A Folder-Based Graphical Interface for an Information Retrieval System

by

Aidan Low

Submitted to the Department of

Electrical Engineering and Computer Science

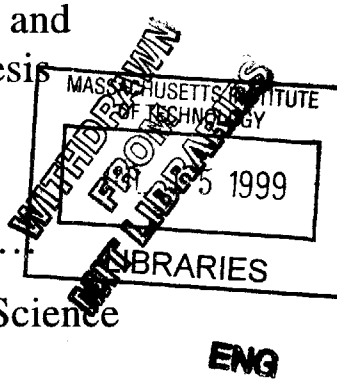
in partial fulfillment of the requirements

for the degree of Master of Engineering in

Electrical Engineering and Computer Science

at the MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1999



Author .....

Department of Electrical Engineering and Computer Science

Certified by .....

David R. Karger

Associate Professor, Thesis Advisor

Accepted by.....

Arthur C. Smith

Chairman, Department Committee on Graduate Students

A Folder-Based Graphical Interface for an Information Retrieval System

by

Aidan Low

Submitted to the Department of Electrical Engineering and Computer Science on May 20th, 1999, in partial fulfillment of the requirements for the degree of Master of Engineering in Electrical Engineering and Computer Science

Abstract

Past advances in user interface design include the move from an application-centric model to the document-centric model found in the file/folder graphical interfaces of modern operating systems. Recently, we have also seen advances in query-based information retrieval technology, such as the search engines of the World-Wide Web. This thesis proposes an interface that combines the power of these two technologies to produce a unified interface that allows users to navigate their information space more easily and more intuitively. This interface, the Haystack Browser, presents users with a file/folder graphical interface that allows users to dynamically categorize their information space by creating "folders" that correspond to queries within an information retrieval system. The interface is implemented as an extension of the Haystack Project, a personalized information retrieval system under development at the MIT Laboratory for Computer Science.

Thesis Supervisor: David R. Karger

Title: Associate Professor

Dedication

This thesis is dedicated to my family: Mom, Dad, Devin, Megan, Capen, Misha, and Chris. In my eyes, they're immortals.

Acknowledgements

I wish to thank Eytan Adar, who not only helped to build Haystack before it was turned over to me, but who has been exceedingly helpful both in my work on Haystack and in the writing of this thesis. Thank you, Eytan.

Contents

List of Figures

List of Tables

Chapter 1.

Introduction

Over the last decade, the role of desktop computers has changed radically. With the rise of the Internet and globally available information, the primary role of computers has changed from problem-solving and information production into new roles as information browsers and archives.

Users today store more and more of their daily lives on their computers, from electronic mail to relevant news articles to technical papers. With the enormous growth in storage capacity of modern hard disk drives, users can store more information on their computers than ever before, and with the explosion of the Internet and the World-Wide Web, users now have access to a near-infinite amount of information electronically. What used to require a trip to the library and looking through shelves and shelves of books can now be accessed in milliseconds from a user's desktop.

Yet, in this age of near-unlimited information, we have not solved the information problem. The issue now is not gaining access, but actually finding relevant information once we have access. Once we wanted a book that we didn't have in our bookshelf; now

we want a book that's somewhere in a basement full of billions of similar-looking books. We now have a real need for systems that facilitate finding information quickly and easily.

Our goal, then, is to help satisfy a user's "information need": her goal of finding some particular information. That information may be a particular known document, or it may any document that contains knowledge on a particular topic. We have a serious problem, however, in translating this need from the user's mind to her computer in a way that the machine can understand so that it can retrieve the relevant information.

The problem is one of organization of information and coordination between the automated systems of the computer and the thought processes of the user. We need some kind of system that can allow the user to organize her information with as little work as possible, but still be able to quickly and easily find the information she seeks.

This thesis presents such a system in the form of a graphical interface to an information retrieval (IR) system. This interface, the Haystack Query Browser, provides the same file/folder interface as current graphical file browsers, but allows users to dynamically categorize their information space. By combining an intuitive interface with an information retrieval system that allows a user to automate the search process, we have created a system that we feel will be of significant benefit to the user in satisfying her information need, though we have not been able to conduct any user studies to verify this at this time. (see Chapter 7, Future Work)

1.1 Information Organization

In today's world, we have access to more information than we can possibly look at and finding information within this sea of data has become near impossible with our old methods of searching. The information space that was once a few dozen word-processing documents on a user's local hard drive is now the combined information of the entire world, all available from anywhere over the World-Wide Web. [Bern89] While the user could once organize her files any way she wanted, she must now cope with an organization that may be quite foreign to her

This is not only a problem associated with the Internet, but a problem that has arisen as the result of a shift towards electronic storage of almost every kind of information and a dramatic increase in the storage space available through today's disk drive technology. Computers have become the repositories for our calendar information, meeting notes, memos, scientific papers, correspondence, and official documents. Further, as computers facilitate the exchange of information with other people, they also bring people into conflict with others' dissimilar organizational structures. A user may receive a collection of documents from a colleague that are organized in a way completely unlike the way the first user would have organized them. However, it would be helpful if the original user could navigate through these documents without manually resorting them all.

This explosion of information calls for new organizational models that can allow users to easily and efficiently navigate their information space and find information of interest to them. We need models that allow dynamic categorization of documents, allowing users

to specify the organization of information at the time of their information need, rather than relying on a static categorization from the past that may not be relevant to the information need of the moment.

1.2 Document-centric Computing

One can argue that the purpose of the user interface is to allow the user to do maximum amount of work with a minimum amount of effort and as little mindless busywork as possible. That said, the somewhat paradoxical concept of an ideal user interface is a "transparent" interface that the user does not even notice.

As user interfaces have progressed from command line interfaces to graphical interfaces, there has been a real shift in the paradigm of how users think about programs and data. [Harr96] The old paradigm of computing is *application-centric*, in that a user thinks of running programs, each of which can access a number of files. A newer paradigm is the *document-centric* model, in which a user views data in the context of discrete documents that she can view and modify using one of a number of programs.

In the application-centric paradigm, a typical action by a user would be the following: The user goes to the directory in which a word-processing program resides. The user starts the program, selects File/Open, goes to the directory in which the data file resides, and then selects the data file and clicks "Open". The file is loaded, and only now can the user actually use the system. The user has to expend concentration and time on dealing with starting the program rather than dealing with the data directly.

Contrast this with an example from the new paradigm: the user goes to the directory where the data file she is interested in, and selects it. The program automatically starts up with that data file already loaded, and the user can immediately begin working. While this still requires that the user locate the data, this is a conceptually simpler interface. The user is free to concentrate on the data that she wants to work with rather than wasting energy and time on dealing with the interface.

This document-centric paradigm is much closer to the ideal user interface. The old paradigm concentrated on "programs", artifacts of the user's need to use a machine to get at her data. In the new paradigm, the user can instead concentrate fully on the data that she is interested in. This is not a simply matter of saving a few mouse clicks; rather it is a conceptual distinction between accessing information through a clean, unified interface versus using a fragmented interface that requires switching between several different methods of control.

1.3 Combining the two

This thesis proposes a system that combines the graphical interface of the document-centric paradigm with a system that allows the user to dynamically organize her information space.

This interface uses a search-based information retrieval system to produce a dynamic content-based organization of information. In this paradigm, rather than associate each document with a particular location in a fixed hierarchy, the system dynamically generates the hierarchy as necessary. At any time, the user can ask the system to create a

category and automatically determine which of the archived documents fit that category. In this paradigm, documents can belong to any number of categories, and far more importantly, the categories used to partition the information space are those important to the user at the time of the search, rather than those deemed important when the static hierarchy was first created. When the user is looking for a history class paper she wrote two years ago, she can create the category of "documents more than two years old about history" and the system will automatically populate it with all documents that fit that category. She could then browse through all the history papers she wrote for school two years ago, and could easily choose the file she wanted out of the three or four presented. This dynamic content-based hierarchy is a relatively new concept that has not been explored by conventional file browsers, and one that we believe is an extremely useful method of organization that will become more and more important as time goes on.

Our goal in this project is to take all the intuitive benefits of the file/folder interface and insert them into the interface for an information retrieval system, while at the same time taking the power of a dynamic categorization system and inserting it into the user's standard file browser. We believe that dynamic categorization systems are the paradigm of document organization of the future. We believe also that combining such a categorization system with the file/folder metaphor will provide users with a unified user interface that will give users a simple and intuitive metaphor for file manipulation along with the organizational power of an information retrieval system. This thesis hopes to provide a user interface that can be used as a user's primary browser for navigating her

personal information space, and enable users to explore this space and find documents more easily and more effectively.

By presenting the user with a file/folder-based interface to an information retrieval system, it is our hope that this more intuitive means of navigating and organizing this information space will allow the user to spend more time working with her information and less time working just to find it. We quote Michael Dertouzos, Director of the MIT Laboratory for Computer Science: "We will do more by doing less." [Dert98]

The Haystack system is a personalized information retrieval system developed under Professors David Karger and Lynn Andrea Stein at the MIT Laboratory for Computer Science. Haystack allows users to organize their information space, be it the files on their local file system, incoming email, or recently visited web pages. The Haystack system archives a user's information as well as extracting as much metadata (author, creation date, subject matter) as possible to assist the user in navigating her information space. The implementation of this thesis provides the main graphical interface to the Haystack system.

1.4 Thesis Roadmap

Chapter 2 will look at background information on document organizational models and user interface paradigms. It will also look at related research projects that also combined information retrieval with document organization. We will then look at the Haystack information retrieval system that has been extended with our graphical interface.

Chapter 3 presents the Haystack Browser, the main component of the new graphical interface that this thesis presents. This novel interface presents the user with a document-centric interface that allows dynamic categorization of her information space. We will look at design goals and the overall design of the interface, and then will discuss its individual components and functions in depth.

Chapter 4 looks at query refinement, the process of incrementally modifying a query to better approximate the information need of the user. This section will look at the mechanisms available in the Haystack Browser that allow users to easily refine their queries, using the Haystack system to automate as much of this process as possible.

Chapter 5 looks at the ideas of Information Search Sessions and Query Signposting. Information Search Sessions are groups of related queries that all look to satisfy the same information need. After the user has found the query that actually satisfies their information need, we can mark all these related queries to refer to that desired query through the mechanism of Query Signposting. By marking these queries, the next time the user issues one of these queries, she can simply follow a "signpost" to the query she is looking for, rather than repeating all her previous work of query refinement. This section will examine these ideas in greater depth and explain the mechanisms within the interface that support them.

Chapter 6 looks at the other component of this graphical interface: the Haystack Root Window. This small-footprint widget is designed to be a user's easy access to the Haystack system, while taking up a minimum amount of space on the user's desktop. This section will look at the details of this component's implementation, as well as discuss the design choices that were made.

Chapter 7 examines a number of avenues of possible future work on the Haystack Browser, ranging from integrating the system into the user's standard file system, user interface, and web browser to additional graphical browsing components to possible user studies that evaluate the effectiveness of this design.

Chapter 8 concludes with an overview of this interface and a discussion of the implications of this proposal.

Chapter 2.

Background and Related Work

This section will first examine the models of document organization and the computer paradigms that exist today. It then examines related work that has influenced and inspired this thesis, looking at a system that combines an information retrieval system with a file system, an internet search engine, and a completely new workspace-based collaborative tool. Finally, this section looks at the Haystack system itself, discussing the details of the system underneath the graphical interface presented by this thesis.

2.1 Document Organization Models

This section will examine the logical progression of document organization on computers from the old days of unstructured collections of files to the ubiquitous hierarchical directory structures of today and then to the dynamic hierarchies that appear to be the future of document organization.

2.1.1 Single directory

In early computer systems, all files were stored in one location; [Tane92] there was no sense of organizing files into different groups at the file system level. Users kept track of their files themselves, organizing them solely by giving their names, usually restricted to be quite short. This paradigm is usually associated with very old computer systems, but it is also often seen today with novice users who are not familiar with the hierarchical directory paradigm.

In this model, the user can find their documents only by using the names of the files. Note that this simple approach provides two separate methods of locating the files. First, the user can remember the name of the file and ask for it by name. This corresponds to a search for a particular known document in an information space. Second, the user can use the names of the files as clues to help her determine which files look like they have a good chance of being what she wants. This is closer to a search for any documents that satisfy a given information need. The naming scheme works well for small collections of items, but it still takes time linear in the number of documents to look over all the names and choose which one is relevant.

While both of these name-based methods work adequately for small sets of data, neither of them scales well to large sets of data. Remembering the name of the file works in cases of recently used files, frequently used files, or when there are only a very small number of files, but once the number of files grows large this becomes increasingly difficult. Using the names of files as clues to find the desired documents when the

collection of files becomes large is possible, but prohibitively slow. While users can continue to use these methods for large sets of files (and some novice users do just this), it becomes highly inefficient once the body of documents grows large. Exceptions exist, however, in situations where the name of the file can be determined from external information. For example, a system with a directory for each user is navigable if the user knows the username of the person whose home directory she is trying to find. A system with a separate directory for every product can be easy to move around in if the user understands the naming scheme.

There are additional problems of display, that sometimes not all the filenames can be displayed at the same time, and so it may be impossible to use all of the filenames to decide which one to choose. [Furn97] It is clear that we need to find a better way of organizing our files.

2.1.2 Static tree hierarchy

A slightly more sophisticated solution is to organize documents into static hierarchies. Whenever a document is created, it is placed into a hierarchy of directories. When the user wants to find the file again, she must somehow figure out which directory the file is in, and then must retrieve it from there. This is similar to the naming solution, in that both files and directories are named, but the power of the static hierarchy lies in the logarithmic depth of the tree. If directories are well chosen and the user can understand the categorization structure, a particular sought document can be found in time logarithmic in the number of documents, rather than in linear time.

Some early systems, such as CTSS, only had a single layer of directories, separating their documents into categories all rooted at the same level. [Vlec97] In later systems, such as MULTICS [Feie71], users could create directories within directories, and place their files at any node in this tree. This allowed the users to organize their data by grouping files together into directories and creating a classification hierarchy.

This is a very powerful organizational system, and has established itself as the primary method of local file organization over the last few decades, overcoming other competing models of organization such as hypertext [Nels65], though hypertext has established itself as the dominant organizational model of the World-Wide Web

The static tree hierarchy paradigm is so powerful because it combines simplicity with the exponential organizing power of trees. Users in Western society today are very familiar with the concept of hierarchies, and so the metaphor of this organizational structure is very intuitive for them to use. In addition to being simple and intuitive, placing documents within a tree structure means that as long as the user knows which paths to follow to reach her documents, navigating to a desired file takes logarithmic time.

Examples of static tree hierarchies include traditional file systems on modern operating systems and semantic classification systems that organize documents based on the subject matter, such as Yahoo. [Yaho99]

While static classification systems provide a very easy system for finding information, it is very expensive to add information to them, as it requires classifying it into the appropriate place or places within the system. While this may be reasonable if it means carefully choosing where to save the two or three documents a user creates in a day, it is not going to work if we try to categorize every webpage that user visits. The problem is that we are still relying on the user to manually decide how to group their information, rather than doing it automatically.

Further, the classification hierarchy is fixed, and the system only works because every document in it has been evaluated with respect to its structure. If the classification hierarchy is ever changed, all documents within the changed part of the hierarchy must be manually reclassified before the system can be used again. This sort of system is clearly not appropriate for a user who wants to be free to categorize their system based on their moment-to-moment information need.

2.1.3 Search

A radically different solution that does provide this dynamic organization is using a search tool. This method improves lookup time from logarithmic to constant time, at the cost of some reliability. Rather than store the files by names within a hierarchy, this solution puts all the files in a pile and then "indexes" the documents, computing a representation of each document that will allow the system to quickly find matches to "queries". When the user wants to find a document, she comes up with some terms that appear within the document and passes those to the search system as a query. The search

system consults its internal representations of the documents and then gives the user references to all the documents that match the query. This is the solution used today in Internet search engines such as Alta Vista. [Alta99] For the user, this now takes constant time, though it may require several iterations to find a query that will pick out the document under consideration.

There are a number of obstacles to using this as our primary method of document organization. First, this system is usually stateless, so after we have completed a query there is no memory of that query. We cannot create a categorization system and then explore it later unless we are willing to reissue all of those queries manually. Further, when in the process of a query, we can usually only backtrack along a linear path, and if we take a fork in that search path then all the work done down another path will be lost. For example, imagine a user issues a query on "NATO", and then refines that ten times before backtracking to the "NATO" query and instead issuing "NATO Warsaw". All the work that she did in the initial refinement process is lost, and she will have to do that work again if she decides that she did indeed want the result she got at the end of that process.

Second, these systems are usually implemented separately from the primary file browser, so in order to use the results of a search we need to use an additional program. This extra level of indirection goes against the goal of user interface transparency that we discussed earlier. Ideally, we should provide a mechanism that can be accessed from inside of any

application, or at least provide a specialized application that also contains the functionality of a user's traditional file system browser.

2.1.4 Dynamic Hierarchies

The next step in this progression is to provide the user with *dynamic hierarchies* that are created on demand, organizing documents automatically based on their contents and on the classification parameters the user believes to be most useful at the moment. This is a field that has yet to be really explored, and there are only a handful of systems that try to achieve this. We feel, however, that this is definitely the future of document organization. This thesis implements a dynamic hierarchy-based organizational system, in hopes that it will inspire others to experiment with this powerful paradigm and continue research in this area.

The primary goal of a document organization system based on dynamic hierarchy is to provide a mechanism to allow users to use a number of different organization hierarchies. Not every hierarchy will be appropriate for every search; some hierarchies are much better suited to particular queries than others. By providing a number of such hierarchies, we can improve the chances that one of them will work. However, if we can allow the user to specify these hierarchies at the moment of the information need, we are almost assured that the hierarchy will be well suited to the query. This system should allow the user to create this organization as necessary, creating the categories that best helps her find the information she is looking for. If we can achieve this, we have in some sense

created a system that always has the ideal hierarchical organization, which would be extremely helpful to users.

The Semantic File System (see 2.3.1 Semantic File System) is an example of a dynamic hierarchy. The SFS system allows users to create hierarchies of virtual directories that contain all files matching a particular query. Once the hierarchy has been created, it can be accessed repeatedly to check new documents against the organization.

2.2 User Interface Paradigms

This section will examine the logical progression of computer interfaces from the application-centric model to the document-centric model.

2.2.1 Application-Centric Interfaces

The first computer systems had no persistent storage, so users would load a program, load some data, and then execute the program.[Tane92] With the advent of persistent storage, this paradigm continued, for the most part. Programs and data would be loaded onto a system, and then users would execute these programs, which would in turn use the data.

Most early operating systems are examples of this, including DOS [Wolv93] and almost all versions of Unix. [Ritc74] In all of these systems, the user selects an application to use, and then selects the data to use with that application. The user's focus in this paradigm is concentrated on the program, with data files a secondary item used by these programs. A problem with this paradigm is that it goes against the goal of interface

transparency, forcing the user to waste her energy and attention dealing with the interface rather than interacting with her data.

This is not about saving a few mouse clicks; rather, it is a matter of presenting the user with an interface that allows her to work focused on her information. Computer interfaces should try to stay out of the user's way as much as possible, maximizing the amount of time the user works on her data, and minimizing the amount of time the user has to deal with the specifics of the interface. This isn't a matter of the time to click the mouse a few extra times to start the application and choose the file, but a question of the concentration requirements being placed on the user. The user should not have to know which applications are appropriate to use for a particular file; the applications are in some sense artifacts of the system that the user would ideally not even know about. There is a conceptual distinction being made here: the user should be able to concentrate on the information and not the method of manipulating it, and should therefore be able to open documents without knowing anything about the applications or programs required to open them.

2.2.2 Document-centric Interfaces

An improvement to the application-centric paradigm is the document-centric model, pioneered by Xerox PARC's Xerox Star interface in 1989 [John89] and followed by a number of other systems, including Apple MacOS [Appl93] and Microsoft Windows 95 [Kay99]. Individual documents are represented by on-screen "file icons", and collections of these documents are represented by "folder icons." In this model of operation, every

document has a type, and associated with every type is one or more applications that can be used to open files of that type.

The intuitiveness of a user interface is critical to its usefulness. The interface, in order to avoid long learning periods before a user can use a system, should use familiar metaphors to users to access their data as easily as possible. The metaphor of files and folders is highly familiar to most users in Western Society, filing cabinets being fairly ubiquitous within our office culture. Explaining to even a novice user that "this icon that looks like a manila folder represents a collection of documents", and "this icon that looks like a piece of paper represents a single document" is very simple. When operating within a familiar metaphor, users have an immediate intuition of how the paradigm operates and how they can use it to get their work done. [Lean91]

In the document-centric model, the user first selects which piece of data to open, and then chooses an application with which to view it, or often lets the system decide which application is appropriate. In most cases, the default application is the desired one, and the user can open a file simply by double-clicking on the icon. This is a much more powerful model, as it correctly gives the user the ability to interact with their data without explicitly dealing with applications.

If all applications had ideal user interfaces as well, the user might be able to interact with her documents without having any idea of which programs were being used to handle the interaction. This idea of an "object-oriented file system", in which documents contain

both data and the programs to access and display that data, is beyond the scope of this thesis, though it is the next logical extension of the document-centric model.

In some sense, the file/folder metaphor is the combination of a hierarchical structure with a visual representation of that structure, and both parts of this are important. The hierarchical structure allows a user to organize their documents in an efficient way, if she can figure out how to manipulate the directory structures. The graphical representation provides users with a convenient method for organizing these hierarchical structures, making the hierarchical organization structure available to users who might not have been able to understand it in the context of a command-line interface. Further, the representation of documents and folders as readily recognizable objects allows users to leverage their knowledge about documents and folders in the real world, and gain immediate intuitions about how these objects relate to one another in the computer interface.

2.3 Related Work

This section presents three pieces of research that share this thesis' subject matter. The first project examined is the Semantic File System, which provided a combination of a traditional network file system with an information retrieval system to allow dynamic categorization of information. The second system is DLITE, a graphical interface to a number of information retrieval systems developed at Stanford as a part of their Digital Library project. The third system is Northern Light, a World-Wide Web search engine that provides automatic sub-categorization of the result set of a query.

2.3.1 Semantic File System

The direct inspiration for this thesis, the Semantic File System (SFS) [Giff91] was the seminal work in dynamic classification hierarchies. This project provided the user with an information retrieval system accessible via the same interface as the file system. Rather than use a separate program or browser to locate files, learn their "true" location in the file system, and then open the files in an application, users can access the information retrieval system directly from within any application that can open files.

The Semantic File System was an implementation of the Network File System (NFS) [Sun89] specification, extended with the power of *virtual directories.* By using special directory names, the user could signify to the system that she wanted to execute some sort of query against the stored documents. The system would present her with what looked like a directory containing all the documents that matched that query. This "virtual directory" was possible because this information retrieval system was integrated with the file system, so it could provide all normal file system functionality as if these virtual directories were real, though they were actually just the collection of files that matched a query. Users specified these virtual directories in a manner very similar to the way they specified real directories, and could access them from the command-line or within any application that could access the file system.

For example, the user could type

```
>cd /foo
```

to change to the directory named "foo" in the root directory of the system, just as in traditional file systems. She could, however, also type

```
>cd /sfs/owner:/aidan
```

to change into a virtual directory, containing softlinks to all documents that belonged to aidan. To applications, the directory appeared to be like any other directory, and could be examined by any application that could normally browse the file system.

It also supported nested queries, so a user could type

```
>cd /sfs/owner:/aidan/dategt:4.22.99/sizegt:4k
```

to change into a virtual directory filled with all the files that belong to aidan, were created after 4/22/99, and have size greater than 4k. The user could use `cd ..` to change into a directory where the most recent restriction was relaxed, `cd ../sizegt:2k` to change the current restriction, and so on.

This project was important for three main reasons. First, it recognized that an information retrieval system is a much more natural method of retrieving files than relying either on user memory of file location or on the clues provided by directory names. Given an appropriate query, it could show the user the documents she is looking for in O(1) time, rather than the O(n) time that it would take the user to visually scan through a long list of documents. It provided a dynamic organization of the files, categorizing the files based on the distinctions relevant to the user at search time. In this way, it could always give the user what she was looking for in a quick and intuitive way.

Second, it recognized that the information retrieval system must be embedded within the user's normal file browser. If the user has to use a separate system to locate her files,

then go back to their terminal window to change directories to get to the file and only then be able to do useful work, the system will be much less useful. Rather than concentrating on the documents they wish to deal with, users must expend energy and concentration to deal with the complexities of the interface. We would like to make it as easy as possible for a user to use their files once she has found them. To this end, the SFS system took the information retrieval system and placed it within the standard file system, so that the user only needed to use one tool to get at their information. This is something that this thesis strives towards, but has not yet achieved. (see Chapter 7, Future Work)

Third, it recognized that users will be much more likely to use an information retrieval tool that is an extension of their current file browser, rather than a system with a novel interface that the user must additionally learn. By basing SFS on the NFS standard, users could use this interface and still retain all the functionality of their old file browser. When they wanted to use its information retrieval capabilities, they could do so, but when they simply wanted to use the file browser as they did before, they could do that as well. They could continue to use their systems exactly as they had before, rather than adjust to a new interface that gave them the power of information retrieval but lacked the full functionality of their old system. While the current Haystack Browser is not integrated into a standard file browser, it does seek to mimic the semantics and interface of traditional file browsers as much as possible. Future work on this browser may actually integrate it into the user's normal browser. (see Chapter 7, Future Work)

The Semantic File System was a very powerful idea, and perhaps did not achieve wide inclusion in commercial operating systems because of the computing requirements of indexing and searching. When the paper was published in 1991, most desktop computers did not have the necessary power to run an information retrieval engine alongside their file system. In today's world of massive surpluses of computing cycles, information retrieval systems are slowly being incorporated into the file systems of modern operating systems, such as Microsoft's Windows 2000. [Micr99]

2.3.2 DLITE

The Digital Library Integrated Task Environment (DLITE) system [Cous97], developed at Stanford for their Digital Libraries project, took a novel approach to providing a graphical interface to an information retrieval system.

DLITE was based on the idea of *workcenters*, a workcenter being "a place where the tools are ready-to-hand". Each workcenter was a desktop with a number of components that are relevant to the workcenter's designated task. A separate workcenter was customized for each task. The developers of DLITE envisioned these workcenters being maintained by librarians or other information managers, though the changes one user makes to the workcenter were visible to other users as well, allowing collaboration between users.

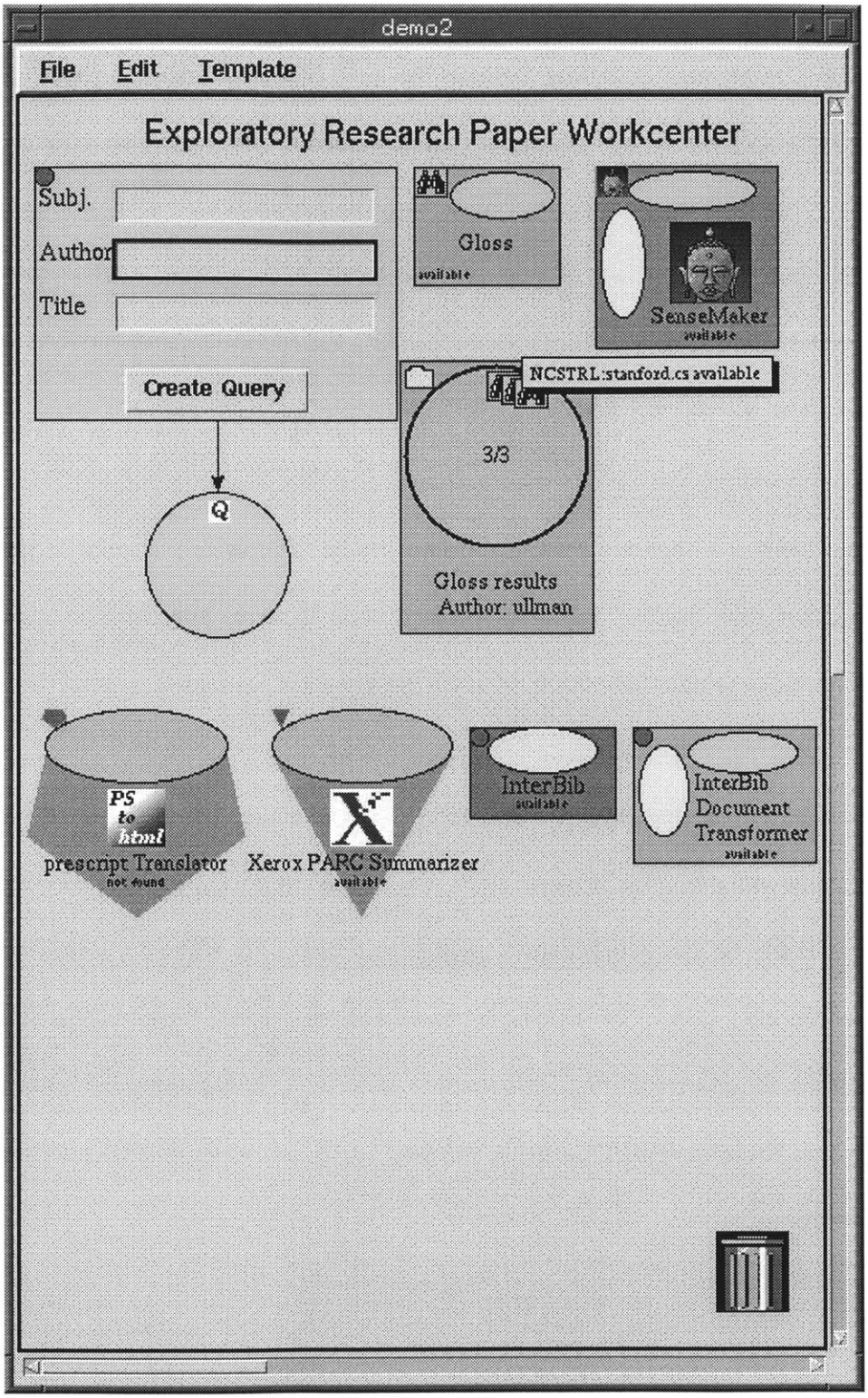


Figure 1. DLITE System

The goal of this system was to allow users to naturally establish the components that are useful to them in the context of the information search that they are currently exploring. Workcenters could contain documents, document collections, queries, and other objects. Users could issue queries to populate document collections, and then manipulate and view the objects within those collections. Workcenters also contained a number of services that could be used to work on behalf of the user. After a query had been created, the user could drag it to a query service to issue it and get the result set back. After a query result set had been returned, a user could drag it to a bibliography service to automatically generate a bibliography containing properly formatted entries for all the documents dragged to the service.

Users working at a particular task would naturally create the tools appropriate to that task, and then future users would be able to use those tools when they came to the workcenter to work on similar tasks. Over time, each workcenter would acquire the necessary tools and document collections for a particular task. Users could then simply choose which workcenter was appropriate for their task, and find it already populated by the appropriate tools.

For example, the user using the DLITE workcenter shown in Figure 1 has created a new query "ullman" and dragged it to the Gloss query system. The system then created a new result set collection with three matching documents, and the user can click on the representative icons and interact with the files that way. She can also drag those icons to the InterBib service icon, which will automatically create a bibliography object. The

DLITE system is useful because it provides access to heterogeneous services through a homogenous interface, making it very easy for a user to manipulate documents from a wide array of sources at the same time.

The DLITE system is an excellent example of how a graphical interface can be combined with an information retrieval system to provide a much more useful tool than a standard search tool. It provided a simple collection of components to allow users to easily interact with the system in an intuitive manner. It also provided support for a collaborative model of search, which is something that will be very important to Haystack in the future.

The DLITE system differed from this thesis in several notable ways. First, DLITE was very much a separate application from the standard file browser. Users used the DLITE system to find, view, and even publish documents, but it did not try to act as the standard interface for users to interact with their files. Second, DLITE did not store history information. While a user could set up a workcenter that was useful for them to interact with a particular set of documents, this was a manual process. The system did not help users to discover commonly explored query sequences or help them to avoid repeating work in the future by reminding them that the last time they issued query X, they ended up refining it to query Z. Third, the DLITE system uses an interface that is very different from traditional user interfaces. While this allows them to design some really novel and interesting methods of interaction, it also means that users will need to undergo a learning period before the system can really be effective for them. One of this thesis' priorities is

to keep this learning period as short as possible, to allow users to benefit from the system immediately.

2.3.3 Northern Light

Northern Light is a particularly interesting example of a dynamic organization system, because it was implemented in the context of an Internet search engine. [Nort99] Northern Light had all the power and functionality of a standard search engine, but provided additional functionality in the form of automatic categorization. After a query was issued, the system provided the result set of documents that matched, but it also automatically grouped these results into a dozen groups, displayed as folders along the side of the display. If the user selected one of these folders, she was shown the elements of the result set that fell into that group. These elements were in turn grouped into subfolders, so the hierarchy of folders could be recursively traversed.

This system effectively produced a dynamic organization of the information space of the entire Internet, using the user's first query to narrow the collection of documents to a manageable size. By use of these folders, the user could quickly see the groups of information that had been returned by her query, and could select the topic that she was looking for. Further, the system provided a number of "power search" features that allowed the user to specify additional constraints on the documents returned, ranging from date range to subject matter to the name of the publication to search in.

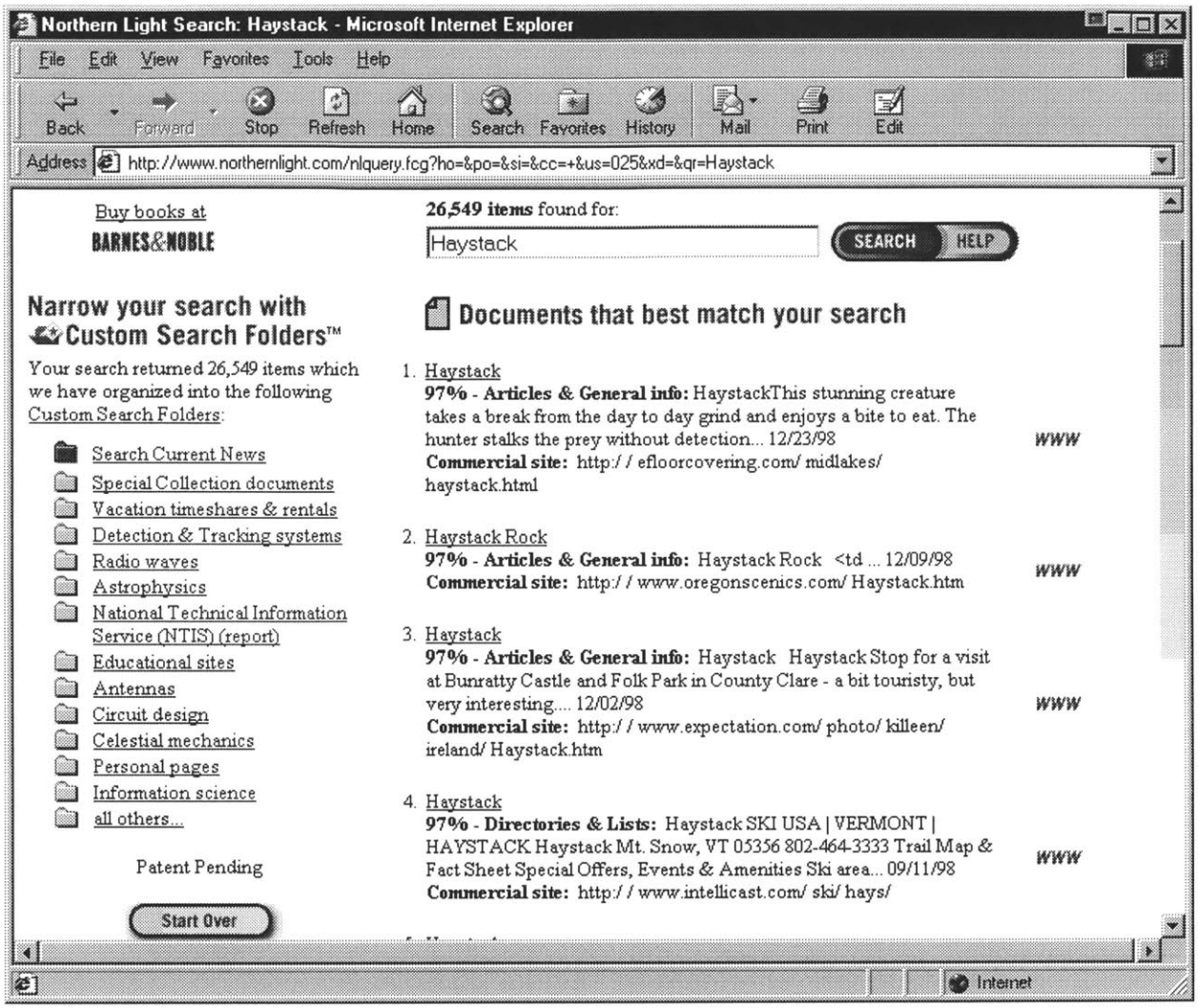


Figure 2. Northern Light

This was a very powerful idea, though in implementation it was limited somewhat by the context of running within a web browser. The system provided no state other than the standard back/forward buttons of the web browser, so a user could not establish a number of categorizations and then revisit them in the future. Further, this system was constrained to searching the Internet (though the technology has been licensed to a number of companies for use on their web sites) rather than a user's local file space. It

would be interesting to see a version of this system designed for use on a user's local file system.

Finally, the system assumed that the user would view the document in their web browser, and doesn't support the standard operations of a file system browser. A naïve implementation on a local file system would still require the user to use a separate application to access their files once she located them with this system.

2.4 Haystack Information Retrieval System

This section will describe the Haystack Information Retrieval System, developed at the MIT Laboratory for Computer Science under Professors David Karger and Lynn Andrea Stein. This section will first look at the goals of the project, and then look at the system at a high level. We will examine both the design of the service model and the data model, and then close by looking at the current status of the Haystack project.

2.4.1 Project Goals

The purpose of the Haystack project is to provide a personalized information retrieval system that can help users to find pieces of information from within their information space quickly and easily. In the context of the Haystack project, a user's information space includes the documents stored on her hard drive, her email, any web pages she has visited, and any other electronic information that the system can get its hands on. The main goal of the project is to let the computer help users to find their information, rather than force the users to manually search through the veritable "Haystack" of information

for the "Needle" that they want. Further, the vision of this project is that a user's documents are all interconnected in different ways. The Haystack system seeks to represent and exploit these interconnections to allow the user to navigate through their information space more easily and more effectively.

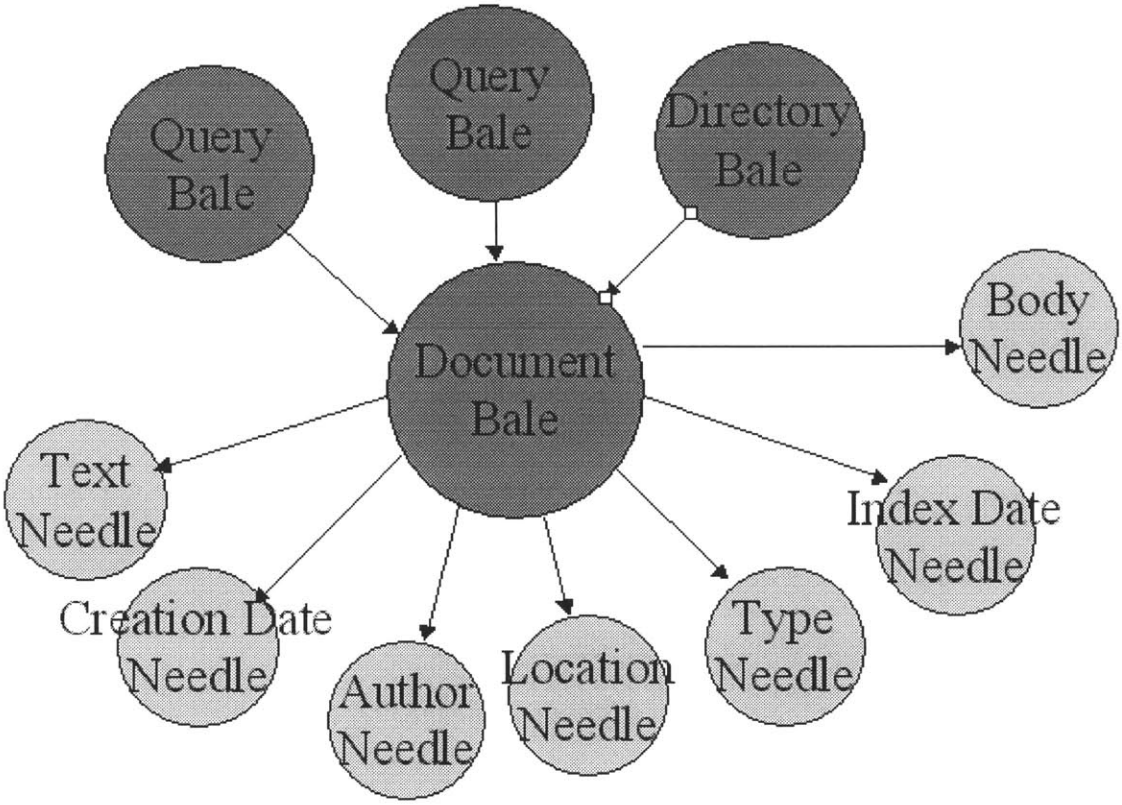


Figure 3. Typical Haystack subgraph

2.4.2 System Overview

The Haystack system is divided into three major sections: the data model, the service model, and the user interface.

The data model of Haystack is a collection of Straws (objects) connected together by Ties (links). There are a number of subtypes of Straws, most notable among them Needles (wrappers around pieces of data, like an integer or a piece of text), Bales (collections of Straws), and even Ties themselves. The paradigm we've created here is that a document is represented by a Bale, which contains Ties connecting the Bale to a Needle for each property of a document. A typical document Bale might contain Ties pointing to Needles for author, date created, location on the file system or the web, file type, creator and text representation of the document.

The service model of Haystack is a collection of services that create, examine, and maintain these data structures. There are services for querying the data structure, archiving new documents in the Haystack structure, finding a textual representation of a document, extracting metadata information like author and creation time from a document, and a variety of other tasks. The Haystack system is decentralized and largely event-driven, so these services watch for opportunities to act and then perform their function, rather than being driven by some higher-level manager system.

The third part of Haystack, the user interface, is the focus of the remainder of this thesis.

2.4.3 Haystack Data Structure

The Haystack data structure is a large web-like structure of Straws and Ties. Straws that represent pieces of information are connected together by Ties that represent context. This structure that associates data elements together also provides an excellent tool for browsing the data structure.

Each Tie represents a relationship between two pieces of data, and following a link corresponds to asking a question about the origin node. By focusing the user's attention on a single Straw and then allowing her to traverse these links, she can ask a number of questions, as shown in Table 1.

Author	Who created this document?
Date	When was it created?
Type	What type is this Straw?
Body	What is the content of this Straw?
Casual	What caused this Straw to be created?
Contains	What does this Straw contain?
ExtractedFrom	What was this Straw extracted from?
Hyperlink	What does this Straw contain a hyperlink to?
MatchesQuery	What Straws match this query?

Table 1 . Tie Links

Figure 3 shows a typical subgraph within the Haystack data structure. The forward links from the document Bale include the author of the document, creation date, and type Ties. The back links from the document lead to the directory this document is in and also to two queries that this document matches.

2.4.4 Haystack Services

The Haystack Service Model [Adar98] is a decentralized object-oriented system in which a number of concurrent services interact in an event-driven manner. These services register their interests with the Haystack dispatcher, which informs and activates them when relevant data is placed into the system. For example, the "textifier" service (a service that extracts a textual representation of any document) might tell the dispatcher that it is interested in any document Bales that contain a body but not a text representation of that body. The tar-extractor service might be interested in document Bales that have a Tie to a tar archive but not an expanded version of that tar file. These interests are represented by "star graphs", or particular graphs consisting of a node and its neighbors that satisfy some given constraints. The Haystack dispatcher uses a regular expression-style pattern matcher to match these interests with subgraphs within the Haystack data web.

The Haystack Service Model is based on the architecture of the CORBA [CORB99] object-oriented system. Services are objects that provide functionality from behind a clearly-defined and understandable interface. Most services run within the main Haystack program, the Haystack Root Server, though communication services allow Haystack services to run on different Java virtual machines or even on different computers.

A number of commonly used Haystack services are listed below in Table 2.

Service	Function
HsGUI	Provides the graphical user interface
HsArchive	Archives documents into the Haystack repository
HsIndex	Indexes text representation of a document into an IR system.
HsQuery	Mediates queries between UI components and IR system
HsTypeGuesser	Determines the type of a document
HsPostscript	One of a number of "textifier" services, this service extracts a text representation of a PostScript file.
HsLaTeXFieldFinder	One of a number of "field finder" services, this service looks within a LaTeX document and extracts as much metadata (Title, Author, Date, etc.) as possible for use in constructing Ties from this Haystack object to other objects
HsTar	One of a number of "extractor" services, this service looks within a "tar" archive file and examines all the documents stored within it.

Table 2 . Some Haystack Services

The Haystack Service Model provides a framework in which a number of different services can cooperate in order to act on behalf of the user. By working in a decentralized framework, we can introduce new services easily without needing to modify any kind of central control structure, so the system is highly extensible.

2.4.5 User Interface

Prior to the development of this thesis, there were two primary user interfaces to the Haystack system. There was a command-line interface, where users can issue query and archive commands and the system will return references to the Haystack objects that are found or created as a result. However, the interface requires that users manually follow the links. An example of this interface is shown in Figure 4. This interface was designed primarily for debugging purposes, but it serves as a basis for comparison. Clearly, we can do better.

The other interface to the Haystack system was the web server interface. The user could start a web server within Haystack, and then connect to it via any web browser. From within this interface, users could query and archive, and this time they could simply click on the representations of Ties to follow links.

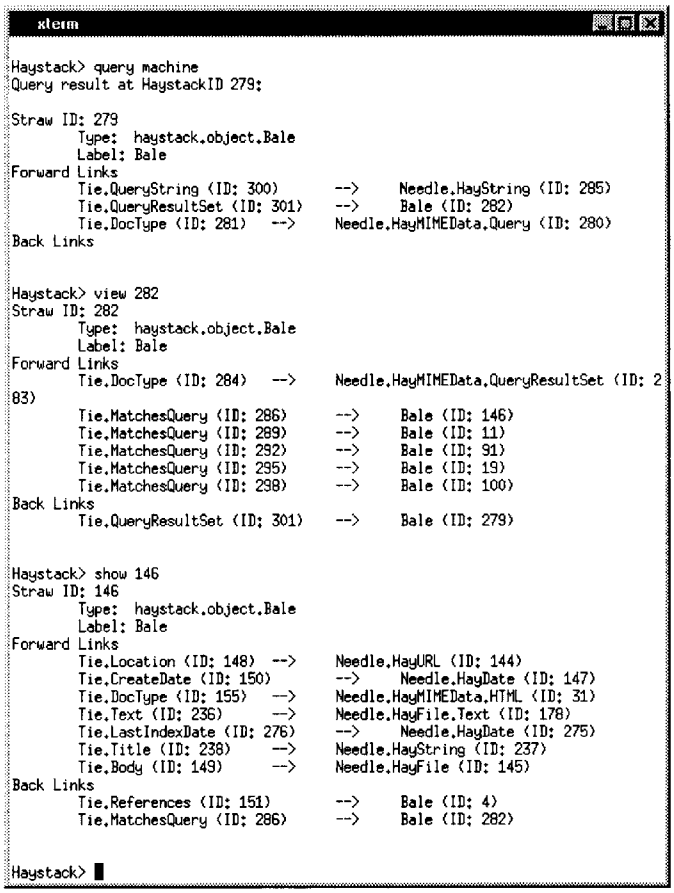


Figure 4. Haystack Command-Line Interface

The web server interface provided templates that allowed users to specify how particular types of documents could be viewed. A postscript template might display the text of the postscript document with key elements highlighted. Metadata located by extractors and field finders might be boldfaced to show the user which elements of the document were

noticed by Haystack as important. By providing the user with this template system, the Haystack system allowed her to view all of her Haystack documents from within a single application, but still allowed customization of the display process.

One possible avenue of future work on the Haystack Browser is allowing it to be run as an *applet* within a web browser, allowing a user to use the graphical interface from any web browser connected to the World-Wide Web. (see Chapter 7, Future Work)

The web interface is much improved from the command-line interface, though the user is still presented with a view of browsing World-Wide Web-type graph structures rather than the view presented by a traditional file browser. This thesis proposes a graphical user interface in the style of traditional graphical file browsers that tries to be as simple and intuitive as possible, while at the same time allowing users maximum functionality with minimum effort.

2.4.6 Project Status

The Haystack project is currently fully implemented in its second incarnation and currently addressing scalability issues. The system is implemented in Java, and as such is reasonably platform independent, though the chief platforms we support are Solaris and Linux. Microsoft Windows support is currently in progress, and other flavors of Unix are relatively straightforward to port to.

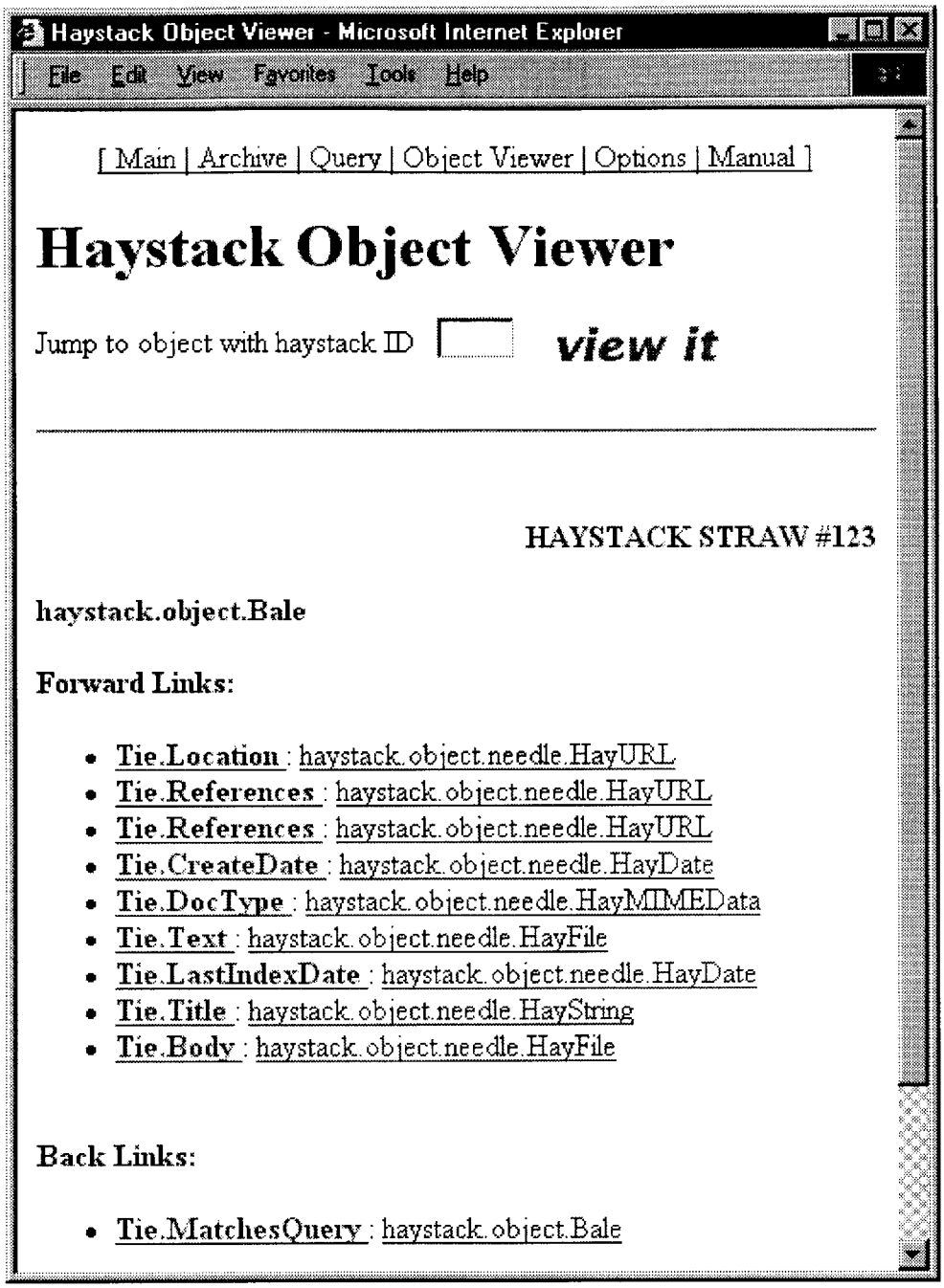


Figure 5. Haystack Web Interface

The current system allows users to archive any file, and we can currently extract textual representations of a variety of types of files, including ASCII text, LaTeX, PostScript, tar-compressed files, and HTML. The graphical interface is complete, and can be used to query against the document repository, archive new documents, or navigate through the Haystack data structures.

The current limitations of the system fall into two chief categories: scalability and reliability. We are currently working on installing a database back-end to the system, as well as implementing a logging system to allow us to recover from crashes of the system and pick up where we left off. These changes should overcome these limitations and make the Haystack system an extremely useful addition to any user's desktop system.

Chapter 3.

Haystack Browser

This section will look at the Haystack Browser, the new graphical interface to the Haystack system. This component allows the user to issue queries, follow links through the Haystack data structures, and open her documents in applications.

3.1 Overview

The heart of this graphical interface is the Haystack Browser. Intended to serve in a similar function to Microsoft Windows' Explorer or the Apple Macintosh's Finder, the Haystack Browser can serve as the day-to-day browser a user uses to find their files and interact with them. This tool also allows users to navigate their archived Haystack documents, following Ties between related data objects. The interface changes to provide appropriate controls and functionalities based on what type of Haystack object is being displayed.

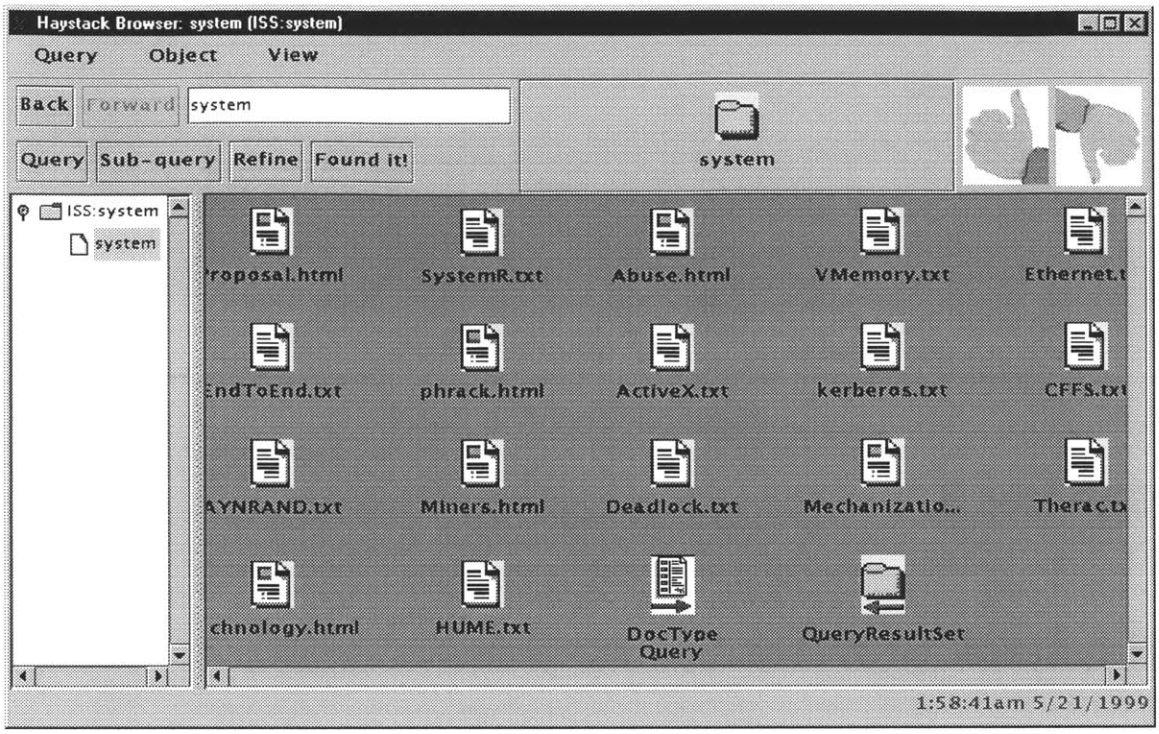


Figure 6. Haystack Browser

The Haystack Browser is essentially a window pane focused on a single Haystack object, filled with icons that represent the Ties connecting to that object. Most Ties are represented by icons that represent the type and direction of the Tie. Ties that point to document Bales and to query result Bales are represented differently. Ties that point to Document Bales are represented by document icons that show the type of the document, and Ties that point to query result set Bales are represented by folder icons. An icon at the top of the screen represents the Haystack object on which the browser window is currently focused.

Icons for documents and query result sets are special in that the interface allows an additional set of actions on them, in addition to the normal actions the user can perform on all kinds of Straws.

3.2 Design Goals

There are a number of design goals for the Haystack Browser, all centered around allowing the user to use this interface as her primary tool to navigate her information space.

The central metaphor of this interface is the combination of the file/folder paradigm with a query system. Just as traditional folders contain documents that have been placed within the category that corresponds to that folder, in this interface these folders contain the documents that match the query that corresponds to this folder. This metaphor actually lends itself very well to this search-based model, where a user creates a new folder for a new query and then opens it to examine the documents that fit within the category defined by that query.

The Haystack Browser should be intuitive, and match up as much as possible with the look and feel of other similar interfaces, such as the Windows Explorer or the Macintosh Finder. We want to emulate these other interfaces for three primary reasons. First, this file/folder metaphor is familiar to modern users, and by using it we can ease their transition to using our system, reducing the amount of time needed to learn how to use our system. Second, we need to provide the same functionality as these conventional file browsers if users are going to use the Haystack browser as their day-to-day file browser. Third, these interfaces represent a well-explored implementation of a document-centric interface, and we can leverage this prior design work rather than try to reinvent the interface ourselves.

The Haystack Browser should allow the user to determine which elements of a set of icons are relevant to their information need without needing to follow every link individually. Our system won't be particularly useful if it degenerates into a linear search again, so we need to make sure that from a window of icons, the user can understand what each icon represents as quickly as possible. To that end, we should clearly mark icons by type as well as extend them with information about what is at "the other end" of the link. These markings will clearly need to be different for different Haystack object types, but all icons should be marked so that the user knows what they represent.

Finally, the Haystack Browser should allow the user to create a categorization hierarchy and then store it for use later. Just as a static hierarchy is useful for users to organize their information and then remember where it was stored, it can be useful within the context of a dynamic hierarchy as well. We need to make sure that in moving to a dynamic hierarchy we do not lose this power of the static hierarchy. For this reason, we want to allow the user to carefully craft a hierarchy of categories and store this hierarchy so that she can access it again at a later time.

3.3 Navigating the Haystack data structure

The first major function of the Haystack Browser is to provide the user with a control that allows them to navigate through the graph-based Haystack data structures quickly and easily.

The Haystack Browser window represents a focus on one particular object within the Haystack structure. The window pane holds a collection of icons which represent Ties connecting this object to other objects in the Haystack data structure. The user can select any these icons and execute different operations on it.

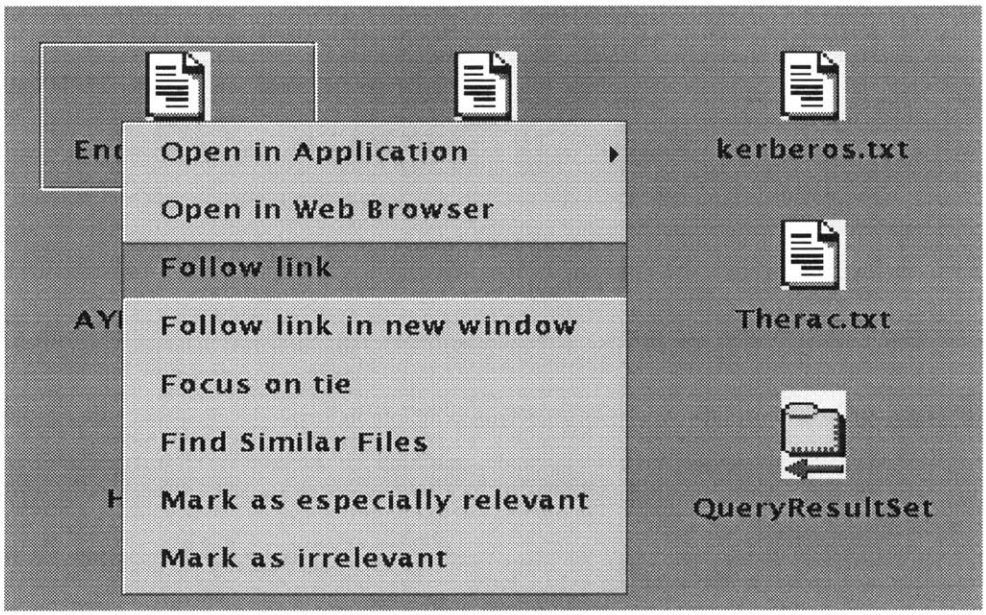


Figure 7. Right-click on icon

3.3.1 Follow Link

The user can select any of the links connected to this object and "follow" that link, moving the focus of the window to another object in the data structure.

In the Haystack Browser icon pane, each icon represents a Tie connected to this object. Each Tie is labeled with an icon symbolizing the Tie type, as well as an extremely brief

textual representation of the data at the other end of the Tie. The text representation of objects depends heavily on the type of the Tie and object to which the Tie points, and clearly some Ties (author) will be easier to summarize than others (body). Holding the mouse cursor over any one of the Ties will display the Tie's full name and Haystack ID.

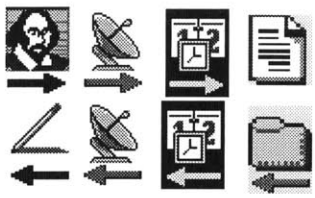


Figure 8. Tie Icon Examples

Each Tie icon is either a "forward icon" (representing a Tie from the ForwardLinks array) or a "backward icon", (representing a Tie from the BackLinks array) and also uses a graphical icon to represent the meaning of this Tie. Note that the graphic associated with a Tie may not be the same in each direction, as the meaning of the Tie has changed. For example, the Author tie on the left of Figure 8 is represented by a picture of William Shakespeare when the link is forward, because the Tie here means "X was written by Y." However, the graphic for a backward Author tie is a pen, because the Tie here means "X wrote Y". On the right side of Figure 8, the forward link is represented by a document, meaning "X is a query that document Y matches", though the backward link is represented by a folder, meaning "X was matched by query Y".

Double-clicking on any of these icons will cause the Haystack Browser to move to focus on the object on the other side of that Tie. The user can also right-click on these Ties and

select an option that opens a new Haystack Browser window that focuses on that object, rather than changing the focus of the current window. Finally, because the Ties are themselves Haystack objects, the user can right-click on a Tie and move the focus of the window to the Tie itself, rather than the Straw to which it points. This allows the user to examine her Ties for information about their history and to look at any annotations which may have been attached to them.

One of the greatest powers of an icon-based paradigm is the easy, intuitive method of local graph navigation that it provides. At every step, the user is presented with all the possible links that she can follow, and it is simple and obvious how to follow those links to a new node. Further, traversing any one of these links corresponds to asking a question. The topic of the question is the type of the link, the subject is the start of the link, and the answer is the end of the link.

One goal of the graphical interface for Haystack is to combine the power of icon-based graph navigation with the questions that can be formed with these local links, allowing users to easily navigate through Haystack's data structures and find answers to their questions.

By following these links, the user is effectively asking Tie-specific questions about the document.

- Who created this document?
- When was it created?
- What kind of document is it?
- What documents were created on last Thursday?

By following multiple links in order, the user is effectively asking questions about the previous answer, and can thus obtain the answer to highly specific questions.

- Who created this document?
- What other documents did that person create?
- Which authors did this person collaborate with on those documents?

By using a sequence of these simple *local* questions, the user can ask very complicated questions, such as "who was the co-author on the other paper that this person wrote?" Clearly, we can give users a great deal of power to ask questions about their Haystack simply by providing a simple, intuitive, and convenient method of navigating the links within their Haystack data structures.

3.3.2 Open in Application

Another basic operation that one can enact on a document is the one typified by the document-centric user interface: opening in the document in some kind of application. The system allows users to associate applications with given document types, and by

double-clicking on a document icon, the document is opened within that application. This allows users to quickly open their document, look at a graphic, or browse an HTML page in the most familiar application to them. Further, this gives the Haystack Browser sufficient functionality that it can (ideally) act as the user's primary method for getting at their documents, since it now has the power to start up the applications that the user most commonly uses.

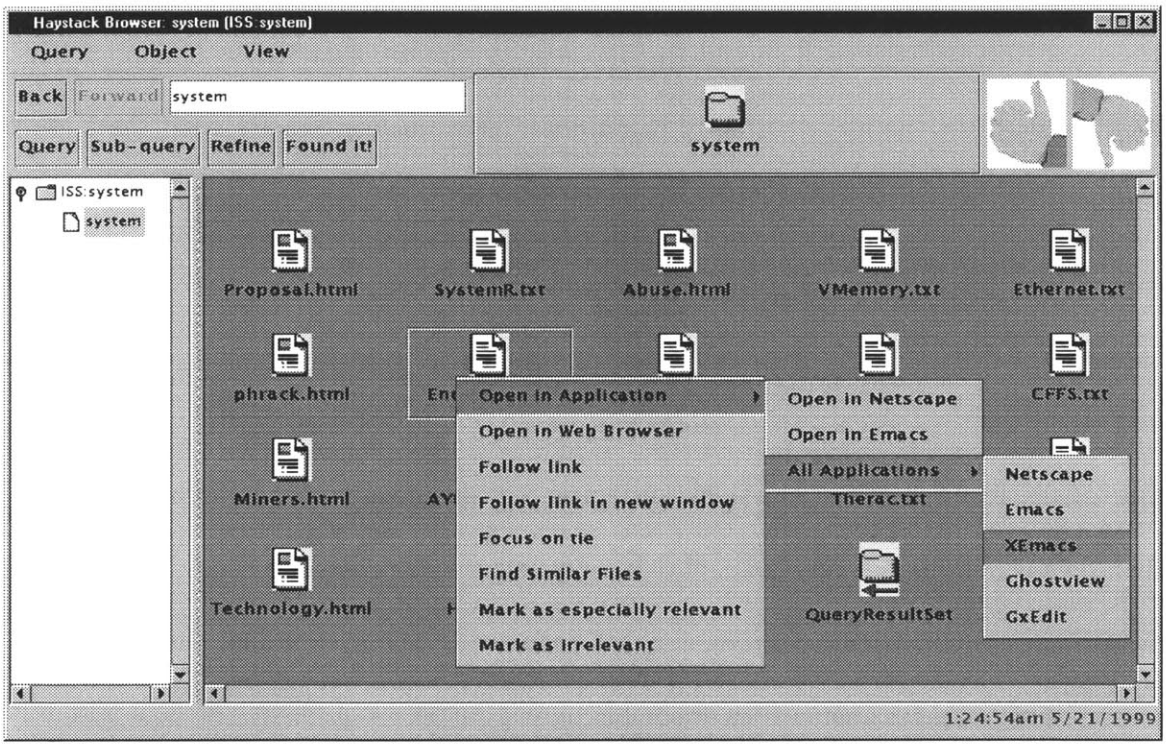


Figure 9. Open in Application

This operation can also be enacted by right-clicking on the document, which provides the option to open the document in the primary application associated with this document. This also provides a cascading list of other secondary applications associated with this document, as well as a list of all the applications the Haystack system knows about. This allows a user to use a web browser to view their HTML files most of the time, but also

makes it easy to open the page in their favorite editor as well. Occasionally, one will want to open a document with an application that is almost never associated with that document, and so we also provide the ability to use any of the applications that Haystack knows about in case the user wants to do something out of the ordinary.

While typical interfaces allow only documents to be opened within applications, the Haystack Browser is general enough that it can open arbitrary data Straws in applications, though this functionality is currently incomplete. It is unclear exactly what the correct semantics should be for opening non-document Straws in an application, and this is another possible direction that future work on the Haystack Browser could take. (see Chapter 7, Future Work)

One of the features of the Haystack system is that documents can be archived even if their data is not immediately available. If a document is expensive to access or store, the system can instead store a *promise* for that document. The system can later *fulfill* that promise and access the data of the document only if the user requires it. To deal with this promise functionality, if a user uses the Haystack Browser to open a document that is stored as a promise, a dialog box will appear and ask the user for confirm that she wishes to fulfill this promise. If the user chooses to fulfill the promise, the document will be opened in the appropriate application like standard documents.

It may seem that in the name of interface transparency, the Browser should hide the promise from the user and simply fulfill it on demand, but this is not quite correct.

Promises exist because the data that they represent is non-trivial to acquire. This may represent a fee-based retrieval system, or perhaps extracting a file from a large archive. In these cases, access to the document may be limited. For example, it may not be possible to open a document from within an archive, edit it, and save it as one can do with normal documents. The user must be aware of the "promised" nature of the Straw so that she can make an informed decision about whether to fulfill that promise, and so that she can understand the limitations of access to this document.

3.3.3 Open in Web Browser

The user can select any Straw in the Haystack system and use this option to open the Haystack web interface to view this object. By allowing the Haystack Browser to open Haystack documents within this web server, we can leverage all the work that went into this template-based display mechanism. We thus give the user a highly developed means of displaying all of her different Haystack Straws in a consistent application, while still having the power to customize the presentation of that matter.

3.3.4 Notification of new matches

Because the system is constantly adding more information to its objects via new Ties, a Haystack object that is the focus of a Haystack Browser window may suddenly be connected with additional Ties.

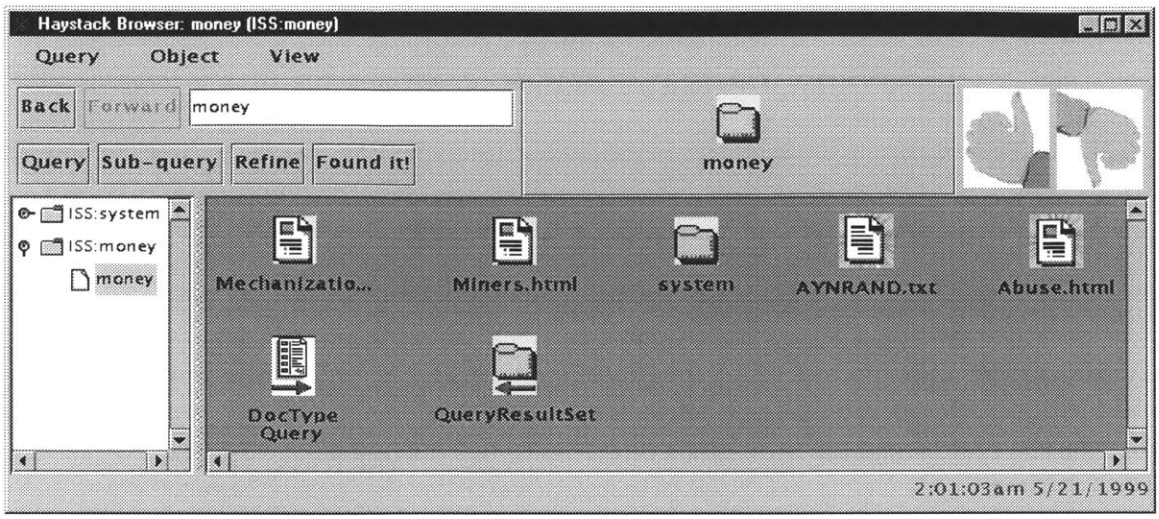


Figure 10. Example of blue "glow"

When this happens, we immediately add the icons for the new Ties to the already open window, marking the icons with a blue "glow" to indicate to the user that they are new items. This glow will go away when the user accesses those icons for the first time, since this demonstrates that she has noticed the presence of these icons. The glow will also go away when the window is closed, as the purpose of the glow is to alert the user that the set of Ties to this object has changed since she opened the document.

3.4 Display of Query Results

The second primary function of the Haystack Browser is to display the results of a query. Queries can be matched by two kinds of Haystack objects at present: previous queries and document Bales. This section will look at the special operations on these types of Straws that the Browser provides and the functionality in the Browser that allows new queries to be issued.

3.4.1 Find Similar Documents

While queries are usually presented to the system simply as a sequence of terms that represents the user's information need, a query can also be specified as "find me documents like this one." The Query Navigator allows the user to select a document Bale in the result set and choose the "Find Similar Documents" option from the Document menu. This will compare the document to other documents in the Haystack archive, and determine the terms to use for a query that best represents this document. The system will then look for other documents that match this query, and return to the user a set of documents that are, in some sense, similar to the original document (see Chapter 4. Query Refinement). The system records this information for future reference, adding a SimilarTo Tie between the original document and each of the similar documents. The next time the user looks at this document, she will be able to follow the SimilarTo Ties to find the similar documents quickly and easily.

3.4.2 Additional Queries

The user can also issue additional queries from within the Haystack Browser. These can be additional queries that the user thinks may be a better approximation to her information need or a query against the results of this query rather than against the entire Haystack repository.

New Queries

New queries can be issued in a number of ways. The user can enter the query into the Haystack Root Window (see Chapter 6, Haystack Root Window), which will create a

new Haystack Browser window focusing on the result set of that query. The user can also type the query within the query text field of the Haystack Browser and click "Query" or the user can use the menu option "Query/New Query" and type the query into the resultant dialog box.

Sub-Queries

While users will often want to search their entire Haystack repository for documents, when trying to narrow a result set to a manageable number of documents, a user may wish to issue a query only against the objects returned by a particular query. To that end, we provide the ability to issue a query against the result set of another query, so that the only objects returned are the objects from the result set of the earlier query that also match the new query.

Sub-queries can be issued by typing text into the query textfield and clicking "Sub Query" or by selecting the menu option "Query/Query Within Results" and entering the query into the resultant dialog box. The query that results from this will be displayed in the history tree as a child of the query whose result set it was against.

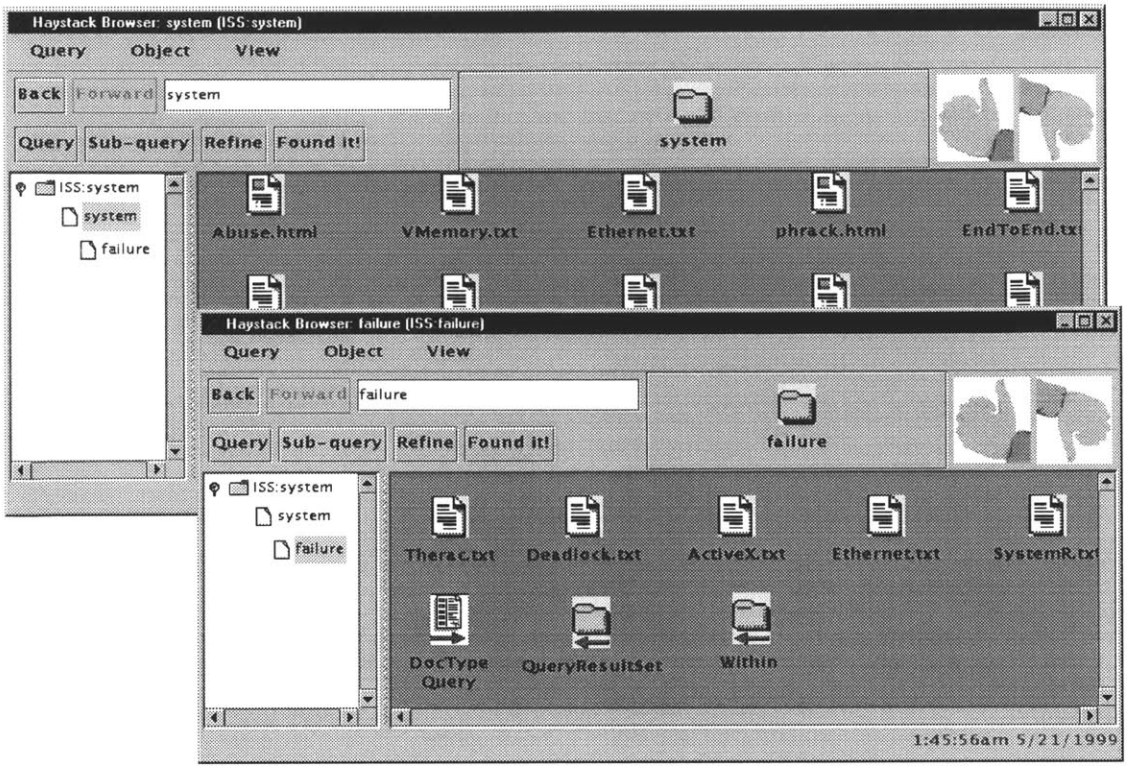


Figure 11. Sub-Query

Extending the Haystack query service, we have added the "Query Within" function, which allows a user to search through a subgraph of the Haystack data structures. The user can use the standard query to search all documents for those that match query A. By using the "Query Within" function, the user can then search all the documents that match query A for the ones that also match query B. A typical session might be as shown in Table 3.

"Haystack, show me all documents from my ancient history class."

haystack> query roman greek history → returned query bale 143
Haystack> view 143
/home/papers/rome.txt
/home/papers/romedraft1.txt
/home/papers/trojanwar.txt
/home/papers/mythology.dvi
/home/papers/odyssey.tex
/home/papers/thracians.dvi
/home/papers/termpaper.tex

"Haystack, show me the documents from my ancient history class that are about Thrace."

haystack> query thrace -within 143 → returned query bale 175
haystack> view 175
/home/papers/thracians.dvi
/home/papers/termpaper.tex

Table 3. Abstract Query Refinement Example

One of the goals of Haystack is that every query that is issued becomes a first-class object that is itself retrievable later. However, this query within another set of documents does not currently correspond to any actual particular query object within the Haystack system.

To allow us to represent these "query within" objects, we extended the Haystack system to create a Within Tie. We create the new query Bale with the items that match both queries, and then attach a Within Tie from the new query back to the first query, to indicate that the second query is not against the entire corpus of the Haystack system, but only against this subpart.

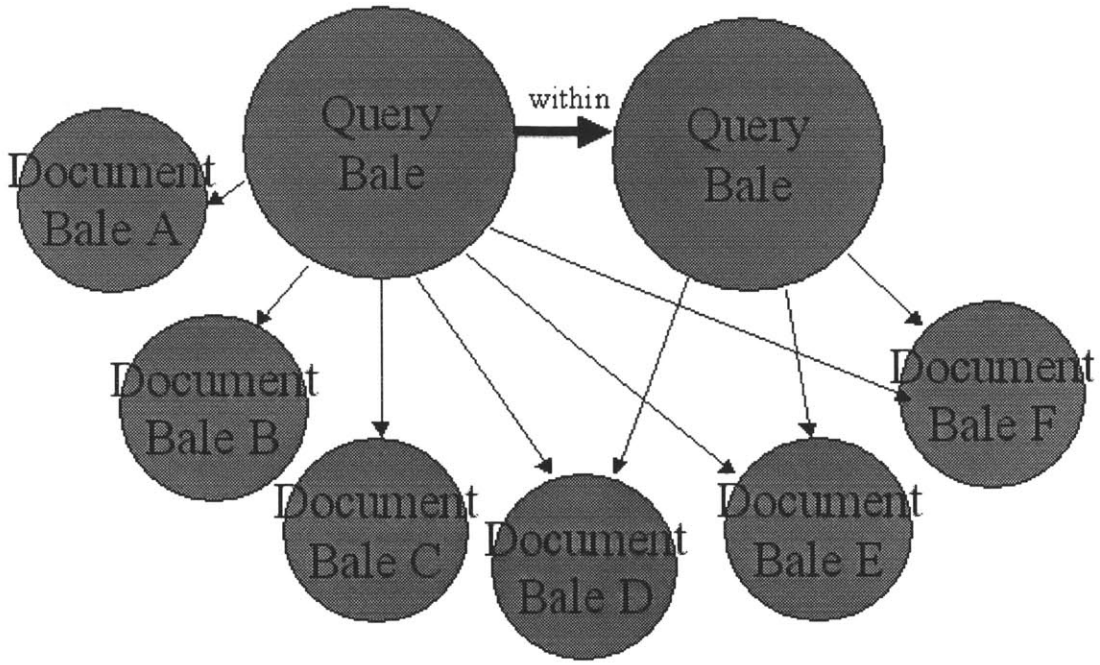


Figure 12. Haystack Subgraph formed by Query Within

The result set of the new query Bale is computed by taking both queries and then computing the intersection of the two. Since the result set for the original query has already been computed, the system hashes each document in that set, and then issues the second query. Each element of the result set of the second query is hashed into the same table, and only if it finds a match is it stored as a result to this query. A new query result set Bale is created, the matching elements are attached to this Bale, and a Within Tie is attached from the new query Bale to the query Bale of the first query. Note that we do not create a query result Bale for the second query alone, only for the intersection of its results with those of the first query.

Query Updating

The Haystack system normally stores every issued query as a new object (query within being one of a handful of exceptions). A query Bale collects the string representation of the query along with all the documents that match the query. For purposes of indexing, the Bale is represented as a combination of the features of all the documents that matched the query, so future queries may well match against this query and return it as a result. In this way, a user looking for information may be reminded of a previous query she issued, perhaps a query she was trying to recreate but could not remember exactly.

One limitation of this system, however, is that it creates a new query Bale every time a query issued. If a user frequently queried the system for documents about NATO, for example, then a new query object would be created every time that query was issued. While one could imagine cases in which this would be useful, in general this leads to a contamination of the Haystack data structures with multiple copies of what should be the same object, the Haystack representation of the query "NATO".

To overcome this limitation, we extend the Haystack query service with the following construct: when the user issues a query, we first look up that query to see if it has been issued before. If so, we return to the user a reference to the existing query Bale. If not, then we issue the query, store the result so that we can find it next time, and then return a reference to the query to the user. This provides a *unique query,* meaning that a particular text query will be associated with a single unique query Bale within the Haystack system, rather than a number of queries that correspond to the query being

issued multiple times. This unique query system is built on the unique Needle service implemented in Haystack by Damon Mosk-Aoyama.

The unique query subsystem overcomes the problem of contaminating the Haystack data with duplicates of the same query, but if we do nothing else, the query will be forever frozen as at the time it was first issued. To fix this problem, we introduce another service, one that revisits queries and checks to see if any new documents match the query. This service operates in the background, going around to all of the query result Bales in the Haystack and refreshing them one by one by reissuing the query and adding any new documents that match.

When a user looks at an old query, she is presented with the latest information that the service has provided, along with information about when that result set was last updated. The user can also explicitly ask Haystack to immediately refresh the query, directing this updating service to update the relevant query Bale with the latest matching documents. We refresh most often the Bales that have been recently referenced, and update much less often Bales that haven't been visited in a while.

One issue that arises in a dynamic categorization system like Haystack that is not a problem in static hierarchies is the problem of consistency. While in a static system it is relatively quick to determine which documents are in a particular directory, it requires a certain amount of computation to determine which documents in the system match a particular query. As a result, every query result set is time stamped with the last time that the query was updated. When a user opens a query, therefore, the result set shown may

be out of date, and might not correspond to the actual documents in the corpus. However, the user can always click the "Refresh" button to direct the query update service to update the current query immediately. By displaying the time of the last update in the window, the user can decide whether the results are recent enough, or if a query update is necessary. By not forcing a query update each time the user opens a query, we can allow the user to quickly navigate her information space, spending the time to reissue a query only when she deems it necessary.

Chapter 4.

Query Refinement

This section will discuss the process of query refinement and the mechanisms provided within the Haystack Browser to allow the user to refine her query easily and efficiently.

4.1 Overview

In an ideal world, a user would know the right query to use for any desired piece of information. However, this is rarely the case. Searching for information is almost always an iterative process: the user issues a query, looks at the results, and then adjusts the query to try to get a better approximation to what she is looking for.

In most systems this adjustment must be done completely by the user. The user looks at the results, sees which terms show up that she is interested in, sees which terms show up that she is not interested in, figures out how to adjust the terms of her query to improve the result set, and then types in the new query. The process continues until the user finally gets the query she is looking for. Table 4 shows a typical query refinement

process, executed on the Alta Vista search engine, and Table 5 shows an abstract query refinement process that a user might want to do with Haystack .

Query	Results	Topics in top ten	Top result
Haystack	25320	Ski resorts, radio astronomy, Haystack project	Radio astronomy
Haystack LCS	39050	Ski resorts, radio astronomy, Haystack project	Radio astronomy
Haystack LCS -snow -radio	15569	Haystack project, summer programs	Professor Karger's homepage
+Haystack +LCS -snow -radio	19	Haystack project, Haystack database (unrelated), MIT offices	Professor Karger's homepage

Table 4. Typical manual query refinement

"Haystack, show me documents about failure"

haystack> query failure → returned query bale 121

Haystack> view 121

/home/papers/apollo13.tex

/home/papers/therac.txt

/home/papers/systems.tex

"Haystack, I'm not interested in Apollo 13, but I am interested in the paper I wrote on hackers. That said, show me the documents that I am interested in."

haystack> exclude 121 /home/papers/apollo13.tex

haystack> include 121 /home/papers/phrack.txt

haystack> refine 121 → returned query bale 145

haystack> view 145

/home/papers/therac.txt

/home/papers/systems.tex

/home/papers/phrack.txt

/home/papers/mitnick.tex

/home/papers/internetworm.dvi

Table 5. Abstract query refinement example

To automate this process as much as possible, the Haystack Browser provides a tool for automatic query refinement. After a user is given a set of documents that match a query,

she can mark some of those documents as not being what she wanted. She can also include other documents that were not returned by the initial query. She then instructs the Haystack system to refine the query, and it will automatically generate a query that includes the documents she marked as relevant, excludes the documents she marked as irrelevant, and finds other documents in accordance with these positive and negative examples.

4.2 Design Goals

The Haystack Browser should make it easy to modify and refine queries. The entire point of this system is to allow users to easily navigate their information space, and a critical part of that is to make it simple for users to improve their queries. We need to allow users to easily mark documents for inclusion or exclusion within a query, and then provide automatic functionality to do the actual query refinement. Whatever we can do to make this process easy to use and easy to learn will be a benefit to the user, so using standard metaphors like drag-and-drop will be very helpful.

4.3 Interface Implementation

This section will explain the different ways the user can use the graphical interface to refine her query. Fundamentally, the interface allows the user to mark some of the items in the query result set as irrelevant, mark some items as especially relevant, and mark other documents that were not returned as relevant also. The user can then instruct the system to compute the refined query.

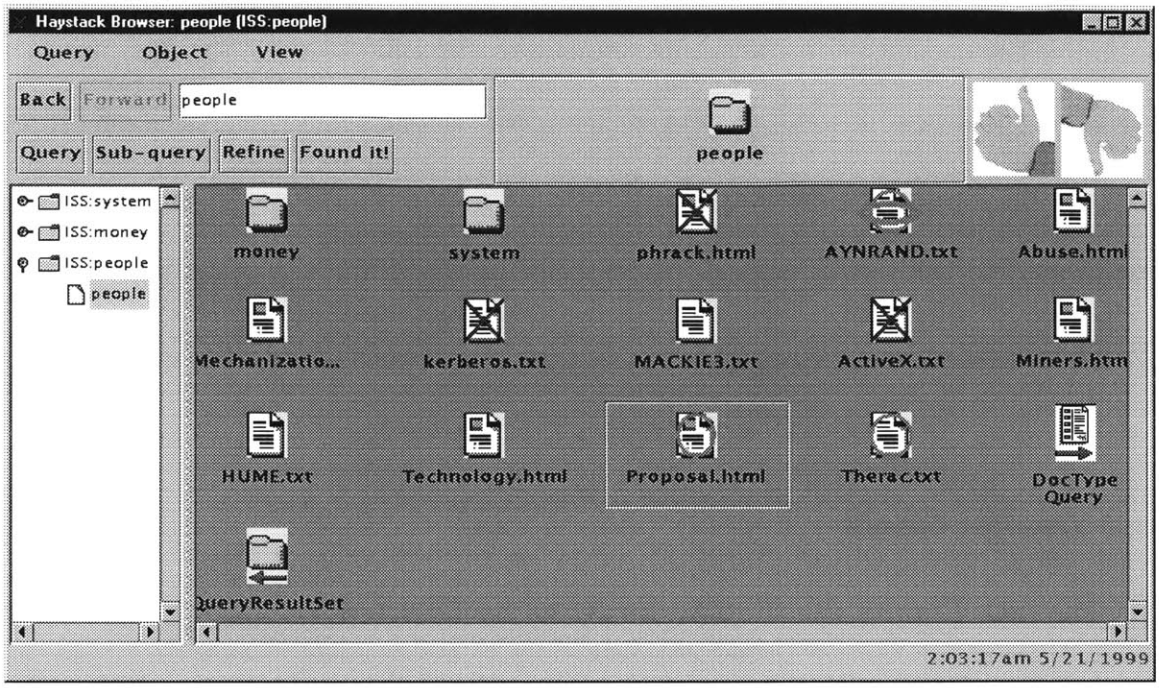


Figure 13. Query Refinement

The user has three different ways to mark documents in the result set that are not relevant to the information the she is looking for. These icons will be marked with a red X symbol, showing that they will be excluded from the refined query. She can select on the icon and select "Mark as irrelevant" or select the icon and then choose the "Mark as irrelevant' option from the Object menu. She can also click the "thumbs down" button, which will turn the mouse cursor into a small "thumbs down" symbol. Each document she clicks before unselecting the "thumbs down" button will be marked as irrelevant. As the user will typically mark many documents as irrelevant at the same time, this makes the operation slightly more convenient.

To add in documents that are relevant to the information search but that were not returned by the current query, the user can either drag the document icon into this window from another Haystack Browser window or by selecting "Mark additional relevant document" from the Object menu. This will add the selected Haystack objects to this query, marked as relevant. The icons for these documents will be marked with a green halo, signifying that they have been selected for inclusion.

The user can also mark document as being especially indicative of the types of documents that she is looking for. When the user right-clicks on a document and selects "Mark as especially relevant" the document is marked with a double green halo to indicate its importance to the query. The user can also click the "thumbs up" button to turn the mouse cursor into a thumbs-up, allowing her to mark the documents as especially relevant with a single click, just as in the "thumbs down" icon described earlier. Finally, the user can select the icon and choose "Mark as especially relevant" from the Object menu. In the current implementation of this thesis, the especially relevant documents are treated the same way as the documents added to this query, though in the future the refinement algorithm will use this information more constructively. (see Chapter 7, Future Work)

When the user selects the "Refine " button or "Refine Query" from the Query menu, the resultant query will be the refined version of current query, excluding the documents marked as irrelevant, but including the documents marked as relevant and other documents that fit this implicit pattern.

Note that this inclusion and exclusion are absolute: regardless of whether the documents marked for inclusion and exclusion would normally be returned by the refined query, these documents will always be returned or excluded, as appropriate. Further, these documents will continue be marked with this green halo when returned by the refined query, so that the user will know that these documents were returned because she asserted that they were relevant to the query, not because Haystack simply believed that they were.

By providing both the power to easily include and exclude documents from a query, the user has a far more powerful method of query refinement than the standard "query, examine results, manually figure out a better query, revise query terms, repeat" method of query refinement.

4.4 Refinement Implementation

After the user has marked additional documents as being relevant and other documents as irrelevant, the system has a collection of positive and negative examples of documents that the user believes to be indicative of the information she wants to find.

The Haystack system then automatically generates a corresponding query. It takes the set of documents returned by the initial query that made the cut, and adds in the documents that were added. It generates a vectorspace representation of all of those documents, by looking at the frequency of each term in each document, weighted by the number of times the term appears in all documents in the archive. This method is known as "term-

frequency, inverse document-frequency" (TFIDF) [Salt83] and is a standard algorithm in information retrieval. However, to take into account the documents marked as irrelevant, we must extend this scheme by subtracting off the representations of the documents that were discarded. To that end, we use Rocchio's Algorithm [Rocc71], shown in Figure 14.

$$Q_1 = Q_0 + \beta \sum_{i=1}^{n_1} \frac{R_i}{n_1} - \gamma \sum_{i=1}^{n_2} \frac{S_i}{n_1}$$

Figure 14. Rocchio's Algorithm

Finally, the top N terms of the final query are chosen and used as the basis for the new query, which is issued to the system. (N is currently set at 5, though this will likely change when as the system is developed and tuned) This system makes heavy use of the profiling system developed for Haystack by Ilya Lisansky, [Lisa99] which generates the term frequency vectors for each document and for the repository as a whole.

Effectively, what this does is allow the user to say: "No, those are not exactly the documents I want. I want documents like these ones here that you gave me, but I do not want documents like these ones over here that you also returned. Here, these are examples of the sorts of documents that I want, even though you did not find them." The system then goes off and tries to find documents that fit the pattern the user has implicitly described.

Because the system generates a new query based on the most common terms in the set of desired topics, the query is represented exactly as if the user had done all of this

processing in her head and then typed in the new query. For this reason, we do not need to make any changes to the way query results are stored, and do not need to record information like "this is the Bale that represents the results of query N after documents A1, A2, and A3 were added and documents B1 and B2 were taken away". The query that is sent to the Haystack system is just like any other query, so it can be stored in the same way and recalled in the same way. However, in keeping with the Haystack philosophy of never throwing away any information, we add a RefinedFrom Tie from the original query to this refined query indicating that the latter query was based on the former.

Note also that the act of finding similar documents is just a special case of query refinement. When a user wants to find other documents like a particular document, she can just through away all other documents and then refine the query. The Haystack graphical interface provides a shortcut to do this by selecting the document and then selecting the "Find Similar Documents" option, but this still uses the same underlying machinery as query refinement. We do not include the RefinedFrom Tie, however, when we use this mechanism to find similar documents.

4.5 Data Model Implementation

In order to support the absolute nature of inclusion and exclusion alluded to above, we need to store this information along with the query, so that we do not lose this information the next time the refined query is updated. To this end, we attach Ties between the new Query Result Bale and these documents. "Includes" Ties connect the

Query Result Bale with documents that will always be included, and "Excludes" Ties connect the Query Result Bale with documents that will always be excluded.

One aspect that this discussion of query refinement has failed to cover thus far is how this is implemented with respect to a Haystack data model that is based on immutable objects, depending on annotation for future changes. We don't only want to create a new refined query, but we want to actually change the original query so that the next time we open the query we will see the refined version.

The answer is that we annotate refined queries with RefinedTo Ties pointing to their refined counterparts. The next time the user visits that query, she is automatically redirected to the most refined form of that query available, though the user can still access the earlier queries if she wishes.

This is very similar in nature to the Query Signposting feature of the Haystack Browser, which will be discussed in Chapter 5.

Chapter 5.

Information Search Sessions and Query Signposting

This section will discuss two key technologies of the Haystack Browser that allow grouping of related queries and designation of a particular query or object as the desired result of this group of queries. Information Search Sessions provides the grouping mechanism, and Query Signposting provides the designation functionality.

5.1 Information Search Sessions

One of the primary ideas of the Haystack Browser is that of Information Search Sessions (ISS), distinct sequences of navigations and queries within the Haystack system designed to arrive at a particular goal. Every link transition and query that a user issues within the browser is considered to be within the same ISS, though a new ISS can be started with a menu option. By maintaining this connection between queries issued within the system, Haystack can monitor the user's progress and store information about the search history.

Because the system knows that a new query issued from a Haystack Browser window is associated with the same ISS as that window, it can construct a link between these two queries, since they are related. We add a CreatedFrom Tie to represent this connection. The next time a user accesses this query, she will be presented with a Tie pointing to the Haystack query that she created the last time she was here.

5.2 Query Signposting

Another key component of assisting the user with many queries over time is *query signposting.* A typical query session will originate with a user's desire to find information on a certain topic. The user might try a number of queries before finding the one that seems best suited to the information she needs. She might then use refinement (see Chapter 4. Query Refinement) to narrow the search down to what she was looking for in the first place.

The problem is that if the user tries to find this information again, she is highly likely to repeat this entire process. What we would like to do is to store the information of the user's final query, and somehow link that to the previous queries used in finding this data. By doing this, the next time the user issues that first query, she will be presented with a *signpost.* This signpost is a link from all the queries associated with the ISS to the final query that the user was looking for. This link indicates that when the user issued a particular query in the past, she really wanted another query. The user can follow the link, and can avoid repeating all the work that she did the first time.

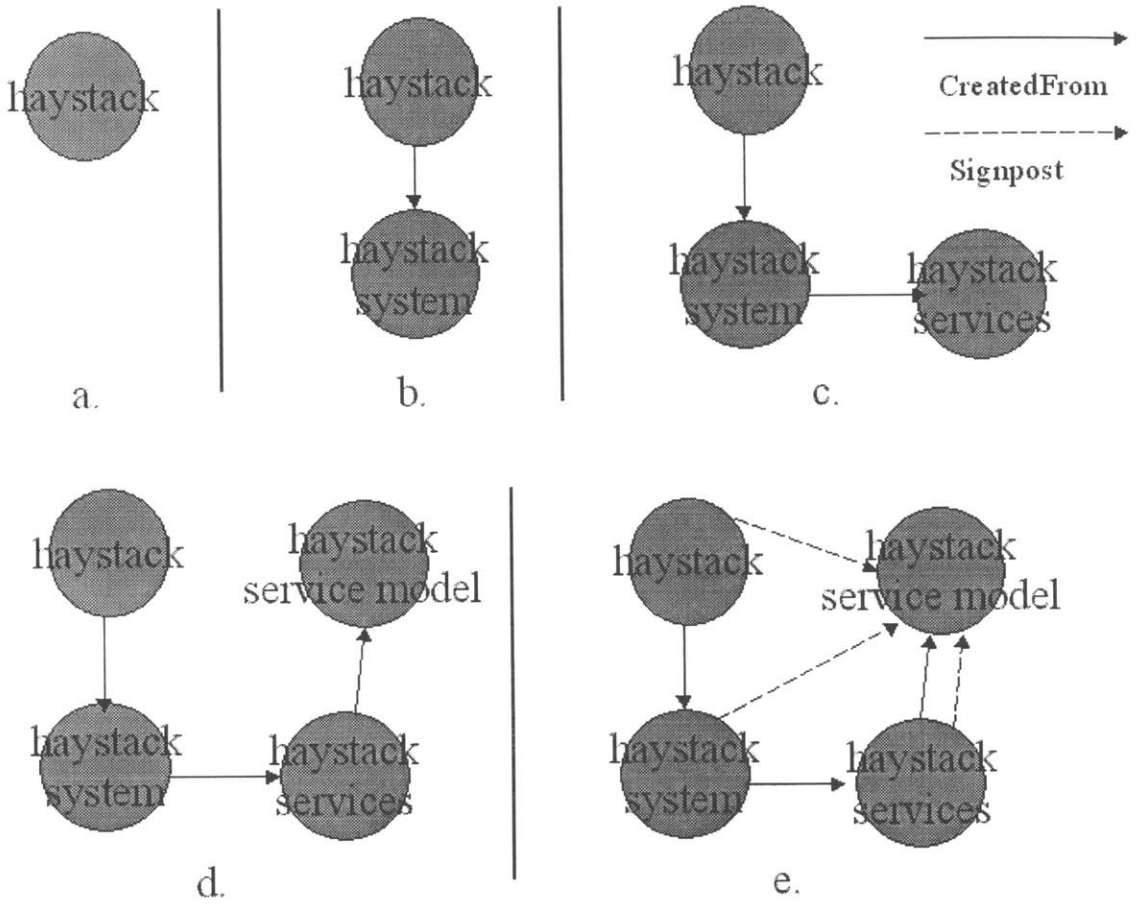


Figure 15. Signposting

(a. initial query. b-d. query modification. e. user signposts fourth query)

This requires a number of additions to the system. The first is a way for the system to know when a user is beginning a new query session. We implement this by making a distinction between new queries towards the same information goal and queries towards a new information goal. As the system cannot understand the queries of the user and know which queries are similar, we need user assistance. For this reason, we have a "New Information Search" button in the Haystack Browser, distinct from "New Window" which opens another window in the same query session. The sidebar that shows the query history also organizes visited queries by their ISS. When the user clicks the "New

Information Search" button, a new ISS is created, and all subsequent operations in the new Haystack Browser window are associated with that ISS.

The second addition to the system is a way for the system to know when a user has found the information that she is looking for. We might try to do this by looking at the length of time the user looks at the window, or the number of documents in this window she looks at, or if she closes Haystack after finding this query, or any of a number of other heuristics. However, this signposting will not be useful to users unless it is reliable, and for that reason we again require user assistance. For this reason, we ask the user to notify the system when she has found the query that she was looking for.

We provide a "Found it!" button in the Haystack Browser, which indicates that all queries visited previously in this ISS should be signposted to point at the object the window current focuses on. The user will, if she chooses, click this button once she has found the object that she was looking for. Because this is an "opt-in" system, if the user chooses never to click the "Found it!" button, then no signposts will be created, and the user can easily ignore this functionality. By designing the system so that users can choose to use it or not, we can ensure that it does not clutter up the system if users do not wish to use it.

Note that the signposting functionality is not restricted to pointing at a query result set, but can point to any Haystack object. If a user was looking for a particular document, she can focus the Haystack Browser on that document and click "Found it!" to designate that Haystack object as the intended target of all the queries in this ISS.

5.3 Implementation Details

This section will examine a number of implementation details that are affected by the inclusion of the ISS/query signposting functionality.

5.3.1 History Tree

The Haystack Browser needs to make sure that the user can navigate not only throughout her information space, but also through the history of her recent query sessions. The interface should keep track of where the user has been recently, and allow her to easily backtrack and try another path. In addition, this interface should not prune paths when a different fork is taken, as traditional Back/Forward systems do, but should instead store all recently visited Haystack objects, since this has been shown to be the optimal way to display history information for the purposes of assisting user navigation [Taus97]. The Haystack Browser should allow users to make mistakes and change their minds without forcing them to redo work. Whenever possible, we should avoid making the user duplicate previous work manipulating the user interface, this manipulation already much more taxing on the user than it should be.

The tree to the left of the icon pane of the Haystack Browser informs the user of their navigation history. The tree is organized first by ISS, and each element at the first level of the tree representing a different ISS. Within each ISS, each object that was visited is shown at the second level, sorted by the recency of the user's last action in that window.

Query result sets, because of their importance in the browser process, have a different icon from other Haystack objects. Also, if one query was issued against the contents of another query, then it is placed as a child of the other query result set in the tree. When the user has found the information that she is looking for, the object that was signposted as the desired object is boldfaced in this window.

The back and forward buttons in the toolbar are also part of this. By clicking the back button, the user will move the focus of the Haystack Browser to the previous object visited. The forward button is really just an undo for the back button. Note that backing up to a window doesn't move it to the top of the recency list unless the user does something in that window, otherwise we would loop forever over the two most recent elements as we clicked back.

The history tree is always highlighted with the current location in the search space, so the user knows where she is. The current ISS is expanded, but all other ISSes are collapsed by default so that the history window doesn't grow unmanageably large. The user can double-click any query listed in the tree to move the window to look at that query, even if it is from a different ISS.

5.3.2 Colored Windows

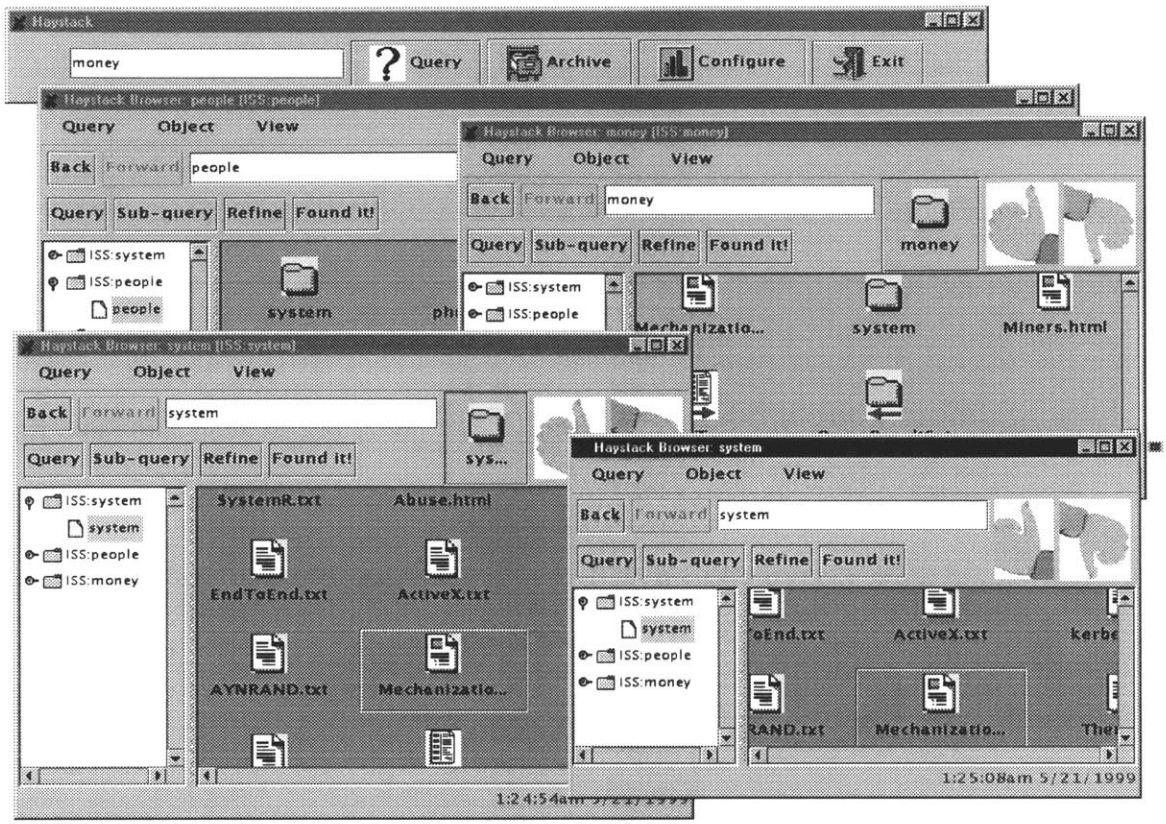


Figure 16. Example of Colored ISSes

The window also indicates which ISS it is associated with. This is done in three ways. First, the title bar of the window will display this information textually, displaying either the first query issued in this ISS or a textual representation of the signposted object, if the user has signposted this ISS. Second, the history pane shows all the currently active ISSes, and highlights the ISS that this window is associated with. Third, the background color of the window is colored differently based on which ISS it is a part of, providing a method of immediately determining which windows are in the same ISS. With a quick glance, the user can now immediately know which windows are associated with which

ISSes, and with another glance at the title bar of any of the like-colored windows she can see which ISS that is.

As mentioned before, it is very important to allow users to create their own hierarchy of "favorite queries" and then store and retrieval that to be able to combine the power of a hand-crafted static hierarchy with an automatic categorization system. For this reason, the user can select options from the Query menu that allow her to load and save the history state of the current Haystack Browser window to a text file. When the user retrieves one of these history files, it will open a Haystack Browser with the stored history information, and the user can quickly access the queries that were in her history when she saved this state.

5.3.3 Haystack Data Model

The knowledge that a group of queries is related into an information search session is extremely useful, and we should not lose this information when the user closes the Haystack Browser. To this end, we create a Bale within the Haystack data structures that corresponds to this ISS, and attach Ties from that Bale to all the queries that are associated with this ISS. In the future, this ISS may be returned as the result to a future query, and the user can then browse through these related queries to perhaps find their information more easily.

Chapter 6.

Haystack Root Window

The Haystack Root Window is a complementary component to the Haystack Browser window, providing a small widget that can live on the user's desktop consuming as little screen real estate as possible, yet still provide rapid access to the Haystack Browser when necessary.

6.1 Overview

The Haystack Root Window simply provides the user with a quick and easy way of using the Haystack system. It provides a small graphical widget that allows users to query and archive documents with only a few clicks. The goal of this widget is to be always active on the user's desktop, ready to open up the larger components of the graphical interface only when the user desires.

6.2 Design Goals

The entire purpose for the Haystack Root Window is to provide quick access to the Haystack system. The user could always start Haystack from the command-line, archive documents there, and then start the Haystack Browser window. The point of the Haystack Root Window is that this takes a lot of time and energy, and the user will often want to get a Haystack Browser Window immediately, especially if the user is using this component as their primary file navigator. For this reason, the Haystack Root Window must provide rapid access to queries and archivals, with as little necessary manipulation by the user as possible.

To provide this rapid access, the Haystack Root Window must be on the user's desktop all the time, and as a result it must be as small as possible. The user might be able to minimize the window, depending on their window manager, but we want to have this root component be something that the user can leave running on their desktop all the time, and not get in the way. Ideally, this widget could be embedded with the user interface of the operating system, as in the GNOME Panel [Icaz99] or the Microsoft Windows toolbar systems.

Finally, this window should be customizable. Some users may wish to have a very simple window that is easy to use and clear about what it is doing. Some users may want to sacrifice some of this clarity for raw speed, but enabling heuristics that allow the user to type into the window and hit return, and have the system figure out whether this was

an archival request or a new query. By supporting both the novice and the power user, we can better tailor the Haystack Root Window to the needs of the user.

6.3 Implementation

The implementation of this widget is straightforward. It consists of a text field and four buttons: query, archive, configure, and exit. The user can click on the triangle on the right side of the text field to open a pull-down menu with the last twenty things entered into the text field. The Configure and Exit buttons are self-explanatory, but the behavior of the Query and Archive buttons depends slightly on whether the component is in "novice" or "power user" mode.

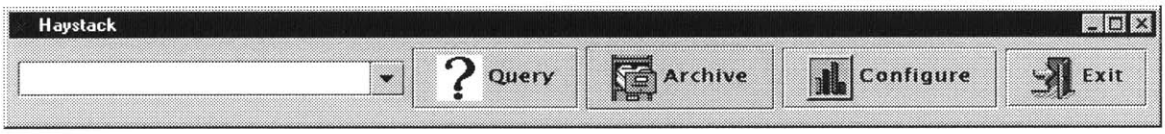


Figure 17. Haystack Root Window

6.3.1 Novice Mode

The text field allows users to enter queries. If the user types in some terms and then clicks "Query", the system will bring up a Haystack Browser window corresponding to the results of that query. If the user clicks on the "Archive" button, then a dialog box will come up to allow the user to select a URL or a local file to archive. By providing these clear and unambiguous query and archive functions, the control should be easy to use by novice users, and it will be difficult for them to make mistakes.

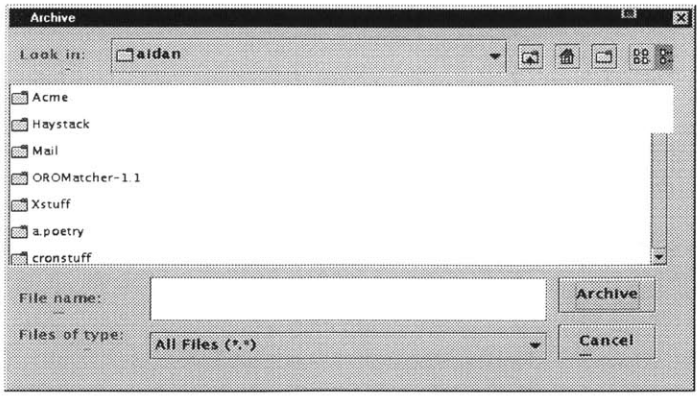


Figure 18. Archive File Chooser

6.3.2 Power User Mode

In this mode, the text field is overloaded to support both archival and querying. If the user clicks Query, then the text field data is used to create a new Haystack Browser centered on that query. If the user clicks Archive, the text field data is used to archive the file or URL specified in that textbox. (if there is no data in the text field, then the Archive File Chooser is opened as before)

The user can also use the Configure option to enable the "power heuristics." When these are enabled, when the user types some text into the textfield and then presses the Enter key, the Haystack Root Window will use a number of heuristics to try to determine if the text represents a query or an archival request, and then do the appropriate thing.

The power heuristics are the following:

- If the text begins with file:// or http://, then it must be a archival request
- If the text begins with a /, then it must be an archival request
- If the text contains a space character, then it must be a query

- If the text refers to a file that exists on the local file system (i.e. something like "www/haystack/index.html") then it is probably an archival request.

We arrived at these heuristics by guesswork and then revising based on the heuristics that turned out to be the most frustrating to us.

Another option is "Enable Power Heuristics with Check". When this option is enabled, the Haystack Root Window will bring up a dialog informing the user of its guess and asking permission to continue when it uses these heuristics. This dialog box will accept keyboard input, so the user can continue to use the system without leaving the keyboard. This functionality allows the user to query or archive using only the keyboard, which can substantially improve the throughput of archival if the user so desires.

Chapter 7.

Future Work

This section will look at a number of topics that might be explored in further development work on the Haystack graphical interface.

7.1 Haystack Graph Browser

It might be nice to actually be able to see a visual representation of the Haystack graph structure as nodes and links. This would require a combination of filtering and a good layout engine to be able to present the user with something useful but also comprehensible. This sort of browser could provide an excellent companion to the Haystack Browser.

7.2 Automatic Categorization

Rather than relying on the user to come up with the categorizations to use in the system's dynamic organization of the data, the system might be able to automatically classify a set

of documents by some kind of clustering algorithm. (e.g. the user might say "classify these documents into a reasonable hierarchy, starting with author") If the system could automatically do this sort of organization, it might be more useful to the user to free them from thinking of the appropriate queries and then figuring out how to type them into the system to get the best results.

7.3 Real Interface Integration

Though the current implementation of the Haystack Browser provides an interface that is very similar to many modern operating system interfaces, it is not the actual interface used by the user on her computer. A natural next step would be to integrate Haystack code into the actual user interfaces used on the user's computer.

This has a number of obvious problems, ranging from tying the code to a particular platform to the proprietary nature of most user interface code. However, the power of embedded Haystack within a user interface is undeniable, as it will make it even more natural for users to use the Haystack system in combination with their normal activities on their computer.

7.4 File system integration

One limitation of the Haystack system is that it needs to be explicitly told what to archive, or else use a proxy system (such as the web proxy or the mail proxy) that monitors the user and archives the information it sees. However, we do not have any

kind of file system proxy at the moment, so we cannot automatically archive a document when the user saves it. While the user can explicitly archive each file as she saves it, it would be useful if we could somehow get Haystack into the file system and archive each document as it was saved.

7.5 User Experiments

While we contend that this interface will allow users to find and use their information more easily and more efficiently, we have not yet run any user studies to back up these assertions. We would like to conduct a number of experiments where we would provide users with a number of tasks, both with and without this interface, and determine whether it is actually a useful tool and how easy it is for users to learn how use this tool effectively. These experiments would not only validate this work, but would give us additional feedback on how to improve the system so that it could be even more useful to users for navigating their information space.

7.6 Haystack Browser Templates

At present, only document Bales and query result Bales have additional operations that can be performed on them. We would like to generalize this paradigm, so that the user could define additional operations to use with any particular Straw type. The user would be able to extend the Haystack Browser to present information in a way that was appropriate for them and for that type of Haystack object, which would clearly be a piece of flexibility that could prove very useful to users who wanted to customize their Haystack Browser.

7.7 Applet-Based Haystack Browser

One of the opportunities presented by the World-Wide Web is the ability to write programs as "applets" that can run within standard web browsers. Such programs are then accessible from any web browser connected to the Internet, and can be run from any of these machine without the normal headache associated with setting up a program to be executed remotely. Future work on the Haystack Browser might produce an applet-based version of the interface, which would allow users to connect to their Haystack from any Internet-connected terminal worldwide and use all the tools provided by the Haystack Browser interface.

7.8 Open Straw in Application

The Haystack Browser currently has the ability to open documents within applications, but this is less general than we would like. It would be nice if a user could open the text representation of an object in an editor, or load the name of an author into a contacts manager application, or use other applications with information in the Haystack data structures that is stored in a non-document Straw. To that end, future work might extend the Haystack system so that arbitrary data Straws could be opened in applications. The main work in this area is to identify what it means to open different types of Straws in applications, and exactly what information will be opened.

7.9 Advanced Refinement Algorithms

The knowledge that a particular document is "especially relevant" to a user's information need is a very valuable piece of information. However, with the current implementation's

crude refinement algorithm, there isn't much that the system does with this information. Another future area of work might be to improve the refinement algorithm to exploit this knowledge to produce a better refinement for the user.

Chapter 8.

Conclusions

This thesis has presented a graphical interface to the Haystack information retrieval system that combines the power of an information retrieval system, a document-centric user interface, and a dynamic classification system. We looked at the evolution of computer systems in user interface and document classification design. We looked at the Haystack system in general, and then took a very detailed look at the implementation of this new graphical interface.

We live in an age of information. We need to be able to use our computers to use this information efficiently, but we need to concentrate our attention on the work we really need to do, not the busywork that today's interfaces demand just to let us get real tasks done. We need to avoid wasting time looking for files with inappropriate classification systems. We need to avoid spending time wrestling with the user interface when we could be working with the data we are looking for. And we need to always maintain a simple, intuitive interface that allows users to get their work done without spending time and energy worrying about what they need to do with the interface in order to start

working on their task. This thesis presents a graphical interface that satisfies all of these conditions.

Even if Haystack does not become the ubiquitous user assistant of the next millenium, it is our hope that this thesis will help to guide those who work on the user assistants of the future.

A user's time is valuable. We should not ask her to waste it by fighting against a user interface and an organization system that get in her way rather than help her. We should provide a system that allows her to get to work as quickly and as easily as possible. One can define the ideal user interface as "the system that works on the user's behalf to make sure she never notices it". We should strive "to do more by doing less", as Prof. Dertouzos says, and this thesis is a step in that direction.

Bibliography

[Adar98] Adar, Eytan. Hybrid-Search and Storage of Semi-structure Information Master's thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, May 1998

[Alta99] Alta Vista website, accessed 16 May 1999 <http://www.altavista.com>

[Appl93] Apple Computer, Inc. Staff, "MacIntosh Human Interface Guidelines" Addison-Wesley Pub Co, January 1993

[Asdo98] Asdoorian, Mark. Data manipulation services in the Haystack IR System Master's thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, May 1998

[Bern89] Bernes-Lee, Tim; Information Management: A Proposal CERN, March 1989, May 1990

[CORB99] Object Management Group. CORBA/IIOP index. accessed 16 May 1999 <http://www.omg.org/corba/csindx.htm>

[Cous97] Cousin, Steve, et al. "The Digital Library Integrated Task Environment(DLITE)" Computer-Human Interactions '97, March 1997

[Dert98] Dertouzos, Michael. The Way It Will Be HarperCollins,1999

[Feie71] Feiertag, R. J. and Organick, E. I., "The Multics Input-Output System", Proceedings of the Third Symposium on Operating Systems Principles, ACM, Oct 1971. pp 3541

[Frak92] Frakes, William B.; Baeza-Yates, Ricardo, Information Retrieval Prentice Hall, 1992

[Furn97] Furnas, George; Effective View Navigation Conference proceedings on Human factors in computing systems (CHI97), March 1997

[Giff91] Gifford, David K. et al. Semantic File Systems 13th ACM Symposium on Operating Systems Principles, October 1991

[Giff91a] Gifford, David K. and O'Toole, James. Intelligent File Systems for Object Repositories Operating Systems of the 90s and Beyond, Springer Verlag, 1991

[Harr96] Harris, Andy; Communication and the World Wide Web

last modified 16 July, 1996, accessed 16 May 1999
<http://klingon.cs.iupui.edu/~aharris/mmcc/mod2/abwww.html>

[Hays99] Haystack web site accessed 16 May 1999 <http://haystack.lcs.mit.edu>

[Icaz99] de Icaza, Miguel, Blandford, Jon "Components in the Gnome Project"
Red Hat Software, May 1999

[John89] Johnson, J., et. al. The Xerox Star: a retrospective
Computer (1989), 22(9):11-26,28-29

[Kay99] McKay, Everett, Developing User Interfaces for Microsoft Windows
Microsoft Press, 1999

[Lisa99] Lisansky, Ilya. Something about Haystack
Master's thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, January 1999

[Lean91] Reaching through analogy: a Design Rationale perspective on roles of analogy
Allan MacLean, Victoria Bellotti, Richard Young and Thomas Moran
Conference proceedings on Human factors in computing systems (CHI91), April 1991

[Micr99] Microsoft Corporation, Microsoft Windows 2000 Beta Training Kit
Microsoft Press, 1999

[Nels65] Nelson, Ted; The Hypertext
Proceedings of the World Documentation Federation, 1965

[Nort99] Northern Light Search, accessed 16 May 1999 <http://www.northernlight.com>

[Otol92] O'Toole, James W. and Gifford, David K. Names should mean What, not Where
ACM 5th European Workshop on Distributed Systems, September 1992

[Ritc74] Ritchie, D. M. and Thompson, K., "The UNIX Time-Sharing System",
CACM 17, 7. July 1974. pp 365-375

[Rocc71] Rocchio, J.J. "Relevance Feedback in Information Retrieval"
In Salton G. (Ed.), The SMART Retrieval System
Prentice Hall, 1971

[Salt83] Salton, G., and McGill, M.J.
Introduction to Modern Information Retrieval
McGraw-Hill, 1983.

[Sun89] NFS Network File System protocol specification
Sun Microsystems, Network Working Group

Request for Comments (RFC 1094) March 1989
Version 2

[Tane92] Tanenbaum, Andrew; Modern Operating Systems
Prentice Hall, 1992

[Taus97] Tauscher, Linda and Greenberg, Saul.
Revisitation Patterns in World Wide Web Navigation
Computer-Human Interactions '97, March 1997

[Vlec97] Van Vleck, Tom; The IBM 7094 and CTSS
modified 12/18/97, accessed 16 May 1999 <http://www.best.com/~thvv/7094.html>

[Wolv93] Wolverton, Van "Running MS-DOS"
Microsoft Press, 1993

[Yaho99] Yahoo! website, accessed 16 May 1999 <http://www.yahoo.com>