finding the answer is simply a matter of taking the "best" zone, locating the "best" bucket within this zone, and displaying its content together with the zone's title to the user.

In some cases, however, especially among the zones that do not directly represent a successful outcome of our search, it is hard to know which one of them will be the most useful for the user. Instead of just displaying the zone that the designer of the DRS-System deemed best, zones can have pointers to other zones, thus indicating *alternatives* among certain answers that each might be of value to the user. Thus, if only an ambiguous answer could be found, a DRS-System still presents the user with the "best" zone that seems to be relevant to her search, but offers links to possible alternative zones, in case the answer could not be found in the original zone presented to the user.

In theory, all zones could have associated alternative zones, but in practice this only makes sense for lower quality zones. However, it is up to the final implementation of a DRS-System within each domain to decide when it makes sense to offer alternatives to the user.

Filtering Summary

Let us summarize this process again. First, a simple heuristic *analysis function* α categorizes each reference P according to a number of predefined features $f_1, \ldots f_m$. These features are computed by combining the title t, URL u, snippet s and eventually full-text information d of the document together with the values $w_{i1}, \ldots w_{in_i}$ of a query Q specified by the user and the results $r_{i1}, \ldots r_{in_i}$ of a set of orthogonal sources O. Each reference is then sorted into a *bucket* K_i that represents its particular feature combination $v_{i1}, \ldots v_{in_i}$ and is labeled accordingly. The buckets are arranged in an m-dimensional space, called a *DRS-Table*, with better values towards the origin. The Table is itself subdivided into a comparatively small number of *zones*, which combine buckets that have slightly different features, but the same overall quality (i.e. their feature differences are only of cosmetic nature). Once all references are sorted into the Table's buckets, the desired answer is found in the best zone that contains at least a single non-empty bucket. The best zone is either drawn from a predefined ranking of all available zones, or simply by their distance from the center of the table. Within this "best" zone, the content of the "best" bucket, that is, the bucket closest to the origin of the Table, is then displayed to the user, together with the label of its zone as a high level description of the output.

3.2.3 Learning

An opportunity that hasn't been discussed before is the ability of DRS-System to use techniques of machine learning to improve performance over time. The following paragraphs will explain the notion of *URL pattern extraction* and *URL generation*, and how these concepts can enable a DRS-System to find references even though they can't be found by using its standard base reference source.

A great number of personal assistants and Internet services learn about the preferences of a single user.[12] However, these approaches all demand a large number of examples given by each individual user, or, in the case of collaborative filtering systems a large number of users, before the system can acquire the data necessary for sufficient performance.

Instead of learning about user preferences of an often very limited user base, a DRS-System can harness the power of many users to learn about the Web itself (i.e. its structure). For each successfully located reference found, it extracts a general description of its location. Later, this can be used for directly locating pages that could not be found in the base reference set.

Since domain specific heuristics give a DRS-System enough added value even without any learning, it does not fall prone to the familiar problem of many machine learning systems, which perform only well once a sufficient number of people have used it, while many users resent using it until it exhibits a sufficient level of performance. As more and more users provide search examples, a DRS-System can instead incrementally add more and more expertise in its field, without requiring an initial number of users to suffer through bad performance for the sake of later performance improvements.

The learning process in DRS-Systems can be subdivided in two parts, which are described in the following paragraphs. The basic idea is to remember successful searches by *extracting* general patterns from correct URLs, and later using these patterns to *generate* new URLs which enable the DRS-System to find the correct page even though it could not be found in the base reference set.

URL extractor

Given a URL of a relevant reference and the according query parameter (*e.g.*, in the homepage domain, the person's firstname, lastname, and institution) a DRS-System tries to extract the *variant* parts of the URL, i.e. those that depend on the specific information the page is dealing with. For each domain, a specific extraction module has to be defined, which is able to analyze the URL looking for domain specific patterns and strings. Together with the current query parameters, and standard domain heuristics, the DRS-System can then create a pattern, with so called *placeholders* reserving parts of the URL that are query dependent. We call such an extracted pattern a *general hypothesis*.

Let us look at an example in the academic paper domain. The DRS-System successfully found a paper co-authored by Oren Etzioni at the following address:

`http://www.cs.washington.edu/research/softbots/pub/etzioni/cacm94.ps`

Using domain specific heuristics together with the query parameter, the DRS-System is able to extract the following pattern:

[12] *e.g.*, WebWatcher [AFJM95], Syskill & Webert [PMB96], Mitchell's Calendar Apprentice [MCF+94] or Lang's Newsweeder [Lan94].

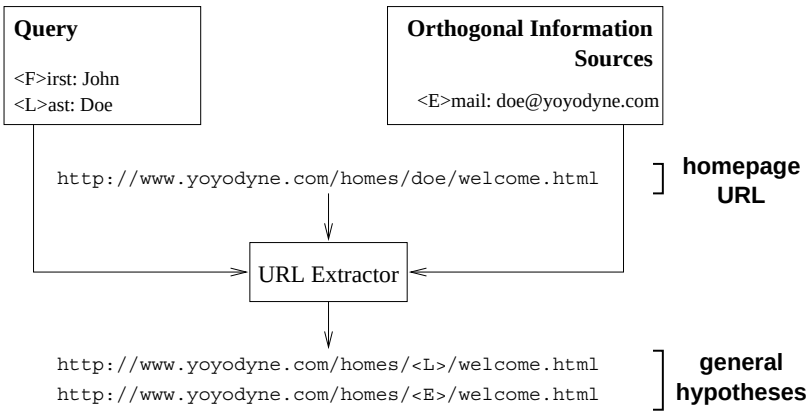


Figure 3.3: **URL Extractor.** Using the URL of a successfully located homepage, together with information from an orthogonal reference source (email directory) and the query itself, the DRS-System generates two *general hypotheses* by substituting the *placeholder* `<L>` and `<E>`, representing lastname and email-username, for the string `doe`.

```
http://www.cs.washington.edu/research/softbots/pub/<L>/<F>.<S>
```

`<L>`, `<T>` and `<S>` are *placeholders* for the URL's *variable* parts of the authors lastname, the papers filename, and suffix, respectively. The rest of the pattern is called the *fixed* part, and is assumed constant for other elements of the reference set (i.e. academic papers, in our example) that share common parts (such as the institution). Figure 3.3 shows another example from the personal homepage domain. In the next section we will explain how these patterns can then be used to increase both recall and precision in subsequent searches.

URL generator

The URL generator part is responsible for applying the extracted patterns in case the DRS-System could not find any promising references in the base reference set. The same domain heuristics that enabled it to map the variable query parameter appearing in the original URL to general *placeholders* can now be used to *reinstantiate* a full URL from the general pattern and the new, yet unsuccessful query.

To continue with our previous example, another user might try to locate Dan Weld's "Least-commitment planning" paper. Given the query parameter `firstname = dan, lastname = weld` and the knowledge that Dan Weld is a professor at the University of Washington (`instname = university of washington`), a DRS-System could generate the following *instantiated hypotheses*:

```
http://www.cs.washington.edu/research/softbots/pub/weld/lcp.ps
http://www.cs.washington.edu/research/softbots/pub/weld/lcp.html
http://www.cs.washington.edu/research/softbots/pub/weld/lcp.pdf
```

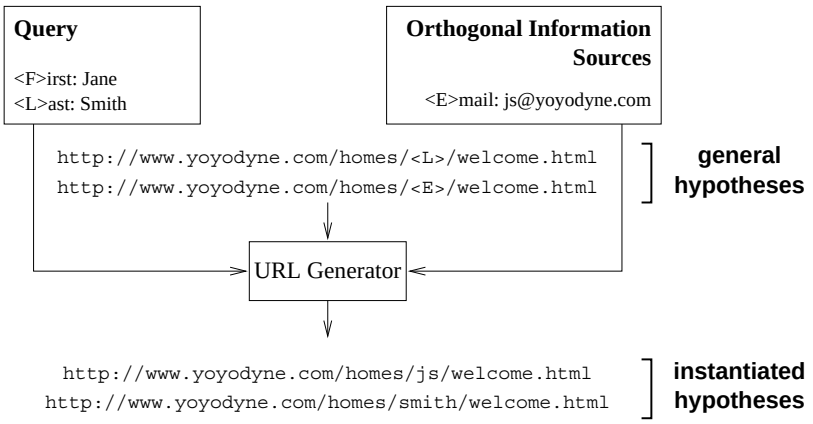


Figure 3.4: **URL Generator.** Using previously acquired general hypotheses, together with information from an orthogonal reference source (email directory) and the query itself, the DRS-System is able to generate two *instantiated hypotheses* by substituting current query values of lastname and email-username for the placeholders `<L>` and `<E>`. Note how the single homepage URL in figure 3.3 has led to two instantiated hypotheses.

The reader might object that it is highly unlikely that we could locate any paper this way, for although lastname and file-suffix provide a reasonable small space, the mapping from paper title to filename proves to be too arbitrary. Since the exact mapping from the single, extracted *general hypothesis* to a list of possible *instantiated hypotheses* is dependent on the domain specific implementation, we might in the case of academic papers refrain from generating direct URLs that point to specific papers, but choose to only generate index URLs like

```
http://www.cs.washington.edu/research/softbots/pub/weld/
```

in order to get a default index file of Weld's paper directory, which might then be searched for a link to the correct link.

However the final mapping from *general* to *instantiated hypothesis* turns out to be implemented, the usage remains the same: In the case of a failed search, a DRS-System is able to directly search for the desired page by using successful searches from the past in the form of *general hypotheses*, which are then mapped to a number of *instantiated hypotheses*. Figure 3.4 shows this process for the example from figure 3.3.

Pattern selection

One final question remains, though, and that is how a DRS-System can select the hypothesis to instantiate if more than one *general hypotheses* has been extracted previously.

A first selection is made by determining the correct institution to start a direct search at. For personal homepages or academic papers, this might either be obvious from the given query (where the user can explicitly specify the institution she believes stores the desired information), result from additional information through an orthogonal reference sources (like an email address indicating the company the person is working for), or stem from direct user soliciting (i.e. prompting for selecting a site to search at). In other domains we might notice a common repository address for many successful searches, and thus choose a small number of possible repositories to search based on the above mentioned information (i.e. query information, orthogonal sources, or user soliciting).

Once a list of institution and/or departments/server have been established, the list of known hypotheses at these sites are ordered according to statistics regarding successful and unsuccessful applications of each single hypotheses in previous direct searches. Statistics could simply be keeping track of which hypothesis has been most recently successful in locating a personal homepage, or they could include the number of times a particular hypothesis has been successfully applied, or even take into account how long ago it was that is had been successful.[13]

When examining these statistical values, a DRS-System can make use of the hierarchical nature of the Web and solicit information higher up in the hierarchy in case the individual server data is not sufficient. For example, should the data gathered for hypotheses on the server `www.cs.washington.edu` not provide enough confidence for imposing a ranking among the different hypotheses, we might examine data gathered among *all* servers within the CS department. Should still be data missing, we could solicit the performance of all hypotheses across the University of Washington, or even educational sites in general.

3.2.4 Sample domain implementations

Tables 3.3 and 3.4 give brief examples of each DRS element for the six domains listed in section 3.1.2. A more detailed account of a particular domain, personal homepages, is given in the next chapter, while chapter 5 describes two additional, brief example implementations for academic papers and jokes.

3.3 Summary & Discussion

The World Wide Web is an ever growing, vast information space. When trying to find specific information on it, intelligent agents in general, and softbots in particular, can significantly reduce the amount of effort needed on behalf of the human. Softbots help people using the Internet and Unix by using the same tools as human do. Using softbots as *information agents*, we propose a new kind of softbot called *Dynamic Reference Sifters* (DRS-System).

[13] Section 4.3.5 describes an experiment that contrasts a number of possible measures.

Element	Transportation	Product reviews	Business homepages
Query	Origin, destination, mode of transportation, transportation company, date of travel, route number/identification.	Company name, location, field, product.	Product title, manufacturer, category.
Base references	Series of Web indices queries for origin and destination, along with mode of transportation.	Search for company name, eventually grouped as a phrase.	Query for product title, including standard review words like "review" or "description".
Orthogonal references	Geographical database, which would allow associating cities and regions with countries and neighboring cities. Database of transportation corporations world wide.	Yellow pages or investment services could provide postal address or telephone number.	If category given, use Web directories review section for list of sites that offer reviews in this category. If a manufacturer given, look up homepage using an institutional database.
Heuristics	Page contains table like structure; origin and destination occur on page; times and route numbers appear on page.	Name of the company featured in either title or content of the document or (eventually in abbreviated form) in URL; copyright notices, company terms (such as "Inc" or "Co"), high number of images or imagemaps on page; page URL close to server root.	Contains the product name and eventually 'review keywords' (like "review") in either text or URL; product title appears in the title, or abbreviated in the URL.
Bucketing	Degrees of matching for country, origin, destination, mode and transportation provider, 'timetable' likelihood.	Degrees of matching for country and server name, product and company name, and location in servertree (distance to root).	Product name match, category match, "review"ness, location (at known reviewers site, at manufacturers site).
Learning	City-to-provider associations, geographic associations, Web page organization on a provider to provider basis.	Cache company homepages, compile own list of companies on the Web.	Review repositories, Reviews per product.

Table 3.3: **Implementation Samples.** This table lists the domain examples given in section 3.1.2 together with possible implementations of the generic DRS-System features.

Element	Personal homepages	Academic papers	On-line jokes
Query	Firstnames, lastname, institution, email address, country.	Paper title, authors, institutions, subject keywords.	Words, phrases remembered from the joke.
Base references	Query containing only the persons first- and lastname to a Web index.	Search for author and title (as a phrase). In addition, search for links featuring the paper title, as well as pages with the paper title as the document title	Query for phrase and keywords of joke, along with "joke" words, such as *humor* or *fun* in text, title or URL.
Orthogonal references	A database of institutions supplies a list of sites at which the homepage could be located, in case the query included affiliation information. Email whitepage services are queried with the persons name and supply possible login names that might appear in the URL.	Query database (i.e. INSPEC, MEDLINE) to obtain the principal authors affiliation or co-authors. Institutional database supplies URL information for this institution. Homepage Finder DRS-System can supply authors personal homepages.	None
Heuristics	First- and lastname appearing in the full text, the title, or the URL of the reference; Analysis can handle nicknames, abbreviations, titles (i.e. Mr.) and additional firstnames.	Author names near title string in document. Paper title in document. Author names, publication keywords (i.e. 'research', 'publication') in URL path.	Keywords close to phrase, or within paragraph; "joke" words present.
Bucketing	Degrees of matching for country and institution, as well as name match and 'homepageness'.	Degrees of matching for the institution, the number of authors found, the title match, the proximity of title and authors to a link, and the type of page linked to (PS, PDF, HTML format).	"Joke" words in URL, title, text; phrase in title text; number of keywords in title text; closeness of keywords and phrase.
Learning	Mapping from given name to URL, on an institution per institution basis.	Discover paper repositories on an institution basis, as well as for certain areas, or on a per author basis.	Joke repositories, phrase/keywords-to-joke mappings.

Table 3.4: **Implementation Samples (continued).** Implementations for the three domain examples listed here are discussed in more detail in chapter 4 and 5.

DRS-Systems and their key elements

DRS-Systems have six key elements — a base reference source, a cross filter using orthogonal information sources, heuristic filters, buckets, and URL extractor and generator – that fall into three main areas: Information gathering, filtering, and learning.

A DRS-System gathers information from standard Web indices (this forms the *base reference set*), which is augmented with additional data from orthogonal sources (*orthogonal reference set*) that allow *cross filtering* of irrelevant references.

Then it uses *domain specific heuristics* to asses a number of predefined features for each element in the base reference set, sorting it into labeled *buckets*, with each bucket representing a distinctive feature combination. The buckets are arranged in an n-dimensional space, called a *DRS Table*, along the axes formed by the different features analyzed. By collating these buckets into *zones* of various degree of 'quality', a DRS-System is able to present the user with a very precise, labeled answer, always returning the best available information and providing alternatives in case certain information was not what the user was looking for.

Finally, a DRS-System is able to remember successful searches using a *URL Extractor*, which uses the query parameters, information from orthogonal information sources and the URL of successful pages to generate *general hypotheses* that contain *placeholders* for the variable parts of the address. In case a later search should fail, the DRS-System can then use a *URL Generator* to create *instantiated hypotheses* by replacing the placeholders in general hypotheses with current query parameter and/or available orthogonal information.

DRS Set Characteristics

DRS-Systems are useful in domains that exhibit a large number of highly dynamic member pages, which are at least partially indexed by standard Web search indices. During any given search, a user is typically interested in a single element of this set. Examples for suitable domains include personal homepages, academic papers, product reviews and transportation schedules.

DRS Design Goals

Using a standard search service, a user can usually vary the *granularity* of her search using keywords and other additional query syntax. It is this choice of query words and terms that lead to the familiar tradeoff between coverage and accuracy: Formulating the query overly restrictive (i.e. including a large number of keywords, and using phrases and conjunctive clauses) will help retrieving only few *irrelevant* references, but may also miss many of the *relevant* ones. Using too few restrictions (i.e. using only a small number of keywords and allowing disjunctions) will have a higher chance of including most, if not all of the *relevant* ones, but also cluttering the results with countless *irrelevant* entries.

Finding exactly the right combination of keywords and query terms that will include all *relevant* references while excluding all *irrelevant* ones, is the pinnacle of automated search. But instead of trying to vary these query parameters to achieve optimal performance, DRS uses a more pragmatic approach: Starting with an overly *inclusive* query, we first make sure that most, if not all of the *relevant* references are retrieved. Then, *sifting* through all of the references one by one, we sort each reference into "buckets" containing references with identical features. Once all of the references have been categorized this way, it is simply a matter of locating the bucket that contains the most promising feature combination and presenting its content to the user, together with a corresponding label describing the particular feature combination.

This strategy ensures the three major design goals: The comprehensive reference set ensures *high recall*, the domain specific labeling and sorting methods allow *high precision*, and the flexible sorting scheme allows for *graceful degradation*, making sure that the maximal relevant information is always presented.

Offering both high precision *and* high recall is a good example for a technique where the total is more than the sum of its single parts: Not only is the user presented with *fewer* references (high precision) *more* often (high recall), she also saves a tremendous amount of time once such a technique *fails* to find a reference. Given the premises, the user can have a high confidence that the desired information is indeed not on the Web, leaving time to plan the next step, instead of spending more and more time establishing the non-existence of this information. Furthermore, by displaying additional information even if the main piece of information could not be found, a DRS-System can also add explicit value to a failed search, providing important clues for the users further actions.[14]

[14] *e.g.*, contacting the author of a unfindable paper, using the displayed email address.

4 Ahoy! The Homepage Finder: A case study in the homepage domain

This chapter describes our DRS case study in the personal homepage domain, called Ahoy! The Homepage Finder. Ahoy! is a fully implemented Web service that is publicly accessible since May 1996 and has handled over half a million queries since.

Building upon our discussion of the general DRS framework in the previous chapter, we will introduce our domain specific implementation in 3 steps. First, we will describe the domain this particular DRS-System is operating in. Then we will proceed with the historical development and module overview of the system, where we present the implementation of the three main components of a DRS-System – Information Sources, Analysis & Classification, and Learning – as well as the Input & Output of the system, and discuss implementational aspects and performance issues. Finally, we will report the results of our experiments, which compare the accuracy and coverage of Ahoy! with four popular Web search services, and examine the effect of three different statistical measurements on Ahoy!'s learning module.

4.1 Finding personal homepages on the Web

Before we set out to give a description of our domain specific implementation, we will have to describe the domain it is operating in. We will first describe the concept of a *personal homepage*, and then briefly summarize how the three search methods described in section 2.2 – Web directories, Web indices, and manual search – can be used to find pages in this particular domain.

4.1.1 Domain description

The initial concept of a homepage was that of a personal "gateway" to the Web: The first "browser" applications, which allowed a person to explore the web of interconnected pages with a simple click of a mouse (*i.e.*, to *browse* the Web) offered the installation of a "home page", a hypertext[1] document that would be the very first document displayed after the application was started [NCS97].

Today, many different types of homepages have begun to supersede this initial definition. However, the concept of a "gateway" remains the same: Business home-

[1] Any text that contains links to other documents. See the glossary in appendix A or [NCS97].

pages provide links to press releases and product descriptions; University and Departmental homepages offer degree information and course lists; Class homepages cover additional reading material and past midterms; while Project homepages list publications and experimental results.

Maybe the most popular type of hompage is that of a *personal homepage*. Personal homepages offer starting points to explore a person's work, hobbies, friends and family. With the sharp rise in the number of commercial Internet Access Providers, personal homepages are not only for researchers and college students anymore: thousands of individuals use it to promote their private business, exchange favorite recipes or put up pictures of latest vacation for friends and family. Offering a search service that would reliably find specific pages of this category would provide an immediate benefit to a large number of users, addressing our initial goal of building *truly useful systems*.

4.1.2 Current Methods

This section describes the three most commonly used methods for finding personal homepages. Each method uses one of the search strategies described in section 2.2, and we will specify briefly how they can be used in the personal homepage domain.

Web Directories

Some Web directories such as Yahoo! have attempted to create directories of homepages by relying on users to register their own pages. As of June 1997, Yahoo! contains about 75,000[2] personal homepages. It is difficult to say how many personal homepages are on the Web, but it is clear that Yahoo!'s list represents only a small fraction of the total. For example, it contains only between one and ten percent of the homepage samples used to test Ahoy!, and less than two percent of the roughly 30,000 personal homepages created by Netcom subscribers [SLE97]. The result is that for the majority of all searches, looking up a personal homepage in a directory will result in a "No entries found" message.

Web Indices

Many general-purpose indices like AltaVista and HotBot make query syntax available that is tuned to find people. This approach to finding personal homepages avoids the problems of manually creating a list, but the output of such searches frequently contains an inconveniently large number of references. For example, searching AltaVista for a person named Oren Etzioni using the (advanced) query "Oren NEAR Etzioni" returns about 300 references. A similar search using HotBot's specialized "People Search" produces over 700 matches. A separate problem is that many users do not bother to learn such specialized query syntax and thus conduct an even less precise search.

[2] See `http://www.yahoo.com/Entertainment/People/`.

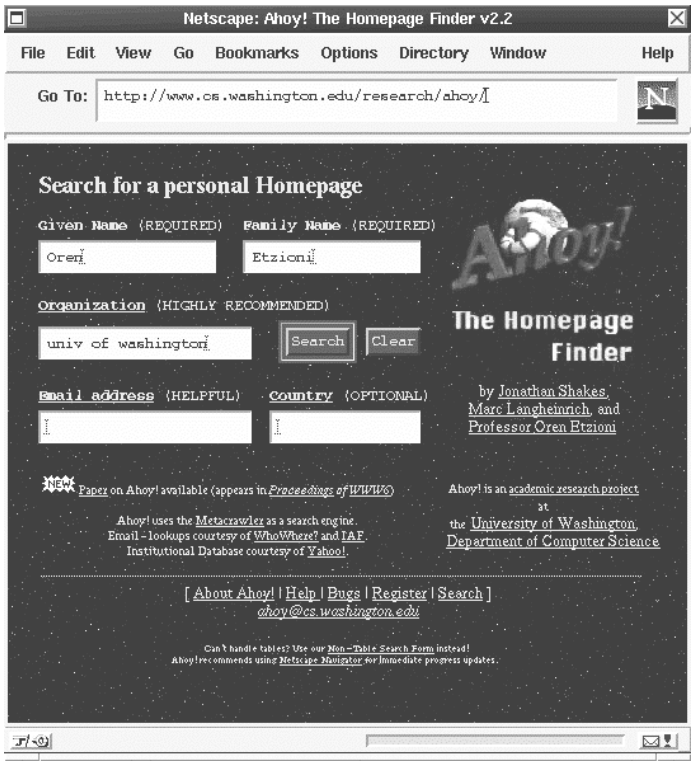


Figure 4.1: **Ahoy! Search Form.** The user is asked for the first- and lastname of the person she is looking for. Additionally, the institution this person is affiliated with, as well as any (partial) email and country information available can be entered. The latter fields are optional and help Ahoy! increase its accuracy, but without limiting its coverage.

Manual Search

Knowing enough about a person, one could find his or her homepage by first finding the Web site of the person's institution, then possibly searching down to the person's department, and finally locating a list or index of homepages for people at that site. Unfortunately, this method can be slow. If, for example, one were looking for a biologist named Peter Underhill at Stanford University, several minutes might be spent looking through Web pages of dozens of departments that might reasonably employ a biologist.

4.1.3 Using Dynamic Reference Sifters

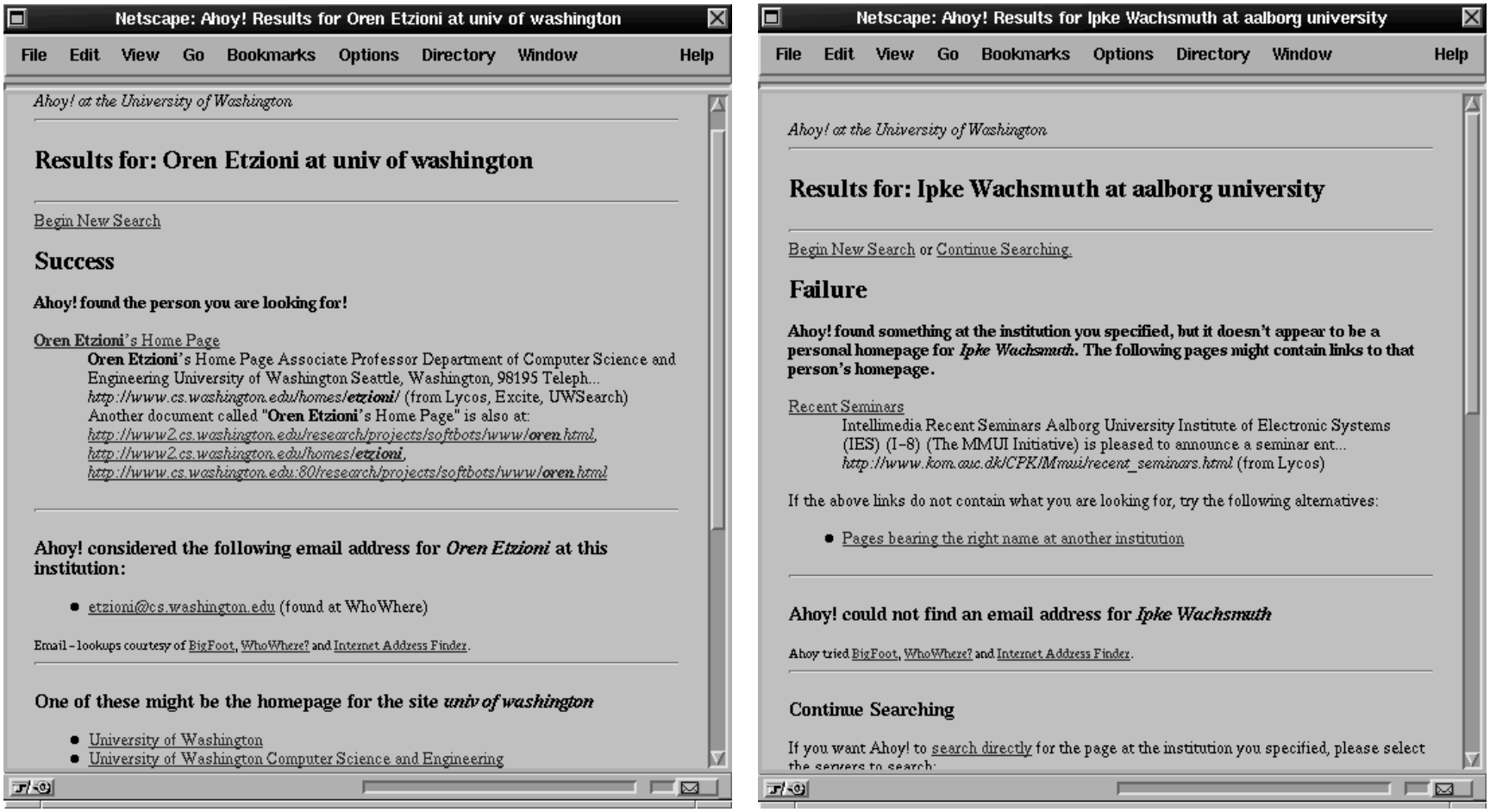
The Ahoy! softbot represents a fourth way of searching for personal homepages. A snapshot of the Ahoy! search form is shown in figure 4.1. The user enters the firstname and the lastname of the person she is looking for, and additionally the institution the person is affiliated with (*i.e.*, which is most probably storing the homepage). Also, the search from provides fields for the email address and the desired country

Figure 4.2: **Ahoy! Status Report.** Addressing the design goal of *reactivity*, Ahoy! provides a detailed status report during its search, even allowing the user to directly use any preliminary results by following a hyperlink without waiting for the final results.

of the person. Giving additional information to Ahoy! does increase its accuracy, but without limiting its coverage. Standard search engines often fall prey to over restrictive searches when too much, or accidentally wrong information is given.

After submitting the query, the user is presented with a detailed status report, (see figure 4.2) which already contains preliminary information about possible homepage candidates. This way, an impatient user, who recognizes the brief reference as it comes in, can immediately follow the link without waiting for Ahoy! to finish its search.

Finally, the answer, along with a brief search summary, is presented on a single page (see figure 4.3(a)), typically featuring less than 5 references, in most cases only one or two. Additional information like the institutional homepage or the email address of the person search for is given, in case none of the returned references turn out to belong to the right person. Also, in case Ahoy! could not establish sufficient confidence in the references to declare it an unambiguous success, it will try to present the best possible answer, along with a number of alternatives, as shown in figure 4.3(b).



(a) Success

(b) Failure

Figure 4.3: **Ahoy! Results.** In case Ahoy! could unambiguously find relevant references, it displays those together with a brief description and any additional information (email addresses, institutional homepages) available (a). If only a suboptimal result could be found, Ahoy!'s *graceful degradation* mechanism allows it to label the presented references as such, and offer pointer to possibly related references in other categories.

With its novel approach, the Ahoy! softbot combines the advantage of manually created Web directories – their relevance and reliability – with the advantage of general-purpose Web indices like AltaVista– their enormous pool of indexed pages. In fact, due to the *URL generator*, Ahoy! is able to find and return homepages that are not listed in *any* search index. Finally, Ahoy! provides the advantage of speed: when it returns a negative result (*i.e.*, it reports that it cannot find a given homepage), it saves its users from scanning through tens or hundreds of "falsely relevant" references returned by a general-purpose search engine. Last not least, Ahoy! returns its results much faster than a manual search would.

The next section will examine the Ahoy! architecture and implementation in more detail, describing more closely its control flow and module structure.

4.2 The Ahoy! Web Service

We will begin this section with a short history of the Ahoy! project, and how it lead to the design of a DRS. Several early architectures will be presented and their problems discussed. We then go on to give an overview of Ahoy!s current module structure, and describe the flow of control for a number of common situations.

Our three DRS areas introduced in section 3.2 – information sources, filtering (analysis and classification), and learning – will provide the structure for our description of the Ahoy! modules, with two extra sections concerning Ahoy!'s input and output operations.

4.2.1 History and Overview

Early implementations

The Ahoy! softbot project started in summer of 1995, in response to a challenge posed by UW Computer Science Departments chairman Ed Lazowska: Building a softbot to find personal homepages. The author joined graduate student Jonathan Shakes and professor Oren Etzioni in September after a number of approaches had been discussed, as outlined in table 4.1.

The most promising approach this far had been the *best-first search*: Given the name, institution and departmental affiliation of a person, an early prototype would follow the following four steps:

1. Look up institutional homepage in existing list and download it.
2. Find link that leads to departmental homepage and download it.
3. Find link that leads to homepage list page and download it.
4. Find link that leads to personal homepage and show it to user.

Approach	Description	Problems
Centralized database User-created	Users register their homepage manually	How to achieve initial size to make service attractive? How to ensure accuracy of the database? How to prevent bogus entries?
Centralized database Spider-created	Use specialized Web-Crawler which indexes only homepage	How to achieve large enough coverage? How to unambiguously decide if a page is a homepage?
Centralized database Assembling smaller databases	Periodically copy existing homepage databases and assemble into centralized database	Often impossible to download all existing entries, unless special arrangement with local maintainer. Labour intensive to add new services. Bandwidth consuming to update entries.
Guess correct URL	Use institution name to "guess" institution's URL and homepage path until correct one is found, using previously learned patterns.	Bandwidth & time consuming.
Best-first search	Use institutional URL as root of a search tree, then perform heuristic best-first search.	Hard to code universal evaluation functions that chooses next link to explore. Time consuming. Depends strongly on initial guess about correct institution to search.
Specialized query to standard Web indices	Create specialized query to standard Web indices.	Easy to over- or underconstrain query (*i.e.*, too few or too many results). Multiple iterations of query string ('tuning') takes long time.
Specialized Meta-Search	Compile list of directories or indices that specialize in people (*i.e.*, people at a certain institutions or working in a certain field). On basis of input information, choose correct sources to query.	Tedious to compile and maintain list. Needs large number of specialized "wrapper" that decode responses of each information source (makes maintenance even harder).

Table 4.1: **Initial Architectural Approaches for a Homepage Finder** Before developing the general DRS-framework described in chapter 3, a number of approaches had been tried to create a homepage finder.

At any point, the system would keep track of the links seen so far, including their likelihoods for leading to a specific page type searched for by the program – departmental homepages, list pages, and personal homepages. Performing a depth-first search, it would analyze each link and its surrounding text, asses likelihoods for the three page types, and follow them in the order of highest probability. Depending on the state of the system (*i.e.*, trying to find the departmental page, the list page, or the homepage itself), it would use one of the three values as the rank indicator. In case all values on a current page were worse than any other values encountered before, the system would backtrack to the best seen link so far and keep exploring from there on.

Although the system showed a surprisingly well initial performance, its limitations soon became obvious, when it should be expanded to handle arbitrary homepages. The initial prototype had specialized in finding personal homepages of computer science professor at North American universities, and already for the next small improvement – finding professors at North American universities in general – creating the necessary heuristic filter to asses the likelihood values became increasingly difficult: How was the system to find the departmental homepage of the botany department? Was it to look straight for "botany", or should it look under "biology" first? And what if all these disciplines where grouped under the "natural sciences" pages? Soon, the need for a general hierarchy of departments became obvious.

Also, with the introduction of a wider range of possibilities to explore (instead being focused on computer science departments), the chance for following irrelevant links increased rapidly. The more often the system had to backtrack in situations like these, the more time it took to find the page.

While this initial approach had been highly motivated by user studies, who would find homepages in the very same way (*i.e.*, finding the persons institutional homepage, and the searching from there), the vast general knowledge used by people during these searches was crucial to the success of the system.[3]

Our research focused then on a less knowledge intensive task, which was nevertheless also motivated by user studies: While "expert" users would seem to prefer our first methods, many occasional users reported simply issuing a query for the persons name to a popular search engine that supported keyword searches, and then looking through the returned references one by one until the desired page was found.

The work on this second prototype, together with parts of some of the initial approaches shown in table 4.1,[4] finally led to the general design of Dynamic Reference Sifters, inspired by user's manual and often laborious *sifting* through tens and hundreds of references.

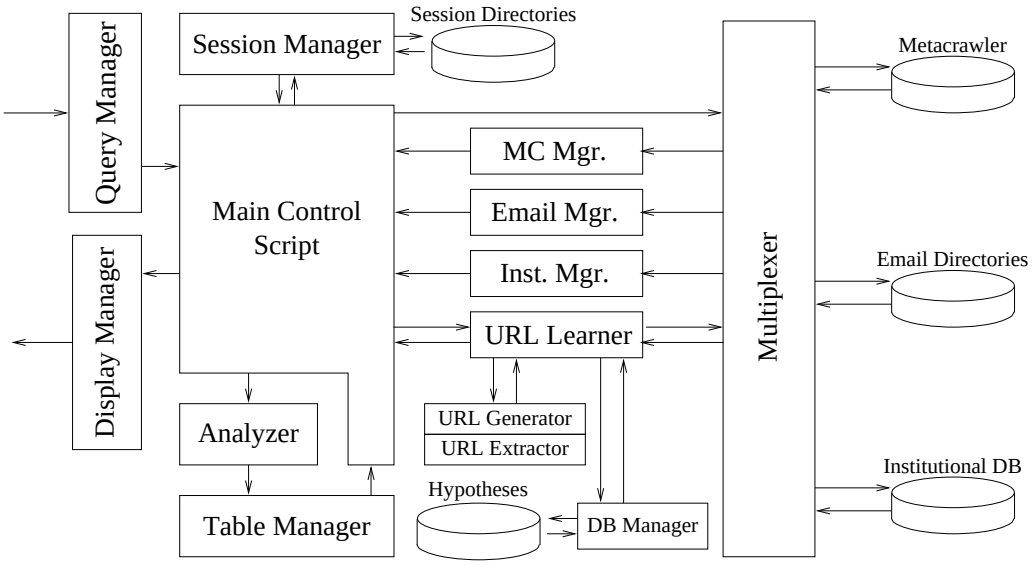


Figure 4.4: **Ahoy! Module Overview** More than a dozen main modules compromise the Ahoy! System. Query and Display modules handle *input* and *output*; MC, Email and Institution modules query and parse Ahoy!'s information sources; the Analyzer and Table modules handle *Analysis & Classification*; and URL Generator and Extractor provide Ahoy!'s *Learning* capabilities.

Module overview & Control Flow

Figure 4.4 shows the architectural overview of the parts that compromise the Ahoy! system. Ahoy!'s main modules can be divided into the three areas of a DRS-System: Information Sources, Filtering, and Learning. In addition, the Ahoy! system contains modules for reading a user query (*input*) and giving back a response (*output*). Table 4.2 lists each of these areas with its corresponding modules.

The flow of control in Ahoy! is similar to the general DRS mechanism first shown in figure 3.1: After parsing the users request, Ahoy! issues an appropriate query to the MetaCrawler, as well as to each of its supporting email services (WhoWhere, Internet Address Finder and BigFoot). Together with information from its internal institutional database, the returned references are then filtered and classified as described for the general DRS-System in section 3.2.2. Depending on the results, any relevant references are shown to the user (after Ahoy! extracted any new *general hypotheses* from them), or it uses its database of previously acquired patterns to generate a list of *instantiated hypotheses* that will then be tried directly.

The following sections will briefly describe each module in the order shown in table 4.2. Most modules represent a single object that will be created and used

[3]See chapter 5 for a brief description of the WebFinder system, which uses a very similar approach, and a discussion of its performance.

[4]Specifically, the "Guess correct URL" approach led to the concept of URL Generator and Extractor, while the "Specialized query to standard Web indices" approach provides the base reference set.

1. Input	**4. Output**
Query Manager	Display Manager
Session Manager	**5. Learning**
2. Information Sources	URL Learner
MC Manager	URL Extractor
Email Manager	URL Generator
Inst DB Manager	**6. Miscellaneous**
3. Filtering	Main control script
Analyzer	Multiplexer
Table Manager	DB Manager

Table 4.2: **Ahoy! Modules by Category:** Ahoy!'s main modules can be divided into 5 areas: input, output, information sources, filtering, and learning. In addition to the modules shown here, Ahoy! has a number of auxiliary modules which are listed in appendix B.3.7.

in the Main control script, although others might simply provide functions or more complex data types. A number of smaller program parts not shown in figure 4.4 are also briefly described in the appendix.

4.2.2 Ahoy! Input: Searches and sessions

This section gives an overview about Ahoy!'s frontend: The query decoding module and the persistent session management.

Query decoding

The Query Manager enables Ahoy! to decode requests coming from a users browser when the Ahoy! script is called by the local Web server. Ahoy! knows the following five fields:

`firstname`	The first names of the person
`lastname`	The last name of the person
`instname`	The name of the institutional affiliation
`country`	The country of the institution
`email`	Email address of the person (or parts of it)

Before offering the values of these fields to the rest of the system, the query module first asserts that the user entered sufficient information to provide an answer. Ahoy! knows 3 different modes of operation, which are set by the query module according to the fields the user filled out:

1. **Name-only search:** `firstname` and `lastname` given, but no institutional information. Ahoy! tries to find a homepage at any institution.[5]

2. **Institution-only search:** User specified `instname` only, without giving either `firstname` or `lastname`. Ahoy! simply tries to locate the corresponding institutional homepage in its database without querying any external sources.

3. **Name & institution search:** `firstname`, `lastname` and `instname` given.[6] In this mode, Ahoy! will try to locate a relevant page at the correct institution.

After decoding the information from the user and determining the mode of operation, the query module returns control to the Main control script, where either an error message is displayed (in case of invalid fields),[7] or execution continues with the creation of a Session Manager object.

Session creation

Each search with Ahoy! is called a "session". Every input to the main search form will always start a new session, even if exactly the same information has been entered before. Only searches that use Ahoy!'s URL Generator (after an initial search using the *base reference set* has failed) will *continue* an existing session.

Each session uses a private directory, where session dependent files are stored. The list of files per session include the original output of the *base reference set*, the MetaCrawler (for later inspection by the user); result and alternatives pages; status pages; and email information pages (see table 4.3).

File	Description
`results.html`	Results page shown to user
`status.html`	Copy of status report, for reference
`alt_*.html`	Alternative results, if any
`previous.html`	Previous results in case of a continued search
`email.html`	Separate email page, if more than 3 addresses found
`session.data`	Statistical information
`url.data`	Information about URLs followed

Table 4.3: Contents of Ahoy! Session Directory

In order to be able to continue a session after an answer has been sent to the user, Ahoy! encodes a unique session identifier in the results page. Once the user submits a request to continue this search (in case it failed before and Ahoy! has at least one

[5] If any `country` information is given, Ahoy! will first try to restrict its search for domains in this country only.

[6] Alternatively any institutional information from the `email` field will be used, if available.

[7] Currently Ahoy! does not handle initials for either first- or lastname, nor does it allow searches for firstnames or lastnames only.

general hypothesis to locate the page directly), this session identifier is sent back to Ahoy! and can be used to look up the current state of this particular search in the corresponding session directory.

4.2.3 Information Sources

Each of the three information sources manager, MC Manager, Email Manager and Inst Manager, provides the Main control script with an appropriate query string to the source (given the current user request), as well as containing methods to decode the information that is sent back from each source.

The Main control script simply submits each query string to the Multiplexer module, which allows the parallel connection to a number of remote and local sources (see section 4.2.7), and then translates the incoming results into Ahoy!'s internal data structures using each managers decoding methods.

The MetaCrawler: The base reference set

To achieve a large *base reference set*, the Ahoy! query to the MetaCrawler simply submits the first- and lastname of the person. In addition, it specifies the "as a person" switch, which instructs the MetaCrawler to use this feature with all search services that support it.

Depending on net traffic and timeout values, the MetaCrawler usually returns between 40 and 120 references, which are immediately converted into Ahoy!'s internal reference format, analyzed and sorted into a *DRS-table* (as soon as sufficient information from orthogonal sources is available).

Orthogonal information: The Institutional Database

The *institutional database* consists of a local copy of the Yahoo! index for companies, universities, as well as governmental and military sites. Using the freely available `glimpse` package,[8] the Institution Manager allows keyword searching on the entries consisting of the *institution name*, its principal *URL*, as well as any optional *nicknames*, such as "UW" for the University of Washington.[9]

Instead of providing a single query string, the Institution Manager returns a *list* of queries to this index, arranged in descending degree of restrictiveness: First, an exact match for the information entered in the `instname` field is tried, which is then gradually loosened (*i.e.*, allowing different word order, spelling mistakes, etc), until at least a single match could be found.

[8] Glimpse (which stands for GLobal IMPlicit SEarch) is an indexing and query system that allows searching through a large number of local files very quickly. See the glimpse home pages in `http://glimpse.cs.arizona.edu/`.

[9] These nicknames currently need to be entered manually.

Once a list of matches is available, the Institution Manager uses filtering methods to find the best match in a possibly large list of candidates.[10] Filtering constraints include the correct country of the institution, as well as a metric for the closeness of the institution name and its nickname.

Orthogonal information: Email Directories

Ahoy! uses information from the three largest email directories available: Bigfoot [Big97],[11] Internet Address Finder [Dou97], and WhoWhere [Who97].

The queries to each email service are similar to those to the MetaCrawler, simply stating first- and lastname of the person. After the decoding method has extracted all available email references in the output of the services, the Main control script tries to unify this list of possible email addresses with the list of institutions obtain by the Institution Manager. This step is described in the next section.

Cross filtering of orthogonal information

Before analyzing the references returned from the MetaCrawler, the Main control script tries to determine the "correct" institution, based on the information obtained from both the Institution Manager and the Email Manager.

Ahoy! tries to "synchronize" these two sources by finding overlaps between email addresses and institution URLs, keeping only those emails and institutions that have the highest possible overlap. An example is given in figure 4.5. The overlap is later used during filtering to decide if references from the MetaCrawler are at the correct institution (see section 4.2.4).

4.2.4 Filtering: Heuristic Analysis & Classification

Ahoy!'s filtering is done by the Analyzer module and Table Manager. The Analyzer computes the relevant features from each reference, while the Table Manager implements a *DRS-Table* with the appropriate axes and zones.

Feature detection: The Analyzer

A Web document usually features a number of common characteristics, which are used by the Ahoy! system to determine whether a particular reference constitutes a personal homepage for the desired person.

The following three features are examined in order to characterize a reference as being a personal homepage for a certain individual:

1. **Ownership:** The degree of match between `firstname` and `lastname` (as specified by the user) and any personal name featured prominently in the reference.

[10] Once the first queries using exact matches have failed, a relaxed query allowing for example single spelling mistakes can easily return several hundred matches.

[11] Bigfoot was recently substituted for the much larger OKRA service [DoCS96], a student operated project at the University of California at Riverside, as the latter one had to be shut down due to lack of support (the ultimate fate of Ahoy! :().

Code	*Name*	*Description*
`M`	with the right name	In title, full match of name (all user specified parts found). In URL, found possible login name (as inferred from emails) at correct position.
`P`	with a similar name	Some user specified parts (*i.e.*, second firstnames) could not be found, but at least one first and lastname was found.
`S`	with a similar sounding name	Ahoy! uses the Soundex algorithm [Knu73] to match lastnames in spite of spelling mistakes (although this algorithm only works with English names). This is only useful if the MetaCrawler found a reference even though the user misspelled the name, which is only likely with very unique name combinations.
`L`	with the same lastname	Found only the (correct) lastname, but no firstname. If also a non-matching firstname is found, see class `U`.
`R`	with a similar sounding lastname	Same as class `L`, but using the Soundex algorithm for matching the lastname.
`A`	with a possible login name	Found a "generic" login name (*i.e.*, generated from first- and lastnames combinations) in URL (not used in title analysis).
`F`	with the same firstname	Found only one or more correct firstnames, but no lastname. If also a non-matching lastname is found, see class `W`.
`E`	without the name	Could not find any part of the specified name in title or URL. This does not necessarily mean that there is no name at all, only that no part of the desired name could be found.
`U`	undecided	This is the fall-back code, whenever none of the other cases could be found.
`D`	with a different firstname	Found correct lastname, but preceded with a non-matching firstname.
`W`	with a wrong lastname	Found one or more correct firstnames, but also a non-matching lastname.

Table 4.4: **Ahoy! Analyzer: Ownership codes.** Ahoy! uses a list of 10 ownership codes to categorize a given reference as bearing the name of the right person.

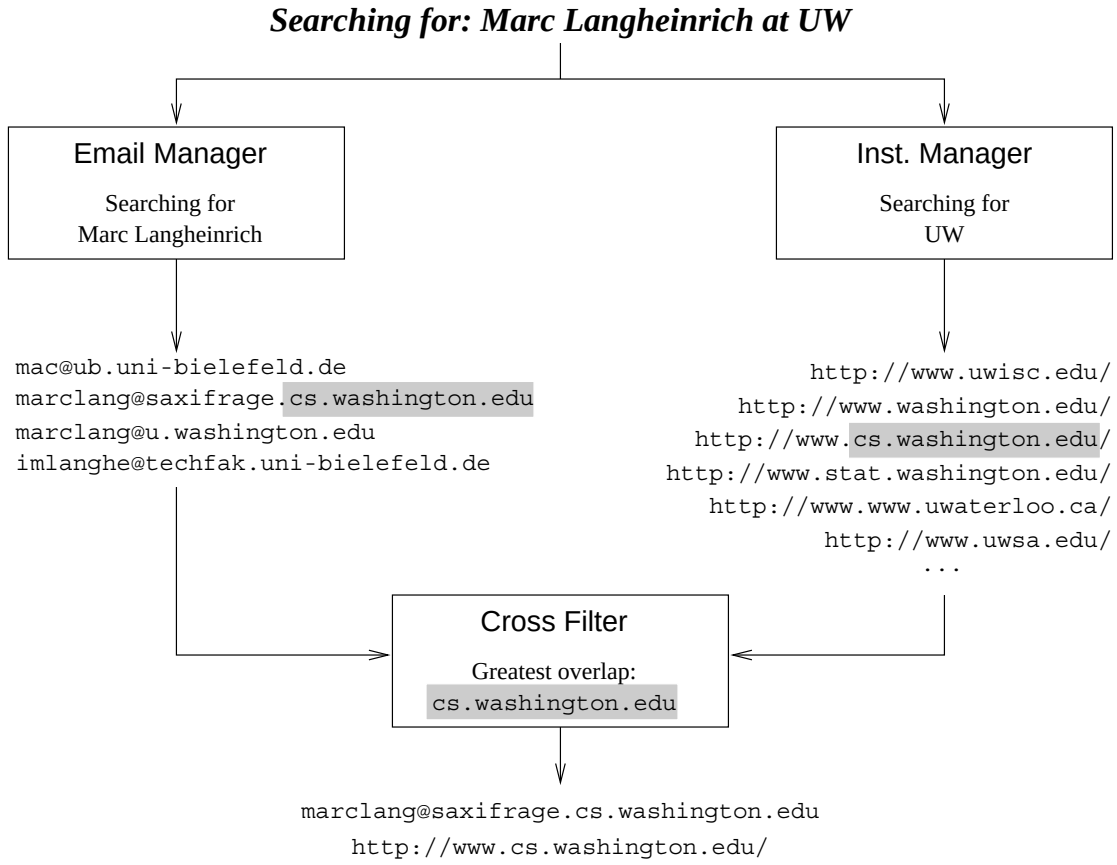


Figure 4.5: **Ahoy! Cross Filtering.** By finding the largest overlap between institutional URLs and email addresses, the cross filter in the Main control module unifies email and institutional information. Overlaps are later used to determine the "correct" institution of the *base references* from the MetaCrawler.

Ahoy! uses both *title* and *URL* information to compute this feature. Ten possible values (M,P,S,L,F,R,E,U,D,W) can be obtained analyzing the *title* of the reference, three possible values are computed (M,A,E) from the *URL* (see table 4.4). The final value that is reported for this feature is simply the better of the two individually obtained codes.

2. **Location:** Ahoy! determines a three-digit binary value regarding the Location of the reference, indicating the correct *country*, *zone* (if applicable) and *site-name* of its URL.[12] In case the user did not specify an institution, the value for the sitename is always zero. See table 4.5.

3. **Type:** Ahoy! analyzes both the title and the URL in order to determine whether a reference is indeed a homepage (independent from belonging to the right person or being at the correct institution).

[12]See [BL89] for a description of the various parts of a URL.

Code	*Description*
111	at the right institution
110	in the wrong country
101	in the wrong zone
100	at a similar named institution
011	in the right zone and country
010	in the right zone
001	in the right country
000	in a different country

Table 4.5: **Ahoy! Analyzer: Location Codes.** The first four codes, 111-100 are only used if the user specified institutional information.

Code	*Description*
5	explicitly labeled homepage
4	name only in title (common for personal homepages)
3	URL in homepage format
2	list page (*e.g.*, faculty lists, student directories, etc)
1	undecided
0	explicitly labeled as non-homepage

Table 4.6: **Ahoy! Analyzer: Page type codes.** Ahoy! differentiates between 5 types of pages. Codes 5 through 3 are considered being a homepage, while pages with codes 2 through 0 are only displayed in case of failure as alternatives.

First, it tries to find a domain specific indicator, such as "home page" or "personal page", or even negative indicators like "publications" or "resume", in the page title. In either case it assigns an unambiguous maximum (value 5) or minimum (value 0) value to its internal 'type' scale (see table 4.6). It also has a special "list page" type (value 2) that is triggered by strings like "directory" or "list". If it fails to find any of these strings, it will try to establish if the name of the person forms the complete title – a form very common among personal homepages (value 4).

If none of the above cases matches, Ahoy! analyzes the *URL* in order to find additional indicators for the reference being a homepage. For example, many homepages don't have an explicit filename in their URL, or use the default filenames `index.html` or `welcome.html`. If this analysis also fails to find a path pattern common to personal homepages (value 3) in the URL, the page type is left open (value 1).

This analysis is done as soon as information becomes available, even if only parts of the three values, ownership, location and type, can be established. As soon as all three values has been assessed, each reference is sorted into an appropriate *bucket*

and placed in the *DRS-Table*, as described in the next section.

Bucketing: the Table Manager

Once we have established all three value for each returned reference in the base set, Ahoy! can sort the references into its buckets, which will allow it to dynamically find the most promising reference along these three features.

Each bucket in Ahoy! is uniquely identified by three values: page type, ownership and location. An example for a bucket label would be: "Homepage with the correct name at the right institution", corresponding to a feature combination "`M-111-5`".[13]

Figure 4.6 shows a picture of Ahoy!'s table, with each of the three features forming an axis in a three dimensional space, and each axis ranging over all possible values of the respective feature.

In order to decide which references to display, the Table Manager provides methods for selecting the "best" available bucket in the most promising *zone*. As described in section 3.2.2, *zones* provide a way to arrange the available buckets in a table into a small number of groups featuring similar characteristics.

Zones in Ahoy! are defined by only two axes: *ownership* and *location*. The third axis, *page type*, is used only as a preference function for buckets with otherwise identical features. Ahoy! uses a list of eight zones to group possible general outcomes of the search. Table 4.7 shows all existing configurations, consisting of a short description; one of three statuses (success, partial success or failure); ownership and location range; and alternatives. As mentioned before, the page type is not directly used in zone selection, but only during the final display of the references. Please refer to tables 4.4 (page 56) and 4.5 (page 58) for the ownership and location codes used in the table.

Figure 4.7(a) shows the layout of these eight zones in the *DRS-table*. If no institutional information is available, a reduced table is used (as shown in figure 4.7(b)), featuring only zones `1-3` and `8`.

After all references have been analyzed and sorted into corresponding buckets, the Table Manager simply checks zone by zone to see if at least one of their buckets is filled. If so, it will take the best bucket within this zone (*i.e.*, the one closest to the center) and return its content to the Main Control Script, together with an appropriate description depending on the zone, the bucket, and the page type of the reference (See the *Description* field in table 4.7). Using the Display Manager, this information can then be returned to the user.

4.2.5 Ahoy! Output: The Display Manager

Ahoy!'s Display Manager provides methods and functions to the rest of the system that allow each module to return customized message to the user. Depending on

[13] Due to the historic development of these features, the page type feature is actually not a separate axis. Instead, all references assigned to a particular bucket are automatically kept in an ordered list, according to their page type value. However, this implementation detail can be directly translated into the framework described above.

	Description	Status	Ownership	Location	Alternatives
1.	**Pages bearing the right name (at the given institution)**	Success	M, P, S	111, 110, 101, 100	(2, 3)
2.	**Pages bearing the lastname (at the given institution)**	Partial Success	L, R	111, 110, 101, 100	4, 3
3.	**Pages bearing the firstname (at the given institution)**	Partial Success	F, A	111, 110, 101, 100	4, 5
4.	**Pages bearing the right name at another institution**	Partial Success	M, P, S	011, 010, 001, 000	6, 7
5.	**Pages without a name at the correct institution**	Failure	E	111, 110, 101, 100	6, 7
6.	**Pages bearing correct last or firstname at another institution**	Failure	L, R, F	011, 010, 001, 000	–
7.	**Pages bearing another name at the given institution**	Failure	U, D, W	111, 110, 101, 100	8
8.	**Pages with no apparent connection to your search**	Failure	E, U, D, W	011, 010, 001, 000	–

Table 4.7: **Ahoy! Zones.** Using a list of eight zones, which are defined by only two axes, *Ownership* and *Location*, Ahoy! selects the best references to display as the search result. The third axis, *page type*, is used only as a preference function for buckets with otherwise identical features. Zones `1-3` have different descriptions depending on available institutional information. See also figure 4.7.

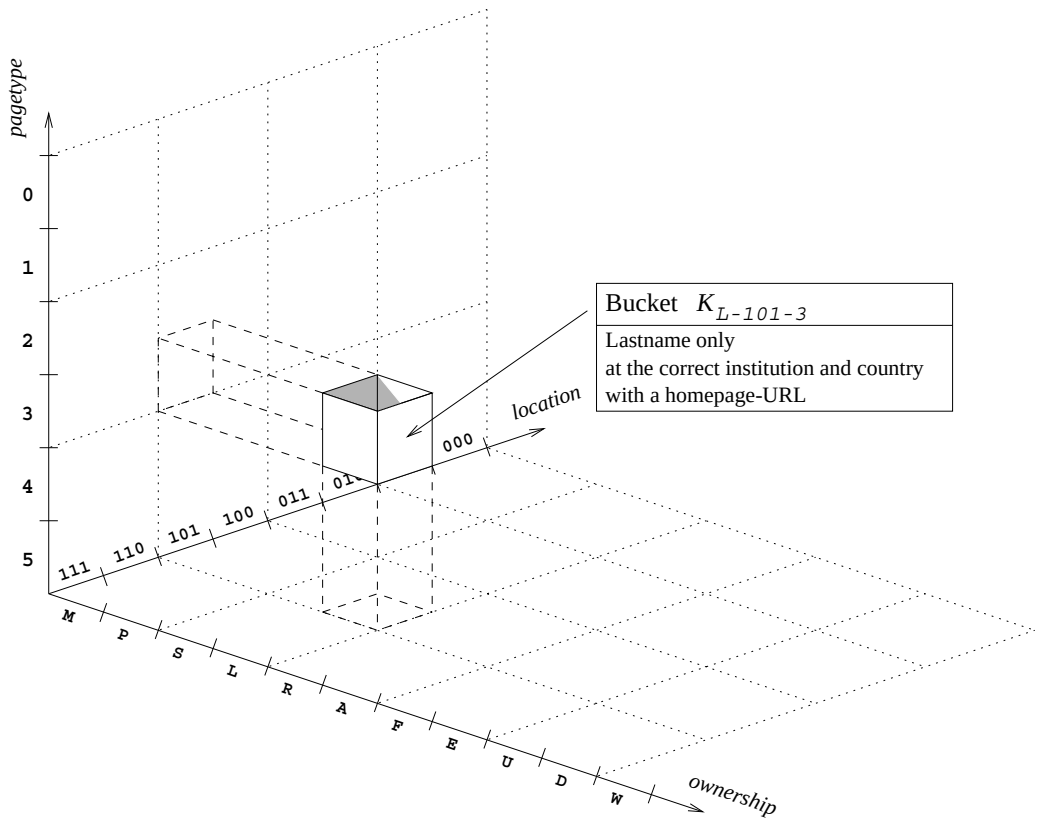


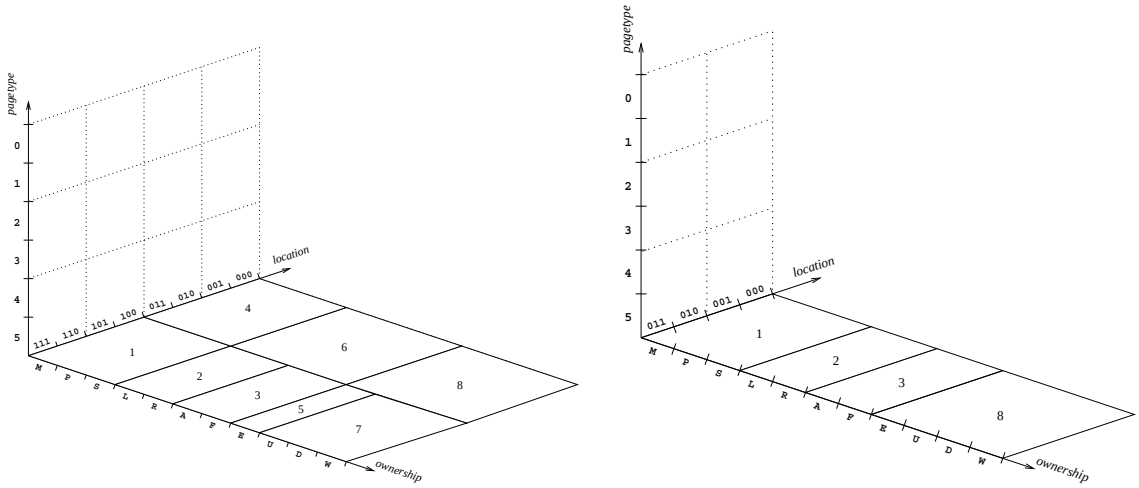
Figure 4.6: **Ahoy! Table.** The three analyzed features – ownership, location and page type – form the axes of a three dimensional space. A sample bucket, $K_{L-101-3}$, is shown, containing references featuring "the lastname only, at the correct institution in the correct country, with the URL indicating a homepage".

the user's Web Browser, the Display Manager will *allow* (for Netscape browsers) or *suppress* (all other browsers) intermediate output that is used to update the user regarding the progress of the search (*i.e.*, Ahoy!'s *reactivity* feature). It also handles any overhead resulting from the HTTP protocol[14] for starting or ending a dynamic Web response properly.

4.2.6 Learning

Learning in Ahoy! follows the scheme described in section 3.2.3: Successful searches are used to create *general hypotheses* by extracting patterns from the URLs of the top references, while unsuccessful searches lead to the generation of *instantiated hypotheses*, which are then tried directly in order to find the desired page. Three modules provide Ahoy! with its learning capabilities: the URL Extractor *acquires* new general hypotheses by extracting patterns from successful URLs, while the URL Generator

[14]The HyperText Transfer Protocol is the communication protocol of Web based services.



(a) institutional information available (b) no institutional information

Figure 4.7: **Ahoy! Zones.** The eight available zones divide the *DRS-table* as indicated in figure (a). In case no institutional information could be obtained, a collapsed table shown in figure (b) is used, featuring only four of the zones, `1-3` and `8`.

generates instantiated hypotheses from previously extracted patterns using current query parameters. Both modules are controlled by the URL Learner module, which provides a high level access to Ahoy!'s directed search capabilities for the Main Control Script, as well as *evaluating existing* general hypotheses by recording success of failure of the according instantiated hypotheses.

Hypothesis Acquisition: the URL Extractor

The URL Extractor module allows the URL Learner module to map successful URLs to a number of corresponding *general hypotheses*. A given URL is searched for a (query dependent) list of strings that can be replaced with general *placeholders*.[15]

Table 4.8 shows a list of currently implemented placeholders in Ahoy!, although more mappings can be added at any time. The *unknown placeholder* `<_>` is used whenever domain specific heuristics found indicators of a user specific URL part, but failed to find an appropriate mapping from a given query parameter or orthogonal information. For example, when looking for "Erik Selberg" and finding his homepage at `http://www.cs.washington.edu/~speed/`, domain heuristics would indicate that "speed" is indeed the user specific part in the URL (using the tilde ~ as an indicator), but none of the standard mappings would allow it to substitute the word "speed". In this case, the extraction method creates an hypothesis containing the "unknown placeholder": `http://www.cs.washington.edu/~<_>`.

[15]See figure 3.3 on page 36.

Code	Description	Example
`L`	Lastname	`smith`
`F`	Firstname	`john, peter`
`fL7`	Firstname initial, filled up with up to seven letters of lastname	`jsmith, psmith`
`F7l`	Firstname, up to seven letters, followed by lastname initial	`johnsmit, petersmi`
`ffl`	All available firstname initials, followed by lastname initial	`jps`
`U`	Email user-name	`smith93, jsmith`
`_`	Unknown mapping	

Table 4.8: **Ahoy! Placeholders.** Six "known" and one "unknown" placeholder are used in Ahoy!'s URL Extractor to form *general hypotheses* from successful URLs. The examples in the rightmost column assume a query for "John Peter Smith" and finding two email addresses with user-names `smith93` and `jsmith`.

Generating new URLs: the URL Generator

The URL Generator module allows the URL Learner to reuse *general hypotheses* that were acquired during previous, successful searches, by re-instantiating the general placeholders in such a hypothesis with actual value from the current, unsuccessful search.[16]

When encountering an *unknown placeholder* <_> in a hypothesis, the URL Generator will use *all available placeholders*, as if it encountered the same hypothesis multiple times, each with a different placeholder. With multiple *unknown placeholders* in a single hypothesis, this could easily lead to a very large number of instantiated hypotheses. In order to prevent this, the current version of Ahoy! will treat all *unknown placeholders* as being the *same* placeholder, thus preventing an exponential explosion at the loss of generality.

Coordinating direct search: the URL Learner module

In case of a successful search, the URL Learner uses the URL Extractor module to map every URL of the candidate homepages to a number of corresponding *general hypotheses*. Those new hypotheses are simply added to its list of known hypotheses (if they aren't already in there) and their *performance statistics* (see below) are initialized with the "unknown" value.

In case Ahoy! could not find a successful hypothesis, the URL Learner will search the existing hypotheses-lists for applicable hypotheses. If the user failed to specify an institution and none could be inferred from the emails found, Ahoy! will not use any of its hypotheses. Otherwise, it will use the institutional information from Ahoy!'s

[16] See figure 3.4 on page 37.

orthogonal information sources (*i.e.*, the "overlap", see figure 4.5 on page 57), to select a small number of sites to search. The existing general hypotheses at these sites will then sorted according to their past performance and used to generate a list of *instantiated hypotheses*, which can then be used directly to find the desired page. This list of *instantiated hypotheses* represents Ahoy!'s overall hypothesis where the information for this particular search request is most likely to be stored.

The performance statistics of a general hypothesis are updated every time one of its instantiated hypotheses is used to find a specific homepage. If it was successful, a *positive feedback* is given for the *general hypothesis* that was used to generate this URL. If it fails to find the correct page, *negative feedback* is given instead. However, is *none* of the available instantiated hypotheses is able to find a page, all feedback from this search is discarded. This is done in order to avoid accumulating a large amount of negative feedback from queries that might simply not have an answer (*i.e.*, an existing homepage) at all.

When selecting the list of hypotheses to use in case a particular search failed, these statistics are used to order the available hypotheses at a given site according to their performance in the past. Generating *instantiated hypotheses* is done in the order suggested by their past performance, which is also kept during the actual direct Web search, so that instantiated hypotheses from successful general hypotheses are tried first. This not only ensures short direct search times by Ahoy! since it will try most promising hypotheses first, but also helps to prune erroneous hypotheses that are not useful for finding homepages.[17]

If a particular general hypothesis lacks sufficient feedback (*e.g.*, it was only recently acquired), the URL Learner will solicit feedback at higher levels in the domain hierarchy, as described for the generic DRS-architecture in section 3.2.3.

Finally, it should be noted that these performance statistics are not used to determine the "correct" format of homepage URLs at a given institution. Instead, the hypotheses ranked top by the URL Learner reflect both user demand for a particular set of people *and* a lack of availability through standard sources.

4.2.7 Miscellaneous

This section lists the modules mentioned in figure 4.4 that do not fit into any of the five preceding categories. Additional smaller modules are briefly described in the appendix.

Main control script

The Main Control Script coordinates all modules of the Ahoy! system. Two different scripts are used, depending on the type of search that should be performed. The default script, `nph-ahoy.cgi`, handles new searches and uses all of the modules described above. In addition, the `nph-continue.cgi` script is called whenever

[17] Though no pruning is done in the current version of Ahoy!, subsequent experiments in this area should provide enough data to incorporate cut-off values based on the kept statistics.

Ahoy! asks the user to specify a site that it should search directly in case of a failed search. This "continuation" script does not need to connect to any information sources, and only contains the modules that allows it to *directly search* for the desired pages. All modules that deal with information sources (MC Manager, Inst Manager and Email Manager) are replaced with dummy methods that re-read the information stored in the matching *session directory* (see section 4.2.2 on page 53) of the initial search.

Multiplexer

The Multiplexer module allows the Main Control Script to access a number of Internet and local sources in parallel. After registering each request as specified by the information sources modules, the Main Control Script simply calls the `wait` method, and is then notified as soon as any information becomes available. This is used to provide the user with continuous status information during a search.[18]

DB Manager

Ahoy! uses various DB Manager, which share a uniform interface, to access and store temporary information on disk. Ahoy!'s DB Manager transparently handle the organization of data files into hierarchical directories (in order to prevent performance problems when accessing directories featuring a large number of files), as well as providing access to plain text and *Berkeley DB* [Sof] files (the latter ones are used to efficiently store the large amounts of data available during cross-domain learning).

4.3 Evaluating Ahoy

This section gives an overview of the experiments we conducted to measure Ahoy!s performance compared to the standard search services currently available on the Web.

We will first examine the standard measures used in the field of information retrieval (IR) to asses the performance of a retrieval system, *recall* and *precision*, and describe why we can not use them directly; both in the field of Web Search Systems in general and with the task of DRS-System in particular. We will then introduce our own measurements, *recall'* and *precision'*, which we will use to approximate the standard measures, and describe our experimental setup in more detail.

After presenting the results of our main performance experiments, which deal with Ahoy!'s search capabilities, we will briefly describe our second set of experiments, which examine the effects of using the hypothesis performance statistics we described in section 3.2.3 on page 37.

[18] Only possible if the user is using a browser that supports multiple pages of output, such as the *Netscape* browser.

4.3.1 IR Performance Measures

Several aspects have to be considered when examining the performance of a system. [Sal67] argues that many types of evaluation environments exists: different *systems* (operational, testbed, laboratory) as well as different *concerns* (users, managers, operators). Instead of evaluating managerial issues such as maintenance efforts and costs, we want to focus in our experiment on the ability of an *operational system* to *satisfy the user*. [Cle65] lists the following six criteria all of which affect user satisfaction:

1. **Coverage:** The extent to which the system includes relevant matter.
2. **Time lag:** The average interval between the time the search request is made and the time an answer is given.
3. **Presentation:** The form of presentation of the output.
4. **User Effort:** The effort involved on the part of the user in obtaining answers to his search requests.
5. **Recall:** The proportion of relevant material actually retrieved in answer to a search request.
6. **Precision:** The proportion of retrieved material that is actually relevant.

We already gave brief qualitative comparisons of the first four points for the systems examined here in section 2.2 (see also summary in table 4.9). In the remainder of this section, we will focus instead on the last two criteria – most popular performance measures in the field of information retrieval – *precision* and *recall*.

Recall & Precision

Recall and precision are standard performance measures in the field of information retrieval. *Recall* is defined as the proportion of relevant matter retrieved, whereas *precision* is the proportion of retrieved material actually relevant [Sal67]:

- **Recall:** A measure for the *comprehensiveness* of a search service. The higher the recall, the more likely it will find the information the user is looking for.

$$\text{Recall} = \frac{\text{number of relevant documents retrieved}}{\text{total relevant documents in collection}} \tag{4.1}$$

- **Precision:** A measure for the *accuracy* of a search service. The higher its precision, the fewer irrelevant entries will return for any given search.

$$\text{Precision} = \frac{\text{number of relevant documents retrieved}}{\text{total retrieved}} \tag{4.2}$$

Criteria	Web Index	Web Directory	Meta Index	DRS
Coverage	High Automatic updating	Low Manual updating	High Uses Web Indices	High Uses Meta Indices
Time lag	Short Simple query techniques allow fast lookup	Short Simple query techniques, small index.	Medium Can be as slow as slowest service	Medium Can be as slow as slowest source
Presentation	Low Simple list, ranked by "relevance"	Medium List augmented by categories	Low Simple list, ranked by "relevance"	High Single labeled answer, including additional information
User Effort	High Manual searching through list; query reformulation	Medium Search through classified list	High Manual searching through list; query reformulation	Low All relevant information presented on single page
Example	AltaVista	Yahoo!	MetaCrawler	Ahoy!

Table 4.9: **Qualitative Performance Comparison.** [Cle65] lists six criteria which affect user satisfaction. Four of them, *Coverage*, *Time Lag*, *Presentation* and *User Effort*, were already discussed in section 2.2 and are summarized here. The remaining two, *Precision* and *Recall* are discussed in this section.

Request: 5 relevant documents
ranks: 1, 2, 3, 20, 21

Cutoff values	1	5	10	15	20	25
Recall	0.2	0.6	0.6	0.6	0.8	1.0
Precision	1.0	0.6	0.3	0.2	0.2	0.2

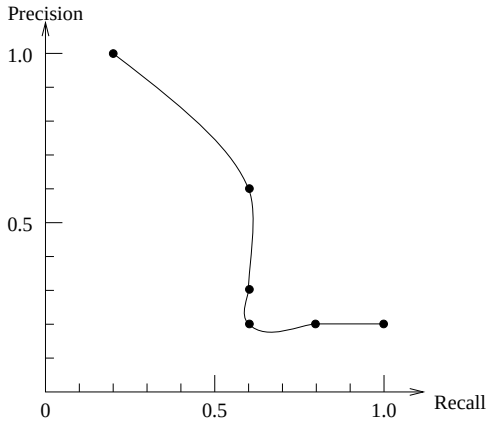


Figure 4.8: **Precision-Recall Graph** A precision-recall graph shows a number of precision-recall pairs for various cutoff values. This example after [Sal67] assumes a response featuring relevant documents at position 1, 2, 3, 20 and 21.

A retrieval system is usually characterized not by a single recall-precision pair, but by a number of these pairs obtained at different *cutoff values*,[19] arranged in a so called *precision-recall graph*, as shown in figure 4.8.

For a given system, it is usually straightforward to asses these numbers. First, compile a set of test queries and tag all elements in the available dataset as either *relevant* or *irrelevant* to each queries. Then perform all queries and compute the average values for *recall* and *precision* according to formulas 4.1 and 4.2, respectively. However, in the context of IR systems on the Web in general and DRS-Systems in particular, two problems arise:

1. Conventional IR systems have hundreds, maybe thousands of documents in their collection,[20] which makes providing relevance judgments for *all* documents with respect to *all* search requests in the test set laborious, but possible. But for the Web, containing over hundred million documents, it is impossible to asses the relevance for every document, even for a very small number of test requests. Thus, we cannot compute recall as given above, because the "total number of relevant documents in collection" is never known.

2. DRS-Systems are best used when searching for tightly bound sets, such as a particular homepage, paper or review (see table 3.2 on page 28). When using a DRS-System, finding a *single* relevant reference is usually sufficient to answer the query – any further "relevant" references are likely to be redundant in most domains (*e.g.*, other addresses for same homepage, other citations of

[19] IR systems usually return a list of ranked elements for a given search. Using a *cutoff value*, the system designer can specify how many elements of the list to show to the user, thus greatly influencing the precision and recall of the system.

[20] The collections evaluated by [Sal67], *ADI*, *Cranfield-1* and *IRE-3*, featured between 82 and 780 documents.

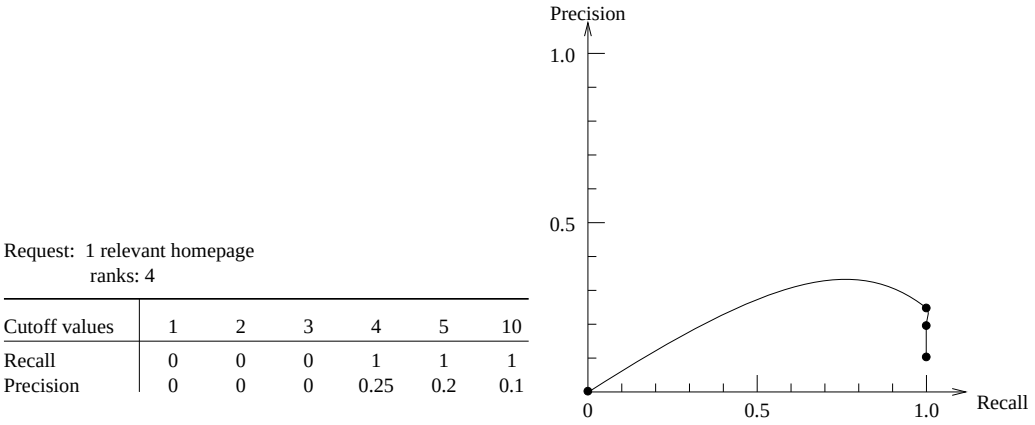


Figure 4.9: **Precision-Recall Graph in DRS-domain.** Using precision and recall to compute performance in DRS-domains such as personal homepages results in a number of "zero precision, zero recall" values, until the cutoff value includes the single relevant reference (if the particular service found it). From then on, all data points feature 100 percent recall and a precision of 1/cutoff. This suggests using different measures in our experiments.

same paper, etc). Thus, computing precision will usually result in either zero or the inverse number of documents retrieved ($1/n$), because the "total number of relevant documents retrieved" is either zero or one.

Even when assuming only a single relevant reference in order to compensate for the first problem (the unknown number of existing relevant documents), our experiments in the homepage domain would still fail to produce reasonable results. Using precision and recall to compute performance would only result in a number of "zero precision, zero recall" values, until the cutoff value would include the single relevant reference (if the particular service found it). From then on, all data points would feature 100 percent recall and a precision of 1/cutoff. Figure 4.9 shows the corresponding precision recall graph.

Alternative Measures

In order to be able to compare the performance of various search services in DRS-domains, we define our own "precision" and "recall" measures, *precision′* and *recall′*:

- *recall′*: Given a list of queries, the percentage of them that found the correct answer within the complete answer set (up to 200 references with most services). This is a measure for the *comprehensiveness* of a search service.

- *precision′*: Given a list of queries, the percentage of correct answers returned as the top-ranked reference by each search service. This is a measure for the *accuracy* of a search service.

Although these measures are named similar to precision and recall, we cannot combine *precision′* and *recall′* into a precision-recall (or *precision′-recall′*) graph, since these two measures no longer share common data points related to a specific cutoff value and a specific query. Instead of a number of recall-precision pairs, a single *precision′* and *recall′* value now characterizes the performance of a retrieval system.

precision′ might be overly strict for practical purposes, however. Often a reference ranked second would also satisfy the user's information need. We can compute variations of the *precision′* measure when increasing the number of references examined, including for example the 10 best references, denoted by $precision'_{10}$. Similarly, $precision'_1 = precision'$ and $precision'_\infty = recall'$.[21]

4.3.2 Experimental Setup

Now that we determined *what* we want to measure, *precision′* and *recall′*, we have to find a way *how* to measure it. With Ahoy! serving thousands of search requests daily, the idea of using a large sample of these real life user queries, taken over a week or a month of service, seems ideally suited to provide an excellent, unbiased test set.

Unfortunately, sampling user queries is impractical. In many cases, it is difficult to determine objectively whether an obscure person mentioned by a query has a homepage or not. Users are also fond of searching for the homepages of celebrities: although the concept "celebrity" is difficult to define precisely, informal log analysis reveals that five to ten percent of searches are for celebrities. Such searches pose an unsolved testing challenge: which page(s) of the dozens devoted to an Oprah Winfrey should be considered her correct homepage(s)? Because we cannot determine "correct" answers to actual queries, we cannot judge Ahoy!'s performance on a representative sample of queries.[22]

Instead of using usage logs as a source of test queries, we use two Web sites that list people along with the URLs of their homepages. One site, David Aha's "List of Machine Learning and Case-Based Reasoning Home Pages" [Aha], contains 582 valid URLs. We call these homepages the *Researchers sample*. Another site, the "netAddress Book of Transportation Professionals" [The], contains 53 URLs for individuals' homepages.[23] We call these homepages the *Transportation sample*. We chose two independently-generated samples from disparate fields in order to evaluate Ahoy!'s scope and breadth. The *Researches sample* provides a good indicator for personal homepages at academic institutions, while the *Transportation Sample* represents the growing fraction of non-academic Web users establishing their Internet presence.

[21] In most cases, infinity would be bound to 200 as an upper value for the number of references returned by most search services.

[22] Table 4.11 on page 74, however, presents statistics that partially characterize Ahoy!'s performance on actual user queries.

[23] Both samples were taken in September 1996. Since those lists evolve constantly, today's versions might contain significantly more or less entries, however.

Methodology

A large number of studies in recent years have attempted to compare the performance of various popular search services on the Web [Pau97], [Pal]. Some studies simply record the number of indexed pages for a given keyword or site [Leb],[24] others try to compare the usefulness of the services through head to head quiz competitions [Lak97].

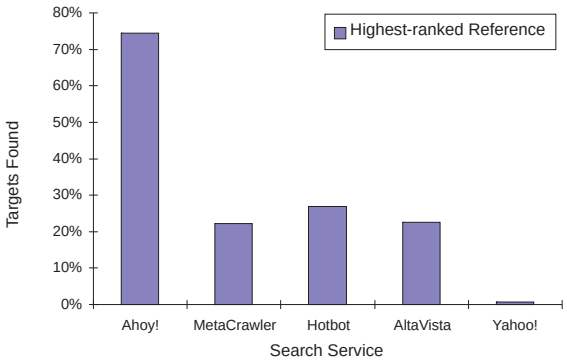
In contrast, our experiments tried to measure the search services performance on finding a *single right answer*, given a particular query. Using a sample set of more than six hundred query-answer pairs, we would send one query at a time to all competing services and then record the position of the expected answer (*i.e.*, the URL of the correct homepage for that person) among each services output, or "missed" in case the answer could not be found.

While testing each competing service on the sample set, we used query syntax that maximized the performance of the service in finding personal homepages. For example, if the target homepage belongs to John Smith, we used AltaVista's "advanced syntax" to search for "John NEAR Smith" and ranked the results using the string "John Smith." Queries to HotBot invoked its person-specific search option. In addition, in the tests for *recall'*, each service was allowed to return as many references as possible – about 200 each for AltaVista and HotBot, usually less than 30 for Yahoo!, and approximately 50 for the MetaCrawler. Ahoy! usually returns one or two references.

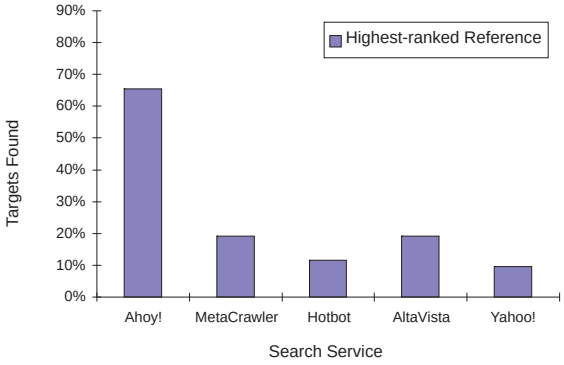
Sample	**Transportation Sample**		**Researchers Sample**	
Service	Average Rank	Std. Deviation	Average Rank	Std. Deviation
Ahoy!	1.03	0.17	1.21	0.83
MetaCrawler	7.82	9.28	7.02	8.06
AltaVista	6.06	9.3	7.42	19.79
HotBot	7.31	6.57	8.0	16.9
Yahoo!	1.17	0.37	1.33	0.47

Table 4.10: **Average Rank of Targets** The "Average Rank" column shows how many references a user will typically have to read through before finding the desired homepage, if the homepage is found. Ahoy! and Yahoo! place the average target page at the top of their output, while the other services tend to place the target page under several non-target references.

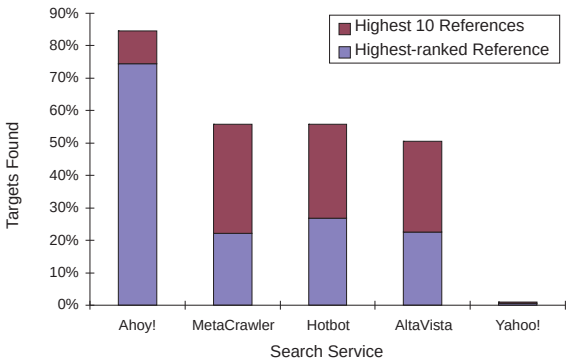
[24] Which raises the question as to what the difference is between, say, 2871 and 13452 returned references — in both cases it is highly unlikely that the user will actually look at more than the first 50.



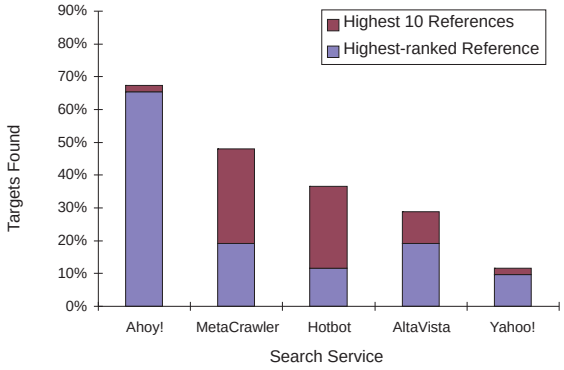
(a) *precision′* Researchers Sample



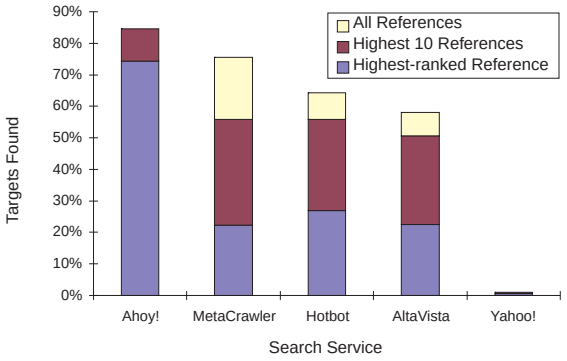
(b) *precision′* Transportation Sample



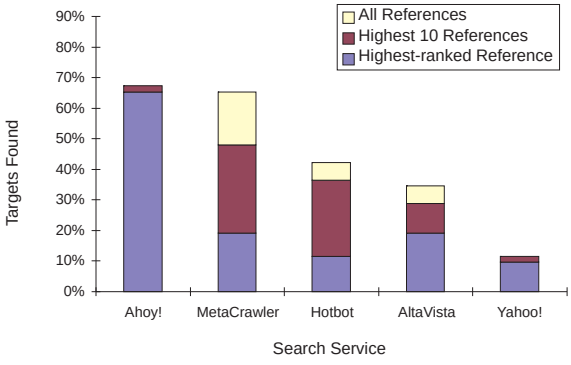
(c) $precision'_{10}$ Researchers Sample



(d) $precision'_{10}$ Transportation Sample



(e) *recall′* Researchers Sample



(f) *recall′* Transportation Sample

Figure 4.10: **Performance Comparison** Ahoy!s accuracy in the homepage domain is significantly higher than that of general purpose search services. Ahoy! surpasses the *precision′* of its closest competitor, HotBot, by a factor of almost three (a),(b). Including the first ten references, Ahoy! still features over 25 percent better performance (c),(d). Even when including *all* available references, Ahoy! demonstrates an up to 9 percent better *recall′*, due to its URL Learner module (e).

4.3.3 Results

Figures 4.10 (a), (c) and (e) show results for the *Researches Sample*; figures (b), (d) and (f) the results for the *Transportation Sample*.

With about three quarter of all homepages in the Researchers Sample ranked top, Ahoy! surpasses its closest competitor, HotBot, by a factor of almost three (figure 4.10 (a)). The same experiment using the Transportation Sample still results in 65 percent of the pages shown as the highest-ranked reference by Ahoy!, compared to less than 20 percent for the MetaCrawler and AltaVista.

Another way of capturing accuracy is to calculate the average rank of the target URL within the output of successful searches (see table 4.10). A low rank is good, because that means the target reference will be displayed prominently near the top of the output. Conversely, a high rank suggests that irrelevant pages have been erroneously ranked higher than the target. A very high rank also lessens the likelihood that the user will bother to scroll down and find the target.

Under the average rank metric, Ahoy!'s accuracy is significantly higher than that of every search service except Yahoo!. Yahoo!'s precision comes at the expense of coverage, however, as demonstrated by Figure 4.10 (e) and (f).

If we take the maximum number of references returned into account (up to 200 for HotBot and AltaVista), we can get an idea for the comprehensiveness of the search service. As one would expect, the MetaCrawler features the highest *recall'* value of Ahoy!'s competitors, since it combines the output from a number of search services, including HotBot and AltaVista. But even though Ahoy! *filters* the output of the MetaCrawler, it has an up to 9 percent higher *recall'*. This increase is due to Ahoy!'s URL Generator, which allows it to generate URLs missing in the output of *all* of its competitors. The *recall'* for Yahoo!, the manually compiled Web directory, is predictably low: Less than one percent of the people in the Researchers Sample have registered their homepage with Yahoo!, compared to about ten percent of the Transportation Sample.

Data from the fielded Ahoy!, as given in table 4.11, helps explain why the URL Learning module is able to increase Ahoy!'s *recall'*: As of November 1996, Ahoy!s URL Pattern Extractor learned more than 23,000 patterns from over 6,000 institutions. In August 1997, this figure had risen to include more than 80,000 patterns from over 30,000 institutions. Although this figure is small compared to the number of sites with a Web presence, it reflects the distribution of sites at which users have found homepages in earlier searches before. In other words, the content of Ahoy!s URL database is skewed in a way that resembles the type of queries posed to Ahoy!.

Because of that, Ahoy! is able to use information from its database in more than 34 percent of all failed searches where the user specified an institution. In 27 percent of all cases when Ahoy! generates a URL, it finds a page that it considers to be a target homepage.

Searches over 3 months	53,000
Unique target names seen	42,000
Clients originating queries	20,000
Clients originating over 20 queries	200
Institutions in Ahoy!'s URL pattern DB	6000
Patterns in Ahoy!'s URL pattern DB	23,000
Searches for which user follows at least one link	26%
Searches considered "Unambiguously Successful" by Ahoy!	50%
Searches in which result is a generated URL unavailable from reference sources	2%

References provided in each search	
Median	2
Mode	1
Mean	3.5
Standard Deviation	5.0

Search speed (seconds)	
Median	9
Mode	6
Mean	13
Standard Deviation	23

Table 4.11: **Cumulative Results.** These data suggest that many people search with Ahoy! regularly. Users search for a wide variety of people at a wide range of institutions. In most cases, Ahoy! returns one or two references, but in some cases – notably, searches for celebrities with dozens of "homepages" created by fans – it returns many more. Data are based on Ahoy! operations during a 3-month period ending late October, 1996 [SLE97].

4.3.4 Discussion

The design of experiments to measure the effectiveness of Web Search Services is difficult. Our above results contain experimental biases both in favor and against Ahoy!.

The discrepancy between the small number of hits returned by Ahoy! and the larger number returned by the other services biases comparisons of *recall'* against Ahoy!: the other services have, in effect, many more opportunities to "get the right answer". Furthermore, references displayed deep in the list are of questionable value; many users are unwilling to scroll past the first ten or twenty references returned [Win96]. Thus, pure *recall'* figures understate the value added by Ahoy!.

On the other hand, the experiments contain two biases in Ahoy!'s favor. First, Ahoy! has the advantage of being domain-specific. Second, queries to Ahoy! contain the name of the target institution, information not provided to the other services. Ahoy!'s domain-specific knowledge, used in the framework of its DRS architecture, is what makes it effective, so removing the first bias would change the fundamental nature of Ahoy!. We might, however, remove the second bias by withholding institutional information from Ahoy! or by providing such information to the other services.

Withholding institutional information would prevent Ahoy! from using its full set of features, including the institutional cross-filter and the URL Generator. Even so, Ahoy!'s precision under these conditions remains over fifty percent higher than that of

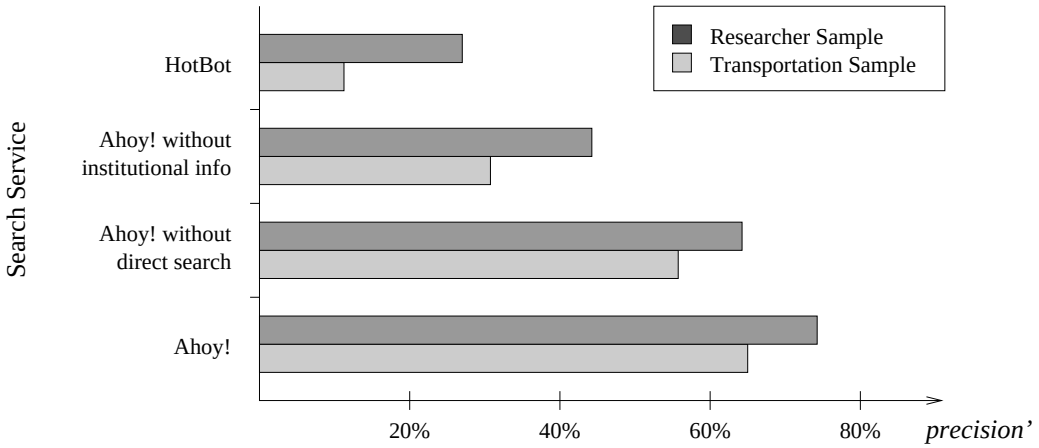


Figure 4.11: **Restricting Ahoy!:** *precision′* **Performance Comparison.** Removing Ahoy!'s directed search feature, or even withholding institutional information, reduces Ahoy!'s *precision′* by up to 50%. Even so, it still remains over twice as high as any other service (only its closest competitor, HotBot, is shown).

the other services, as shown in figure 4.11. Ahoy!'s *recall′*, however, shows significant performance loss when removing the benefit of the URL Generator. As shown in figure 4.12, Ahoy!'s performance is worse than that of the MetaCrawler when not using its directed search feature, and even more so when withholding institutional information completely.

Ideally, Ahoy!'s *recall′* under these conditions should be close to, if not identical to that of the MetaCrawler. But disabling Ahoy!'s URL Generator shows that it not only looses the ability to create the *additional* references that are not found by *any* service, but also misses about 8% of the references that are already included in the MetaCrawler output (see figure 4.12). These references apparently fail to obtain favorable values during Ahoy!'s feature analysis because of uninformative titles or unusual URLs. But using its URL Generator, Ahoy! is able to recreate these URLs by instantiating previously acquired *general hypotheses* and then generating direct requests for each of them. Obviously, Ahoy! could compare each reference in its *base reference set* with its *instantiated hypotheses* to find these missed entries *before* trying to locate them directly.[25]

The reason for the second performance loss – when completely withholding institutional information from Ahoy!– is its internal bias for small answer sets. In many cases, Ahoy! finds a substantial number of homepages for people with the same name. Using its cross-filter, it is able to discard a large number of these, which obviously belong to a different person. However, without any institutional information, the correct target homepage might often be of only suboptimal quality (*i.e.*, featuring only part of the name, or none at all), so that references with a more prominent title,

[25]Ultimately, this step should already be incorporated during the content analysis phase by reinforcing URLs that match such previously acquired *general hypotheses*.

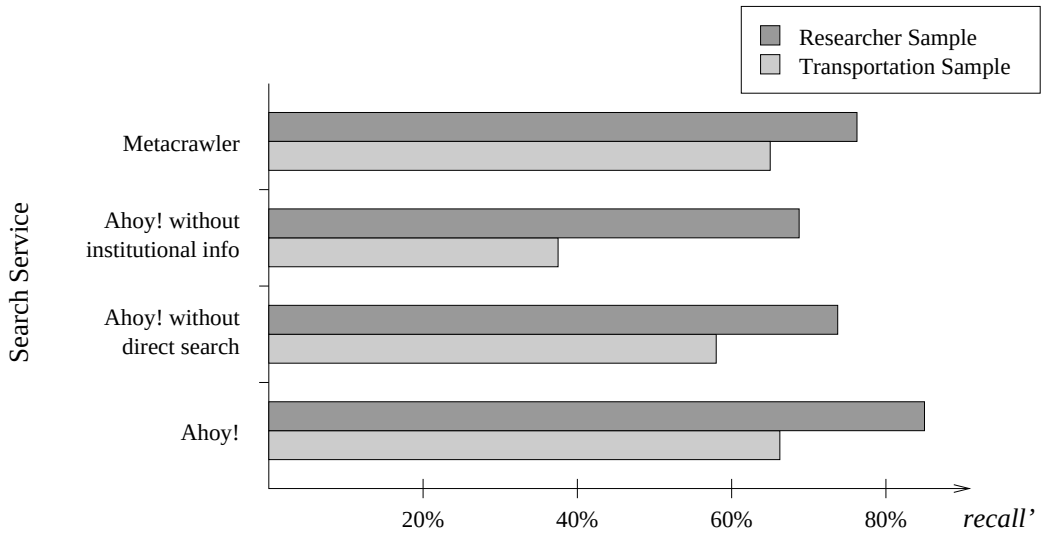


Figure 4.12: **Restricting Ahoy!:** *recall′* **Performance Comparison.** Ahoy!'s performance is worse than that of the MetaCrawler (its closest competitor regarding *recall′*) when not using its directed search feature (but still specifying the correct institution), and even more so when withholding institutional information completely. While the first represents a shortcoming of the current implementation, the reason for the second performance loss lies in the characteristics of DRS-domains in general.

but belonging to the wrong person, will simply look "better". Since Ahoy!'s selection mechanism only displays the content of the "best" bucket as the final answer, the correct homepage gets lost in a lower-ranked bucket. Improving the answer selection mechanism with respect to this apparent insufficiency seems futile, however. Recalling the DRS domain characteristics in table 3.2 on page 28, these searches violate item 2, the "focused attention" constraint: "*... the user can pose a query specific enough to exclude other members.*" With more and more people with similar names providing homepages on the Web, specifying the institution becomes essential in order to explicitly select a single person.

On the other hand, when providing institutional information to the general-purpose search engines, the services' *precision′* increases slightly, and their *recall′* decreases significantly.[26] Why does such additional information decrease performance? This might have several reasons:

- The name of the institution may not appear on the target homepage and thus may confuse the search engines' ranking algorithms.
- The general query syntax appropriate for an institutional search often makes it impossible or impractical to take simultaneous advantage of a services person-specific query syntax.

[26] Only informal studies where made here.

- User-provided institutional information is often ambiguous or incorrect. Ahoy!'s institutional cross-filter is designed to be forgiving of imperfect institutional queries; general-purpose ranking algorithms do not have this luxury. For example, a user may input the query "Oren Etzioni at UW" to Ahoy!, but adding "UW" to an AltaVista query would only serve to confuse it.

4.3.5 The effect of Hypothesis Statistics

In order to evaluate the effect of keeping statistics that record the performance of each hypothesis, we evaluated three different strategies for selecting the order in which prospective hypotheses should be used during directed search.

The simplest strategy, called "as-is" strategy, simply records new hypotheses in the order they are discovered, without trying to reorder the existing hypotheses lists. This method, featured in the first implementation of Ahoy!'s direct search module, was soon replaced by the "last-successfully-used" strategy: Record which hypotheses have been successful during the last direct search, and put these at the top of the list so that they are going to be used first the next time this site is searched.

Finally, recording information both about the times a hypothesis has been used, as well as the number of successful applications, we can compute the "Laplace Accuracy" [CB91], which allows us to compute a 'confidence' value for each hypothesis, providing a performance based ranking for subsequent searches. In its simplest variant, the Laplace accuracy L takes the following form:[27]

$$L = \frac{P+1}{N+1}$$

where

P is the number of positive examples
N is the number of negative examples

This measure is currently used in Ahoy!. It ensures that a hypothesis that was successful 245 out of 250 times is prefered to another one that worked 5 out of 5 times, although the expected accuracy of the latter one would be 100%. Using the Laplace accuracy, we can compute the expected accuracies to be:

$$\frac{5+1}{5+2} = 0.857 \quad \text{and} \quad \frac{254+1}{250+1} = 0.976$$

which is more consistent with our intuitive notion.

Setup & Methodology

During a period from May 1st 1997 until June 30st 1997, we let Ahoy! record additional information that could be used to evaluate different strategies in hypothesis selection.

[27] A derivation of the basic formula can be found in [Nib87].

Site	hypotheses	Number of hypotheses used	Number of hypotheses successful	Number of times used	Number of times successful
washington.edu	207	92	8	212	**80**
compuserve.com	17	17	3	168	**51**
att.com	36	9	2	38	**25**
gte.net	9	7	3	72	**24**
concentric.net	33	33	2	59	**17**
ridgecrest.ca.us	3	3	2	18	**12**
ed.ac.uk	74	11	5	20	**12**
usa.net	4	4	1	14	**11**
uvic.ca	52	8	4	28	**11**
ihug.co.nz	3	3	3	10	**10**
duke.edu	111	5	2	13	**9**
purdue.edu	190	27	5	40	**9**
columbia.edu	82	13	5	37	**9**
infoave.net	4	4	2	11	**8**
ohiou.edu	34	7	4	18	**8**
psu.edu	200	65	5	73	**8**
jaist.ac.jp	11	10	2	32	**7**
marshall.edu	11	8	2	21	**7**
sympatico.ca	16	12	3	27	**7**
monash.edu.au	92	24	5	27	**7**
utexas.edu	225	14	6	15	**7**

Table 4.12: **Successful Hypothesis Application by Site (97/05 - 97/06).** Using the sites with the highest number of successfully used hypotheses ensures that we can observe a large enough effect of keeping statistics. Evaluating the performance of sites with a low number of successful searches would make it hard to examine the benefits of keeping these statistics, since only few data points used for computing these performance figures would actually be available.

For each search that resulted in a Failure, but had enough institutional information available to use Ahoy!'s directed search feature, we recorded all user query information, along with Ahoy!'s selection of searchable servers. In case Ahoy! queried the user to make a choice on which site to search, only the users choices would be kept. Finally, a summary of Ahoy!'s results, together with eventually successfully applied hypotheses would be stored.

After monitoring Ahoy!'s directed searches this way for over 6 weeks, we then collated the conducted directed searches and filtered out those that were unsuccessful. The successful hypotheses of the remaining searches were then sorted by site and ordered according to the total number of successful searches per site (see table 4.12).

Using the sites with the highest number of successfully used hypotheses ensured that we could observe a large enough effect of keeping statistics. Evaluating the

Site	# relevant hypotheses	Rank using statistics Laplace	last-used	"as-is"
cs.washington.edu	84	1	1	42
u.washington.edu	54	1	13	16
compuserve.com	17	1	2	2
att.com	36	1	1	30
gte.net	9	1	1	3
concentric.net	33	1	1	6
ridgecrest.ca.us	3	1	1	1
ed.ac.uk	74	1	18	5
usa.net	4	1	1	2
uvic.ca	52	1	2	2
ihug.co.nz	3	1	1	1
duke.edu	111	1	13	8
cs.purdue.edu	190	5	20	18
cc.purdue.edu	190	1	4	3
cs.columbia.edu	82	2	2	3
cpmc.columbia.edu	82	1	8	41
infoave.net	4	1	1	1
ohiou.edu	34	1	1	2
psu.edu	200	1	1	4
jaist.ac.jp	11	1	3	1
marshall.edu	11	1	2	2
sympatico.ca	16	1	1	2
monash.edu.au	92	6	1	3
cc.utexas.edu	225	1	21	35
ma.utexas.edu	225	2	80	80
Average	73	1.4	8.36	12.25

Table 4.13: **Sample Searches.** The final performance data was obtained by re-issuing one example search for each of the sites in table 4.12 and comparing the position of the recorded, successful URLs, with the proposed ranking obtained by each of the three different mechanisms. While the average position of a correct URL is more than 12 when using no sorting at all, keeping the last successful hypotheses at the top of the list results in over 30% improvement. Using information about the number of successful searches, as well as the overall number of times a certain hypotheses has been used, the Laplace Accuracy is able to almost always rank the correct hypothesis at the top position.

performance of sites with a low number of successful searches would make it hard to see the benefits of keeping these statistics, since only few data points used for computing these performance figures would actually be available.

The final performance data was obtained by re-issuing one example search for each of the above sites, and comparing the position of the recorded, successful URLs, with the proposed ranking obtained by each of the three different mechanisms. In case multiple URLs were successful, the best position within each ranking would be recorded.

Results & Discussion

Table 4.13 shows the results of 25 sample searches from the Ahoy! logs while comparing the highest positions of one of the successful URLs in the lists.

While the average position of a correct URL is more than 12 when using no sorting at all, keeping the last successful hypotheses at the top of the list results in over 30% improvement. Using information about the number of successful searches, as well as the overall number of times a certain hypotheses has been used, the Laplace Accuracy is able to almost always rank the correct hypothesis at the top position.

However, this simple experiments leaves several questions unanswered:

1. Although keeping statistics helps, it is not clear which measure provides the best performance. Although better than a simple "last-successful – first-used" policy, the Laplace accuracy used in the current Ahoy! system still favors sparsely used, but almost always successful hypotheses over those that have been used over a thousand time, and succeeded about half of the time.[28]

2. How well does keeping cross-domain information help guiding the process? In the experiment above, we only used information that was directly recorded at each site. Further experiments would be necessary to examine the benefits of having additional, global information available in case a particular hypothesis has rarely been used at a certain site. Also, it remains unclear how to connect local and global statistics to form a single confidence value.

3. With some sites featuring a couple of hundred hypotheses, a crucial aspect of direct search performance is the correct pruning of these long lists. Trying all available entries simply takes far too long, but cutting off too early might prevent Ahoy! from finding the correct page, even though it might have the information further down on the list.

[28] Simply because the Laplace accuracy for 1 out of 1, $(1+1)/(1+2) = 0.66$, is still better than 600 out of 1200, $(600+1)/(1200+2) = 0.49$.

5 Beyond Ahoy: Using DRS in other domains

5.1 Introduction

In order to demonstrate the domain independence of our approach, we will extend the application of DRS-Systems beyond the initially shown homepage domain and introduce prototype applications in two other domains: Academic Papers and Jokes.

The next two section will examine both domains in details, as well as providing architectural outlines and description of the initial prototypes. Both sections will summarize the results of our brief experiments for each system.

The applications described below represent one-week efforts and are not meant to be as sophisticated as our homepage domain prototype. Instead, they are used to demonstrate the feasibility of our approach in other domains.

5.2 Academic Papers

Research often involves following a large number of citations in a certain article or book. Conventionally, one would try to find cited proceedings or journals in a library, but the large number of publications makes it very hard for libraries to have all titles available. Although inter-library-loans could in most cases order the relevant book from another library, this takes efforts and time. With the rapid growth of the World Wide Web, an increasing number of authors publish their articles online – either directly in HTML, the native markup language of the Web, or in Postscript or PDF format that can be read by specialized viewers. These articles are available from anywhere in the world, anytime of day, as long as the user has a Web connection and browser software.

But finding such online versions of relevant articles on the Web poses a problem in itself. Citations in HTML (hypertext) articles are often directly linked to other online articles, making the retrieval of a cited article a matter of a single mouse click. However, if the original author did not spend the time to find the online addresses for her citations, or when reading a copy on paper or in a non-hypertext format such as Postscript, the reader still has to use conventional search methods in the form of Web indices and directories to find the corresponding article. In these cases, an information agent specialized in academic papers could simplify the search process

by allowing the user to search for the online version of an article, given the names of the author and its title.

This section presents a very simple implementation of a DRS-System in the academic paper domain, which is able to find online versions of an academic paper given the name of its authors and its title. As our initial experiments suggest, we are able to achieve a satisfactory performance with very simple heuristics and our generic DRS architecture.

After briefly describing academic papers on the web, we will examine the 6 elements of Dynamic Reference Sifters, as described in section 3.2, in the domain of academic papers, leading to a rough architecture sketch which is described in section 5.2.2. This is followed by a section describing our experiments and their results.

5.2.1 Description of the Paper Domain

Academic Papers share many features with the previously described homepage domain. We can examine this class of web pages according to the characteristics outlined in section 3.1.2 and summarized in table 3.2 on page 28:

- **Availability:** *Many, but not necessarily all, of their members are available using a more traditional reference source.* As more and more authors provide on-line links to papers they wrote, or those they cited in their on-line articles, a large number of academic papers is fairly accessible using standard search engines like AltaVista.

- **Focused attention:** *During a given search, a user is interested in very few, and often only one, members of the class, and the user can pose a query specific enough to exclude other members.* Finding academic papers is often a very focused search, where a user has found a citation in an article she is reading, and now wants to review the referenced article in full.[1]

- **Strong cohesion:** *Their members are easily identifiable as belonging to the class.* As with Ahoy!, any tool in this domain would need to rely on domain specific methods in order to detect pages (or references) belonging to this class. As it should become clear from our experiments, such an identification method is fairly easy to construct, and offers a surprisingly good performance.

- **Large cardinality:** *They are too large or too dynamic to be exhaustively indexed by hand.* Similar to personal homepages, a large amount of new papers is published everyday, and more and more authors provide copies of their articles on-line.

[1] Although less focused searches are possible, like looking for all papers of a specific author on a certain topic, these searches can be mapped onto a focused search by providing a database frontend like INSPEC (containing abstracts and keyword indexed articles sorted by author) which would then provide the necessary focused information (like title and authors of a relevant paper).

- **Widely dispersed:** *No central repository exists where all or most members can be found.* Although some institutional repositories exist, most online versions of academic articles are published on local servers only.

Academic Paper come in a number of different formats. While Personal Homepages are standard HTML documents that provide elements like *title* or *snippet* in the search service output, academic papers can also be in Postscript, PDF[2] or an archive format such as `tar` or `gz`. Documents in these formats are not indexed by standard Web indices, and thus will not show up in the output of any of the standard services. In order to find articles in these formats, we will have to find *links* to these files from standard HTML pages, for example from the authors personal homepage.

5.2.2 Paper Mate: A Case study

User Interface

The search form for Paper Mate, as shown in figure 5.1, simply contains a field for the name of the authors as well as one for the title of paper. The author field accepts it input in the most common formats used in citations (such as "firstname lastname", "f. lastname", "lastname, firstname", "et al", etc.), so that the user can simply copy the given author name(s) from any available citation. The title field also accepts a substring of the paper title, although care should be taken not to change the word order or omit single words. Also, using a fairly specific part of the title instead of a common one will have a much higher chance of finding the correct article.

During the search, Paper Mate will continuously update the user about its progress[3], before it will present the final answer consisting of references to the paper, citations with links to online versions of it, or possible documents containing information about it (see figure 5.2. In this simple prototype there is still a chance of receiving an uninformative "nothing found" answer, without giving any further, orthogonal information. Future work could include links to the homepage of the author or corresponding institution or conference in the results, in order to allow the user to manually continue her search or contact the author of the article directly. Also, future improvements might allow the user to specify additional information in the search form, like subject keywords, any known institutional affiliation of the author or even the corresponding conference name the article was accepted at.

Control Flow

The fact that some articles might not be indexed directly by standard Web indices has two consequences. First, the query for the *base reference set* has to be specifically geared towards finding not only HTML versions of academic articles, but also documents that contain *links* to non-indexed versions of these papers. Second, for

[2] Adobe's **P**ortable **D**ocument **F**ormat, the successor of the Postscript format.

[3] This feature requires a Netscape browser.

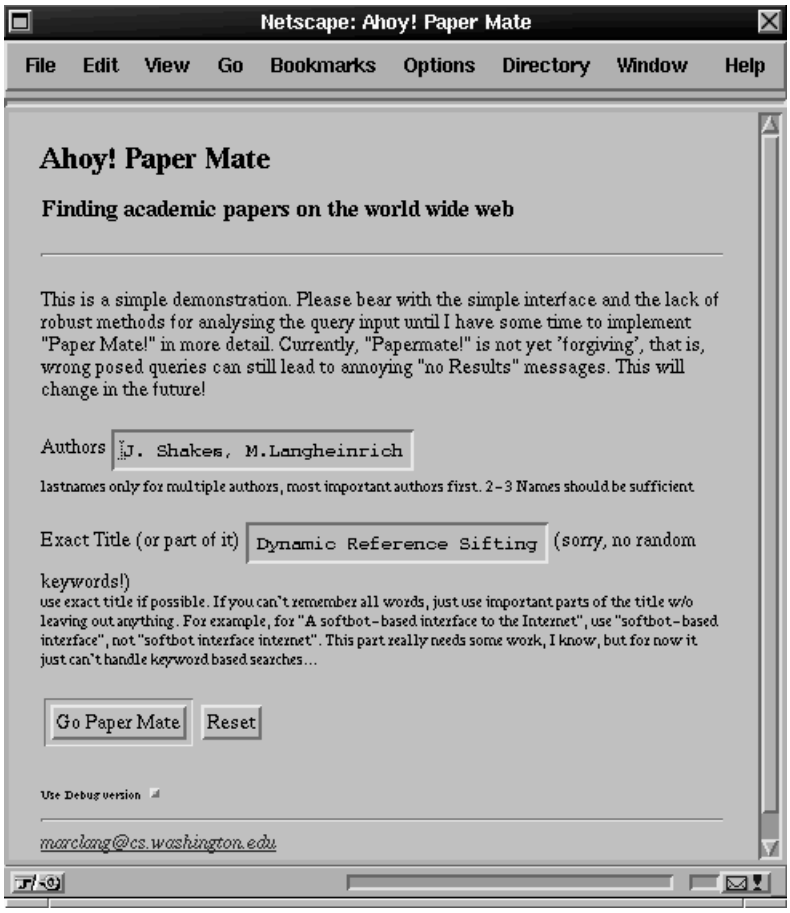


Figure 5.1: **Paper Mate Search Form.** The user is asked for the author name(s) and title of the article. Paper Mate accepts a wide range of name forms common to scientific citations. The title field also accepts a substring of the paper title, although care should be taken not to change the word order or omit single words.

the latter kind of documents the relevant information (*i.e.*, the *link* to the article) will not be directly available in the output of the search services. Instead, the system will have to select a number of references to download and then parse their full text in order to find any of these links.

Figure 5.3 shows the three separate queries that are used to address the first consequence. Instead of a single query to the MetaCrawler, Paper Mate uses the advanced query syntax that is offered by a few Web indices such as AltaVista or HotBot,[4] to search directly both for HTML versions of the article (which feature a corresponding title) and for documents containing *links* to online versions of the paper. In order to compensate for the lack of comprehensiveness when using a single Web index, an additional third query to the MetaCrawler is made, which can be used

[4]The current implementation uses only AltaVista.

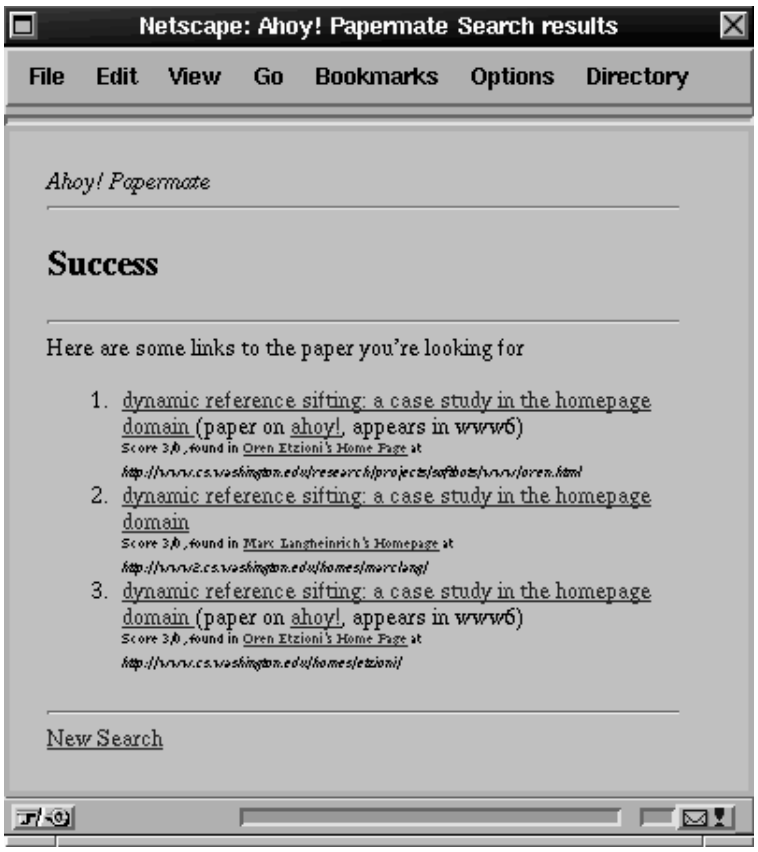


Figure 5.2: **Paper Mate Results Screen.** In case Paper Mate was able to find one ore more references to the desired article, or even the article itself, it displays those together with a brief description (the title and URL of the document that contained the reference).

Query 1 – for HTML articles:

```
author1 NEAR author2 NEAR ...  AND title:"titlephrase"
```

Query2 – for links to articles:

```
author1 NEAR author2 NEAR ...  NEAR "titlephrase"
        AND (link:ps OR link:pdf)
```

Query3 – backup query:

```
author1 AND author2 AND ...  AND "titlephrase"
```

Figure 5.3: **Paper Mate Base Reference Set Queries.** Instead of a single query to the MetaCrawler, Paper Mate uses the advanced query syntax that is offered by a few Web indices such as AltaVista or HotBot to search directly both for HTML versions of the article (which feature a corresponding title) and for documents containing *links* to online versions of the paper.

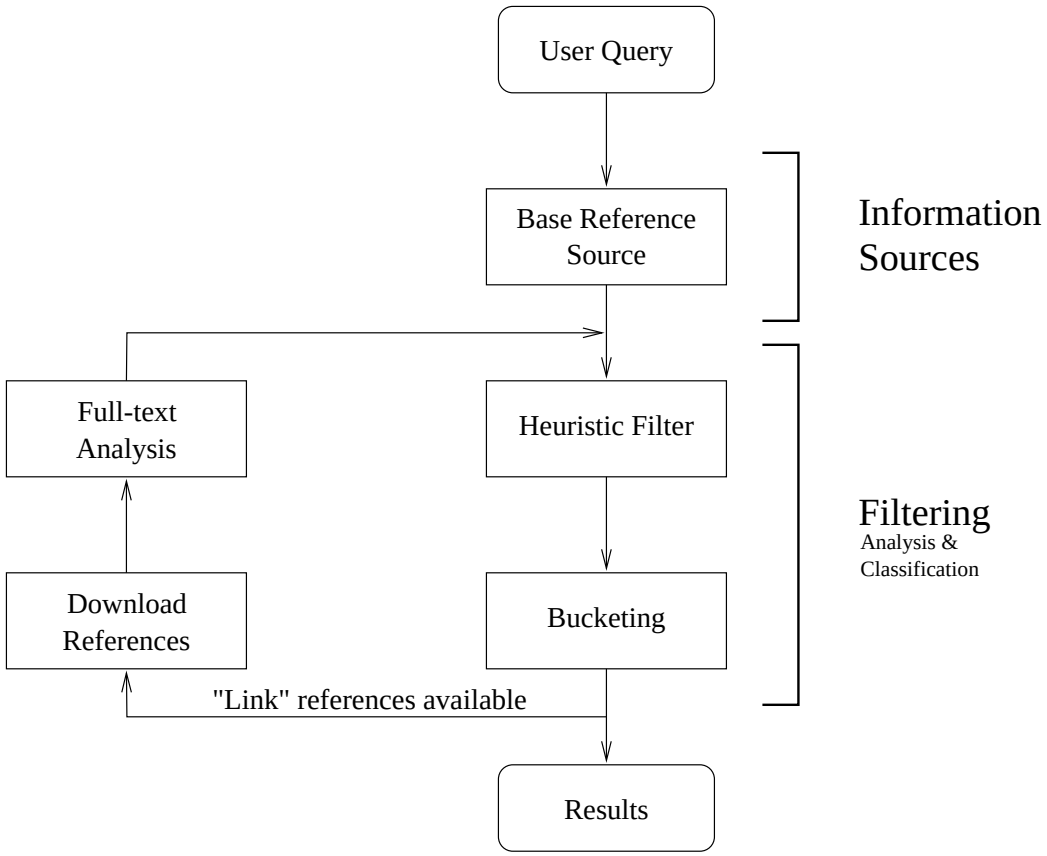


Figure 5.4: **Paper Mate Control Flow.** After sending its three queries to the corresponding search services, the returned references are analyzed and sorted into buckets. Any promising "Link" references are downloaded and their full text searched for embedded links to the desired paper. Any such citation is rewritten as a standard reference and sorted into the appropriate bucket. Once all "Link" references have been downloaded, or a sufficient number of references are available for display, the response is sent back to the user.

as a "backup" in case the first two queries should come up empty.

The second consequence, the need to download the full text of some of the references, can be split up into two problems:

1. **Identifying promising pages to download.** In order to be able to offer a response in a timely fashion, the system should download as few pages as possible.

2. **Finding and extracting the information within the full text of a page.** Simply providing the user with documents containing the link to the desired article is not enough. If the user has to spend a long time reading the whole page herself in order to find the contained link, the added value of the DRS-System is greatly reduced.

The control-flow outlined in figure 5.4 shows the corresponding two-step approach in Paper Mate: After analyzing the initial references from the *base reference set* (which were obtained using the three queries in figure 5.3), a first bucketing phase sorts them into three categories: html-version, contains-link and none.

The first bucket, html-version, holds those references that already feature the correct article title as the HTML title. These references should already constitute valid answer to the user's query. However, a cutoff value determines additional downloads of references in the contains-link bucket. Downloading of additional references continues until all elements in this bucket have been requested, or the minimal number of references have been found.[5]

The full text of each downloaded reference from the contains-link bucket is scanned for embedded links and each link is examined together with its surrounding area (according to HTML markup, *i.e.*, the paragraph, the list item, etc.). After analyzing a number of features, such as the "paper-title match", the "author match", the "link type" and the "size of surrounding area", a *virtual reference* is created, just as if the part containing the link had been found directly as a reference in the *base reference set*.

These newly created references are sorted into the existing table, now allowing two more buckets containing paper-citations, which holds successful citations that include a link to a Postscript or PDF version of the paper, and citations-w/o-link, which contains those that simply cite the paper but do not offer a corresponding link.

As soon as a sufficient number of references has been sorted into both the html-version and paper-citation bucket, the download is interrupted, and the content of these buckets is displayed to the user. If none of these buckets could be found, the most promising references of the other buckets are shown instead.

5.2.3 Experiments

Experimental Setup

We used a simple, qualitative experiment to test the feasibility of our DRS-System in the Academic Paper domain. Using a set of academic paper citations that are known to have an online version on the Web, we used their author and title information to query Paper Mate and recorded the number of papers we could find this way. In addition, we compared its performance to the WebFind system [ME96], a (non-public)[6] Web service similar in scope to Paper Mate, which was developed at the University of California, San Diego.

WebFind uses knowledge guided search, similar to the early Ahoy! prototype described in section 4.2.1:

1. A query to the Melvyl database, a traditional information retrieval system for magazines and journal articles at the University of California, determines the

[5] This is done to increase the chance of "getting it right" for the simple prototype.

[6] Thanks to the authors for allowing access to the WebFind system during our experiments.

institutional affiliation of the principal author. By specifying the author's name and a set of keywords, the user is first presented with a list of articles by the author on the given topic, and can then select the articles WebFind should search for.

2. Using the publicly accessible Netfind directory, which finds Internet addresses given the name of an institution, a candidate set of Internet domains is compiled which allow WebFind to find the URL of the institutional homepage.

3. Starting from the institution's homepage, WebFind analyzes and follows hypertext links in order to find the authors personal homepage.

4. A similar strategy is then used to find the online version of the desired paper off the authors homepage.

Since WebFind takes a list of *keywords* instead of the paper title, we first manually tried a number of keywords (or none, to list all available articles of the author), as well all names of co-authors, to find the matching entry of the desired paper in the Melvyl database.

The list of papers was taken from the proceedings of a recent Web conference [Gen97]. Choosing articles at random, one or two references were selected that appeared in the bibliography and included a URL address for the online version of the cited article.

Experimental Results

Figure 5.5 gives an overview of the results. Paper Mate is able to find over 90% of the test examples, while WebFind fails to find a single one. The breakdown in figure 5.5(b) helps explain the dismal failure of WebFind: Only about half of the papers in the test set were indexed in the Melvyl database, WebFind's principal source for the institutional affiliation of the author. For half of the remaining references, Netfind was then unable to find a corresponding server, so that WebFind could continue its search on the institutional homepage. Many of these references featured institutions in Europe, which are apparently not accessible via Netfind.[7] Finally, from only two of the remaining 9 institutional homepages WebFind was able to find a link to the apparent homepage of an author – one for a completely different person, the other one a simple `mailto:`[8] link. Not surprisingly, WebFind could not locate a link to the desired papers from any of the two.

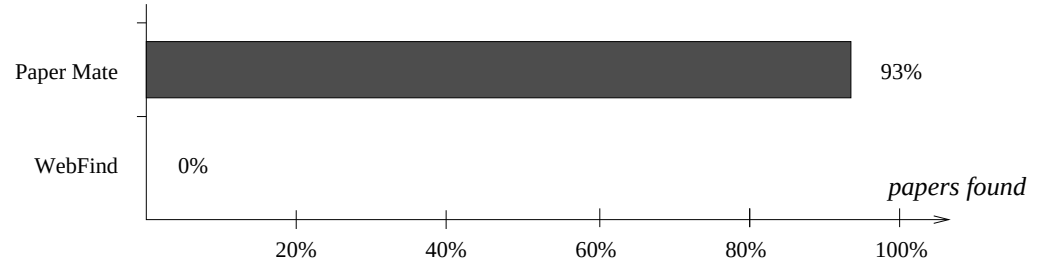
The performance of WebFind is a good example for the problems of such a sequential approach: at each step, the chances of failure increase exponentially. Demonstrating a roughly 50% chance of failure at each step, the 4-step WebFind algorithm

[7] However, Netfind was also unable to find a server for the institution named "Lab. for Comput. Sci., MIT, Cambridge, MA, USA".

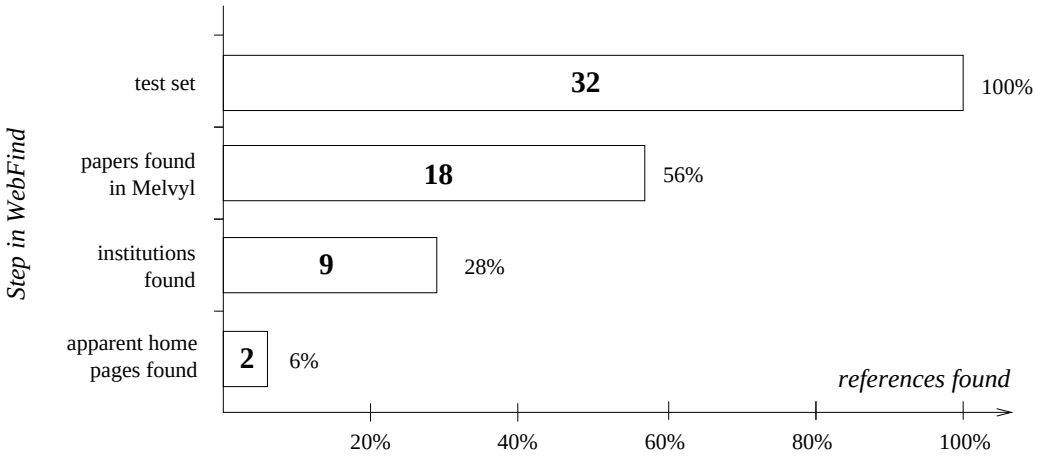
[8] Instead of a `http:` link, which connects a page of hypertext to another page of hypertext (or at least to a picture, video or sound), the `mailto:` link only prompts the users browser to send an email message to the specified email address.

	Paper Mate		WebFind		
Article	found	time (s)	found	time (s)	problem
1	yes	42	no	-	Netfind
2	yes	54	no	-	Melvyl
3	yes	49	no	-	Melvyl
4	yes	39	no	-	Melvyl
5	yes	25	no	-	Netfind
6	yes	35	no	-	Melvyl
7	yes	27	no	-	Melvyl
8	yes	15	no	210	no paper
9	yes	17	no	-	Melvyl
10	yes	34	no	150	no hp
11	no	117	no	90	no hp
12	yes	47	no	190	no hp
13	yes	43	no	95	no hp
14	no	36	no	160	no hp
15	yes	24	no	-	Melvyl
16	yes	23	no	120	no hp
17	yes	40	no	-	Netfind
18	yes	39	no	-	Netfind
19	yes	85	no	130	no hp
20	yes	41	no	120	no paper
21	yes	39	no	-	Melvyl
22	yes	43	no	-	Netfind
23	yes	33	no	120	no hp
24	yes	36	no	-	Melvyl
25	yes	19	no	110	no hp
26	yes	44	no	15min	timeout
27	yes	47	no	-	Netfind
28	yes	29	no	-	Melvyl
29	yes	40	no	-	Netfind
30	yes	44	no	-	Melvyl
31	yes	21	no	-	Netfind
32	yes	28	no	-	Melvyl
Total	**30**	**∅ 39**	**0**	**∅ 135**	

Table 5.1: **Paper Mate Results per Article.** A list of 32 articles were used to compare the performance of Paper Mate with the similar WebFind system. Paper Mate failed to find 2 of the papers, while WebFind was unable to locate a single article from the test set. The reason for WebFind's failure is given in the last column: "Melvyl", "Netfind", "no hp" and "no paper" correspond to WebFind's four steps algorithm described in section 5.2.3.



(a) Final Results



(b) WebFind Breakdown

Figure 5.5: **Paper Mate Experimental Results.** Out of the 32 papers in the test set, Paper Mate finds 30, or 93% (a). WebFind fails to find a single one, mostly due to its sequential search algorithm. (b) shows how at each step WebFind fails to find about half of the references. Only 18 papers could be found in Melvyl, and for only 9 of them WebFind was able to find an institutional homepage. At these 9 institutions, WebFind only found 2 apparent homepages of the authors, both of which were incorrect. As a result, WebFind could not find a single reference.

Step	# of references
Original Reference Set	32
Containing full name information	8
Ahoy! found personal homepage	8
Manually located online version of paper	5

Table 5.2: **Results of Follow-up Experiment.** Of the 32 papers checked with Paper Mate, only 8 included the full firstnames (instead of the initials only) of the authors. Of these 8 references, Ahoy! always found a personal homepage for at least one of the authors, if not all of them. The input to Ahoy! was only the given first- and lastname, since no institutional information was present in the citations. By following links from these homepages, 5 papers could be located manually.

(Melvyl, institutional homepage (using Netfind), author homepage, paper link) has only a $1/2^4$, or about 6%, chance of success. When confronted with a similar low percentage of found papers during their experiments, the WebFind authors argue that "many authors have not yet put papers online" [ME96]. However, as our qualitative result suggests, even those articles that are already available are hard to find with such an approach.

Follow-up Experiment

One might argue that a specialized paper finder such as Paper Mate is unnecessary, once a system like Ahoy! exists: Instead of locating the desired article from scratch, one could follow the approach taken by WebFind and simply locate the personal homepage of the author in order to find a link to the paper.

In a follow-up experiment, we tried to determine the number of papers we could find using Ahoy!– first searching for the personal homepage of one of the authors and then locating the link to the relevant article manually. The results are summarized in table 5.2.

Of the 32 papers checked with Paper Mate, only 8 included the full firstnames (instead of the initials only) of the authors (Ahoy! currently requires a firstname) – the remaining 3/4 of the existing citations were not suitable for Ahoy!. Of these 8 references, Ahoy! always found a personal homepage for at least one of the authors, if not all of them. The input to Ahoy! was only the given first- and lastname, since no institutional information was present in the citations. By following links from these homepages, we were than able to manually locate 5 papers, including one that had not been found before when using Paper Mate.

Although over half of the personal homepages we found featured direct or indirect links to the corresponding articles, the success of such a resource discovery approach using best-first link traversal (as in WebFind) remains questionable, given the number

of indirect links to follow[9] and the increasing use of graphics and imagemaps for such links. However, using Ahoy! as an orthogonal information source in the current Paper Mate system might be a promising approach to further increase its coverage, as well as providing helpful information (in form of the author's homepage) in case a search should fail.

5.3 Jokes

Our final example tries to apply the concept of DRS in a very different domain. While Paper Mate's academic papers were conceptually very close to Ahoy!'s personal homepages, our last domain, online jokes, are hardly found on the personal homepages of researchers.

How often have you found yourself in the situation where you just recently heard this great joke but could not remember the punch line anymore? Simply enter some (distinctive) keywords, together with some longer phrase that you still remember, and Joker!, our DRS-System in the on-line joke domain, will find the full text of the almost forgotten joke for you.

5.3.1 Description of the Joke domain

We begin again by examining our standard DRS-Set characteristics from table 3.2 on page 28:

- **Availability:** The simplest way to share jokes on the Web is as simple text file or as plain HTML document. Both formats can be indexed by standard Web indices to form Joker!'s *base reference set*. Only few of the larger repositories offer database entries, which are usually not indexed by traditional search services.

- **Focused attention:**In our case, the goal is to find the exact text of an almost forgotten joke. Although a similar scenario would be possible, where users would search for new jokes in a certain category, this would not fit into our DRS framework.

- **Strong cohesion:** Although it seems hard to analyze the full text of a document in order to determine whether or not it constitutes a joke, our experiments show that relative simple heuristics, together with the Web's richness of information, can make identification of these pages fairly straightforward.

- **Large cardinality:**Joke pages are typically maintained by individuals who like to share their favorite laughs with others on the web. Many maintainer ask for new jokes sent to them via email, and try to post new entries every so often.

[9]Two of the personal homepages were pointing to the homepage of the authors' research group, which in turn pointed to a repository of technical articles.

Figure 5.6: **Joker! Search Form.** The user is asked for a prominent phrase and additional keywords of the joke she is trying to find. Although Joker! accepts searches featuring only keywords or only a phrase, using both input fields together allows highest precision.

- **Widely dispersed:** Very few archives feature more than couple of hundred jokes, more often only one or two dozens.

Jokes usually appear on HTML pages or simple text pages (without any HTML markup). Some pages feature a single joke, but more often a number of jokes are found listed on a single page, separated by HTML markup (paragraphs, list elements) or text elements (horizontal lines, numbers, etc.). If Joker! can find the full text of the desired joke on a page by itself, returning this reference should be sufficient. However, simply returning a reference to a page containing the desired joke next to tens or hundreds of others is not enough – in this case the system has to extract the relevant part of the page only.

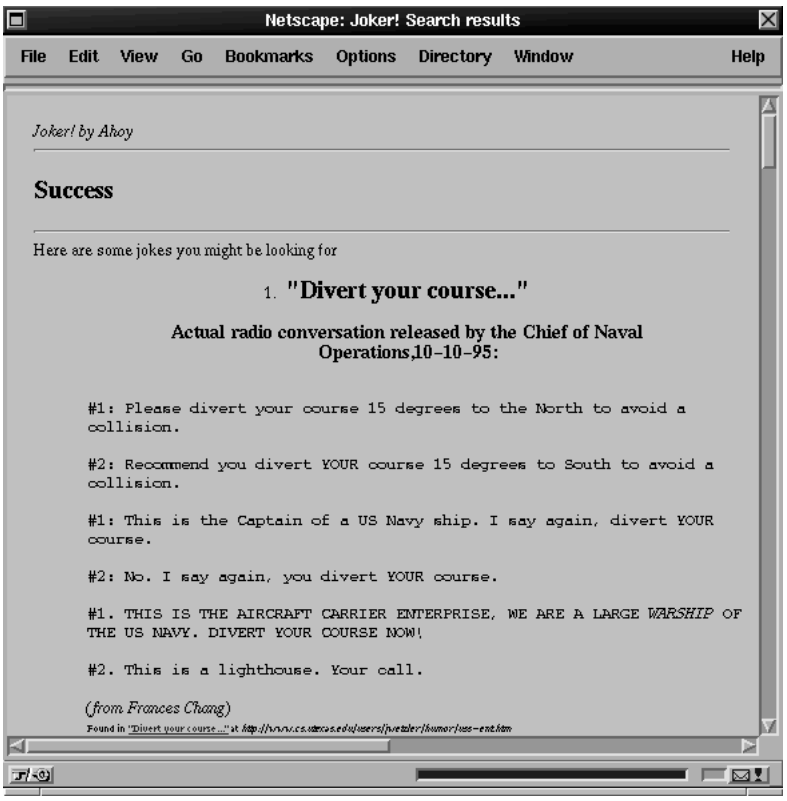


Figure 5.7: **Joker! Results Screen.** In case Joker! was able to find one ore more documents featuring the desired joke, it displays its full text those together with a brief description of its source (*i.e.*, the title and URL of the document that contained the joke).

5.3.2 Joker!: A Case study

User Interface

The search form for Joker! is shown in figure 5.6. The user supplies preferably a distinctive phrase (or part of it) of the joke she is trying to find, together with a number of words that appear in the joke. Using a fairly uncommon phrase of the joke, together with any unusual words used, increases the chances of finding the correct joke.

After updating the user of the search progress, Joker! finally lists the full text of each joke that it found, together with a link to the page it found this joke on (see figure 5.7. In case it was unable to extract the full text properly, it gives a link to the page only, together with an appropriate message. Joker!, too, can return a "nothing found" answer.

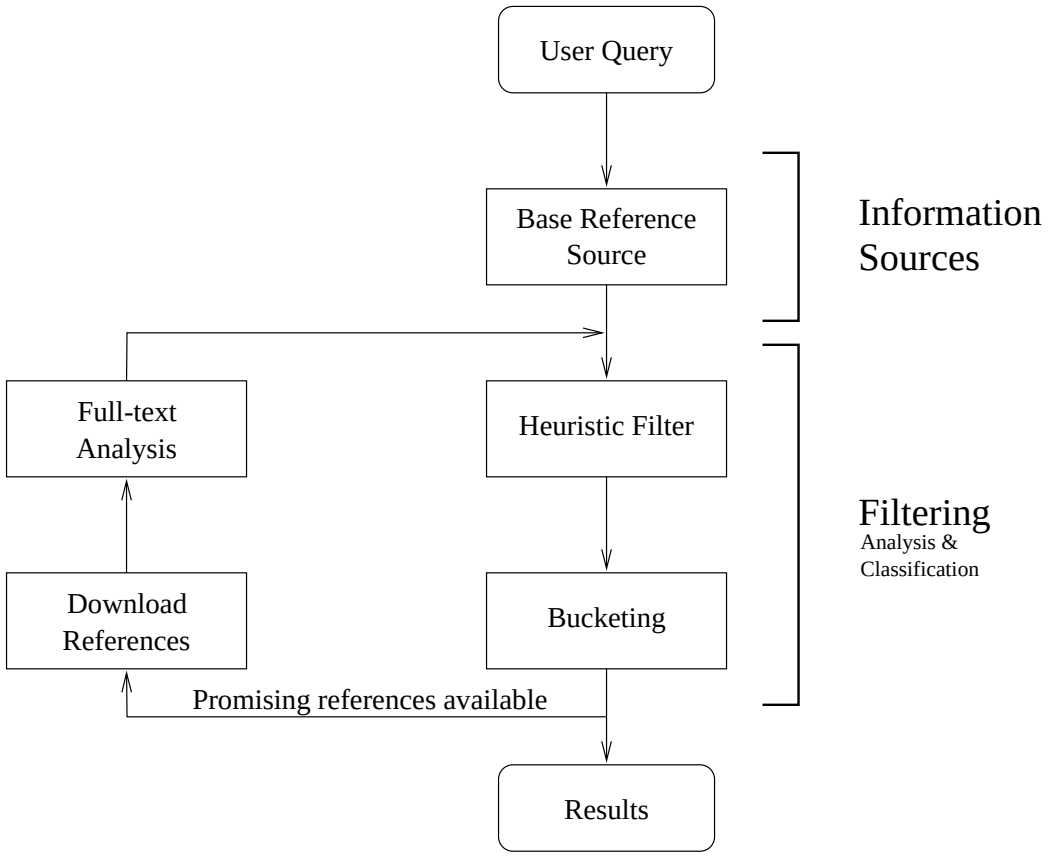


Figure 5.8: **Joker! Control Flow.** After sending its two queries to the corresponding search services (the MetaCrawler and AltaVista), the returned references are analyzed and sorted. Any promising references are downloaded and their full text searched for sections or paragraphs containing the keywords and phrase specified by the user. Any such section is extracted and a corresponding entry is created, which can then be displayed to the user.

Control Flow

The main problem of searching for jokes is the fact that they often contain very common words. When using a query to a standard search engine, a request featuring common words can easily result in a very large answer set. Although we like to maximize Joker!'s recall, we will have to take care not to formulate a too general query. Instead, we want to make sure that our *base reference set* contains a large number of "jokes-pages" featuring the desired words, and not just regular pages.

Joker! tries to provide a slightly focused query by adding a number of so called "joke words" to the user supplied keywords and phrases. Currently, three of these words are used: "joke," "fun," and "humor;" although this might unnecessarily biases Joker! to find predominantly jokes in English. Joker! uses a query to the MetaCrawler, as well as an advanced query to AltaVista. The former query simply

requires all keywords, plus any of the "joke words", and groups the words of the phrase together. The latter query additionally requires any of the "joke words" to appear in the URL or title of the document (thus having a higher chance of finding a "joke-pages").

Once all references have been returned, Joker! sorts them according to the number of keywords in title, snippet or URL; if a phrase match in title or snippet could be found; and if any "joke words" appear in the title or the URL of the document. Then, similar to the Paper Mate system, each promising reference is downloaded and the full text analyzed.

On each downloaded page, Joker! tries to isolate the piece of text containing the relevant keywords and phrase in order to build a "joke" reference containing the full text of the joke together with some information about the page it was found on. In case it was unable to successfully extract the full text, but found all relevant parts of the joke, it will create a reference to the corresponding document only, so that the user has a chance to manually search for the desired joke.

Finally, when enough "joke" references have been extracted,[10] or no more pages can be downloaded, Joker! displays the list of extracted jokes to the user, as shown in figure 5.7. However, if Joker! was unable to extract any joke, any available links to promising pages are shown instead.

5.3.3 Experiments

In our brief experiment, we tried to asses if our simple implementation of a DRS-System would prove sufficient in a highly unstructured domain such as online jokes. The idea was to take a list of jokes that are known to be available online and try to find the full text using Joker!.

Experimental Setup

With the help of fellow students and professors in the Softbots group at the University of Washington we compiled a list of 22 jokes. Every joke had either reached one or the other participant via email or had been found on the Web. For each joke, we asked to supply a phrase, together with a number of keywords. The obvious bias in our experiments, the selection of the phrase and keywords, could be eliminated by using queries that were posed by online users, once the service would be publicly accessible. However, in order to get a qualitative result, this test set should be sufficient.

Experimental Results

Joker! was able to find the full text of 21 out of the 22 jokes, with one of them being a variation of the original joke (only some proper names had been changed). Although these results are not sufficient evidence for the *recall'* or *precision'* of the service, nor

[10] Similar to Paper Mate, Joker! tries to find a small number of identical references before ending the search, in order to increase its chances of "getting it right".

do we have any data on how existing services can be used to find these jokes, our experiment at least demonstrates feasibility of our approach.

5.4 Summary & Conclusions

DRS has shown to be an effective IR tool in domains other than personal homepages. With two simple, 10 day efforts, we were able to provide two high quality search services using our generic DRS architecture: assembling a large reference set from standard sources, augmenting these with additional information, using heuristics to sort and filter, and providing flexible display methods for returning usable information even in case of failure. Although both prototypes used only minimal mechanisms in each of the areas and did not investigate the possibility of learning in these domains, our results suggest that fielded systems could show the same proficiency in their respective domains as the current Ahoy! system.

6 Conclusions and Future Work

6.1 Summary & Conclusions

This thesis described an attempt at a fully implemented system that satisfies the increased expectations towards an effective information retrieval system on the Web. The technology used in current Web Search Services like Directories or Indices has been developed long before the recent success of the World Wide Web: Web Directories, using techniques of manual indexing that work well for only moderately fast growing fields, such as Yellow Pages or a library, cannot keep up with the full size of the rapidly expanding Web. Indices, using techniques developed to maximize recall on often fairly static datasets, often feature comprehensiveness beyond usefulness. Using intelligent agents that allow users to delegate time consuming, tiresome tasks, we can create systems that reuse existing services and offer a functionality that is more than the sum of its parts.

Our work on DRS and the prototype systems, Ahoy!, Paper Mate and Joker!, contributed to the field of Information Retrieval in a number of ways. This thesis

- *demonstrated the feasibility of providing high accuracy without sacrificing coverage*, even in highly unstructured domains such as the World Wide Web. The Ahoy! system achieves a remarkable performance when compared to standard search services, showing more than twice the accuracy of its closest competitor, while still offering the largest coverage.

- *developed a domain-independent architecture that works in a variety of domains*, enabling the rapid prototyping of simple, yet powerful prototypes. Using the general DRS architecture, we were able to quickly construct two simple DRS-Systems in different domains, Joker! and Paper Mate, both offering high accuracy combined with high coverage on the available test sets.

- *proposed a novel way of resource location on the Web*, by learning about the structure underlying a specific domain. In one experiment, Ahoy! was able to find nine percent more references than its closest competitor by using its URL extraction and generation methods.

- *examined simple machine learning techniques and their effects on retrieval performance*, by comparing three different strategies for resource ordering in Ahoy!'s URL learning module. Using the *Laplace accuracy*, the Ahoy! system is

able to increase the effectiveness of its *direct search* feature by a factor of 12 compared to unsorted URL generation.

But our work on DRS-Systems has only just begun. As Etzioni puts it, we "start out with something useful, and then promise that we will add intelligence afterwards" [Etz96]. After having demonstrated the feasibility of our concept, future work in the field of DRS could lead to new insights into the nature of the Web and the retrieval of information from large, distributed full-text databases.

6.2 Future Work

DRS-Systems are positioned advantageously as publicly available Web search services. Without trying to compete with large commercial services like Lycos or AltaVista, they can still attract a significantly larger amount of users than other academic Web prototypes. With the data available of more and more users using a fielded DRS-System, several future work areas open up.

6.2.1 Extending the functionality of existing DRS-Systems

Although the existing systems already exhibit a satisfactory performance, many things are still far from perfect. Extending the capabilities of existing implementations would not only make these services more attractive to the user, but also increase the amount of data available for experiments – an important advantage of fielded systems.

Extending Ahoy!'s query capabilities

The experiments in the academic paper domain have shown that in a lot of cases, the full name of a person might not be known. Extending Ahoy! to handle firstname initials would not only attract more users, but also make it possible to use it as an orthogonal reference source in Paper Mate. Using additional, advanced query features of popular Web indices it would even be possible to search for the lastname only, as long as institutional information is available.

Publicly fielding Paper Mate and Joker!

Before offering the services of Paper Mate and Joker! to the public, these simple prototypes would need a more robust implementation. Including more Web indices such as HotBot or Infoseek could increase the number of high quality base references used in each system (compared to the fall-back set obtained through the MetaCrawler). Additional orthogonal sources, such as university wide magazine and journal databases, or even Ahoy!, could extend Paper Mate to augment a search with the names of additional authors, perform title correction, provide the institutional affiliation of the authors, and allow keyword searches for a more general topic. In case of a failed

search, Paper Mate could offer the personal homepages of the authors and their institutions. Joker! could for example profit from including a number of large jokes repositories into its base reference set. Finally, both systems could use the concept of a URL Learner to further increase their coverage.

Field additional DRS-Systems

As outlined in section 3.2.4, DRS-Systems could be implemented in a number of other domains. This could not only confirm the design of the generic framework, but also add useful conceptual additions to the existing architecture.

6.2.2 Extending the generic DRS framework

The current DRS framework is very simple. Using data from the fielded prototypes, many components of the architecture could be improved or generalized. A number of research areas suggest themselves in this context:

Combining heterogeneous information sources

Although the process of cross filtering, where we combine the information of available orthogonal information with the information coming from the base reference set, resembles a database *join*,[1] and has been widely studied in the context of heterogenous information sources (*i.e.*, Levy's Information Manifold [KLSS95]), we have to be cautious when using unstructured information sources such as Web indices or directories.

With these Web resources, any join between two result sets can only be of probabilistic nature, since there is no guarantee that the results of a query are necessarily related to the concept searched for.[2]

Instead of explicitly developing an algorithm for the respective combination of information sources used in each system, a declarative semantic could be introduced in order to characterize each orthogonal information source and then use a generic mechanism to combine the results, no matter what sources are combined in a specific domain.

Minimizing false hypotheses

Further investigating the effects of different statistical measures to rank existing *general hypotheses* could significantly improve the performance during the direct search of a DRS-System. Also, with more and more hypotheses acquired per site, we

[1] A *join* is a binary operation of relational algebra, combining two sets that share common fields while enforcing equality on those fields that appear in both sets. See any introductory database textbook, such as [KS91].

[2] This is both due to shortcomings of the query language and the full text nature of standard search indices/directories.

need to have a method of pruning unsuccessful hypotheses, as well as preventing the DRS-System from relearning them.

Cross domain learning

Although Ahoy! already uses the concept of *cross domain learning*, its effects are not yet understood. In the experiment described on page 77, we only used information that was directly recorded at each site. Further experiments would be necessary to examine the benefits of having additional, global information available in case a particular hypothesis has rarely been used at a certain site. This could also clarify how to connect local and global statistics to form a single confidence value

Learning domain features

While current DRS-Systems have to be manually tuned in order to achieve optimal feature analysis, future systems could examine the use of machine learning techniques to gradually improve an initial set of heuristics over time. Using direct user feedback *i.e.*, the user explicitly labels a returned reference as incorrect or correct, a DRS-System could revise suboptimal results and update its feature detection algorithm as new features become relevant.

Bibliography

[AFJM95] R. Armstrong, D. Freitag, T. Joachims, and T. Mitchell. Webwatcher: A learning apprentice for the world wide web. In *Working Notes of the AAAI Spring Symposium: Information Gathering from Heterogeneous, Distributed Environments*, pages 6–12, Stanford University, 1995. AAAI Press. To order a copy, contact sss@aaai.org.

[Aha] David Aha. David Aha's list of machine learning and case-based reasoning home pages. See `http://www.aic.nrl.navy.mil/~aha/people.html`.

[BDH+94] C. Mic Brown, Peter B. Danzig, Darren Hardy, Udi Manber, and Michael F. Schwartz. The Harvest information discovery and access system. In *Proceedings of the Second International World Wide Web Conference*, pages 763–771, 1994. available from ftp://ftp.cs.colorado.edu/pub/cs/techreports/schwartz/Harvest.Conf.ps.Z.

[BDH+95] Mic Bowman, Peter Danzig, Darren Hardy, Udi Manber, Mike Schwartz, and Duane Wessels. The Harvest homepage broker, 1995. See the file `afs/transarc.com/public/trg/Harvest/demo.html` on `http://harvest.transarc.com/`.

[Big97] BigFoot Partners L.P. Welcome to bigfoot - bigfoot homepage, 1996, 97. See `http://www.bigfoot.com/`.

[BL89] Tim Berners-Lee. Uniform resource locators (urls), March 1989. See `http://www.w3.org/pub/WWW/History/1989/proposal.html`.

[BM85] D.C. Blair and M.E.Maron. An evaluation system of retrieval effectiveness for a full-text document retrival system. *C. ACM*, 28(3):289–299, 1985.

[Bra92] R. Brachman. "Reducing" CLASSIC to Practice: Knowledge Representation Theory Meets Reality. In *Proc. 3rd Int. Conf. on Principles of Knowledge Representation and Reasoning*, October 1992.

[Bro91] R. Brooks. Intelligence without representation. *Artificial Intelligence*, 47(1–3):139–159, 1991.

[BS92] Christine L. Borgman and Susan L. Siegfried. Getty's Synoname^{tm} and its cousins: A survey of applications of personal name-matching algorithms. *Journal of the American Society for Information Science*, 43(7):459–476, 1992.

[BS95] Marko Balabanović and Yoav Shoham. Learning information retrieval agents: Experiments with automated web browsing. In *Working Notes of the AAAI Spring Symposium: Information Gathering from Heterogeneous, Distributed Environments*, Stanford University, 1995. AAAI Press.

[Bus45] V. Bush. As we may think. *Atlantic Monthly*, 1945. See the file `~duchier/misc/vbush/vbush.html` on `http://www.isg.sfu.ca/`.

[Cai93] Robert Cailliau. A little history of the world wide web, 1993. See `http://www.w3.org/pub/WWW/History.html`.

[CB91] Peter Clark and Robin Boswell. Rule induction with CN2: some recent improvements. In *Machine Learning - EWSL-91. Proceedings of the European Working Session on Learning.*, pages 151–163, Porto, Portugal, March 1991.

[CFM+97] Mark Craven, Dayne Freitag, Andrew McCallum, Tom Mitchell, Kamal Nigam, and Choon Yang Queck. Learning to extract symbolic knowledge from the World Wide Web. In *Proceedings of the 14th International Conference on Machine Learning*, 1997.

[CKPT92] D. Cutting, D. Karger, J. Pedersen, and J. Tukey. Scatter/gather: A cluster-based approach to browsing large document collections. In *15th Annual Int'l SIGIR92*, 1992.

[Cle65] C. W. Cleverdon. The testing and evaluation of the operatin gefficientcy of the intellectual stages of information retrieval systems. In P. Atherton, editor, *Int. Conf. on Classification Research*, Munksgaard, Copenhagen, 1965.

[CSO96] Hsinchun Chen, Chris Schuffels, and Richard Orwig. Internet categorizations and search: A self organizing approach. *Journal of Visual Communication and Image Representation*, 7(1):88–102, 1996.

[DEC95a] DEC. The AltaVista search engine homepage, 1995. `http://altavista.digital.com/`.

[Dec95b] John December. List of homepage providers, 1995. See the file `cmc/info/culture-people-lists.html` on `http://www.december.com/`.

[Dei95] Peter Deitz. HouserNet, homepage repository, 1995. See `http://www.housernet.com/`.

[DEW97] R. Doorenbos, O. Etzioni, and D. Weld. A scalable comparison-shopping agent for the World-Wide Web. In *Proc. Autonomous Agents*, pages 39–48, 1997.

[DoCS96] University of California Riverside Department of Computer Science. OKRA directory service, 1996. Out of Service. See `http://okra.ucr.edu/okra/`.

[Dou97] DoubleClick, Inc. Internet Address Finder, 1997. See `http://www.iaf.net/`.

[Eic94] David Eichmann. Ethical web agents. In *Proceedings of the Second International World Wide Web Conference '94: Mosaic and the Web*, Chicago, IL, 1994.

[ES92] Oren Etzioni and Richard Segal. Softbots as testbeds for machine learning. In *Working Notes of the AAAI Spring Symposium on Knowledge Assimilation*, pages 43–50, 1992.

[Etz96] O. Etzioni. Moving up the information food chain: softbots as information carnivores. In *Proc. 13th Nat. Conf. on AI*, 1996. available from `http://www.cs.washington.edu/homes/etzioni/`.

[EW94] O. Etzioni and D. Weld. A softbot-based interface to the Internet. *C. ACM*, 37(7):72–6, 1994.

[EW95] Oren Etzioni and Daniel Weld. Intelligent agents on the Internet: Fact, fiction, and forecast. *IEEE Expert*, 10(4):44–49, 1995.

[FG97] S. Franklin and A. Graesser. Is it an agent, or just a program?: A taxonomy for autnonomous agents. In J.P. Mueller, M.J. Wooldridge, and N.R. Jennings, editors, *Intelligent Agents III. Agent Theories, Architectures, and Languages. ECAI '96 Workshop (ATAL) Proceedings. Budapest, Hungary. 12-13 Aug. 1996*, pages 21–35. Springer-Verlag, Berlin, Germany, 1997. See the file `~franklin/AgentProg.html` on `http://www.msci.memphis.edu/`.

[Gen97] M. Genesereth, editor. *Proceedings of the Sixth International WWW Conference*, Santa Clara, CA USA, 1997. See `http://www6conf.slac.stanford.edu/index.html`.

[Gib94] William Gibson. *Neuromancer*. Ace Books, 3rd edition, 1994.

[Goy97] Ambuji Goyal. The promise of information technology, June 1997. UW-CSE Colloquium, June 4th 1997.

[HBML95] Kristen Hammond, Robin Burke, Charles Martin, and Steven Lytinen. FAQ finder: A case-based approach to knowledge navigation. In *Working Notes of the AAAI Spring Symposium: Information Gathering from Heterogeneous, Distributed Environments*, pages 69–73, Stanford University, 1995. AAAI Press. To order a copy, contact sss@aaai.org.

[Hot96] Hotbot. The hotbot search engine homepage, 1996. `http://www.hotbot.com`.

[Hot97] Hotbot. Altavista is history. *Wired Magazine*, 5(7):164, July 1997.

[Hoy95] Rhese S. Hoylman. People Pages, homepage repository, 1995. See `http://www.peoplepage.com/`.

[Inf96] Infoseek Inc. Infoseek search engine homepage, 1996. `http://www.infoseek.com`.

[Inf97] Infoseek Inc. Ultraseek search engine homepage, 1997. `http://ultra.infoseek.com`.

[Int97] Internet Literacy Consultants^{tm}. ILC glossary of internet terms, 1994–1997. See the file `files/glossary.html` on `http://www.matisse.net/`.

[Joh97] W. Lewis Johnson, editor. *Proceedings of the First International Conference on Autonomous Agenst*. ACM press, February 1997.

[Kay90] Alan Kay. User interface: A personal view. In Brenda Laurel, editor, *The Art of Human Computer Interface Design*, pages 191–207. Addison-Wesley, 1990.

[KLSS95] T. Kirk, A. Levy, Y. Sagiv, and D. Srivastava. The Information Manifold. In *AAAI Spring Symposium: Information Gathering from Heterogeneous, Distributed Environments*, pages 85–91, 1995.

[Knu73] Donald E. Knuth. *The Art of Computer Programming*, volume 3. Addison-Wesley, Reading, Massachusetts, 1973.

[Kos95a] Martin Koster. Guide for robot writers, 1995. See the file `mak/doc/robots/guidelines.html` on `http://www.webcrawler.com/`.

[Kos95b] Martin Koster. A standard for robot exclusion, 1995. See the file `mak/doc/robots/norobots.html` on `http://www.webcrawler.com/`.

[KS91] Henry F. Korth and Abraham Silberschatz. *Database System Concepts, 2nd edition*. McGraw-Hill, 1991.

[KSC94] Henry Kautz, Bart Selman, and Michael Coen. Bottom-up design of software agents. *C. ACM*, 37(7):143–146, 1994.

[KWD97] N. Kushmerick, D. Weld, and R. Doorenbos. Wrapper Induction for Information Extraction. In *Proc. 15th Int. Joint Conf. on AI*, 1997.

[Lak97] Matthew Lake. Search engine shootout! *PC Computing*, August 1997. See the file `pccomp/interdot/intermar/web_srch.html` on `http://www.zdnet.com/`.

[Lan94] Ken Lang. NewsWeeder: Learning to filter netnews. *somewhere*, 00(0), 1994.

[Leb] Alexander Lebedev. Best search engines for finding scientific information in the Net. See `http://www.chem.msu.su/eng/comparison.html`.

[Lot97] Mark Lottor. Internet domain survey, 1997. See the file `zone/report.doc` on `http://www.nw.com/`.

[LRO96] A. Levy, A. Rajaraman, and J. Ordille. Query-answering algorithms for information agents. In *Proc. 13th Nat. Conf. on AI*, 1996.

[Lyc96] Lycos, Inc. Lycos Internet Directory homepage, 1996. `http://a2z.lycos.com`.

[MCF+94] Tom Mitchell, Rich Caruana, Dayne Freitag, John McDermott, and David Zabowski. Experience with a learning personal assistant. *C. ACM*, 37(7):81–91, 1994.

[ME96] Alvaro E. Monge and Charles P. Elkan. Integrating external information sources to guide worldwide web information retrieval. Technical report, Computer Science and Engineering Department, University of Calfornia, San Diego, 1996. See `http://dino.ucsd.edu:8000/`.

[Mel97] Melee Inc. Melee's indexing coverage analysis (MICA), 1996–1997. See the file `mica/index.html` on `http://www.melee.com/`.

[Mon97] Louis Monier. Talkback: AltaVista CTO responds, March 1997. See `http://www4.zdnet.com/anchordesk/talkback/talkback_13066.html`.

[NCS97] Glossary for ncsa mosaic and the world wide web users, July 1997. See the file `SDG/Software/Mosaic/Glossary/GlossaryTable.html` on `http://www.ncsa.uiuc.edu/`.

[Nib87] Tim Niblett. Constructing decision trees in noisy domains. In *Progress in Machine Learning (Proceedings of the 2nd European Working Session on Learning)*, pages 67–78, Wilmslow, UK, 1987.

[Pal] Bruce Palmer. Search engine comparisons around the web. See the file `~bwp2/isearch8.html` on `http://jan.ucc.nau.edu/`.

[Pau97] Kathryn Paul. Ongoing search engine analysis, 1997. See `http://burns.library.uvic.ca/searchengineanalysis.html`.

[PDEW97] M. Perkowitz, R. Doorenbos, O. Etzioni, and D. Weld. Learning to understand information on the Internet: An example-based approach. *J. Intelligent Information Systems*, 1997. To appear.

[Pet96] Charles J. Petrie. Agent-based engineering, the web, and intelligence. *IEEE Expert: Intelligent Systems & their Applications*, pages 24–29, December 1996.

[Pik97] John Pike. Talkback: Shocked by search engine indexing, March 1997. See the file `anchordesk/talkback/talkback_11638.html` on `http://www4.zdnet.com/`.

[Pin94] Brian Pinkerton. Finding what people want: Experiences with the WebCrawler. In *Proceedings of the Second International World Wide Web Conference*, Chicago, IL, 1994. See the file `SDG/IT94/Proceedings/Searching/pinkerton/WebCrawler.html` on `http://www.ncsa.uiuc.edu/`.

[PMB96] Michael Pazzani, Jack Muramatsu, and Daniel Billsus. Syskill & Webert: Identifying interesting Web sites. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence and the Eighth Innovative Applications of Artificial Intelligence Conference.*, volume 1, pages 54–61, Portland, OR, USA, August 1996.

[PS96] Anandeep S. Pannu and Katia Sycara. Learning text filtering preferences. In *Symposium on Machine Learning And Information Access. AAAI 96 Symposium Series*, March 1996.

[Sal67] Gerard Salton. *Automatic Information Organization and retrieval.* McGraw-Hill Book Company, 1967.

[Sal89] Gerard Salton. *Automatic Text Processing: The Transformation, Analysis, and Retrieval of Information by Computer.* Addison Wesley, 1989.

[SE95] Erik Selberg and Oren Etzioni. Multi-Service Search and Comparison Using the MetaCrawler. In *Proc. 4th World Wide Web Conf.*, pages 195–208, Boston, MA USA, 1995. See `http://www.cs.washington.edu/research/metacrawler`.

[SLE97] J. Shakes, M. Langheinrich, and O. Etzioni. Ahoy! the home page finder. In *Proc. 6th World Wide Web Conf.*, Santa Clara, CA USA, 1997. See `http://www.cs.washington.edu/research/ahoy`.

[Sof] Sleepycat Software. The Berkeley Database. See the file `/db/index.html` on `http://mongoose.bostic.com/`.

[The] The Intelligent Transportation Systems Program. netAddress book of transportation professionals. See the file ~dhb/TRANSPORT_NAB/ on http://dragon.princeton.edu/.

[Tra97] TradeWave Corporation. Net Citizens, homepage repository, 1993–1997. See http://www.einet.net/galaxy/Community/Net-Citizens.html.

[Ven96] Alladi Venkatesh. Computers and other interactive technologies for the home. *C. ACM*, 39(12):47–54, 1996.

[WE94] Dan Weld and Oren Etzioni. The first law of robotics (a call to arms). In *Proc. 12th Nat. Conf. on AI*, pages 1042–1047, 1994.

[web95] *The Merriam-Webster Dictionary*. Merriam-Webster, Inc., 1995.

[Wel97] Daniel S. Weld. Software agents that reason about information content & quality. In *Proceedings of the first international conference on Autonomous Agents*, pages 472–473, 1997.

[Who97] WhoWhere? Inc. WhoWhere? E-mail Addresses homepage, 1995–1997. See http://www.whowhere.com/.

[Win96] Nick Wingfield. Engine sells results, draws fire, June 1996. See http://www.news.com/News/Item/0,4,1635,00.html.

[Yah95] Yahoo! Inc. Yahoo! homepage, 1995. http://www.yahoo.com.

[Yan97] Jerry Yang. Yahoo! finding needles in the internet's haystack. In *Architects of the Web*, chapter 6. John Wiley & Sons, 1997. Online version at http://www.architectsoftheweb.com/jw/yang1.html.

[ZEMK97] Oren Zamir, Oren Etzioni, Omid Madani, and Richard M. Karp. Fast and intuitive clusturing of web documents. In *Submitted to KDD-97*, 1997.

Appendix

A Glossary

The following is a subset of the *ILC Glossary of Internet Terms* [Int97], featuring many of the technical terms used in this document.

Browser A Client program (software) that is used to look at various kinds of Internet resources.

See Also: Client, URL, WWW, Netscape, Mosaic, Home Page (or Homepage)

CGI (Common Gateway Interface) – A set of rules that describe how a Web Server communicates with another piece of software on the same machine, and how the other piece of software (the *CGI program*) talks to the web server. Any piece of software can be a CGI program if it handles input and output according to the CGI standard.

Usually a CGI program is a small program that takes data from a web server and does something with it, like putting the content of a form into an e-mail message, or turning the data into a database query.

You can often see that a CGI program is being used by seeing *cgi-bin* in a URL, but not always.

See Also: cgi-bin, Web

cgi-bin The most common name of a directory on a web server in which CGI programs are stored. The *bin* part of *cgi-bin* is a shorthand version of *binary*, because once upon a time, most programs were refered to as *binaries*. In real life, most programs found in cgi-bin directories are text files – scripts that are executed by binaries located elsewhere on the same machine.

See Also: CGI

Client A software program that is used to contact and obtain data from a Server software program on another computer, often across a great distance. Each Client program is designed to work with one or more specific kinds of Server programs, and each Server requires a specific kind of Client. A Web Browser is a specific kind of Client.

See Also: Browser, Server

Cyberspace Term originated by author William Gibson in his novel Neuromancer [Gib94]. The word Cyberspace is currently used to describe the whole range of information resources available through computer networks.

Domain Name The unique name that identifies an Internet site. Domain Names always have 2 or more parts, separated by dots. The part on the left is the most specific, and the part on the right is the most general. A given machine may have more than one Domain Name but a given Domain Name points to only one machine. For example, the domain names:

```
matisse.net
mail.matisse.net
workshop.matisse.net
```

can all refer to the same machine, but each domain name can refer to no more than one machine.

Usually, all of the machines on a given Network will have the same thing as the right-hand portion of their Domain Names (matisse.net in the examples above). It is also possible for a Domain Name to exist but not be connected to an actual machine. This is often done so that a group or business can have an Internet e-mail address without having to establish a real Internet site. In these cases, some real Internet machine must handle the mail on behalf of the listed Domain Name.

See Also: IP Number

Finger An Internet software tool for locating people on other Internet sites. Finger is also sometimes used to give access to non-personal information, but the most common use is to see if a person has an account at a particular Internet site. Many sites do not allow incoming Finger requests, but many do.

FTP (File Transfer Protocol) – A very common method of moving files between two Internet sites. FTP is a special way to login to another Internet site for the purposes of retrieving and/or sending files. There are many Internet sites that have established publicly accessible repositories of material that can be obtained using FTP, by logging in using the account name anonymous, thus these sites are called anonymous ftp servers.

Gopher A widely successful method of making menus of material available over the Internet. Gopher is a Client and Server style program, which requires that the user have a Gopher Client program. Although Gopher spread rapidly across the globe in only a couple of years, it has been largely supplanted by Hypertext, also known as WWW (World Wide Web). There are still thousands of Gopher Servers on the Internet and we can expect they will remain for a while.

See Also: Client, Server, WWW, Hypertext

hit As used in reference to the World Wide Web, *hit* means a single request from a web browser for a single item from a web server; thus in order for a web browser to display a page that contains 3 graphics, 4 *hits* would occur at the server: 1 for the HTML page, and one for each of the 3 graphics.

emphhits are often used as a very rough measure of load on a server, *e.g.*, *Our server has been getting 300,000 hits per month.* Because each *hit* can represent anything from a request for a tiny document (or even a request for a missing document) all the way to a request that requires some significant extra processing (such as a complex search request), the actual load on a machine from 1 hit is almost impossible to define.

Home Page (or Homepage) Several meanings. Originally, the web page that your browser is set to use when it starts up. The more common meaning refers to the main web page for a business, organization, person or simply the main page out of a collection of web pages, e.g. *Check out so-and-so's new Home Page.*

Another sloppier use of the term refers to practically any web page as a emphhomepage, e.g. *That web site has 65 homepages and none of them are interesting.*

See Also: Browser, Web

Host Any computer on a network that is a repository for services available to other computers on the network. It is quite common to have one host machine provide several services, such as WWW and USENET.

See Also: Node, Network

HTML (HyperText Markup Language) – The coding language used to create Hypertext documents for use on the World Wide Web. HTML looks a lot like old-fashioned typesetting code, where you surround a block of text with codes that indicate how it should appear, additionally, in HTML you can specify that a block of text, or a word, is linked to another file on the Internet. HTML files are meant to be viewed using a World Wide Web Client Program, such as Netscape or Mosaic.

See Also: Client, Server, WWW

HTTP (HyperText Transport Protocol) – The protocol for moving hypertext files across the Internet. Requires a HTTP client program on one end, and an HTTP server program on the other end. HTTP is the most important protocol used in the World Wide Web (WWW).

See Also: Client, Server, WWW

Hypertext Generally, any text that contains links to other documents - words or phrases in the document that can be chosen by a reader and which cause another document to be retrieved and displayed.

Internet (Upper case I) The vast collection of inter-connected networks that all use the TCP/IP protocols and that evolved from the ARPANET of the late 60's and early 70's. The Internet now (July 1995) connects roughly 60,000 independent networks into a vast global internet.

internet (Lower case i) Any time you connect 2 or more *networks* together, you have an internet - as in inter-national or inter-state.

See Also: Internet, Network

IP Number (Internet Protocol Number) – Sometimes called a dotted quad. A unique number consisting of 4 parts separated by dots, e.g.

```
165.113.245.2
```

Every machine that is on the Internet has a unique IP number - if a machine does not have an IP number, it is not really on the Internet. Most machines also have one or more Domain Names that are easier for people to remember.

See Also: Domain Name, Internet, TCP/IP

ISP (Internet Service Provider) – An institution that provides access to the Internet in some form, usually for money.

See Also: Internet

Login Noun or a verb. Noun: The account name used to gain access to a computer system. Not a secret (contrast with Password). Verb: The act of entering into a computer system, e.g. Login to the WELL and then go to the GBN conference.

See Also: Password

Mosaic The first WWW browser that was available for the Macintosh, Windows, and UNIX all with the same interface. Mosaic really started the popularity of the Web. The source-code to Mosaic has been licensed by several companies and there are several other pieces of software as good or better than Mosaic, most notably, Netscape.

See Also: Browser, Client, WWW

Netscape A WWW Browser and the name of a company. The Netscape (tm) browser was originally based on the Mosaic program developed at the National Center for Supercomputing Applications (NCSA).

Netscape has grown in features rapidly and is widely recognized as the best and most popular web browser. Netscape corporation also produces web server software.

Netscape provided major improvements in speed and interface over other browsers, and has also engendered debate by creating new elements for the HTML language used by Web pages – but the Netscape extensions to HTML are not universally supported.

The main author of Netscape, Mark Andreessen, was hired away from the NCSA by Jim Clark, and they founded a company called Mosaic Communications and soon changed the name to Netscape Communications Corporation.

See Also: Browser, Mosaic, Server, WWW

Network Any time you connect 2 or more computers together so that they can share resources, you have a computer network. Connect 2 or more networks together and you have an internet.

See Also: internet, Internet, Intranet

NNTP (Network News Transport Protocol) – The protocol used by client and server software to carry USENET postings back and forth over a TCP/IP network. If you are using any of the more common software such as Netscape, Nuntius, Internet Explorer, etc. to participate in newsgroups then you are benefiting from an NNTP connection.

See Also: Newsgroup, TCP/IP, USENET

Node Any single computer connected to a network.

See Also: Network, Internet, internet

Password A code used to gain access to a locked system. Good passwords contain letters and non-letters and are not simple combinations such as virtue7. A good password might be:

```
Hot$1-6$
```

See Also: Login

Port 3 meanings. First and most generally, a place where information goes into or out of a computer, or both. E.g. the serial port on a personal computer is where a modem would be connected.

On the Internet port often refers to a number that is part of a URL, appearing after a colon (:) right after the domain name. Every service on an Internet server listens on a particular port number on that server. Most services have standard port numbers, e.g. Web servers normally listen on port 80. Services can also listen on non-standard ports, in which case the port number must be specified in a URL when accessing the server, so you might see a URL of the form:

```
gopher://peg.cwis.uci.edu:7000/
```

shows a gopher server running on a non-standard port (the standard gopher port is 70). Finally, port also refers to translating a piece of software to bring it from one type of computer system to another, e.g. to translate a Windows program so that is will run on a Macintosh.

See Also: Domain Name, Server, URL

Server A computer, or a software package, that provides a specific kind of service to client software running on other computers. The term can refer to a particular piece of software, such as a WWW server, or to the machine on which the software is running, *e.g.*, Our mail server is down today, that's why e-mail isn't getting out. A single server machine could have several different server software packages running on it, thus providing many different servers to clients on the network.

See Also: Client, Network

TCP/IP (Transmission Control Protocol/Internet Protocol) – This is the suite of protocols that defines the Internet. Originally designed for the UNIX operating system, TCP/IP software is now available for every major kind of computer operating system. To be truly on the Internet, your computer must have TCP/IP software.

See Also: IP Number, Internet, UNIX

UNIX A computer operating system (the basic software running on a computer, underneath things like word processors and spreadsheets). UNIX is designed to be used by many people at the same time (it is multi-user) and has TCP/IP built-in. It is the most common operating system for servers on the Internet.

URL (Uniform Resource Locator) – The standard way to give the address of any resource on the Internet that is part of the World Wide Web (WWW). A URL looks like this:

```
http://www.matisse.net/seminars.html
telnet://well.sf.ca.us
news:new.newusers.questions
```

The most common way to use a URL is to enter into a WWW browser program, such as Netscape, or Lynx.

See Also: Browser, WWW

Web See: WWW

WWW (World Wide Web) – Two meanings - First, loosely used: the whole constellation of resources that can be accessed using Gopher, FTP, HTTP, telnet, USENET, WAIS and some other tools. Second, the universe of hypertext

servers (HTTP servers) which are the servers that allow text, graphics, sound files, etc. to be mixed together.

See Also: Browser, FTP, Gopher, HTTP, Telnet, URL, WAIS

B The Ahoy! System: Maintenance & Troubleshooting

B.1 Implementation Details

Ahoy! and other DRS-Systems discussed in this work have all been implemented in the Perl5 scripting language. Perl was invented by Larry Wall in 1987 as a general tool for parsing output of UNIX commands and generating simple reports. With the new object oriented features in version 5.004, Perl moves from a simple scripting language to become a serious programming language, which is especially suited for rapid prototyping. Perl source code is read at runtime, compiled into byte code and then interpreted. However, the Perl community already works on a project to provide a compiler to machine code which would significantly cut down execution time for an application as large as Ahoy! (almost 15,000 lines of code, plus about 10,000 lines of standard libraries, which are read and compiled *everytime* a search is started).

- **Language:** Perl5, patchlevel 4
- **Lines:** 14,864
- **Architecture:** DEC OSF[1]
- **Server:** DEC Alpha 200Mhz, 256MB main memory, Andrew File System (AFS)

B.1.1 Main scripts: searching, guessing and following

Ahoy! uses three main scripts to handle requests: `nph-searching.cgi`, the "searching" script; `nph-guessing.cgi`, the "guessing" script; and `nph-follow.cgi`, the "following" script.

The *following* script is invoked everytime a user follows a link to a homepage presented on Ahoy!s results page. It will record which URL the user followed, and then send the user to the desired page by return a HTTP redirect command. This data could be used for additional learning (currently not exploited by Ahoy!), but is of only statistical value right now.

[1] Perl is architecture independent and runs on most UNIX and many non-UNIX (*e.g.*, DOS, Windows, Mac OS) systems.

The *searching* script is actually two scripts: a *proxy* script, `nph-proxy.cgi` first checks the number of active searches on the machine and decides if enough resources are available to start another search. If not it will ask the user to come back later, otherwise it calls the "real" search script, `nph-ahoy.cgi`, with the user defined query values. The real search script then performs the search as described in the preceeding chapters, up to the point where it either ends with a results page, or prompting the user to select the servers to search further (in case the search failed and it found a number of hypotheses to at the specified institution).

The *guessing* script, `nph-guessing.cgi` forms the continuation of such a user prompt, picking up where the initial search script stopped, by instantiating the relevant hypotheses at the selected institution with the search parameters and directly accessing the generated *instantiated hypotheses*. It either returns a result page similar to the one the initial search script would have shown in case of a successful query, or with another prompting for selecting yet another possible location. Thus, the user can initiate a number of these directed searches in a row, each time invoking the guessing script again. The guessing script should probably also have a proxy frontend checking for the availability of resources, but the low usage number (1 continued search for every 10 searches) made this a low priority during development.[2]

B.1.2 Managing multiple server

The Ahoy! system is designed to run on a number of servers concurrently. Although every process runs only on a single server, each invocation of a script can be on different server. This is because all servers in the Ahoy! cluster are accessible through a single DNS alias (`ahoy.cs`), which randomly sends a user to one of the machines in the cluster.

Because of this, all server need a common directory structure, so that continuation script invocations on one machine can use data from searches that ran on other machine. Also, all maintenance scripts have to be able to run on multiple machines. In order to avoid multiple invocations of system wide scripts on different machines, one machine in the Ahoy! cluster is the designated *main* machine. This main machine is the only one that runs system wide scripts that collate Web server statistics or archive session directories.

B.2 File List

The following is a list of files and directories that constitute the Ahoy! system. See also the files `MANIFEST` and `ROADMAP` in the Ahoy! root directory (`<root>`).[3]

<root>/
bin/ Binaries used by the Ahoy! system

[2] Although the longer runtime of `nph-guessing.cgi` might consume more resources.
[3] The Ahoy! root directory is currently `/afs/cs/home/ahoy/`.

doc/	On-line Documentation
scripts/	Ahoy! maintenance shell scripts
server/	Web server related files
system/	Ahoy! system related files (shared)
MACHINES	Lists machines currently in the Ahoy! pool
MANIFEST	File list
README	Top Level information about on-line documentation
ROADMAP	Directory Structure

<root>/bin/

afssh	Shell replacement to use with AFS
gzarchive	Tool for archiving entire directories
gzlistarchive	Lists archived directories
gzunarchive	Entpacks archived directories (shared)

<root>/doc/

HOWTO/	Howto Guides
CODING	How to make changes to the source code
DEBUG	How to debug Ahoy!
HTSTATS	Information about the Web Statistics package
METACRAWLER	Information about Ahoy!'s MetaCrawler interface
MODULES	List of modules & brief descriptions
TROUBLESHOOTING	How to solve problems with the Web Service
mc.command	sample commandline to the MC as used by Ahoy!
mc.output0797	sample output of the MC as expected by Ahoy!

<root>/doc/HOWTO/

HOWTO-Count_hypotheses	How many hypotheses does Ahoy! have?
HOWTO-Create_Institutions_DB	Adding a new institution or nickname
HOWTO-Deny_access	In case a particular site bombards Ahoy!
HOWTO-Install_edit_version	Adding development versions
HOWTO-New_Root	Installing Ahoy! in a new location
HOWTO-New_Server	Adding a new machine
HOWTO-Setup-Environment	UNIX packages needed to run Ahoy!
HOWTO-Troubleshooting_System	If the program dies or performs badly

<root>/scripts/

server/	Web server maintenance
system/	System maintenance

<root>/scripts/server/

_crontab.server	Crontab file for every Ahoy! machine
_crontab.system	Crontab file for main Ahoy! machine
change_root.pl	Changes default directories in scripts
check-server	Is a Web server running?
create_crontab.pl	Installs crontab files
create_new_server_root.pl	Adds directory for new Ahoy! machine
restart-server	Restarts Web server

`rotate-logs`	Rotates Web server logs
`scavenge-old-logs`	Collates Web server logs accross multiple machines
`start-server`	Starts Web server (if necessary)
`stop-server`	Stops currently running Web server

<root>/scripts/system/

`_logtdiff.pl`	Subscript for `_merge_logs.pl`
`_merge_access_logs`	Subscript for `_merge_logs`
`_merge_logs`	Used by `merge_all_server_logs` to merge logs
`_merge_logs.pl`	Subscript for `_merge_access_logs`
`handle_sess_dirs`	Archives old session directories
`merge_all_server_logs`	Used by `scavenge-old-logs`
`small_update_htstats`	Hourly Web statistics update
`update_htstats`	Nightly Web statistics update

<root>/server/

archives/	Holds archived server logs
root/	Web server roots for each machine

<root>/root/

`_access.conf.global`	Global access configuration file (shared)
`_httpd.conf.global`	Generic httpd configuration file (copied)
`_mime.types.global`	Global mime types (shared)
`_srm.conf.global`	Global mapping configuration (shared)

<root>/system/

CGI/	Ahoy! program code (CGI scripts)
HTML/	Ahoy! HTML documents
resources/	Hypotheses, institutional DB, etc.
tools/	Miscellaneous helper scripts

<root>/CGI/

Ahoy/	Ahoy! related sub-modules. See section B.3.7, pp. 138
LWP/	Additional Web library modules
WWW/	General WWW sub-modules
`nph-ahoy.cgi`	Main Ahoy! search script
`nph-down.cgi`	Shows "Ahoy! is down" message
`nph-follow.cgi`	Protocolls every answer URL the user follows
`nph-guessing.cgi`	Continues direct Ahoy! search
`nph-proxy.cgi`	Proxy script, blocks access if service overloaded
`nph-searching.cgi`	Links either to `nph-ahoy.cgi` or `nph-down.cgi`
`nph-status.cgi`	Shows status information on current searches

<root>/system/HTML/

doc/	FAQ and additional information
gif/	Graphics
htstats/	Web server statistics
sessions/	Individual session directories

`index.html`	Points either to `table-search-form` or `out-of-service`
`non-table-search-form.html`	Simple search from (no tables)
`out-of-service.html`	Out of service message
`robots.txt`	Blocks robots from indexing parts of Ahoy!
`table-search-form.html`	Standard search form

<root>/system/resources/

`glimpse_index_command.txt`	How to build inst. DB
`glimpse_index_update_command.txt`	How to update inst. DB

<root>/system/tools/

`add_title`	Augments acquired hypotheses with server names
`count_hypos`	Counts total number of acquired hypotheses
`deletefromDB_file.pl`	Deletes entry from Berkeley DB file
`traverse.pl`	Generic module for traversing a directory
`viewDB_File.pl`	Lists content of Berkeley DB file

B.3 On-line documentation

The Ahoy! systems includes a number of on-line documentation files (see file list above) that are reprinted here.[4] Please refere to the filenames given underneath each section title to locate them on-line.

B.3.1 General Information

Filename: `<root>/README`

Ahoy! The Homepage Finder

1996, 1997 J.Shakes, M.Langheinrich, O.Etzioni

This directory contains the Sources for the Ahoy! Web Service. See the `ROADMAP` file in this directory for a description of the various subdirectories in this distribution.

The `MACHINES` file in this directory lists the machines currently (7/97) in the Ahoy! system, and their tasks (*i.e.*, what crontab jobs are running on them)

See the file `TROUBLESHOOTING` in the *doc/* subdir if you need quick help with problems. See the `DEBUG` guide in the *doc/* directory, as well as corresponding `HOWTO` guides in the *doc/HOWTO/* directory for more information about specific problems with Ahoy!

See the `CODING` guide in the *doc/* directory for more information on how to edit the Ahoy! sources.

If you have problems, please don't hesitate sending me mail.

Marc Langheinrich

[4] The versions printed here have been marked up for easier reading – the original version are text only.

marclang@cs.washington.edu
marc@ub.uni-bielefeld.de

B.3.2 Ahoy! Directory Guide

Filename: <root>/ROADMAP

Ahoy! Roadmap

Main Overview

The following is a break-up of the major directories of the Ahoy! System: (i.e. the sub-directories in this current directory)

```
1. CVS/          #  Version Control Information. See doc/CODING
2. bin/          #  Contains additional helper files for Ahoy!
                 #  ( or symbolic links to them )
3. development/
     edit/       #  Contains sources for active development.
     trial/      #  Contains sources for testing.
4. doc/          #  Help & Documentation
     HOWTO/      #  General Q&A's
5. scripts/
     server/     #  Scripts related to server upkeep/maintenance
                 #  (run by all machine)
     system/     #  System related scripts (run only by one machine)
6. server/
     archives/   #  Where old server logs are stored
     root/       #  Configuration- and Log-Files for each machine
7. system/
     CGI/        #  Ahoy! System Software (called by httpd server)
     HTML/       #  Document root of httpd server
     resources/  #
       dicts/            #  words for insitution module
       hypotheses/       #  where all hypotheses are stored
       institutions/     #  contains institution DB (glimpse index)
     tools/      #  Contains scripts for counting hypotheses, listing
                 #  contents of db-files, etc. (i.e. 'tools')
```

All directories are accessible via /afs/cs/home/ahoy/ahoy (<root>), although they might reside on different filesystems and appear only as symbolic links underneath that directory.

Detailed Description of items 3.,5.-7.

3. `development/`
   Here are links to the two alternative Ahoy! versions. The `trial/` version is supposed to be a pre-fielding version, for use inside the department only. The `edit/` version is the one where files are changed, bugs are tracked down, etc etc. Only if the edit version is stable, one should copy the changed files to the release version.

   See `doc/CODING` for more details.

5. `scripts/`
   On all Ahoy! machines there is a constant need of daily upkeep, in order to ensure HTTP server availability and logfile analysis. As long as Ahoy! pages are served by a separate Apache-Server, these scripts need to be installed as nightly crontab jobs on all machines that are accessible via `ahoy.cs.washington.edu:6060`.

   Ahoy! scripts come in two flavours – "system" specific, and "server" specific. Server specific scripts should be run on a machine by machine basis, and handle each individual HTTP server running on each machine. System specific file should be run by one machine only and handle system wide tasks, like merging access logs accross all machine into a single one, collecting statistics or cleaning up session directories. If two server would both be executing some of these scripts, they would definitely trip over each other and bad things would happen!

   (a) `server/`

   - check-server [-q]
     Should be run every 10 minutes or so, to ensure that the server is up and running. In case of a problem, it will call restart-ahoy or start-ahoy to re-initialize server.
     It has to be run on the machine that should be checked and will produce no output during normal operation when using the "-q' flag. This is handy for use as a crontab job. (No output means that a server is running ok. Of course, if something goes wrong, it will show an error message even when using the "-q" flag)
   - start-server [-q]
     stop-server [-q]
     restart-server [-q]
     Starts, ends, or restarts the ahoy server respectively. Has to be run on the machine where the server should be stopped, started or restarted. Will produce no output during normal operation when using the "-q' flag. This is handy for use as a crontab job. (No output means that the script went through ok, and the server is now started (or stopped or restarted). Otherwise it would, even "-q" mode, give an error msg

- rotate-logs
  Renames current access_log and error_log of current machine, and gzips it. Restarts server to force creation of new log files.
- scavenge-old-logs
  This should be run each night just after rotate-logs has been called. It will gzip the freshly rotated log files of the machine it's run on, and move them to the central archive directory (see `server/archive/` below), where the system script merge_all_server_logs (see below) will merge them into a single `access_log` file for all Ahoy! servers.
- change_root.pl
  create_new_server_root.pl
  create_crontab.pl
  _crontab.server
  _crontab.system
  These scripts should be used to create a new directory for an added Ahoy! machine. `change_root.pl` should only be run *before* any server directories are created (i.e. when starting anew in a whole new environment), in order to adjust the basic parameters in each script for the ahoy default port and directories. (see `doc/HOWTO/HOWTO-New_Root`). Then `create_new_server_root.pl` can be used to actually create a fresh server subdirectory underneath the `server/root/` directory (see `doc/HOWTO/HOWTO-New_Server`). Finally, create_crontab.pl should be called to automatically install a corresponding list of crontab jobs on this machine. _crontab.server and _crontab.system contain the default jobs installed on each machine and on the single machine that runs 'system'-wide jobs.

(b) `system/`

- update_htstats
  Calls httpd-analyze with right parameters to update the monthly Ahoy! access statistics. Should be run each night, in order to have up to date data.
- small_update_htstats
  Calls httpd-analyze for real quick access log update. This should be run every hour or half an hour or so on the main machine (the one receiving most of the hits), to give an idea on momentarily performance.
- merge_all_server_logs
  Collects all gzipped access and error logs for all machines and creates merged version of the access logs (see merge-logs). Will not merge error logs. Should be run before update_htstats. (So that the httpd_analyzer crontab job always finds some access log in the place where it expects it to be, otherwise it will report no hits for the day)

- _merge-logs (called by merge_all_server_logs)
  _merge_access_logs (called by _merge_logs)
  _merge_logs.pl (called by _merge_access_logs)
  Subscripts of merge_all_server_logs.
- handle_sess_dirs
  Handles all session directories up until yesterday. This should be called at the end of the day (like 11pm or so) each night so that the session dir of the day before gets stored away. Right now, it will simply delete all session directories that are older than yesterday, but eventually one could call a special data extraction script on each session directory to gather some statistical data here. Deleting only yesterdays logs, and this really late in the day helps to answer fan-mail/bug-reports, which usually concern problems from the day before.

6. `server/`
   This directory contains all parts of Ahoy! that relate to keeping copies of the HTTP server running and serving pages from the Ahoy! HTML tree.

   (a) `root/`
   This subdirectory contains a directory for each machine used in the ahoy server pool, in the format `$machine-$port`. Each of these contains in turn a `conf/` and a `logs/` directory, where the server configurations files and logs are stored.

   i. `logs/`
   Holds `access_log`, `error_log` and the process ID (`httpd.pid`) of the currently running server.

   ii. `conf/`
   Each Ahoy! server uses basically the same configuration by simply using symbolic links to the global configuration scripts in the `<root>/server/root` directory. However, the `httpd.conf` file for each configuration has to be adjusted to feature the correct root-directort for each machine. This is usually done automatically by the server-installation scripts described above

   (b) `archives/` (symbolic link to some large disk space!)
   Old Server logs are compressed and stored in a month/day subdirectory structure by scripts/system/merge_all_server_logs.

7. `system/`
   This directorie contains all files needed for presenting and running Ahoy! queries, i.e. the HTML-Files needed for providing the forms and help texts, the graphics on the Ahoy! pages, the CGI-scripts that are called upon pressing "Submit" on the forms, and, last not least, the libraries needed by the Perl-CGI scripts.

(a) `HTML/`
Ahoy! uses two different versions of its search page: one for table-aware browser, and one non-table version:

```
table-search-form.html
non-table-search-form.html
```

In addition, an Out-Of-Service file that can be substituted for the search form via the `index.html` symbolic link, allows closing down Ahoy! for public use (i.e. for updating). Please make sure that you also re-link the search script in `system/CGI/`, nph-search.cgi to the nph-down.cgi cript in order to prevent people using external search forms (like the "All-in-One" search page) from issuing queries.

All additional documentation (like help pages or references) are kept in the `HTML/doc/` subdirectory. All graphics are in the `HTML/gif/` subdirectory (although they aren't neccessarily gif files)

(b) `CGI/`
Four main CGI scripts provide access to the Ahoy! system. They all start with the string '`nph-`' in order disable buffering by the apache-server.

nph-status.cgi provides status information about currently running searches on this machine (Symbolic link to nph-proxy.cgi, which checks for the name of the script called in order to provide different functionality. See below).

nph-guessing.cgi continues search in case the user was asked to select the sites to search.

nph-follow.cgi redirect-script that is called whenever someone follows a link to a page found by Ahoy!

nph-search.cgi symbolic link pointing to the main entry script for searches. This should point to one of the following files:

nph-proxy.cgi Script that checks the current load of the Ahoy! system and either continues with the nph-ahoy.cgi script, or presents an 'overload' message to the user. If called using the nph-status.cgi link, it will simply give a status overview.

nph-ahoy.cgi *real* Ahoy! main script!

nph-down.cgi BlockOut script that displays down-message. Has to be edited to match time and day mentioned in the text.

nph-ahoy.cgi and nph-guessing.cgi use Perl-modules in the form `Ahoy::Modulename`, `LWP::modulename` and `WWW::modulename`. These modules are found in the `Ahoy/`, `LWP/` and `WWW/` subdirectory, respectively.

i. `Ahoy/`
This subdirectory contains all Perl modules needed for Ahoy!. See the file `MODULES` in the `doc/` directory for more information about how

the code is divided into different files. Three subdirectories contain various submodules:

`Analyzer/`	contains modules for analyzing page content.
`Buckets/`	contains modules for manipulating the Bucketing-Table content.
`Query/`	contains additional files needed by the Query module.

ii. `LWP/`
This subdirectory contains additions to the standard Perl library `libwww` which allow parallel Web access. These files are part of the Parallel User Agent package. See the file `homes/marclang/ParallelUA/` on the server `http://www.cs.washington.edu/`.

iii. `WWW/`
This subdirectory contains a single additions to the standard Perl library `libwww` allowing the use of the Berkeley DB [Sof] together with the RobotRules module.

(c) `resources/`
Contains resources needed by the Ahoy! system, such as

`dicts/`	contains extra keywords needed by the Inst-Manager module.
`institutions/`	Ahoy!'s institutinal database (see the file `HOWTO-Create_Institutions_DB` in `doc/HOWTO/`)
`hypotheses/`	Ahoy!'s general hypotheses.

(d) `tools/`
Contains additional scripts that are handy for administering Ahoy!

add_title Augments hypotheses with server titles for nicer presentation to the user (i.e. when she's asked to make a choice where to continue looking). Currently run nightly by `centauri-prime`.

count_hypos Counts the total number of (general) hypotheses in Ahoy!'s database (either globally, per zone or per domain).

viewDB_File.pl Lists the content of a Berkeley DB file.

B.3.3 The Ahoy! machine cluster

Filename: `<root>/MACHINES`

The Ahoy! machine cluster

1997 Marc Langheinrich

The following machines are currently running part of the Ahoy! system: (see also the `server/root/` subdirectory)

`draz.cs`	Main Ahoy! server. Runs system wide jobs (use the '`crontab -l`'-command to list) to merge server logs accross servers and delete old session directories.
additional servers, not yet mapped to `ahoy.cs` alias:	
`centauri-prime.cs`	Runs nightly job to augment hypotheses with server/site titles
`vorlon.cs`	-

Each of these machines runs an httpd server and 2 crontab jobs for keeping this server running: rotate_logs and check_server. Any output of the crontab jobs is sent to the (local) `ahoy` account on each of the machines (i.e. you'd see a "you got new mail" message when you log in into each of them), but a `.forward` file in the `ahoy`-user directory should forward these messages to the maintainer of the system (currently this crontab output, if any, is forwarded to `marclang@cs`).

If you add a new machine, make sure that it runs the 'server' crontab files. Use the create_new_server_root.pl script in `<root>/scripts/server/` to create a server root directory, then use the create_crontab.pl script in the same directory to initialize its crontab jobs.

If you remove a server, be sure to take a look at its crontabs (using '`crontab -l`'), and move any system wide tasks to another machine (*i.e.*, if you decide to remove `draz`, you will need to add the system wide crontabs, say, to `centauri-prime`). You can use the create_crontab.pl script for doing so, by specifying the `-s` option.

Note: The crontab job currently running on `centauri-prime` that augments the hypotheses with corresponding server titles is not part of the default system-crontabs package and has to be installed manually!

B.3.4 How to make changes to the source code

Filename: `<root>/doc/CODING`

Ahoy! Coding Guide

1997 Marc Langheinrich

The Ahoy! source code is maintained using the CVS source control system. Although this system, unlike RCS, does not impose access restrictions (*i.e.*, you can edit any source, any time), it is much better to take some steps before and after changing some source code.

Tip: Use '`man cvs`' for an introduction to cvs, or view the 'info' entries in your emacs using '`C-h i`'. Using '`cvs --help-commands`' on commandline lists all available commands, while '`cvs -H [command]`' gives help for the specific command.

General Directive: If you want to make changes to Ahoy!, you should first edit a development copy in the `<root>/development/` directory!!! Only after you have

extensively tested the changes, you should update the sources in the "real" Ahoy! distribution (*i.e.*, the sources in `<root>/system/`)

The only reason to work on the code of the 'real' system is to fix some serious problems that prevent the service from running at all. If Ahoy! is running ok, but some responses are suboptimal, you should first try out your changes on a non-public version!!

Before you change a source:

1. **Check to see if you are editing the latest version**

   ```
   draz% cvs -n update MyModule.pm
   M MyModule.pm
   ```

   This compares the local copy of MyModule with the one stored in the CVS repository (currently located in `/cvsroot/`). The `-n` option prevents cvs from making any changes to your source file. The first letter indicates the difference detected between your local copy, and the one in the repository:

   `M file` The file is modified in your working directory. This indicates that you have changed the original, but haven't yet checked in the changes into the repository. You probably want to update the repository copy to the current version of the file, before you add any more changes. See 3. below for how to commit changes to the repository. Alternatively, the file was also changed in the repository, but the system is able to successfully merge the two versions together.

   `C file` A conflict was detected. This means that someone (maybe yourself?) already checked in a new version of the file *after* you checked out this local copy, and now the system is unable to merge the two versions (the changes you made to your local copy) together.

   If you remove the '`-n`' option, cvs will propagate any changes in the repository to your local copy. In case a Conflict was detected ("C"), it will mark the conflicting areas with "`<<<<<`" and "`>>>>>`" markup, so that you will have to manually check and resolve any problems.

   If the version in the repository hasn't changed since you checked out your local copy of the file, your file will not be changed. However, if you made any changes unrelated to the one you are about to add, you will probably want to commit those changes first to the repository, as outlined in 3.

   If the 'update' command doesn't list anything, your sourcefile and repository entry are identical.

2. **Edit your file as usual.** Make sure everything works by running a couple of sample searches.

3. **Propagate changes to the repository**

   ```
   draz% cvs commit MyModule.pm
   ```

   This will prompt you for a log message, and check in the updated version of your Source.

4. **Update the copy of your source in the 'real' Ahoy!** If you made your chances in a development directory (as you should), you should sooner or later update the 'real' version of Ahoy! to use the new, updated, source:

   ```
   draz% cd ~/ahoy/system/CGI/Ahoy/
   draz% cd cvs diff MyModule.pm
   ```

   will list the changes you made, and

   ```
   draz% cd update MyModule.pm
   U MyModule.pm
   ```

   will update the copy in the Ahoy! directory with the changes you made. (You might want to use '`cvs -n update MyModule.pm`' first to see what would happen. Having to clean up a conflict in the 'real' Ahoy! source is probably not a good idea, since the `<<<<` markup will def. prevent Ahoy! from compiling.)

B.3.5 Web Statistics using htstats

Filename: `<root>/doc/HTSTATS`

Troubleshooting Ahoy!'s Web Statistics

1997 Marc Langheinrich

This documents explains how to troubleshoot problems with the web statistics package, *e.g.*, if the statistics Web pages indicates zero hits for a number of days in a row, although the system was up most of the day. Most probably this is due to problems with the nightly crontab jobs.

1. Login to an Ahoy! machine (preferably `draz`)

2. `cd` into the server log archives

3. `cd` into the apropriate subdirectory, for example

   ```
   centauri-prime-(Alpha)% cd ahoy/server/archives/07-97/07-23-97/
   ```

4. List content

   ```
   centauri-prime-(Alpha)% ls
   access_log.merged.07-23-97.gz
   centauri-prime-6060-error_log.07-23-97.gz
   draz-6060-error_log.07-23-97.gz
   vorlon-6060-error_log.07-23-97.gz
   ```

   If you can find the "access_log.merged" file, continue with 10.

5. If no access_log.merged file exists, check the status of the crontab jobs.

   Check you mail on the draz machine (or whichever machine runs the htstats job) to see if mail has been sent due to a problem during execution. Sometimes an AFS server is temporarily down, and some of your scripts could not execute. Usually, the problem should go away the next day, when the crontab jobs are executed again. If you want, you can execute the scripts manually (if you're in a hurry).

6. Check each servers local 'log' directory.

   Login to _each_ server you want to check, and cd into their apropriate server root directory, usually something like

   ```
   centauri-prime-(Alpha)% cd ~/ahoy/server/root/draz-6060/
   ```

   Be sure to be on the correct machine for each log – draz won't be able to see the log files created on vorlon, and vice versa (well, maybe see them, but they'll be 0 bytes).

7. Check for gzipped access/error logs in the `<server-root>/log` directory.

   If you can't find those, then maybe the server hasn't been restarted last night? You can simply wait another day, then next day's crontab jobs should make things ok again. If you can't wait, you could also rotate the logs manually

   ```
   centauri-prime-(Alpha)% ~/ahoy/scripts/server/rotate_logs
   ```

   (this should be repeated for each apropriate machine)

8. Merge Ahoy! access logs.

   If you can find .gz files in the directories, it means that the system wide 'merge_logs' script has not been executed. You can do so manually by calling

   ```
   centauri-prime-(Alpha)% ~/ahoy/scripts/system/merge_all_server_logs
   ```

This should only be executed once. However, you can of course also wait 'til tomorrow, when the next automated call to 'merge...' should bring things in order.

9. Now `cd` back to the archive, to see if the files got merged ok:

```
centauri-prime-(Alpha)% cd ahoy/server/archives/07-97/07-23-97/
```

10. Look at the contents of the existing `access_log.merged` file, especially at the beginning and the end:

```
centauri-prime-(Alpha)% gzcat access_log.merged.07-23-97.gz | more
centauri-prime-(Alpha)% gzcat access_log.merged.07-23-97.gz | tail
```

Make sure no entries overlap to a previous or following day. Sometimes the statistics package gets mixed up if it finds single entries of a different day at the beginning or at the end. Just remove any such entries using a text editor (these two or three requests shouldn't make a difference).

11. Manually restart indexing process

```
centauri-prime-(Alpha)% ~/hoy/scripts/system/update_htstats
```

(or, of course, wait until tomorrow, when it should get automatically fixed)

12. Check the created statistics pages again (eventually reload cached copies). Point your browser to

```
http://ahoy.cs.washington.edu:6060/htstats/
```

B.3.6 MetaCrawler Interface

Filename: `<root>/doc/METACRAWLER`

MetaCrawler Interface Description

1997 Marc Langheinrich

This document provides a rough overview about the MetaCrawler interface used by Ahoy!

Most of the code concerning the MetaCrawler interface is in the `Ahoy::MC` module. However, the function that *decodes* the datastream sent back from the MetaCrawler binary is in the nph-ahoy.cgi script. The relevant sources are:

```
CGI/
 nph-ahoy.cgi:
   _MC_callback                        handles datastream from MC
     _MC_handle_entry                  called per entry
     _MC_handle_crawler                called per crawler

CGI/Ahoy/
 MC.pm:
   new                                 contains MC commandline
   get_chunk                           parses response
     _read_answer_type                 determines MC answer type
```

Here's how it works. nph-ahoy.cgi calls the `MC->new` method to create a new MC object. This will also initialize the MetaCrawler commandline. Then it will take this commandline, register it with the New_Multi.pm Multiplexer-Service and specify the _MC_callback function as a 'callback' function. This callback function is called each time new data arrives on the socket that is connected to the MetaCrawler output.

The _MC_callback function will use the `MC->get_chunk` method to parse the datastream into single entries, and then use the _MC_handle_entry and _MC_handle_crawler functions to file each entry (i.e. document reference) into our Bucket-Table.

If there's something broken, it's either the commandline assembled in `Ahoy::MC::new`, or the data format has changed and `Ahoy::MC->get_chunk` can't decode it anymore.

What to do

Try running nph-ahoy.cgi from the commandline, and add the `debug=1` flag. For example:

```
draz% nph-ahoy.cgi first=marc last=langheinrich debug=1 noemail=1
```

(The `noemail=1` will skip the email directory connections – that makes our output less cluttered). You might want to uncomment the various calls to the inform function in the `Ahoy::MC::get_chunk` method, so that you can better see what's going on. Here's a breakdown of what's happening:

The original MetaCrawler had two modes: Either it would request new information from the search engines (called "f"resh answer), or it would return a cached copy (called a "c"ached answer), if available.

The first case, the *fresh response*, would return special entries of each reference right as they'd come back from the search services, like this

```
<!-- entry -->
#A
http://www.cs.washington.edu/homes/marclang/bio.html
#T
Short bio - Marc Langheinrich
#C
```

```
Short bio - Marc Langheinrich. Short bio. I was born in February 1971
in Frankfurt/M, Germany. After 13 years of schooling I recieved my
Abitur from the
#V
0
#H
1
#S
AltaVista
```

Once all search services returned, it would then return the standard HTML form usually shown in the browsers. This information, only containing the already collected references in collated format, would just directly written to disk by Ahoy! However, in case of a _cached_ reponse, *only* this collated list of references would be returned, so Ahoy! would need to parse this list and create the same entries that were usually created from the entries described above.

The current version of the MetaCrawler (aka Huskysearch) does *not* use a cache anymore, so the response should always be a "f"resh response, guaranteeing that Ahoy! can obtain the references by parsing the above entry structure. However, the code still contains the full functionality for handling this dual answer mode, so that makes it kinda hard to see what's going on.

When the read_chunk function is called the first time, it will first try to determine the answer mode the MetaCrawler response is in, by calling the read_answer_type method (well, again, Huskysearch should *always* return 'f'resh answers).

After this method identified a "f"resh or a "c"ached response, it will then set the `$self->{'answer_type'}`. Then the read_chunk function reads line by line and tries to group together single entries, until it finds the final MetaCrawler HTML header, from which point on it will flag a "f"inal response, which enables Ahoy! to directly store the rest of the answer (i.e. the HTML page) directly to disk.

The file `mc.output.0797` in the `<root>/doc/` directory contains the sample output of the current MetaCrawler version (as of 7/97). You can use the `mc.command` script in `<root>/doc/` to reproduce the query that was used to create this response. If you compare the two formats, it should become clear what changed (Note: The order in which the references return will most certainly be different, so it is probably not a good idea to use 'diff' on these two files).

B.3.7 List of Modules

Filename: `<root>/doc/MODULES`

List of Ahoy! Modules

1997 Marc Langheinrich

This file describes the modules that make up the Ahoy! search service. They are located in `<root>/system/CGI/Ahoy`.

Alphabetical Listing

Analyzer/	Sub-Modules used by Analyzer.pm
Location.pm	Analyzes URL for Location match
Snippet.pm	Analyzes Snippet for name match
Title.pm	Analyzes Title for name match and page type
Analyzer.pm	Analyzes single reference and returns feature values
Buckets/	Sub-Modules used by Buckets.pm
Bucket.pm	Implements a single Bucket object
Entry.pm	Implements a single Entry in a Bucket
Table.pm	Implements the Table object that holds Buckets
Zone.pm	Implements Zones that group single Buckets in Table
Buckets.pm	Creates Table structure (no object code in here) and initializes zones
Cont_InstLookUp.pm	Institution object subclass for Continued searches
Cont_Query.pm	Query object subclass for Continued searches
Cont_Session.pm	Session object subclass for Continued searches
DB.pm	General DB object for storing/retrieving data, now only used to access hypotheses
DB_File.pm	Subclassed from original DB_File class to access cross-domain hypo-db's in Berkely DB format (currently disabled, so this module is unused)
Display.pm	Display object. Handles Output to Browser
EmailLookUp.pm	Email object. Handles connections to email services
Globals.pm	Holds global variable definitions
HP_Locator.pm	HPLocator object, responsible for initiating direct search if no page was found (alternatively, this will just print the list of available server to search, thus setting up a continued search via the nph-guessing script)
Hypothesis.pm	Contains "Hypothesis" object implementations, used by HP_Locator and URLLearner
InstLookUp.pm	Inst object, access Institutional database
Instantiator.pm	This object is responsible for extracting and generating URL's
MC.pm	Metacrawler object, handles all MC access
New_Multi.pm	Multi-Connection module, allow access to spawned processes (i.e. glimpse search, MC) as well as tcp connections (i.e. email services)
New_Query.pm	Query object subclass for new searches
New_Session.pm	Session object subclass for new sessions

`Nicks.pm`	Contains Nicknames (loaded by the Analyzer modules)
`Query/new_vocab.pm`	used by Query object for country name-to-id mapping
`Query.pm`	Generic Query object. Decodes user query and allows access to all fields. Not used directly, buty only through subclasses in two different flavours: New_Query and Cont Query_, which are used when calling nph-ahoy.cgi or nph_guessing.cgi, respectively
`Request.pm`	Request objects, subclassed from generic (libwww default) HTTP::Request object, but with extended fields.
`RobotPUA.pm`	Subclassed from LWP::RobotPUA, in order to provide custom message for Ahoy!
`Session.pm`	Generic Session Object. Creates unique session file for a given query, and allows other objects to save data in there. Not used directly, but rather in its two subclasses New_Session and Cont_Session when using nph-ahoy.cgi or nph-guessing.cgi, respectively
`Statistics.pm`	Statistic objects are part of each hypothesis object, where they keep track of there successful usage.
`Tools.pm`	General Tools (functions) for Ahoy!
`URL.pm`	Generic URL object (subclasses from LWP::URL) used by HP_Locator and friends (i.e. Hypothesis, URLLearner, etc)
`URLLearner.pm`	Manages hypotheses files by providing "ask" and "tell" methods that retrieve or update information stored in Ahoy!'s hypotheses files
`Working.pm`	Defines Root directory of this system. Used to allow 'development' versions to read/write their data to different directories than the "real" Ahoy!
`iHypo.pm`	"instantiated Hypotheses" objects created by the Instantiator object.

Grouped Listing

`Working.pm`	General Modules
`Globals.pm`	
`Tools.pm`	
`Query.pm`	Query Decoding
`Query/net_vocab.pm`	

New_Query.pm	
Cont_Query.pm	
Session.pm	Session directory management
New_Session.pm	
Cont_Session.pm	
Display.pm	Output management
New_Multi.pm	Information sources management
InstLookUp.pm	institutional DB
Cont_InstLookUp.pm	
EmailLookUp.pm	email sources
MC.pm	metacrawler
Analyzer.pm	Reference Analyzer
Analyzer/	
Location.pm	
Snippet.pm	
Title.pm	
Nicks.pm	
Buckets.pm	Reference Sorting/Classification
Buckets/	
Bucket.pm	
Entry.pm	
Table.pm	
Zone.pm	
HP_Locator.pm	Direct Searching
RobotPUA.pm	handles tcp connections
Request.pm	
URLLearner.pm	provides instantiated Hypotheses and handles feedback
DB.pm	accesses .hyp files
(DB_File.pm)	accesses _global.db files (disabled)
Instantiator.pm	maps URLs, hypotheses and instantiated hypotheses
URL.pm	
Hypothesis.pm	
Statistics.pm	
iHypo.pm	

B.3.8 Ahoy! Server Troubleshooting

Filename: <root>/doc/TROUBLESHOOTING

Ahoy! Server Troubleshooting

General Information

Server Access Statistics can be accessed via

```
http://ahoy.cs.washington.edu:6060/htstats
```

The current connections handled by each individual server can be checked via

```
  http://ahoy.cs.washington.edu:6060/server-status
= http://draz.cs.washington.edu:6060/server-status

  http://vorlon.cs.washington.edu:6060/server-status
  http://centauri-prime.cs.washington.edu:6060/server-status
```

The current searches running on a given machine can be checked using

```
  http://ahoy.cs.washington.edu:6060/cgi-bin/nph-status.cgi
= http://draz.cs.washington.edu:6060/cgi-bin/nph-status.cgi

  http://vorlon.cs.washington.edu:6060/cgi-bin/nph-status.cgi
  http://centauri-prime.cs.washington.edu:6060/cgi-bin/nph-status.cgi
```

The following section gives information for two kind of symptoms:

1. Ahoy! is down
2. Ahoy! is slow or shows other bad performance

Ahoy! is down

Server is not responding or search is not working. One of the following symptoms have been observed:

1. server does not accept connections

   Cause: Machine down, wedged

   How to check: determine machine that has problems:

   - try to connect to each machine individually
     `http://draz.cs:6060/`
     `http://vorlon.cs:6060/`
     `http://centauri-prime.cs:6060/`
   - try telnet to it.
   - go to physical location & check console

   Action: Restart machine (press reset button)

Cause: Machine ok, but no server running

How to check: telnet to it. login as user ahoy. use `ps auxwww | grep ahoy` to list currently running processes for user `ahoy`. At least 4-5 httpd processes should be running

Action: use check-server script that does that for you and even starts/restarts the server for you:

```
draz% ~/ahoy/scripts/server/check-server
```

2. service returns no data (message in dialog box displayed by netscape)

   Cause: an error in the perl script that leads to a compile time error (i.e. perl complains when trying to load the script)

   How to check: Run Ahoy! from commandline:

   ```
   draz% cd ~/ahoy/system/CGI
   draz% ./nph-search.cgi first=marc last=langheinrich
   ```

   Watch output. See `the DEBUG` guide if you can't figure out what this error msg means, or how to correct it.

   Action: Fix error reported on commandline. If Unix complains "./nph-foobar: No such file or directory", then there's something wrong with the "`#!/`" line that calls the perl interpreter. Make sure it points to a valid perl executable. If instead a line number/filename is displayed, follow this information and try to fix this problem, then try to run Ahoy! from commandline again.

3. access denied

   Cause: Server on denied list

   How to check: See `_access.conf.global` in `<root>/server/root/`

   Action: remove server from entry, or remove entry completely. See the guide `HOWTO-Deny_access` in the `doc/HOWTO/` subdir on how to remove/add access restrictions.

   Cause: Requesting directory listing where none is allowed (like trying to get `http://ahoy:6060/doc/`)

   Action: If you want to enable directory listings in the particular directory you requestes, you should change the line `Options` in the `<Directory>` entry that contains the requested directory to include the option `Indexes`. If you only want this single directory to be listable, add an extra `<Directory /afs/cs/...>` section for this directory only.

4. Server 500 error
   Output stops right after "Ahoy! searching for ..." message
   Output stops halfway during the search

Cause: Ahoy! script encountered run-time error, i.e. compiled ok, but failed to give any output once it started running, only printed header or died somewhere later during the search.

How to check: Look at error logs for specific server

```
draz% tail -f ~/ahoy/server/root/draz-6060/logs/error_log
```

and/or try to run Ahoy! from commandline

```
draz% cd ~/ahoy/system/CGI
draz% ./nph-ahoy.cgi first=marc last=langheinrich
```

and check for error messages

Action: Debug Ahoy! script from commandline. Follow information in the `HOWTO/HOWTO-Troubleshooting_System` guide.

Ahoy! performs badly

1. Ahoy! is slow

Cause: Heavy server load

How to check: Find out which specific machine is slow (*i.e.*, use `draz:6060` and `vorlon:6060` instead of `ahoy:6060`, but since currently only draz is mapped to the "ahoy" alias, usually it should be draz). Then check currently running server on the machine

```
http://<machine>.cs:6060/server-status
```

as well as the number of searches currently conducted

```
http://<machine>.cs:6060/cgi-bin/nph-status.cgi
```

and maybe even the number of hits received today:

```
http://ahoy.cs.washington.edu:6060/htstats/stats.html
```

(works only for draz) or look at the access logs directly

```
draz% more ~/ahoy/server/root/draz-6060/logs/access_log
draz% tail -f  ~/ahoy/server/root/draz-6060/logs/access_log
```

Action: If a single site is repeatedly hitting Ahoy!, you can exclude it by adding its IP/DNS address into the server `access.conf` file. See access restriction guide in `HOWTO/HOWTO-Deny_access`.

If there are too many searches, you might want to consider lowering the number of maximum searches in the nph-proxy.cgi script that checks if Ahoy! is able to accept a new search by changing the value of the `MAXIMUM` variable in nph-proxy.cgi.

Another reason for many searches piling up and klogging Ahoy! can also be filesystem related. See next section.

Cause: AFS file system problems

How to check: login to the slow machine, and check

- currently running processes

  ```
  draz% top
  ```

- current usage of AFS filecache and server availability

  ```
  draz% fs getcacheparms
  AFS using 217083 of the cache's available 270000 1K byte blocks.
  draz% fs checkservers
  ```

Action: If another memory/CPU intensive process is running, try contacting the owner and stop it. If the AFS chache is using nearly all of the available blocks, try to increase the cache size (see `AFS-README` in the `ahoy/` directory)

2. Ahoy! misses entry (i.e. it shows up in MetaCrawler output but Ahoy! fails to include it in response, or ranks sth else higher)

   Cause: Ahoy! Analyzer function gave suboptimal ranking to this entry

   How to check: Run the query again that showed this behaviour, then edit the "Location" cmdline in your browser and add the field `&dump=1` at the end. This will force Ahoy! to dump its internal Bucket table to the session directory in a file called `table.data`. To access it from your browser, simply follow the link to the "search transcript page" and then change the "Location" to point to `table.data` instead of `status.html`. You can also access it from the UNIX prompt at

   ```
   draz% cd ~/ahoy/system/HTML/sessions/
   draz% cd <date>/<sess-name>
   draz% more table.data
   ```

   (get those last two values from the location field of the search transcript page!)

   Look through this file and locate the reference that was suboptimally placed. The interesting parts (i.e. those that lead to its placement in the table) are:

   ```
   {'home'} => 111
   {'location'} => 111
   {'name'} => M
   ```

   Home is a three digit code for the "homepageness" of the reference, location an indicator for the correct institution, and name a field denoting the correct name in title or URL. See my thesis for a description, or contact me directly, at `marclang@cs`, for more information

   Action: Easy fixes could be to change the list of homepage and non-homepage words in the Title Analyzer:

```
draz% vi ~/ahoy/system/CGI/Ahoy/Analyzer/Title.pm
```

and edit the variables

```
my $homepages = "'s page|homepage|...|personal";
my $nohps = "project|presentation|...|DB&amp;LP";
my $lists = "students|faculty|staff|...|directory"; #$
```

You might want to consult the CODING guide first!!

3. Ahoy! lacks certain search service
   Ahoy! fails to find the MetaCrawler output
   The MetaCrawler found "0 references"
   See the METACRAWLER guide if you are having problems with the MetaCrawler connection.

4. Ahoy! fails to *write* any output
   Ahoy! shows "No Space" page when trying to view transcript or MC page.

   Cause: lacking AFS permissions (or other file permissions)

   Action: See AFS-GUIDE in ahoy

   Cause: disk quota exceeded

   Action: Contact system administrator

   Cause: No space left on device

   Action: Check size of hypotheses database, session directories, etc.

5. Ahoy! fails to find email addresses.
   Email addresses are decoded in the _email_callback function in nph-ahoy.cgi. The variable that is given as the first argument is an array containing the following fields (as defined in Ahoy::New_Multi)

   ```
   [0] - Socket
   [1] - Name of the Source
   [2] - commandline
   [3] - callback reference (i.e. the code you're looking at)
   [4] - timeout in seconds
   [5] - status
   [6] - pointer to array containing output lines
   [7] - last, partially received output line
   ```

   The calls to the Email services are made in Ahoy::EmailLookUp, in the generate_email_searches method. Look there to make sure the request format is correct.

6. The server statistic at http://ahoy:6060/htstats/ says there wasn't a single hit on a particular day!

   See the HTSTATS file in <root>/doc/.

B.3.9 Howto: Count hypotheses

Filename: `<root>/doc/HOWTO/HOWTO-Count_hypotheses`

Howto count the number of hypotheses in Ahoy!'s DB

1997 Marc Langheinrich

Use the `count_hypos` script in the `<root>/system/tools` subdirectory. This script will count the number of hypotheses and unique sites in the directory specified, and all of its subdirectories.

To count the number of hypotheses known in, say, kuwait, use

```
draz% count_hypos /afs/cs/home/ahoy/ahoy/system/resources/hypotheses/kw
[Traversing /afs/cs/usr/ahoy/ahoy/system/resources/hypotheses/kw]
[Traversing /afs/cs/usr/ahoy/ahoy/system/resources/hypotheses/kw/edu]
[Traversing /afs/cs/usr/ahoy/ahoy/system/resources/hypotheses/kw/edu/ku]
[Traversing /afs/cs/usr/ahoy/ahoy/system/resources/hypotheses/kw/ed]
[Traversing /afs/cs/usr/ahoy/ahoy/system/resources/hypotheses/kw/com]
[Traversing /afs/cs/usr/ahoy/ahoy/system/resources/hypotheses/kw/com/kn]
[Traversing /afs/cs/usr/ahoy/ahoy/system/resources/hypotheses/kw/gu]
~/ahoy/system/resources/hypotheses/kw contains 3 hypotheses at 2 unique server
```

To find the total number of known hypotheses, use

```
draz% count_hypos /afs/cs/home/ahoy/ahoy/system/resources/hypotheses
```

B.3.10 Howto: Maintain the Institutional DB

Filename: `<root>/doc/HOWTO/HOWTO-Create_Institutions_DB`

Howto create and maintain the institutional DB

1997 Marc Langheinrich

Add single institutions / nicknames manually

If you just want to quickly add a specific institution to the database, or want to add a 'nickname' like "UW" to an existing institution, you can do so without having to rebuild the index from scratch:

The `edu/` and `com/` subdirectory both contain files called "hand_entered":

```
draz% ls ~/ahoy/system/resources/institutions/*/hand_*
~/ahoy/system/resources/institutions/com/hand_entered
~/ahoy/system/resources/institutions/edu/hand_entered
```

Note: The `edu/`, `com/`, `mil/` and `gov/` subdirectory are just for filing purposes. Ahoy! just searches all of them (using the glimpse index), so it doesn't really matter if the `com/` directory contains educational sites or vice versa!

The format of such a "hand_entered" file is the same as for all other files in the institution DB, containing

1. the URL of the institution

2. its name

3. a list of nicknames/abbreviations

These field are separated with a `|` character, resulting in lines like this:

```
http://www.washington.edu/|University of Washington Home Page|uw
```

If you want to add another instiution, just append a correctly formatted line at the end of one of these two "hand_entered" lists. If you want to add a nickname to an existing entry in these lists, just append the word at the end (separated by spaces).

If you want to add a nickname that is only in one of the regular files (i.e. like `com/comlist.580.txt`), you should copy the corresponding entry from this (automatically!) generated file and add it as a hand_entered entry to one of the two files described above. Otherwise you would have to

1. rerun the glimpse database generation script

2. update this entry again once a new list of institutions is obtain from Yahoo!, since it would overwrite this file!

Create/Update The Institutions Glimpse Index

You should find the following directory structure in the `institutions/` dir:

```
<root>/system/resources/institutions/
     mil/
     com/
     edu/
     gov/
```

`cd` into the 'institutions' directory, then run

```
/uns/bin/glimpseindex -b -B -H . -T mil/ com/ edu/ gov
```

to create the initial index. If you want to update an existing index, run the following command instead (i.e. using the update option '`-f`')

```
/uns/bin/glimpseindex -f -b -B -H . -T mil/ com/ edu/ gov
```

Download the Yahoo! information

```
On Wed, 23 Jul 1997, Jonathan Shakes wrote:
> See
>
> /a/sekiu/sekiu1/softbots/jonathan/inst/yahoo/{mil|com|gov}/yahoo_suck.perl
>
> (which does the first stage, and)
>
> list_convert.perl  in the same directories
>
> which does the second.  I believe one needs to delete all datafiles and
> dbm files before starting, otherwise it will think it's already sucked
> down a lot of the info and not get the updated stuff.  Unless I put in a
> check on the age of the files -- it's been so long...
>
> I'd be happy to spend a couple of hours with someone who wanted
> to actually do it, and who would be assuming this role in the future.
>
> -Jonathan
```

B.3.11 Howto: Block specific sites from using Ahoy!

Filename: `<root>/doc/HOWTO/HOWTO-Deny_access`

Howto deny access to Ahoy!

1997 Marc Langheinrich

Make sure you restart the server using the "restart-server" script after you changed any access restrictions:

```
draz% ~/ahoy/scripts/server/restart-server
```

Access Control Locations

Access restriction for the (Apache) Web server are defined in two ways: Globally and Locally.

1. **Global Access control**
   Global access control is specified in the `access.conf` file located in the Web servers `conf/` directory. All Ahoy! Web servers share the same `access.conf` file by using a symbolic link from their `conf/` directory to the global `_access.conf.global` in `<root>/server/root/`:

   ```
   draz% cd ~/ahoy/server/root
   draz% vi _access.conf.global
   ```

The `access.conf` file contains multiple `<Directory> </Directory>` sections that define access parameters for a certain physical directory and all of its subdirectories, as well as some `<Location> </Location>` sections that define access to services such as the server-status. Here's an example:

```
<Directory /afs/cs/home/ahoy/ahoy/system/CGI>
AllowOverride None
Options FollowSymLinks

<Limit GET POST>
order allow,deny
deny from hawaii.uni-trier.de
allow from all
</Limit>

</Directory>
```

2. **Local Access control**
   Each directory underneath the server's Document root (and whatever else it is serving using aliases) can have a hidden "`.htaccess`" file that will override any of the global settings defined in the `access.conf` file. Here's an example:

```
draz% cat some/directory/.htaccess
AuthUserFile /afs/cs/home/ahoy/.htpasswd
AuthGroupFile /afs/cs/home/ahoy/.htgroup
AuthName ServerStatistics
AuthType Basic

<Limit GET>
require group develop
</Limit>
```

   The contents of a `.htaccess` files should be thought of being implictly encapsulated in a `<Directory>` Section with the corresponding name of the directory the file is located in. For example, the above `.htaccess` file in `/some/directory` would work as

```
<Directory /some/directory>
AuthUserFile /afs/cs/home/ahoy/.htpasswd
AuthGroupFile /afs/cs/home/ahoy/.htgroup
AuthName ServerStatistics
AuthType Basic

<Limit GET>
require group develop
</Limit>
</Directory>
```

Access restriction schemes

The part that limits access is enclosed in the `<Limit>` section. An optional parameter can specifiy the HTTP Method that this `<Limit>` section should be applied to, usually 'GET', but for CGI-scripts also 'POST'. Two basic schemes exist for limiting access to a Web server: On a per-site basis, or using passwords and groups.

1. **Limiting access on a server basis**
   The simplest form of limiting access is only allowing connections from a certain host, or generally allowing all access but denying specific machines.

   The "deny" and "allow" keyword can be used to list specific server, or even domains (like `washington.edu`) that should be restricted. Specifying "all" will include all servers, "none" will match no server.

   The "order" field makes the important distinction of which of the directives should be processed first. Here are examples:

   This denies all access, then allows it to `foo.bar.com`:

```
<Limit GET>
order deny,allow
allow foo.bar.com
deny  all
</Limit>
```

   Changing the order will alter the list of allowed servers as well:

```
order allow,deny
allow foo.bar.com
deny  all
```

   which would first all access to `foo.bar.com`, then deny it to all server, resulting in a blocked access for **ALL** server!!

   As a rule of thumb: The field (deny/allow) that contains an "all" field should always be listed `FIRST` in the 'order' command!!

   Another example: Allow access to all, but deny access to some specific sites:

```
<Limit GET POST>
order allow,deny
allow all
deny  some.site.com
deny  some.other.site.de
</Limit>
```

   Deny access to everybody, but people from within the department:

```
<Limit GET POST>
order deny,allow
allow cs.washington.edu
deny  all
</Limit>
```

2. **Limiting access on a per user/group basis**
   The Directory section will need to contain a description on where to find the password files and users/groups that should be allowed access to this directory. The AuthUserFile, AuthGroupFile specify files that contain this information. The AuthName section is used to give a meaningful description of the restricted Server area when prompting the user for a password (i.e. "enter username and password for XXXX"). The AuthType keyword has only one valid value, "Basic". See the Mosaic guide at

   ```
   http://hoohoo.ncsa.uiuc.edu/Mosaic/auth-tutorial/tutorial.html
   ```

   for more information about this type of access restriction. (note: you will need the `htpasswd` program that comes with the Web server source code. It is installed in `/uns/bin/htpasswd` on `draz`, `zhadum`, `vorlon` and `centauri-prime`)

B.3.12 Howto: install an edit version of Ahoy!

Filename: `<root>/doc/HOWTO/HOWTO-Install_edit_version`

Howto install an edit version of Ahoy!

1997 Marc Langheinrich

This HOWTO describes the prcedures for setting up an 'edit' version of the Ahoy! service.

1. Create a 'development' or similar directory off the Ahoy!-Root directory. For example:

   ```
   draz% cd <root>
   draz% mkdir development
   draz% mkdir development/edit
   ```

2. Populate the newly made directory with a copy of Ahoy!'s system directory. The easiest way (and most space consuming) would be to do

   ```
   draz% cp -R <root>/system <root>/development/edit
   ```

   Of course, this would also copy all of the current session directory, as well as the hypothesis directory. Unless you want to experiment with different hypothesis

formats or session directories, you should probably provide symlinks to those directories instead.

Also, the institutional database would be duplicated this way, so an much better way (but more time consuming!) would be to manually recreate the directory structure found underneath the `<root>/system` directory, providing symlinks for all but the `system/HTML` and `system/CGI` directory.

3. Adjust the entry in `Working.pm` and the mechanism used to translate this value into a pathname in `Globals.pm`.

   If you installed the copy into the directory `<root>/development/edit`, you should only need to specify `edit` in `Working.pm`. If you chose a different naming scheme, you will have to edit the corresponding lines in `Globals.pm` as well.

4. Extend the HTTP server to server the 'edit' version of Ahoy! as well.

   You will have to add an alias to the new HTML directory, as well as to the new scripting directory. To do so, edit the file _srm.conf.global in the `<root>/server/root/` directory. Currently, all servers share this global configuration file by means of a symlink. If you installed local copies of this file in some server `conf/` directorie, you will have to edit these copies directly. The entries to add are

   ```
   Alias  /edit         <root>/development/edit/system/HTML
   ScriptAlias  /cgi-bin/ahoy/edit/     <root>/development/edit/system/CGI/
   ```

   Of course, you can name the path used by the HTTP server anything you like (for example, `/cgi-bin/test/`), just make sure the 'real' path points to the correct edit version you just installed.

5. Restart the server to re-read the new entries

   Use restart-server located in `<root>/scripts/server/` on each machine, or wait until the next day (Ahoy! restart itself every night when rotating the log files). Then try to connect to the `/edit` directory on the Ahoy! server using your favourite browser.

   Note: It is a good idea to change the search form (`table-search-form.html`) in your development version to note this in their `<TITLE>` or so, e.g. "Ahoy! Development Version 3.0xx", so that you can easily distinguish what version of Ahoy! you are using.

   Caveat: If you follow the "Start New Search" link on an Ahoy! results page, it might take you to the 'main' Ahoy! version, the one at the root of your HTTP service. You should bookmark the search page of your development version and use this bookmark to go back to the search form instead.

B.3.13 Howto: install Ahoy! in a new directory

Filename: <root>/doc/HOWTO/HOWTO-New_Root

Howto setup a new Ahoy! directory

1997 Marc Langheinrich

1. **Adjust script defaults:**
   In order for all scripts to use a common 'root' and 'port' number, run the following script **BOTH** in the scripts/server **AND** the scripts/system/ directory:

   ```
   ./scripts/server% ./change_root.pl -r <root-dir> -p <port>
   ./scripts/system% ../server/change_root.pl -r <root-dir> -p <port>
   ```

2. **Adjust httpd config files manually (not yet automated):**
   The Ahoy! default httpd-configuration files are located in <root>/server/root/:

   ```
   -rw-rw-r--  1 ahoy   system    1697 Jun 30 07:43 _access.conf.global
   -rw-rw-r--  1 ahoy   system    5737 Jul  2 15:00 _httpd.conf.global
   -rw-rw-r--  1 ahoy   system    2405 Jun 30 07:43 _mime.types.global
   -rw-rw-r--  1 ahoy   system    6972 Jul  2 15:12 _srm.conf.global
   ```

   The following settings need to be changed in case the root directory should be altered:

   - _httpd.conf.global
     ServerRoot
     Port (this needs to be altered in case the port changes)
   - _srm.conf.global
     DocumentRoot
     Alias
     ScriptAlias
   - _access.conf.global
     <Directory ...>

   Use your favourite editor and search for the given keywords, then change each value to reflect the new Ahoy!-root directory.

3. **Adjust path information in Globals.pm file:**
   The $MAIN_DIR variable in Globals.pm (in the <root>/system/CGI/Ahoy/ directory) should be set to the <root> directory.

   Make sure the $UNS_BIN variable points to the directory containing unsupported programs like 'glimpse', usually /uns/bin.

4. Adjust path information in httpd server stats configuration file
   The HTMLDir variable in the httpd-configuration file 'ahoy.conf' in the directory ¡root¿/system/HTML/htstats/ has to point to the directory it is located in (i.e. ¡root¿/system/HTML/htstats)

5. **Create Glimpse Index (if neccessary):**
   You should find the following directory structure in the `<root>/system/` dir:

   ```
   <root>/system/resources/institutions/
        mil/
        com/
        edu/
        gov/
   ```

   `cd` into the 'institutions' directory, then run

   ```
   /uns/bin/glimpseindex -b -B -H . -T mil/ com/ edu/ gov
   ```

   to create the initial index. If you want to update an existing index, run the following command instead (i.e. using the update option '-f')

   ```
   /uns/bin/glimpseindex -f -b -B -H . -T mil/ com/ edu/ gov
   ```

6. **Check required programs and symbolic links:**
   Ahoy! relies on finding the following programs in their respective directories:

   ```
   /uns/bin/perl        (Version 5.004)
   /uns/bin/glimpse     (Version 4.0) (uses UNS_BIN in Global.pm module)
   <root>/bin/httpd     (Apache, version 1.2)
   ```

   The following symbolic links should be checked in order to make sure they exist and point to the correct programs:

   ```
   <root>/system/HTML/
        ./index.html -> ./table-search-form.html
        (during maintenance, this can also point to ./out-of-service.html)
   <root>/system/CGI/
        ./nph-search.cgi -> ./nph-proxy.cgi
        ./nph-status.cgi -> ./nph-proxy.cgi
   ```

   Ahoy! uses some non-standard perl modules for parallel Web access. These files need to be accessible from the `<root>/system/CGI` directory in a symbolic link named `LWP`:

   ```
   <root>/system/CGI/
        ./LWP            -> /afs/cs/home/ahoy/pua
   ```

Directories that get written to during an Ahoy! search should be put on a file system without diskquota restrictions. Especially the daily sessions can easily grow up to hundreds of megabytes, which would exceed standard quotas of most accounts. In addition, one might consider putting these on a local disk for faster access.

```
<root>/server/root ->
      /afs/cs/project/metaserver/ahoy/server/root/

<root>/system/HTML/sessions ->
      /afs/cs/project/metaserver/ahoy/system/HTML/sessions

<root>/system/HTML/resources/hypotheses ->
      /afs/cs/project/metaserver/ahoy/system/resources/hypotheses
```

7. **Adjust Working directory path:**
   Ahoy! can use multiple configurations, in order to support running different versions simultaneously. The file `Working.pm` contains extra path information for development systems, which are located in a special 'development directory'.

   For the main Ahoy! system, `Working.pm` should contain an empty string in the `WORKING` variable.

   See also the module `Global.pm` in `<root>/system/CGI/Ahoy/` for the use of this variable in the main program.

   Note: Currently, only the main version is set up. In order to create a 'trial' or 'edit' version (handy for experimenting with the source code without disturbing the main service), see the file `HOWTO-Install_edit_version`.

8. **Create a new server-root directory:**
   See `HOWTO-Create_new_Server_Root`

9. **Test ahoy search script:**
   Change directory into the ahoy-CGI directory:

   ```
   cd <root>/system/CGI
   ```

   and execute nph-ahoy.cgi on the command-line:

   ```
   ./nph-ahoy.cgi first=oren last=etzioni inst=washington
   ```

   You should see the verbose HTTP/HTML output send to a Netscape browser during a typical search (i.e. including intermediate status reports). Please refere to the file `HOWTO_Troubleshooting_System` in case this fails.

B.3.14 Howto: Add a new machine to the Ahoy! cluster

Filename: <root>/doc/HOWTO/HOWTO-New_Server

Howto add a new machine to the Ahoy! cluster

1997 Marc Langheinrich

1. **Adjust port number defaults:**
   Make sure that the scripts use the correct port number you want to use. If you want to create a large number of servers running on different machines, but having the same port number, it is probably a good idea to change the defaults in the script:

   (a) change the default port number in the server and system scripts:

   ```
   ./scripts/server% ./change_root.pl -r <root-dir> -p <port>
   ./scripts/system% ../server/change_root.pl -r <root-dir> -p <port>
   ```

   PS: Unfortunately, the current version of change_root.pl requires you to at least specify the root_dir. Even if you just want to change the port number only, you will have to specify the root_dir again, since it is a required parameter.

   (b) change the default port number in the httpd.conf file. The Ahoy! default httpd-configuration files are located in <root>/server/root/:

   ```
   -rw-rw-r--  1 ahoy   system    5737 Jul  2 15:00 _httpd.conf.global
   ```

   Change the port the httpd server is running on by changing the Port variable in the _httpd.conf.global file. Use your favourite editor and search for the Port keyword, then change the value to reflect the correct port number.

   Since most of the server- and system-scripts are not yet able to accept a port argument, you might have to create a second set of scripts in case you want to have Ahoy! running on two different port numbers.

   You can use the change_root.pl script to create one set, copy it to a safe location, and the create another set with a different port number.

   After you created a new server directory using the create_new_server_root script (as described below), you will then have to edit the locally created copy of httpd.conf in each server directory to reflect the correct port number for that directory/server.

2. **Run the sffamily create_new_server_root script:**
   Simply run the create_new_server_root script with the correct machine name supplied via the -m option. If the directory already exists, and you want to overwrite it, use the -f option to force overwriting the current information. Otherwise the script won't overwrite your existing configuration:

```
vorlon-(Alpha)% ./create_new_server_root.pl -m draz -f
Creating new Server Root underneath '/afs/cs/home/ahoy/ahoy/server/root':
        Machine: draz
        Port:    6060
Directory for machine/port 'draz-6060' exists.
Removing... (you specified -f, right?)
Success! Here's what we created
/afs/cs/home/ahoy/ahoy/server/root% ls -l draz-6060/
total 2
drwxrwxr-x   2 ahoy     system       512 Jul  3 23:38 conf
drwxrwxr-x   2 ahoy     system       512 Jul  3 23:38 logs

Don't forget to setup the crontab file on draz using create_crontab.pl!
```

3. **Test the scripts on the newly added machine:**
   Log into the machine for which you just created a server_root, and try to execute the start_server script:

   ```
   centauri-prime-(Alpha)% cd <root>/scripts/server/
   centauri-prime-(Alpha)% ./start-server
   Server started. Checking log files...
   ok
   ```

   If something goes wrong, please check manually to make sure you created the correct directory and are using the correct port numbers in the start-server script (sorry, you can't use a `-p` flag or so to specify a non-default port here :|).

   Now try to stop the server:

   ```
   centauri-prime-(Alpha)% ./stop-server
   All servers halted
   ```

   Restart it (apparently, no server is running, so the script should complain about that)

   ```
   centauri-prime-(Alpha)% ./restart-server
   No server to restart. Start new one.
   Server started. Checking log files...
   ok
   ```

   Then try the check-server script: `centauri-prime-(Alpha)Server running ok: 11993` (or whatever pid your server has)

4. **Install the crontab on the added machine:**
   First, check the parameters in `<root>/server/create_crontab.pl` to see if they fit the crontab executable used on your system:

```
cron_read => "crontab -l",
cron_write=> "crontab ",
skip => 0,  # some crontab append headers (number of lines)
```

Make sure that the first command lists your existing crontab file, and that the second command takes input from STDIN and creates a new crontab file for you. (On some systems this might be '`crontab -`') Finally, some crontab versions add a certain number of 'header' lines, such as the day of the change, to each crontab file (eg. on Linux). You can specify 'skip' to tell the script to skip the first n lines of the current crontab file when adding the Ahoy! jobs.

Then take a look at the crontab file that would be created by running

```
centauri-prime-(Alpha)% ./create_crontab.pl
Reading existing crontab file... (skipping first 3 lines)
crontab: can't open your crontab file.
# <ahoy type=server>
#
...
#
# </ahoy>

Please use the -c switch to write the new crontab file
```

Check the output to see if the correct paths have been used, and if existing crontab jobs have been copied. Once you are certain that eveything is ok, use the `-c` option to commit the new crontab file:

```
centauri-prime-(Alpha)% ./create_crontab.pl -c
```

You can test to see if the crontab jobs are running corrctly for example by stopping the currently running server (using stop-server), and checking after 10 minutes to see if check-server automatically started a new one.

B.3.15 Howto: Install required third-party programs for Ahoy!

Filename: `<root>/doc/HOWTO/HOWTO-Setup-Environment`

Howto setup the Ahoy! environment

1997 Marc Langheinrich

The following is a list of programs neccessary to run Ahoy! For each source, a URL is given where the source code can be obtained.

```
Perl
  Version:        5.004 or higher
  FTP:            ftp://ftp.spu.edu/pub/CPAN/src/latest.tar.gz
  Alternative:    See  "http://www.perl.org/CPAN/SITES.html"
  Web Info:       http://www.perl.com/perl
```

Ahoy! uses the freely available http-analyze package, which in turn needs Thomas Boutells GD library for dynamic GIF creation.

```
http-analyze
  Version:        1.9d or higher
  FTP:            ftp://ftp.rent-a-guru-de/pub/http-analyze1.9d.tar.gz
  Web Info:       http://www.netstore.de/Supply/http-analyze/

gd
  Version:        1.2 or higher
  FTP:            http://www.boutell.com/gd/gd1.2.tar.Z
  Web Info:       http://www.boutell.com/gd/
```

B.3.16 Howto: fix Ahoy! bugs

Filename: `<root>/doc/HOWTO/HOWTO-Troubleshooting_System`

Howto troubleshoot the Ahoy! system

1997 Marc Langheinrich

This HOWTO describes how to overcome problems with the Ahoy! main script, nph-ahoy.cgi. It does NOT describe problems with the Ahoy! HTTP server. See the `TROUBLESHOOTING` guide in `<root>/doc` for problems related to the apache httpd process. Also, for cases where the command line interface seems to work fine, but using the Web based search form mysteriously fails, see the above Server troubleshooting guide instead!

The following is an unordered list of problems you can experience when running the Ahoy! search script on the command line. We assume you `cd`'ed into the `<root>/system/CGI` directory, and issued something like

```
vorlon-(Alpha)% ./nph-ahoy.cgi first=dan last=weld inst=washington
```

Here's what could possibly go wrong:

1. `./nph-ahoy.cgi: Command not found.`
   You specified a wrong executable in the first line of the nph-ahoy script, or an executable perl interpreter could not be found at the directory specified.

   The first line of all "`nph-`" scripts should look something like this:

   ```
   #!/uns/bin/perl
   # $Id: nph-ahoy.cgi,v 1.6 1997/05/21 17:21:34 marclang Exp $
   ```

To make sure you are using the right path, try issuing

```
vorlon-(Alpha)% which perl
/uns/bin/perl
```

2. `Can't locate 5.0030000000000001 in @INC ...`
   Your executable has the wrong version number. Make sure you are using perl version 5.003 or higher. You can use the following command line parameter to find out the version of the perl executable you are using:

   ```
   vorlon-(Alpha)% perl -v
   This is perl, version 4.0

   $RCSfile: perl.c,v $$Revision: 1.1 $$Date: 1993/05/14 18:36:10 $
   Patch level: 36

   Copyright (c) 1989, 1990, 1991, Larry Wall

   Perl may be copied only under the terms of either the Artistic License or the
   GNU General Public License, which may be found in the Perl 4.0 source kit.
   ```

3. Perl complains with an `error in line xxx`
   If it just complains about a missing ';' or other syntax error, this one should be easy enough to fix. Just go to the line number in the file reported, and try to locate the problem.

   You can also enable a ton of (sometimes spurious) debug messages using the `debug=1` value on commandline. Other useful flags are

   ```
   "nomc=1"            disables the use of the Metacrawler
   "noemail=1"         disables the use of external email directories
   ```

   If it's not just a simple syntax error, you might want to start Ahoy! using the Perl debugger. This gives the advantage that it fill dump a stack trace once the script breaks, so that you can find out which arguments got passed:

   ```
   vorlon-(Alpha)% perl -d nph-ahoy.cgi first=marc last=langheinrich

   Loading DB routines from perl5db.pl version 1
   Emacs support available.

   Enter h or `h h' for help.

   HTTP/1.0 200 OK
   Content-type: multipart/x-mixed-replace;boundary=--ClrScr

   --ClrScr
   ```

```
Content-type: text/html

<HTML>
<head><title>Ahoy! Searching for Marc Langheinrich</title></head>
<address>Please wait...</address><hr>
<h2>Searching for: Marc Langheinrich</h2><hr>

<ul>
main::(./nph-ahoy.cgi:4):       require 5.003;
  DB<1>
```

If you just type 'c' for 'continue', the debugger should try to execute ahoy and will report the current stack content once it encounters an error. If you can't figure out what the problem is, send mail to `marclang@cs.washington.edu` with the a detailed description, and at least a copy of what's been printed out when trying to run the Ahoy! system from commandline (w/o debugger)

Common problems are:

- `Can't locate Foo/Bar.pm in @INC (@INC contains: ...)`
  Make sure you installed all the neccessary Perl modules for Ahoy! (*i.e.*, the `libwww` library, which contains `LWP::`, `WWW::`, `HTTP::` and `HTML::` modules). An excellent module for installing additional Perl modules is the CPAN module that ships with Perl5.004:

  ```
  draz% perl -MCPAN -e shell
  ```

  Also make sure that the nph-ahoy.cgi and nph-guessing.cgi scripts contain a

  ```
  use lib '.';
  ```

  at the beginning, which enables it to find the `Ahoy/`, `LWP/` and `WWW/` modules that are locally installed underneath the `CGI/` directory!
- `Can't locate object method "new" via package "Foo::Bar"`
  You might have mistyped the package name when calling the method described in the error message. Make sure you have a `use Foo::Bar` or `require Foo::Bar` in your code. Or, you mistyped the method name. Make sure the method given does indeed exist in the package described. To get a list of all directories where Ahoy! looks for libraries, type

  ```
  draz% perl -e 'join ("\n", @INC),"\n");'
  ```

  Any `use lib "~some/dir"` will be *prepended* to this list!
- `Can't locate auto/LWP/ParallelUA/init_reques.al in @INC`

  The extension modules `LWP::ParallelUA` and `LWP::RobotPUA` are only installed locally. You need to have a explicitly list the path to the directory where they are installed. If the `LWP/` directory is in the same directory as nph-ahoy.cgi, all you need to add is

```
use lib '.';
```

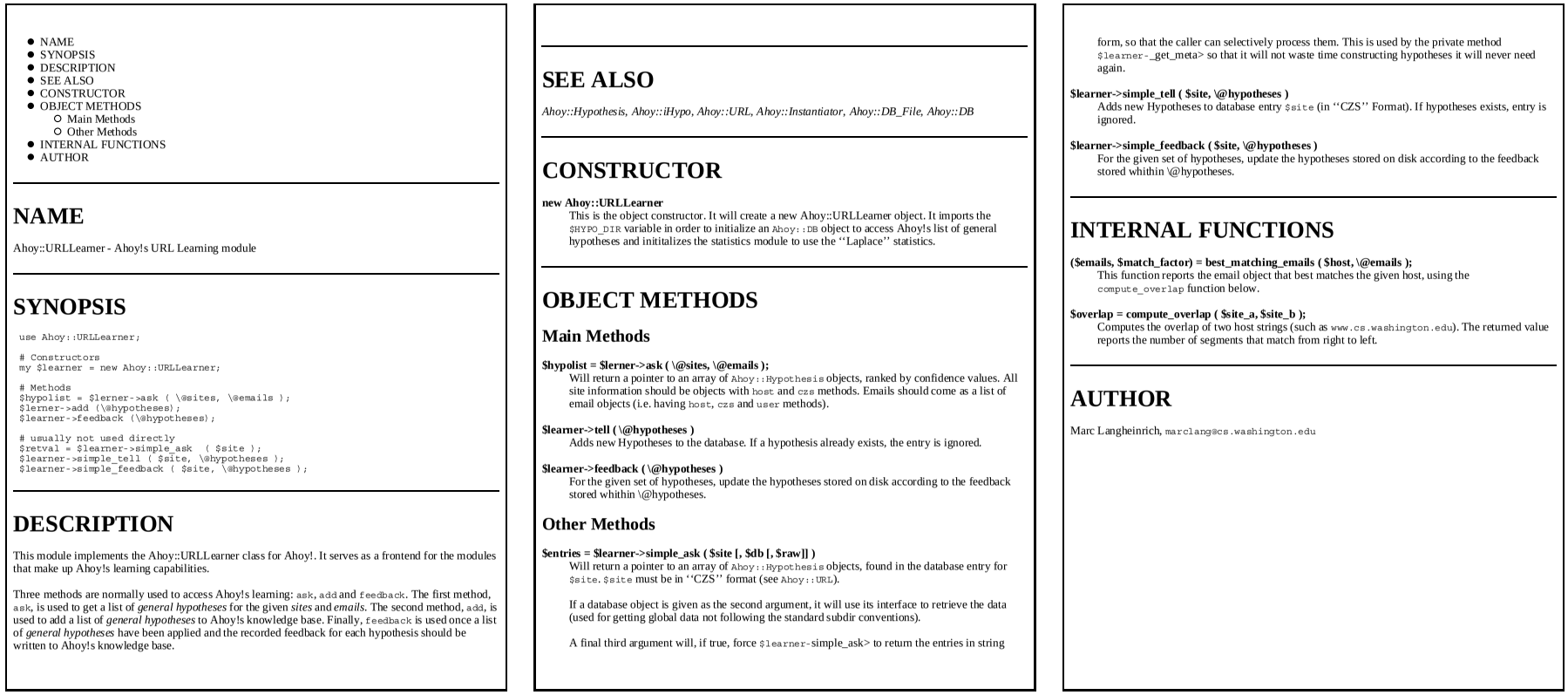
This should fix the problem.

B.4 Embedded Source Code Documentation

Most Ahoy! modules have method and function documentation embedded into the source code. Using Perl's "Plain Old Documentation" (POD) format, the documentation for each module and its routines can be displayed as a manpage (using `perldoc`), as an HTML page (using `pod2html`), as a LaTeXpage (using `pod2latex`) and a number of other formats.

Figure B.1 shows the documentation for the URLLearner module in HTML format. The corresponding source code is listed below:

```
# $Id: URLLearner.pm,v 1.6 1997/05/17 01:43:27 marclang Exp $

package Ahoy::URLLearner;

# Ahoy modules
use Ahoy::DB;
use Ahoy::DB_File;
use Ahoy::URL;
use Ahoy::iHypo;
use Ahoy::Hypothesis;

use Ahoy::Globals qw($HYPO_DIR $session);

# Standard modules
use Carp();
use strict;

=head1 NAME

Ahoy::URLLearner - Ahoy!s URL Learning module

=head1 SYNOPSIS

 use Ahoy::URLLearner;

 # Constructors
 my $learner = new Ahoy::URLLearner;

 # Methods
 $hypolist = $lerner->ask ( \@sites, \@emails );
 $lerner->add (\@hypotheses);
 $learner->feedback (\@hypotheses);
```



- NAME
- SYNOPSIS
- DESCRIPTION
- SEE ALSO
- CONSTRUCTOR
- OBJECT METHODS
  - Main Methods
  - Other Methods
- INTERNAL FUNCTIONS
- AUTHOR

NAME

Ahoy::URLLearner - Ahoy!s URL Learning module

SYNOPSIS

```
use Ahoy::URLLearner;

# Constructors
my $learner = new Ahoy::URLLearner;

# Methods
$hypolist = $lerner->ask ( \@sites, \@emails );
$lerner->add (\@hypotheses);
$learner->feedback (\@hypotheses);

# usually not used directly
$retval = $learner->simple_ask  ( $site );
$learner->simple_tell ( $site, \@hypotheses );
$learner->simple_feedback ( $site, \@hypotheses );
```

DESCRIPTION

This module implements the Ahoy::URLLearner class for Ahoy!. It serves as a frontend for the modules that make up Ahoy!s learning capabilities.

Three methods are normally used to access Ahoy!s learning: `ask`, `add` and `feedback`. The first method, `ask`, is used to get a list of *general hypotheses* for the given *sites* and *emails*. The second method, `add`, is used to add a list of *general hypotheses* to Ahoy!s knowledge base. Finally, `feedback` is used once a list of *general hypotheses* have been applied and the recorded feedback for each hypothesis should be written to Ahoy!s knowledge base.

SEE ALSO

Ahoy::Hypothesis, Ahoy::iHypo, Ahoy::URL, Ahoy::Instantiator, Ahoy::DB_File, Ahoy::DB

CONSTRUCTOR

new Ahoy::URLLearner
This is the object constructor. It will create a new Ahoy::URLLearner object. It imports the `$HYPO_DIR` variable in order to initialize an `Ahoy::DB` object to access Ahoy!s list of general hypotheses and inititalizes the statistics module to use the ''Laplace'' statistics.

OBJECT METHODS

Main Methods

$hypolist = $lerner->ask (\@sites, \@emails);
Will return a pointer to an array of `Ahoy::Hypothesis` objects, ranked by confidence values. All site information should be objects with `host` and `czs` methods. Emails should come as a list of email objects (i.e. having `host`, `czs` and `user` methods).

$learner->tell (\@hypotheses)
Adds new Hypotheses to the database. If a hypothesis already exists, the entry is ignored.

$learner->feedback (\@hypotheses)
For the given set of hypotheses, update the hypotheses stored on disk according to the feedback stored whithin \@hypotheses.

Other Methods

$entries = $learner->simple_ask ($site [, $db [, $raw]])
Will return a pointer to an array of `Ahoy::Hypothesis` objects, found in the database entry for `$site`. `$site` must be in ''CZS'' format (see `Ahoy::URL`).

If a database object is given as the second argument, it will use its interface to retrieve the data (used for getting global data not following the standard subdir conventions).

A final third argument will, if true, force `$learner-`simple_ask> to return the entries in string form, so that the caller can selectively process them. This is used by the private method `$learner-`_get_meta> so that it will not waste time constructing hypotheses it will never need again.

$learner->simple_tell ($site, \@hypotheses)
Adds new Hypotheses to database entry `$site` (in ''CZS'' Format). If hypotheses exists, entry is ignored.

$learner->simple_feedback ($site, \@hypotheses)
For the given set of hypotheses, update the hypotheses stored on disk according to the feedback stored whithin \@hypotheses.

INTERNAL FUNCTIONS

($emails, $match_factor) = best_matching_emails ($host, \@emails);
This function reports the email object that best matches the given host, using the `compute_overlap` function below.

$overlap = compute_overlap ($site_a, $site_b);
Computes the overlap of two host strings (such as `www.cs.washington.edu`). The returned value reports the number of segments that match from right to left.

AUTHOR

Marc Langheinrich, `marclang@cs.washington.edu`

Figure B.1: **Embedded Documentation Sample.** Most Ahoy! modules contain embedded documentation in their source code. Using Perl's "Plain Old Documentation" (POD) format, the documentation for each module and its routines can be displayed as a manpage (using `perldoc`), as an HTML page (using `pod2html`) or as a LaTeXpage (using `pod2latex`). The above example shows the documentation for the URLLearner module in HTML format.

```
 # usually not used directly
 $retval = $learner->simple_ask  ( $site );
 $learner->simple_tell ( $site, \@hypotheses );
 $learner->simple_feedback ( $site, \@hypotheses );

=head1 DESCRIPTION

This module implements the Ahoy::URLLearner class for Ahoy!. It serves
as a frontend for the modules that make up Ahoy!s learning
capabilities.

Three methods are normally used to access Ahoy!s learning: C<ask>,
C<add> and C<feedback>. The first method, C<ask>, is used to get a
list of I<general hypotheses> for the given I<sites> and I<emails>.
The second method, C<add>, is used to add a list of I<general
hypotheses> to Ahoy!s knowledge base. Finally, C<feedback> is used
once a list of I<general hypotheses> have been applied and the
recorded feedback for each hypothesis should be written to Ahoy!s
knowledge base.

=head1 SEE ALSO

L<Ahoy::Hypothesis>, L<Ahoy::iHypo>, L<Ahoy::URL>,
L<Ahoy::Instantiator>, L<Ahoy::DB_File>, L<Ahoy::DB>

=cut

=head1 CONSTRUCTOR

=over 4

=item new Ahoy::URLLearner

This is the object constructor.  It will create a new Ahoy::URLLearner
object.  It imports the $HYPO_DIR variable in order to initialize an
C<Ahoy::DB> object to access Ahoy!s list of general hypotheses and
inititalizes the statistics module to use the "Laplace" statistics.

=cut

sub new {
    my ( $class ) = @_;

    my $self = {
        # database is in the 'hypotheses' subdirectory, 2-char directories
```

```
        # maxumim depth 1 subdir below country and zone subdirs (see DB.pm)
        db => new Ahoy::DB ($HYPO_DIR,2,1),
        wp => undef,  # will hold reference to whitepages object
    };
    bless $self, $class;

    Ahoy::DB_File->init($HYPO_DIR);
    Ahoy::Hypothesis->set_statistics ("Laplace");  # or "Laplace" or "ASR", ...
    $self;
}

=back

=head1 OBJECT METHODS

=head2 Main Methods

=over 4

=item $hypolist = $lerner->ask ( \@sites, \@emails );

Will return a pointer to an array of C<Ahoy::Hypothesis> objects,
ranked by confidence values. All site information should be objects
with C<host> and C<czs> methods. Emails should come as a list of email
objects (i.e. having C<host>, C<czs> and C<user> methods).

=cut

sub ask {
    my ($self, $sites, $emails) = @_;

    # the $meta-hash will hold accumulated statistics for each path in
    # different levels of abstraction: globally, for all .edu domains,
    # etc.
    # Since we usually call 'ask' only once, we make this a local
    # variable, in order to keep the object itself simple in its
    # representation. If we would have several 'ask' requests in
    # sequence, we should probably cache this value accross multiple
    # calls...
    my $meta = {};

    # determine DB entries (in CZS format) to query
    # i.e. www.cs.washington.edu and www.ee.washington.edu are at the same
    # location and should only request one CZS-entry: "edu/washington"
    my %czs;
    foreach (@$sites) { $czs{$_->czs}++; };
```

```
# create hash table with all hypotheses sorted by host
my ($site, %hypotheses);
foreach $site (keys %czs) {
    # load hypotheses for each czs entry
    my $entries = $self->simple_ask ($site);

    my ($entry, $host, $path);
    foreach $entry (@$entries) {
        # and sort per host
        $host = $entry->host;
        $hypotheses{$host} = []  unless $hypotheses{$host};
        push (@{$hypotheses{$host}}, $entry);

        $path = $entry->path;
        # create meta_stats for $site at the same time
        if ($meta->{$site}->{$path}) {
            # merge two stats
            $meta->{$site}->{$path}->statistics->merge($entry->statistics);
            # merge placeholders
            $meta->{$site}->{$path}->placeholder ($entry->placeholder);
        } else {
            # clone hypothesis and statistics
            my $path_hypo = clone Ahoy::Hypothesis $entry;
            $meta->{$site}->{$path} = $path_hypo;
        }
    }
}
undef %czs; # done with that

# domain dependent stuff follows (i.e. email filtering)
if ($emails) {
    # compute email match foreach host
    my ($host, %email_match);
    my $max = 2; # don't bother unless we have at least a match of 2
    foreach $host (keys %hypotheses) {
        my ($matching_emails, $match_factor) =
            &best_matching_emails ($host, $emails);
        if ($match_factor > $max) {# we have a new maximum match factor:
            $max = $match_factor;
            %email_match = ();  # delete all previous entries
        }
        # don't store suboptimal matches
        $email_match{$host} = $matching_emails
            unless $max > $match_factor;
    }
```

```
        # if have any data in %email_match, that means that we could filter
        # out some hosts:
        if (%email_match) {
            # delete all hosts in hypothesis list that are NOT in emails list
            # and expand leftover hypothesis with respective email info
            foreach $host (keys %hypotheses) {
                my $matching_emails;
                unless ($matching_emails = $email_match{$host}) {
                    delete $hypotheses{$host};
                } else { # tell hypothesis which email works for it
                    my $hypothesis;
                    foreach $hypothesis (@{$hypotheses{$host}}) {
                        $hypothesis->emails ($matching_emails);
                    }
                }
            }
        }

    # read this return value (Schwartzian Transform) from bottom to top!
    return [ # sort all remaining hypotheses by their confidence
             # values, best first
             sort { $b->confidence <=> $a->confidence }
             # now we should check each remaining hypothesis to see
             # wether it's 'established' enough. Otherwise consult
             # the hypothesis' next higher knowledge base
             # (i.e. site hypos, zone hypos, etc)
             map { ( $_->established ? $_ : $self->_get_meta($_,$meta) ) }
             # map pointer to hypotheses per site into array
             map { @$_ }
             values %hypotheses ];
}

# get_meta is a helper method for $self->ask. It will try to augment
# the given hypothesis with statistics found in the meta_kb given as
# the second argument. These additional statistics will be used when
# calling the hypothesises ->confidence method.
sub _get_meta {
    my ($self, $hypo, $meta_kb ) = @_;

    my @levels;
    my $global_db = $self->{'gdb'};

    # assemble substring of czs entry: @levels = ('edu/washington', 'edu', '')
    my @czs = split(/\//, $hypo->czs."/");
```

```
    while (pop @czs) { push (@levels, join ("/",@czs)); }

    # Wow. Watch out. Perl Hack (Schwartzian Transform) ahead!
    my $level; my $path = $hypo->path;
    foreach $level (@levels) {
        # create hash entry for this level if needed
        tie %{$meta_kb->{$level}}, "Ahoy::DB_File", "$level/_global.db"
                                   unless ( exists $meta_kb->{$level} );
        # Creation on demand: Keep strings unless we need the information:
        my $meta_hypo = $meta_kb->{$level}->{$path};
        # if we have a string at this position we, create an hypo object
        # and store it in our hypothesis object
        if ($meta_hypo) {
          $meta_hypo = new Ahoy::Hypothesis ( split (/\t/,$meta_hypo, 3));
          $hypo->push_meta($meta_hypo);
          # if we have an established hypothesis here, we stop
          last if $meta_hypo->established;
        } else {
          return $hypo;
        }
    }
    return $hypo;
}

=back

=over 4

=item $learner->tell ( \@hypotheses )

Adds new Hypotheses to the database. If a hypothesis already exists,
the entry is ignored.

=back

=cut

sub tell {
  my ( $self, $hypotheses ) = @_;

  # sort new hypotheses by each site
  my $sites = Ahoy::URL->sort_by_czs ($hypotheses);

  # delete hosts that we couldn't split (most likely numeric addresses)
  delete $sites->{'/no_split'};
```

```
my $success =1; # forces updates
foreach (keys %$sites) {
  $success += $self->simple_tell ( $_, $sites->{$_} );
}

# sanity check: if we didn't tell anything new to the real sites,
# can it be that we have new global info?! no!! So - let's wait
# with adding global info unless we're sure at least one of our
# local db's got updated...
if ($success) {
  # also sort paths by zone and country, as well as globally
  my $hypothesis; my $paths_seen; my $globals = {};
  foreach $hypothesis (@$hypotheses) {
    my @czs = split (/\//, $hypothesis->czs);
    next unless $czs[0]; # unsplittable hosts start with a '/' --
    # ignore those
    # create hypothesis w/o host information, and copy feedback
    my $path_hypo = new Ahoy::Hypothesis $hypothesis->path;
    $path_hypo->placeholder($hypothesis->placeholder);
    while (pop @czs) {
      my $cz = join ("/", @czs, "_global.db");
      $globals->{$cz} = []  unless $globals->{$cz};
      push (@{$globals->{$cz}}, $path_hypo);
    }
  }

  # now record new info by using a tied hash
  foreach (keys %$globals) {
    my %hash; my $path_hypo;
    tie %hash, "Ahoy::DB_File", $_;
    foreach $path_hypo (@{$globals->{$_}}) {
      # read from hash
      my $existing = $hash{$path_hypo};
      my $new;
      if ($existing) {
        # update
        $new = $existing;
        my @parts = split (/\t/,$existing, 3);
        my $ph_obj; my $added;
        foreach $ph_obj (values %{$path_hypo->placeholder_hash}) {
          # check to see if the placeholder appears in the entry
          unless ($parts[2] =~ /\d *$ph_obj\b/) { # if not, append it
            $new.="\t".$ph_obj->stringify;
          };
        }
      } else {
```

```
          $new = join("\t",$path_hypo->stringify);
        }
        # save to hash
        $hash{$path_hypo} = $new  unless $new eq $existing;
      }
    }
  } # if success
}

=over 4

=item $learner->feedback ( \@hypotheses )

For the given set of hypotheses, update the hypotheses stored on disk
according to the feedback stored whithin \@hypotheses.

=back

=cut

sub feedback {
  my ($self, $hypotheses ) = @_;

  # give individual feedback for each site
  my $sites = Ahoy::URL->sort_by_czs ($hypotheses);

  # delete hosts that we couldn't split (most likely numeric addresses)
  delete $sites->{'/no_split'};

  my $success;
  foreach (keys %$sites) {
    $success += $self->simple_feedback ( $_, $sites->{$_} );
  }

  # sanity check: if we didn't tell anything new to the real sites,
  # can it be that we have new global info?! no!! So - let's wait
  # with adding global info unless we're sure at least one of our
  # local db's got updated...
  if ($success) {
    # record feedback for each zone and country, as well as globally
    my $hypothesis; my $globals = {};
    foreach $hypothesis (@$hypotheses) {
        my @czs = split (/\//, $hypothesis->czs);
        # create hypothesis w/o host information, and copy feedback
        my $path_hypo = new Ahoy::Hypothesis $hypothesis->path;
        $path_hypo->feedback($hypothesis->feedback);
```

```
        $path_hypo->placeholder($hypothesis->placeholder);

        while (pop @czs) {
            my $cz = join ("/", @czs, "_global.db");
            $globals->{$cz} = []  unless $globals->{$cz};
            push (@{$globals->{$cz}}, $path_hypo);
        }
    }

    # now record new info by using a tied hash
    foreach (keys %$globals) {
      my %hash; my $path_hypo;
      tie %hash, "Ahoy::DB_File", $_;
      foreach $path_hypo (@{$globals->{$_}}) {
        # read from hash
        my $existing = $hash{$path_hypo};
        my $new;
        if ($existing) {
          # create hypo
          my $original = Ahoy::Hypothesis->new( split (/\t/,$existing, 3) );
          # update
          $original->update($path_hypo->feedback);
          my $placeholders = $path_hypo->placeholder_hash;
          if ($original->new_placeholder($placeholders)) {
            # grab stats and placeholders and add to existing hypo
            $original->placeholder($placeholders);
          }
          # create string
          $new = join("\t",$original->stringify);
        } else {
          $new = join("\t",$path_hypo->stringify);
        }
        # save to hash
        $hash{$path_hypo} = $new  unless $new eq $existing;
      }
    }
  } # if success
}

#
# Private Methods (sort of)
#

=back

=head2 Other Methods
```

```
=over 4

=item $entries = $learner->simple_ask ( $site [, $db [, $raw]] )

Will return a pointer to an array of C<Ahoy::Hypothesis> objects, found
in the database entry for C<$site>.  C<$site> must be in "CZS" format (see
C<Ahoy::URL>).

If a database object is given as the second argument, it will use
its interface to retrieve the data (used for getting global data not
following the standard subdir conventions).

A final third argument will, if true, force C<$learner->simple_ask> to
return the entries in string form, so that the caller can selectively
process them. This is used by the private method
C<$learner->_get_meta> so that it will not waste time constructing
hypotheses it will never need again.

=cut

sub simple_ask {
    my ($self, $site, $db, $raw) = @_;
    $site .= ".hyp";
    my @entries;
    # allow usage of alternate DB scheme
    $db = $self->{db} unless $db;

    # get raw text
    my $lines = $db->read ( $site );
    # create hypotheses unless the user wants raw text
    if ($raw) {
      $lines;
    } else {
      foreach (@$lines) {
        push ( @entries, new Ahoy::Hypothesis ( split (/\t/,$_, 3) ) );
      }
      \@entries;
    }
}

=back

=over 4

=item $learner->simple_tell ( $site, \@hypotheses )
```

```
Adds new Hypotheses to database entry $site (in "CZS" Format). If
hypotheses exists, entry is ignored.

=cut

sub simple_tell {
    my ($self, $site, $hypos, $db) = @_;
    $site .= ".hyp";
    # allow usage of alternate DB scheme
    $db = $self->{db} unless $db;

    my ($entries, $fh) = $db->read_and_keep_open( $site );
    # make sure open was successful
    return unless $fh;

    #
    # Munch new hypos with existing ones:
    #
    # Don't create new hypos here! Wait until we really have to
    my %hypotheses = # read from bottom to top  ("Schwartzian Transform")
      # 2nd: create pair consisting of $hypo->as_string and original line
      map { [ split (/\t/,$_, 2) ]->[0] => $_ }
      # 1st: Start with list of entries
      @$entries;
    # In order to speed up things if we just have some new hypotheses, this
    # hash will exclusively hold _new_ hypotheses. If we didn't update any
    # of the old ones, we can then just _append_ the new hypotheses, instead
    # of rewriting the whole file...
    my %new;

    # Foreach hypothesis given, try to update the ones we just read from disk.
    # We only have to check if it exists. If so, we simply append _new_
    # placeholders to the string lines if neccessary, taking into account
    # our 'tell' semantics which only allow adding new placeholders here, but
    # not increasing 'successes' counts!!
    my ($hypothesis, $original, $updated, $new, @parts, $hypo_string);
    foreach $hypothesis (@$hypos) {
      $hypo_string = $hypothesis->as_string;
      unless ($original = $hypotheses{$hypo_string}) {
        # add new hypothesis
        $new++;
        $hypotheses{$hypo_string} =
          $new{$hypo_string} = $hypothesis;
      } else {
        # for the sake of speed, we don't convert an entry into an object
```

```
        # unless we really have to
        if (ref($original)) { #if it already is an object, go ahead
          my $placeholders = $hypothesis->placeholder_hash;
          if ($original->new_placeholder($placeholders)) {
            # grab placeholders and add to existing hypo
            $original->placeholder($placeholders);
            $updated++;
          }
        } else {
          # if we don't have an object here, don't bother creating one
          # just incrementally add new placholders at the end of the line
          @parts = split (/\t/,$original, 3);
          my $ph_obj; my $added;
          foreach $ph_obj (values %{$hypothesis->placeholder_hash}) {
            # check to see if the placeholder appears in the entry
            unless ($parts[2] =~ /\d *$ph_obj\b/) { # if not, append it
              $original.="\t".$ph_obj->stringify;
              $added++;
            };
          }
          if ($added) { # update entry
            $hypotheses{$parts[0]} = $original;
            $updated++;
          }
        }
      }
    };

    # now save the updated hypotheses back to disk
    if ($new and not $updated) {
      @$entries = map { join("\t", $_->stringify()) } values %new;
      return $db->add ( $site, $entries, $fh );
    } elsif ($updated) {
      # can't sort anymore here (we have a hash of mixed strings and objects)
      # and it would take too long to create new objects every time!
      @$entries = map { ref($_)? join("\t", $_->stringify()) : $_; }
                  values %hypotheses;
      return $db->write ( $site, $entries, $fh );
    } else {
      close ($fh);
      return 0;
    }
}

=back
```

```
=over 4

=item $learner->simple_feedback ( $site, \@hypotheses )

For the given set of hypotheses, update the hypotheses stored on disk
according to the feedback stored whithin \@hypotheses.

=back

=cut

sub simple_feedback {
    my ($self, $site, $hypos, $db) = @_;
    $site .= ".hyp";
    # allow usage of alternate DB scheme
    $db = $self->{db} unless $db;
    # load DB-entries (as strings) into memory
    my ($entries, $fh) = $db->read_and_keep_open( $site );
    # make sure open was successful
    return unless $fh;

    # create hypotheses-hash on the fly
    my %hypotheses = # read from bottom to top  ("Schwartzian Transform")
      # 2nd: create pair consisting of $hypo->as_string and original line
      map { [ split (/\t/,$_, 2) ]->[0] => $_ }
      # 1st: Start with list of entries
      @$entries;
    my %new;

    # foreach hypothesis given, try to update the ones we just read from disk
    # we only have to check if it exists, and if so, if we have new place-
    # holders
    my ($hypothesis, $original, $updated, $new, @parts, $hypo_string);
    foreach $hypothesis (@$hypos) {
      next unless $hypothesis->feedback;

      $hypo_string = $hypothesis->as_string;

      # make note in our status file
      $session->dump_to_file ("feedback.data", "\nHypothesis Feedback:",
                              $hypothesis->feedback."\t$hypo_string");

      unless ($original = $hypotheses{$hypo_string}) {
        # hmm someone must have deleted the original hypotheses on disk
        # or this is a redirection which resulted in a new hypothesis
        # add new hypothesis
```

```
        $new++;
        $hypotheses{$hypo_string} =
          $new{$hypo_string} = $hypothesis;
      } else {
        # we'll have to generate a new hypothesis here, since we'll have
        # to recompute the statistics now...
        $original = Ahoy::Hypothesis->new( split (/\t/,$original, 3) )
          unless ref($original);
        $original->update($hypothesis->feedback);
        my $placeholders = $hypothesis->placeholder_hash;
        if ($original->new_placeholder($placeholders)) {
          # grab stats and placeholders and add to existing hypo
          $original->placeholder($placeholders);
        }
        # replace entry in our hash with hypo-object;
        $hypotheses{$hypo_string} = $original;
        $updated++;
      }
    };

    # now save the updated hypotheses back to disk
    if ($new and not $updated) {
      @$entries = map { join("\t", $_->stringify()) } values %new;
      return $db->add ( $site, $entries, $fh );
    } elsif ($updated) {
      # can't sort anymore here (we have a hash of mixed strings and objects)
      # and it would take too long to create new objects every time!
      @$entries = map { ref($_)? join("\t", $_->stringify()) : $_; }
                  values %hypotheses;
      return $db->write ( $site, $entries, $fh );
    } else {
      close ($fh);
      return 0;
    }
}


#
# functions
#

=head1 Internal Functions

=over 4

=item ($emails, $match_factor) = best_matching_emails ( $host, \@emails );
```

```
This function reports the email object that best matches the given
host, using the C<compute_overlap> function below.

=cut

sub best_matching_emails {
    my ($host, $emails) = @_;

    # reports email objects that best match the given host, including
    # the match factor.
    my ($email, %matches, @emails);
    my $best_match = 0;
    foreach $email (@$emails) {
        my $email_host = $email->host;
        # check cache (%matches)
        unless (exists $matches{$email_host}) {
            # create new cache entry if we have a new host
            $matches{$email_host} = &compute_overlap ($host, $email->host);
            # new maximum?
            if ( $matches{$email_host} > $best_match ) {
                $best_match = $matches{$email_host};
                @emails = ();  # delete all previous entries
            }
        }
        # don't store suboptimal matches
        push (@emails, $email) unless $best_match > $matches{$email_host};
    }
    (\@emails, $best_match);
}

=back

=over 4

=item $overlap = compute_overlap ( $site_a, $site_b );

Computes the overlap of two host strings (such as
C<www.cs.washington.edu>). The returned value reports the number of
segments that match from right to left.

=back

=cut

sub compute_overlap {
```

```
    my $site_a = shift;
    my $site_b = shift;
    my @site_a = reverse split (/\./,$site_a); # we want to compare
    my @site_b = reverse split (/\./,$site_b);     # from back to front!
    &match_factor(\@site_a, \@site_b);
}
# private
sub match_factor { # computes the overlap of two lists
  my $first = shift;
  my $second = shift;

  my $overlap = 0;
  while ($first->[$overlap] eq $second->[$overlap]) {
      $overlap++;
      last unless defined $first->[$overlap];
  }
  $overlap;
}

# keep perl happy
1;

=head1 AUTHOR

Marc Langheinrich, C<marclang@cs.washington.edu>
```