Automating Website Profiling for a Deep Web Search Engine

by

Jeffrey W. Yuan

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

June 2009

Author .....
Department of Electrical Engineering and Computer Science
May 1, 2009

Certified by .....
Michael Stonebraker
Adjunct Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Automating Website Profiling for a Deep Web Search Engine

by

Jeffrey W. Yuan

Submitted to the Department of Electrical Engineering and Computer Science
on May 1, 2009, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

The deep web consists of information on the internet that resides in databases or is dynamically generated. It is believed that the deep web represents a large percentage of the total contents on the web, but is currently not indexed by traditional search engines. The Morpheus project is designed to solve this problem by making information in the deep web searchable. This requires a large repository of content sources to be built up, where each source is represented in Morpheus by a profile or wrapper.

This research proposes an approach to automating the creation of wrappers by relying on the average internet user to identify relevant sites. A wrapper generation system was created based on this approach. It comprises two components: the clickstream recorder saves characteristic data for websites identified by users, and the wrapper constructor converts these data into wrappers for the Morpheus system. After completing the implementation of this system, user tests were conducted, which verified that the system is effective and has good usability.

Thesis Supervisor: Michael Stonebraker
Title: Adjunct Professor of Electrical Engineering and Computer Science

Acknowledgments

First and foremost, I want to thank Professor Stonebraker for his support and advice throughout the course of this thesis work. His creative ideas and insightful comments inspired me in my research efforts. Thanks also to Mujde Pamuk, who offered many valuable suggestions.

I want to thank my friends Li, Danny, James, Bobby, and Wendi, who have made my MIT experience unforgettable. And to Jacob, Andrew, and Holly, thanks for bringing a breath of fresh air from outside MIT.

Finally, I want like to express my gratitude to my parents Lijian Yuan and Dalin Chen. Without their love, understanding, and encouragements throughout the years, none of my achievements would have been possible.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

With the rapid growth of the internet, search engines have gained prominence as tools for sifting through the large quantity of information. In 2008, Google reported that it had indexed more than one trillion unique web pages [2]. While this may seem like an astronomical number, it represents only a small fraction of the content on the internet. Beyond these one trillion pages lies what is known as the deep web, repositories of data that are stored in databases and remain mostly invisible to search engines. Deep web sites are characterized by the generation of dynamic responses to user specified input requests; examples include Kayak.com (travel information) and 411.com (local directory). This is in contrast to the shallow web, sites that search engines can index and consist of mostly textual information, such as CNN.com (world news) and Engadget.com (technology blog). It is generally accepted that the deep web is several orders of magnitude larger than the shallow web [3].

The traditional model for discovering information on the internet is to utilize web crawlers which follow links from page to page. The contents of these pages are cached and indexed. When a user performs a search, the search engine queries its repository of indexed pages and returns a list of results considered most relevant by its ranking algorithm. This is the model employed by the vast majority of search engines such as Google, Yahoo, and Live [5]. While these techniques have worked well on the shallow

web, they are not applicable to the deep web. One of the main reasons for this is that crawlers cannot reach content that do not contain hyperlinks, which is a characteristic of the deep web. Also, because data on the deep web is generated only after some specified input, crawlers do not have the sophistication to tap into these contents. Thus, a new approach is necessary in order to access deep web sites.

The Morpheus system is an attempt to solve the problem of searching for information on the deep web. Originally designed as a system for performing general data integration tasks, Morpheus makes use of transform functions to map input data types to output types [12]. In its second design iteration, the goal of the project is to solve the more specific problem of integrating content from the deep web with the shallow web. Each deep web site is viewed as a function which transforms inputs (user defined parameters) into outputs (the desired information) [22]. Such a function is used to profile the interaction of a website, and is known as a wrapper. The main focus of this thesis is to design and implement a system for efficiently generating these wrappers.

1.2 Wrapper Generation

The primary goal of the wrapper generation system is efficiency, using the least amount of effort to create each wrapper. This is an important requirement for the scalability of Morpheus, since a search service is only useful if it has a sufficiently large pool of sources (in this case wrappers) to draw from. The second goal of the system is to have good usability. Because wrapper generation is a key task in using the Morpheus system, it is important to make the user experience as pleasant as possible. Finally, the wrapper generation system should be modular and extensible. Because web standards frequently change, it is imperative that the system be easily adaptable to keep up with the latest technologies used on the internet.

With these goals in mind, a wrapper generation system was designed and implemented. The general idea of this system is to utilize a human guided approach in the construction of wrappers. A user is asked to visit a set of web sites that he would like to wrap, and for each site submit a representative query and highlight the resulting an-

swer. A tool embedded in the web browser records the activity of the user, including the input information and the highlighted output. A second piece of software converts the logged data into wrappers using a set of heuristics.

After implementing the wrapper generation system, an experiment was conducted to evaluate its effectiveness. Five university students were each asked to research a set of queries, recording their results with the tool embedded in the browser. The result of the experiment was that over 80% of queries were answered with reasonable results, and around 55% of queries had optimal answers. Thus, the human guided method for wrapper generation shows significant promise.

1.3 Thesis Outline

The thesis is organized as follows. Chapter 2 describes the problem of searching the deep web, previous work done in this area, and introduces the approach that Morpheus takes. Chapter 3 gives background on the system architecture of Morpheus. Chapter 4 describes the main contributions of this research in creating a more efficient system for generating wrappers for Morpheus. A detailed discussion is given on several iterations in the design process, as well as the final system that was created. Chapter 5 details experiments conducted to evaluate the wrapper generation system. Chapter 6 concludes with a summary of the contributions of this research, its limitations, and possible future work.

Chapter 2

Searching the Deep Web

This chapter discusses the problem of searching for information in the deep web (section 2.1), surveys existing research in this area (section 2.2), and introduces the Morpheus system as a novel solution to the deep web search problem (section 2.3).

2.1 Background

The term "deep web" was coined by Michael Bergman [3], who compares the approach of traditional search engines to dragging a net across the surface of an ocean; a great deal may be caught in the net, but treasures hidden deep in the ocean are missed. While the deep web generally refers to all web sites not indexed by conventional search engines, we define it more specifically as information that is only available by filling data into web forms. The shallow web, on the other hand, refers to the part of the internet which consists of primarily static textual information and is indexable by traditional search engines. In 2000, a study conducted at the University of Michigan found that deep web contained over 7500 terabytes of data and 550 billion documents, compared to 19 terabytes and 1 billion documents in the surface web. According to these figures, the deep web is 400 to 550 times larger than the surface web [4].

Current search engines have become quite adept at returning results when given keyword phrases such as "New York" or "plane ticket." However, they can not provide satisfying answers to queries such as "What is lowest price for a ticket from New York to

Boston next Friday?" If a user wishes to find the answer to such a question, he typically must remember the URL of a deep website which provides travel information (such as Kayak.com), navigate to that site and fill in the requisite information in a form. Clearly, it would be desirable for a search service to be able to answer a common query like this. That is, we would like a system which can find answers in dynamically generated content that is part of the deep web.

2.2 Previous Work

In recent years, large amounts of research and commercial efforts have been directed in the area of deep web search. We examine four projects in this field: from Google [18, 17], Yahoo [9, 10], Endeca [22], and the University of Illinois [6, 7]. In addition, we briefly discuss two related ideas that have implications to the deep web: federated search [15], and web harvesting [1].

Halevy's approach at Google is to design a system that attempts to "surface" deep-web content by pre-computing submissions for HTML forms and adding the resulting pages into the index of the search engine [18, 17]. For each form found by the web crawler, an algorithm generates combinations of inputs (based on the number and types of parameters) to submit to the form. The results of these submissions are expected to be representative of the data stored in the website. The main advantage of this system is its scalability. By submitting only a set of sample inputs for each form, a relatively short amount of time is spent indexing each website. Furthermore, because the result of the algorithm is a collection of HTML pages to be indexed, it fits neatly into Google's existing search engine framework. Thus, this approach is one which emphasizes breadth of pages over depth of each source. The main disadvantage of this approach is that it does not work for dynamic pages such as the status page for airline flights. Another problem is that semantic information, such as the data type of inputs and outputs, is lost in this "surfacing" process.

Ramakrishnan takes a more depth based approach at Yahoo, and focuses on building a structured portal based on community sources in specific domains on the web

[9, 10]. An example of a structured portal is the Internet Movie Database, which integrates information about movies, actors, studios, etc. A differentiating factor in this approach is the focus on entity-relationship links in the data. This structured approach is intended to be applied to a very specific vertical market, using specialized knowledge and techniques applicable to that market. After spending some months in building a vertical portal, Ramakrishnan hopes to broaden the focus into other markets with less effort. Thus, the aim of this approach is similar to existing domain specific meta-search services such as Kayak and Zillow, in the travel and real estate markets respectively. While this approach allows searches in specific segments of the deep web, its overall scalability is questionable because techniques which work for one vertical market may not be applicable for others.

The approach taken by Endecca is to build custom portals for a few carefully chosen vertical markets. This fits into the general strategy of this company, which has focused on faceted search for the e-commerce market. According to their CTO, they plan to integrate intra-enterprise data that are not typically found on the internet (such as data from SAP or Peoplesoft) using conventional data integration techniques [22]. Thus, the Endecca approach is one of applying well known methods to new and targeted markets. There are no novel insights to offer to the problem of searching the deep web.

Chang from the University of Illinois proposes an entity-based approach in searching and integrating the deep web [6, 7]. Instead of viewing the web as a collection of documents, Chang states that it should instead be seen as a repository of entities or data types. This gives users the ability to define the type of data they are searching for, as well as the context in which it appears. An example of this is the query "Amazon customer service #phone", where phone number is the entity (marked with the # sign) and the rest of the words are context descriptions. An entity extraction engine is used to identify and probabilistically match textual information on web pages to entities. In addition, wrappers are created for each web site in order to extract relationship between the input and output data types. This entity search system offers significant innovation in its emphasis on treating data types as a first class citizen. The idea of us-

ing wrappers to encapsulate the functionality of a web service is also quite interesting. However, because entities are only defined in a probabilistic sense, the system still has not fully captured the semantic meaning of returned data.

A general approach to information retrieval that has been gaining traction in recent years is federated search. It works by taking a search query and sending it to multiple databases with the appropriate syntax. The returned results are merged together with duplicate or irrelevant entries removed [15]. Federated search essentially creates a portal where multiple sites with information can be searched from one location. An example of such a service is science.gov, which is a portal for searching all scientific information provided by U.S. government agencies [20]. Northern Light is another example of federated search; it provides a research portal in market intelligence documents for many industries [14]. The idea of translating one search query into queries into multiple websites is a principle that can be used in searching disparate deep web sources.

In order to create an efficient deep web search engine, it is essential to identify data sources that are creditable and relevant. One approach, known as web harvesting, relies on human expertise to direct crawlers to index web content which is difficult to find [1]. An example of this technique being applied is stumbleupon.com, which allows users to discover and rate interesting websites, and also to explore the recommendations of other users in selected categories [23]. The advantage of using human expertise is that the collection of discovered websites is likely to be high quality, since they have been vetted by humans. However, this approach is also problematic in that the scalability of the service is limited by the number of people willing to contribute and the effort they put forth.

The Morpheus system is aimed at searching the deep web, and takes a novel approach which draws from ideas in some of the previous works discussed above. Like Ramakrishnan's approach, Morpheus is built with an emphasis on data types and entity understanding of the web. Similar to Chang's system, a wrapper framework is used to extract the input-output relationships from websites. However, Morpheus aims to capture more of the meaning of returned data by storing semantic information (un-

like Halevy and Chang). Morpheus also aims to be a general purpose system (unlike Ramakrishnan and Endecca), not one which is restricted to vertical markets. The idea behind federated search is used in Morpheus; multiple deep web sources are searched in parallel, with their results integrated into the final output to the user. Finally, the web harvesting idea provides the inspiration for the wrapper generation system, the focus of this thesis.

2.3 The Morpheus Approach

The Morpheus project originated as a solution to the general problem of data integration. Scientists working in academic fields such as biology, physics, astronomy, and computer science frequently have to work with many disparate schemas when accessing different information databases. Biologists, in particular, have had to integrate data from many large genomic repositories to uncover the function of different DNA sequences. These databases typically use different schemas, data models, and query language. Although systems such as the Genomic Unified Scheme exist to integrate data from sources such as GenBank, EMBL, and DDBJ [8], it remains the exception rather than representative of a general trend.

The data integration problem has also been a hurdle for large enterprises. For example, a multi-national corporation might have branches in New York and Paris. At each location, their human resources system stores information based on the laws and customs of that location. In New York, salaries are specified in gross terms with the currency being the US dollar, and there are no lunch allowances. In contrast, Paris employees have salaries specified in after-tax amounts, in Euros, and include government mandated lunch allowances. In this scenario, any attempt to create a composite list of salaries of employees at the two different branches would not be meaningful, because they are essentially different data types.

A global schema is needed to integrate this human resources system. Individual schemas, representing the local salary storage format for New York and Paris for example, can be mapped into the global schema. These mappings are known as transforms,

written using either general purpose languages (C++, C#) or proprietary syntaxes from specific vendors. But there are significant difficulties in writing these transforms due to the inherent complexity in the schema and the large number of variables. Also, the lack of a common standard in both language and implementation techniques means a steep learning curves for programmers. Finally, most enterprises often end up writing the same transforms multiple times, because programmers cannot find existing ones. The Morpheus system was conceived as a new approach to solving the data integration problem with the goal of addressing these difficulties.

The initial iteration of Morpheus was built as a tool for creating transforms and a repository for storing them [12, 13, 21]. There were two primary goals: make it easy to write transforms, and make it easy to find and reuse transforms written by others. Morpheus includes an easy to use Transform Construction Language (TCT) which promotes a workflow oriented approach to transform construction. The TCT greatly simplifies the task of constructing transforms; a programmer could create a transform in around 10 minutes, after some training. The transforms are stored in a Postgres database, where input and output types are registered as Postgres data types. In addition, Morpheus enforces a requirement where all transforms must be entered into a category hierarchy. The hierarchy allows transforms to be stored in a logical manner, enforcing the semantic relationships that are inherent in the transform input and output data types. A browser tool can be used to traverse along the different levels of the category hierarchy, and also allow keyword searches and filtering to be performed.

The second version of Morpheus aims to build on top of the established framework, and address the problem of searching the deep web [22, 11]. This evolution can be seen as taking a general framework and adapting it to solve a particular problem within the domain of the original design. While the idea of a transform in the previous iteration of Morpheus is a general function which maps between two arbitrary data types, now a transform can be understood as taking in some parameters of a search and outputting the results of the query. Thus, a transform can represent a web service such as Kayak.com or 411.com. The category hierarchy still remains, serving as a logical method for classifying these transforms.

It is on this iteration of Morpheus that much of this thesis is based on. The following chapter takes a more in-depth look at the architecture of Morpheus, and how the main contributions of this research fit into this general framework of the system.

Chapter 3

Morpheus Architecture

This chapter gives an overview of the architecture of the Morpheus system (section 3.1) and describes each of its three main components (sections 3.2 - 3.4). Much of the information presented in this chapter is based on the "Morpheus 2.0" paper by Dobbins et al [11].

3.1 System Overview

The Morpheus system can be divided into three main components: the graphical user interface, the query engine, and a Postgres backend (shown in figure 3-1). The interaction between these components is fairly straightforward. The graphical user interface takes in a query typed in by the user, and sends it to the query engine for processing. The query engine uses some heuristics to decide which wrappers are relevant to the query, and executes the functions corresponding to these wrappers which are stored in the Postgres backend. Finally, the results are passed back to the user interface, which displays them using a graph-based visualization system.

The interactions with deep web sites are through wrappers, which map from user-defined functions in Postgres to actual content on the web page. Each wrapper is placed in a category hierarchy, which allows users to either narrow or broaden the scope of a query. For example, a wrapper for a general travel website may be placed under the "Travel" category, while a wrapper for booking cruise ship tours may be placed

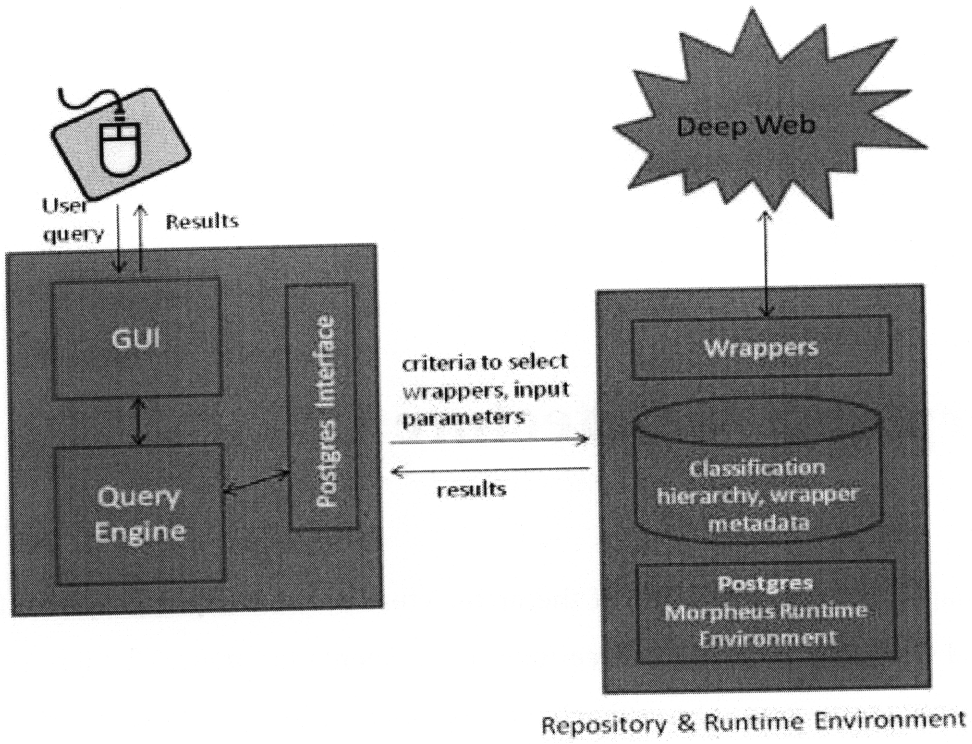

Figure 3-1: The Morpheus system comprises three main components: the graphical user interface, the query engine, and the Postgres backend.

under a subcategory of "Travel" named "Cruises." Wrappers and the category hierarchy are part of the Postgres backend, since that is where they are stored and executed.

The following sections takes a more in-depth look at each of the main Morpheus components, starting from the user facing components and working toward the back-end.

3.2 Graphical User Interface

The purpose of the graphical user interface is to allow users to enter search queries and display the results. For each search query, the user is prompted for three pieces of information. The first is the entity for which the user is requesting information, the second represents the type of information the user is looking for, and the third gives descriptive characteristics of the entity. For example, if the user wishes to find out the

email address of a MIT student named "Jeffrey Yuan", the name "Jeffrey Yuan" would be the first part of the query, "email address" would be the second part, and the third piece of information might be "MIT student".

The result of a search in Morpheus is a collection of outputs from a set of Postgres queries. This information is initially in raw text format, but can be converted into RDF [19]. The advantage of RDF is that it allows data to be stored in a structured format in the entity-relationship model. In Morpheus, all queries are stored as RDF triples, where the "subject" is the input parameters, the function name is the "verb", and the "object" is the output value.

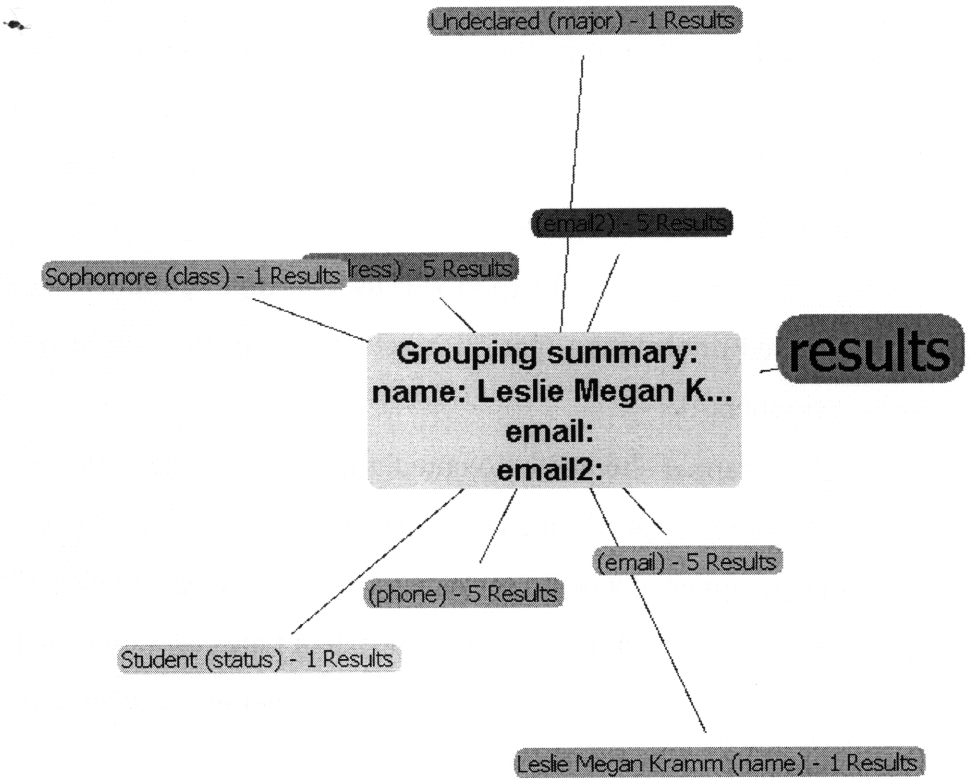

Figure 3-2: This is a sample output displayed on the RDF browser. The node at the center represents the input, in this case is a search for someone named "Leslie." Connected to this central node are results of this search, including information about her major, email, and phone.

An RDF browser displays the output as a graph where the relationships between input and output of the query can be easily visualized (see figure 3-2). At the center of the graph is a node representing the input query, which is linked to all nodes repre-

senting the results. The results are grouped together as clusters by an algorithm which examines their similarity. Factors which affect similarity include the data type of the results as well as the wrapper which produced them. Finally, nodes representing the wrappers are also displayed on the graph, linked to the results they produced. Thus, this graph model gives an entity-relationship view to the results obtained from Morpheus queries.

3.3 Query Engine

For a given query, the query engine decides which wrappers within the category hierarchy should be executed, and then runs the corresponding functions in the Postgres backend. Each Morpheus query contains three pieces of information entered by the user (as described in section 3.2), all of which are passed to the query engine. The characteristics of the entity (part 3 of the three pieces of information) are used to determine which wrappers to use. The entity (part 1) is used as input parameter to these wrappers, and the desired information (part 2) is selected from the output returned from executing the wrappers.

To determine which wrapper should be executed, the query engine performs a relevance calculation for each wrapper. Some heuristics are used in the scoring the relevance, such as favoring wrappers deeper in the category hierarchy over shallower ones, and favoring wrappers which match more of the input data types over ones matching less. The collection of wrappers chosen according to this scheme is executed by running their corresponding Postgres functions.

While the above procedure works for a typical query, cases can arise where the search is not specific enough to narrow down the number of potential wrappers to execute. For example, the query may lack information about the characteristic of the entity. In such a case, Morpheus would ask the user for more information, rather than attempting to run the query over a huge number of wrappers which would produce a large number of low relevance results. If on the other hand, a query could not be matched to any wrapper, then Morpheus would indicate to the user that no informa-

tion could be found. Thus, the design philosophy of the query engine is to give rapid feedback to the user, whether it has too many matches or no match. This is different from most current search systems, which returns some information regardless of how low the relevance of the results might be.

3.4 Postgres Backend

The backend component for Morpheus is a Postgres database which store three types of objects: category hierarchies, data types, and wrappers.

The category hierarchy sorts wrappers and data types into different categories within a tree structure (see figure 3-3). This helps focus a user's search into the right level of specificity, and also allow the query engine to select an appropriate number of wrappers to process for a particular query. An interesting design choice to note is that Morpheus allows a single wrapper to exist under multiple categories within the hierarchy. This choice is made so that user queries can be applied to the most specific category, knowing that all applicable wrappers are included. Also, multiple categorical hierarchies can exist, although the system thus far has run on a single categorical hierarchy.

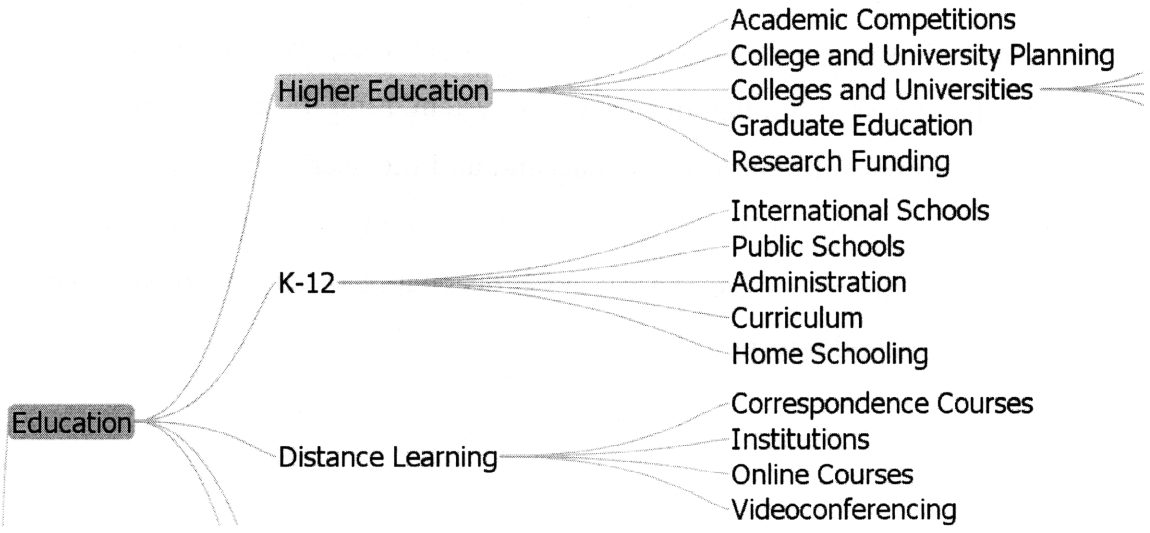

Figure 3-3: The category hierarchy allows objects to be placed into a tree structure with increasing specificity as the depth increases. In this example, the categories within the "Education" field become more specific moving from left to right.

Data types in Morpheus exist to store semantic meaning for queries and results. They help to categorize searches in a logical way, and allow the right wrappers to be chosen for the search. For example, a user request to find the cheapest airline ticket between two locations takes in two pieces of data of type "airport code" for the start and destination locations, a data of type "date" representing the departure date, and optionally a return date if it is a round trip. The result is also a collection of data types; one part could be a number which represents the price of the ticket, another could be of type "airline", etc. When processing such a query, the input and output data types of a wrapper are examined and matched with this query to determine if a particular wrapper is appropriate. Also, the data types allow wrappers to be properly classified under the categorical hierarchy. Wrappers which accept more specific data type are deeper in the hierarchy, whereas those wrappers which only take in very generic data types are shallower in the hierarchy.

Wrappers map from a Postgres user-defined function to the actual details of a web page. Each wrapper includes information about a particular deep web page which it represents: the input data type it accepts, the output data types it produces, as well as the address of the page it describes. It also contains metadata, including information about who created the wrapper, the date it was created, a description of its function, as well as its placement in the category hierarchy.

Overall, the Postgres repository acts as the bridge through which deep web results are obtained and passed through the rest of the system. After the user submits a query, the relevant wrappers are found in the backend and the user inputs are fed to them. After executing the wrapper functions, the result is obtained and returned back to the query engine. To the rest of the system, this process appears as if a stored procedure has been executed and returned.

Chapter 4

Wrapper Generation

This chapter describes the main contributions of this thesis, a system for generating wrappers. The design goals of the system are introduced in section 4.1, and section 4.2 describes the iterative process which leads to the final design. Section 4.3 gives an overview of the system, and sections 4.4 - 4.5 detail its two main components.

4.1 Goals

The primary goal of the wrapper generation system is to make the process of creating a wrapper as efficient as possible; specifically this means that the system should minimize the average time needed to create a wrapper. Because the Morpheus system functions as a deep web search engine, it is crucial that there exist a large repository of potential information sources. Since a wrapper is needed to represent the functionality of each deep web site, having an efficient wrapper generation system is essential to making Morpheus a scalable solution.

Another goal of the wrapper generation system is good usability. One metric of usability is known as learnability, which measures how easy it is for new users to learn the functionality of a system. This is an important consideration since a large number of people could potentially be involved in the task of generating wrappers; it is crucial that they be able to quickly grasp the operations of this system. Another metric of usability is efficiency, the performance of the software for frequent users. This is

also important since this software will be run many times to generate the necessary amount of wrappers for the Morpheus system. Experienced users should be able to use shortcuts and other methods to accelerate the time needed to perform common tasks.

A third goal for the system is that it should be modular and extensible. Because technologies used on the internet changes rapidly, it is important for the wrapper generation system to evolve as well. The system should be adaptable in such a way that a change in web standards could be accommodated easily by changing some definition file.

4.2 Design Iterations

The final design for the wrapper generation system came about after a couple design iterations. Because the final implementation was a result of earlier failed designs, a description of the entire design process is provided in order to properly motivate the final product.

During the early stages of the Morpheus project, wrappers were written by hand. A programmer would identify a relevant deep web site, and programmatically create a wrapper. This process took anywhere from thirty minutes to an hour for each wrapper. Clearly, manually creating wrappers was not a scalable solution; and a more automated system was needed.

The initial design of an automated wrapper generation system took a similar approach to that of the traditional search engines. A crawler was be deployed to traverse the web by following links from page to page. But rather than simply indexing and storing these pages, the crawler looked for specific features on the web pages such as HTML forms. If such a feature was detected, a backend component used a parsing algorithm to extract characteristic information from the page to generate a wrapper.

A crawler which performs some of these functions already existed; the goal of this theis was to make improvements to the feature identification and web page parsing algorithms. The existing crawler only supported web pages which containing WSDL

descriptors, a W3C standard for describing the public interface of web services [24]. However, very few web sites actually utilize this standard, so the existing implementation was not applicable on a large majority of cases.

My plan was to create algorithms for handling additional types of descriptors on web sites: those containing HTML forms, those with hierarchical data types, and those which use AJAX technologies. The plan was to write modules for handing each of these types of sites. When the crawler arrives on a web page, it checks for characteristics for each of these descriptors, and executes the appropriate module. Such a system would be adaptable and modular.

A preliminary prototype was created to test the design, but significant flaws were discovered which made it essentially unworkable. The first problem was that it was quite difficult to create an algorithm which was general enough to be applicable to a large class of websites. For example, it was extremely difficult to conceptualize a method for recognizing AJAX websites and extracting the necessary information from them, since each site usually has a unique display model and data transfer protocol. The second hurdle was that the crawler was not adept at identifying which websites contain relevant information. Because of the vast number of sites in the deep web, it is very difficult to discern automatically which sites contain important information versus others which may have been abandoned for years. It was not feasible for the crawler to save information about each candidate site, since this would create a large amount of false positives. In conclusion, the prototype did not demonstrate that an automatic wrapper generation scheme was any faster than writing wrappers manually.

A new approach was formulated which relies on human guidance to generate wrappers. In the previous design iteration, the main difficulty was in using algorithms to automatically determine which deep web site relevant. However, such a task would be relatively easy for humans. That is, rather than making the entire wrapper generation process automatic, it would be more efficient to use the capabilities of humans in some of the steps.

With this insight, a new process of creating wrappers was devised which consists of two discrete phases. Some users or volunteers are asked to identify deep web sites

that are relevant to Morpheus. This is done by asking them to answer some targeted questions using deep web sites, and their answers are recorded. The second phase consists of converting the information recorded from these users into wrapper code.

This approach has been tested through some prototyping, and has proven to be reasonably effective. The architecture and implementation of this idea are described in details in the next few sections.

4.3 System Overview

The wrapper generation system using the human guided approach (as described in section 4.2) consists of two components: a clickstream recorder which assists a user in marking relevant deep web sites and logging information needed to create wrappers, and a wrapper constructor which converts the data from the clickstream recorder into wrapper code for Morpheus. These two components work together to create wrappers, but exist as separate pieces of software.

The implementation of these two components offers several challenges. First, because the clickstream recorder and the wrapper constructor are logically separate but work together, a clear interface must be defined for their interaction. The XML format is chosen as the method for both components to import and export data. Another challenging aspect is that these two components are written using different languages (see sections 4.4 and 4.5), so it is necessary to deal with the quirks of several programming environments. Finally, the complexity and diversity of the sites in the deep web makes creating a general solution very difficult. This software tries to capture the common case which works for most sites. But there are limitations and cases where it does not work (see section 6.2).

4.4 Clickstream Recorder

The clickstream recorder is used in phase one of the human guided wrapper generation approach; it assists users in identifying relevant deep web sources. In order to

target the category of websites that users should consider, they are provided with a list of questions. In researching these questions, users should naturally find answers in deep web sites. These sites identified are considered to be highly relevant, and their information is saved by the clickstream recorder.

At a high level, the clickstream recorder has two main functions: log the input information entered by users to deep web sites, and save the relevant answer identified by users on the result/output page. Because deep web content is typically accessed using HTML forms, the clickstream recorder logs all forms, including their input fields and the data entered by the user. After submitting the form and obtaining the corresponding results, users should indicate the answer to their research question. The clickstream recorder allows them to highlight these answers (there could be more than one) on the page, and saves information about these selections.

To implement these functions, the clickstream recorder is written as an extension to the popular browser Mozilla Firefox. There are several advantages to integrating this tool inside the browser instead of as a separate piece of software. The main benefit is a more seamless experience for the user, since the clickstream recorder functions within an environment that is already familiar. Also, as an extension to Firefox, it is easy to access the APIs in the browser to obtain state information related to the history of the browser sessions as well as the currently active page. Finally, the Firefox extension framework provides robust updating functionality, so that an updated version of the clickstream recorder can easily be pushed to all users.

Following software design principles, it is desirable to separate the graphical user interface from the data model and backend of a system. The clickstream recorder is designed with this separation in mind; one component handles the onscreen interaction with the user while the other component executes user commands and keeps track of internal state information.

4.4.1 Graphical User Interface

The clickstream recorder tool exists as a toolbar within Firefox (see figure 4-1). The main advantage of a toolbar is that it easily exposes all the functionality of the software

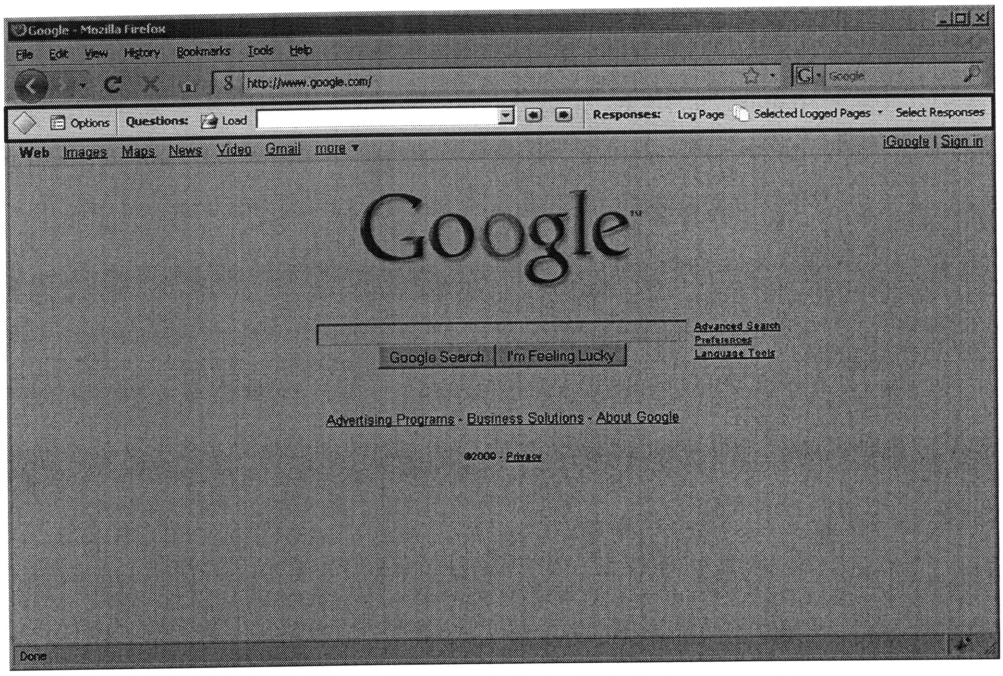

Figure 4-1: The clickstream recorder exists as a toolbar in Firefox, it is located below the navigation toolbar. In the figure, it is the portion enclosed by the box.

to the user without taking up a great deal of screen real estate. This is in contrast to a side panel which takes up a lot of space, or a context menu which is not always present on screen. One other consideration in favor of using a toolbar is the proximity to buttons and menus that form the typical work flow of a user. That is, the location of the clickstream recorder toolbar is very close to the navigation toolbar (which contains the back and forward buttons) and the menu bar. This allows users to work with the clickstream recorder without disrupting their typical usage habits.

The toolbar is divided into three sections (see figure 4-2), separated by gray vertical line separators. The first section contains general functions of the toolbar. The first button, a green diamond in the figure, toggles the toolbar between the active and inactive states. In the active state, all logging and saving functionality is enabled, while in the inactive state these functions are turned off. To the right is the options button, which opens a popup dialog that allows preferences to be changed. Some of these adjustable preferences include the default load/save directory and the naming convention for saving files.

The second section of the toolbar is used to display the list of questions that the

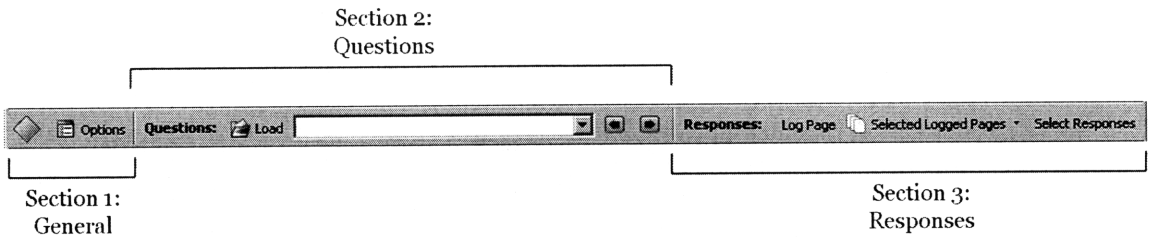

Figure 4-2: The toolbar is divided into three sections: general, questions, and responses.

user is trying to answer. The "Load" button allows the user to browse and select the file which contains the questions. The file containing the questions is an XML file with a predefined format, and is generated using the wrapper construction tool (see section 4.5). After loading the file, the list of questions is populated in the drop down box. The user can use the left and right arrow buttons to select the current question that he wishes to answer or research. Overall, this section of the toolbar functions as a display element, allowing the user to keep track of his current progress. It also serves an organizational purpose, keeping the logged information for each question in separate directories.

The third section of the toolbar is related to answers that users finds to each research question, referred to here as responses. When the toolbar is in its active state, information about the forms on the current page is saved automatically. The button labeled "Log Page" saves a more detailed set of data, including a copy of the page. This is useful if users find the answer to their question on a shallow web page, which is normally not saved automatically. The last button in this section is labeled "Select Responses," it allows users to select answers to the question they are answering on the current page. These responses are associated with the name that is selected in the "Selected Logged Page" dropdown menu, which contains a list of the last ten saved pages (see figure 4-3). By default, the last saved page is selected.

The "Select Responses" button allows users to select answers to their research questions; these answers along with information about the page they appear on are saved by the clickstream recorder to help with wrapper creation. Upon clicking on the "Select Responses" button, a dialog appears which allow multiple selections on the page

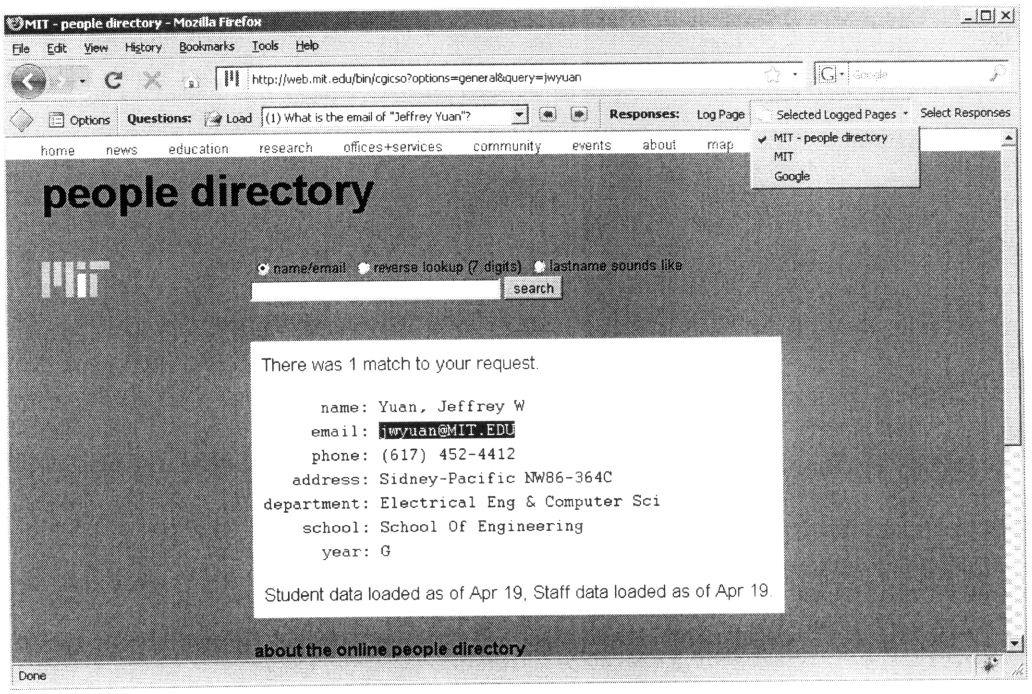

Figure 4-3: In this example, the user is researching the question "What is the email address of Jeffrey Yuan", as can be seen in section 2 of the toolbar. The "Selected Logged Page" dropdown menu contains a list of the most recent saved pages, with the latest one selected. The user can record the selected text as a response by clicking on the "Select Responses" button.

to be named and recorded (see figure 4-4).

To understand the process of creating these selections, an example is given. Suppose a user is trying to answer the question, "What is the email address of a MIT student named Jeffrey Yuan?" After searching in the MIT directory and arriving at the results page, the user clicks the "Select Response" button. When the dialog pops up, he should fill in the "Name" field on the dialog with a description of the result, in this case it can be "Email." Then by highlighting the answer (shown as jwyuan@mit.edu in figure 4-3) and pressing the "Get Selection" button on the dialog, the fields describing the selection are automatically populated with the appropriate data. The start node and end node are the DOM elements at the start and end of the selection. The XPath is a description of the location of the node in the DOM hierarchy, and the offset represents where the selection starts or ends from within the node.

If there are multiple answers on a page, the user could highlight the next one and again press the "Get Selection" button. The "Prev" and "Next" buttons can be used

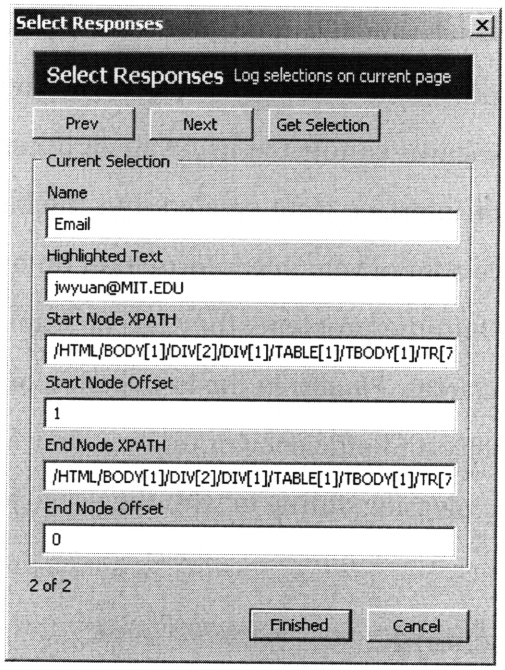

Figure 4-4: This dialog appears after the user clicks the "Select Responses" button on the toolbar. It allows the user to select and label answers on the current. Each selection has properties such as the name, the highlighted text, and description of its location in the DOM hierarchy.

to navigate between the different selections. When all the selections on the page are processed, the user can press the "Finish" button to exit the dialog, at which time the information is saved.

The user interface for the clickstream recorder is specified in XUL, an XML user interface markup language used in the Firefox framework [26]. Within this declarative programming paradigm, the programmer must only specify the location and properties of each user interface element, known also as a widget. The function of each widget is linked directly to procedural code in the backend and model. This type of declarative language enforces good separation of view and model, allowing the look and feel of the user interface to be tweaked without affecting its functionality.

To obtain feedback on the user interface of clickstream recorder, three rounds of user testing were conducted throughout the design and implementation process. The first user feedback session was done informally based on a paper sketch of the interface. A few potential users were asked to describe how they might interact with the interface to perform some hypothetical tasks. Although this study was conducted on

a low fidelity prototype, it gave valuable information on the learnability of the user interface, and the thought process of users as tried to figure out what to do next. The second round of user testing was conducted using an early computer prototype. Here, all the widgets on the user interface were created, but the backend functionality did not yet exist. This test gave a lot of valuable information on how users interacted with the interface in real time and the problems they faced. Improvements were made to address these problematic areas. Finally, in the last session of testing, users gave good responses on the learnability and efficiency criterion. They also commented that the interface was aesthetically pleasing, fitting in with the visual theme of Firefox.

4.4.2 Model and Backend

The backend in the clickstream recorder is responsible for executing the different user commands, including loading and saving files from disk. It is responsible for communicating with the user interface and the model, which is the internal data representation of elements that is displayed on screen. Javascript is the language used to implement all of the backend functionality. This decision was somewhat forced because of requirements of the Firefox framework [16]. But it does have a lot of advantages, the main one being the seamless interaction with the Firefox API.

The clickstream recorder is a singleton class which is instantiated upon the start of a browser session. It attaches a listener to the currently active tab, so that it is notified every time there is a change in the current web page document. Through this method, the clickstream recorder can keep a history of pages visited by the user, and also have access to the currently active document window or tab. Below is a discussion of some key technical aspects of the implementation of the model and backend.

Document Object

Each webpage is represented internally as a hierarchy of nodes in the Document Object Model (DOM). The clickstream recorder attaches a listener to every form element in the current DOM, so that it is notified when a form is submitted by the user. Ele-

ments within a form, such as input box, text area, and radio buttons, also have listeners attached to them, which makes it easy to determine if the user interacted with them.

Selection

The clickstream recorder keeps an internal data type to represent each user selection. Properties of a selection include the location of the node (in the DOM) where the selection starts, the node where the selection ends, and the offset within these nodes. It is tricky to precisely locate a node within a HTML document, because it is not guaranteed to be well formed. Nevertheless, XPath works reasonably well as a selector of nodes within the DOM hierarchy [25]. Although XPath is meant to be used with XML documents, its usage in the clickstream tool is to locate a node using a very specific path specification; it works well unless the HTML is severely malformed.

There are also some difficulties in getting the user selection from the Firefox API. First, there is no support for event handling related to user selections, which makes it difficult to detect when a user selection has been made. This was resolved by manually checking to see if the current selection is an empty string or null, in which case it is assumed that nothing is selected. Also, because the "Get Selection" button is in a popup dialog, it is tricky to get the user selection from the main browser window. In the end, as no efficient solutions were found, the selection is obtained from each opened window or tab. Then, they are each checked to filter out the empty selections (since non-active tabs or windows have selections that are empty), leaving the actual user selection. Overall, the main challenges in implementing the selection object came from the lack of sufficient API support in Firefox, but creative solutions were found to resolve these difficulties.

Form

Forms are the most important object on a web page, because deep web content is accessed by submission of a form. The clickstream recorder saves the structure of every form automatically. This is done by searching for nodes of the type "form" in the DOM of each page after it loads completely.

Two important attributes of forms are the "action" and "method" attributes. The "action" attribute indicates the address where the form is to be submitted, and "method" describes how the form is submitted (either using the GET or POST method). Besides these two attributes, it is important to save the elements encapsulated in the form. These can range from input fields, combo boxes, check boxes, radio buttons, etc.

Preferences

Firefox contains a robust system for managing user preferences. The preferences dialog maps each of the user modifiable options to a corresponding options entry in the Firefox preferences database. Most of the challenge in implementing the preferences component for the clickstream recorder involves setting appropriate default values for new installations. For example, the default file directory for saving response files is set to be the user profile location for each user. This must be carefully managed in cases of multiple user profiles.

Initially, a function was written to export and import preference files, so that a user could synchronize them across multiple computers. However, user testing revealed that this feature was ignored by almost all users. So in the end, it was removed to simplify the user interface.

Input and Output

There are two types of I/O interaction: loading question files and saving response files. The question file is an XML file that is generated by the wrapper constructor tool, and the format specification is well defined. When the user loads such a file, the clickstream recorder simply parses the XML, extracts the questions, and displays it in section 2 of the toolbar (see figure 4-2).

The responses file format is more complex, since it must handle several types of logged file. There are three types of files: form information, selections, and the entire content of the page. Templates for each of these file formats are defined, the appropriate file template is matched and populated with the relevant data, and then written to disk.

4.5 Wrapper Constructor

The wrapper constructor is used to generate wrapper files from data saved by the clickstream recorder. In the Morpheus system, wrappers are stored and executed as Postgres user defined functions in the backend component (as discussed in section 3.4). However, since the query engine is written in Java, it also makes sense for consistency reasons to first generate wrappers as Java classes, so that they can be examined and tested. Existing tools can be used to later convert these wrappers from Java into Postgres functions. Thus, the wrapper constructor takes in as input XML files containing information about forms and selections from web pages, and outputs wrappers written in Java.

The secondary function of the wrapper constructor is to allow question files to be generated. As mentioned in section 4.4, these files contain the questions that users are attempting to answer using the clickstream recorder. The wrapper constructor allows these questions to be entered, and saved into the appropriate format.

The following sections detail the technical aspects behind these two functions of the wrapper constructor.

4.5.1 Generating Question Files

The list of questions for users to answer is stored in a standard XML file. Each entry in this file consists of the text of the question, the category it belongs to, and any notes that are associated with it.

The wrapper constructor allows new question files to be created by entering it into standard document interface similar to a word processor. Each question along with its associated metadata is entered on a single line with the \ character separating them. When the file is saved, the wrapper constructor parses each line into a node in the XML document hierarchy, with the specified metadata as attributes. Overall, the question files management functionality is a fairly straightforward implementation in formatting questions so that they can be read by the clickstream recorder.

4.5.2 Creating Wrappers

To construct each wrapper, two XML files generated by the clickstream recorder are necessary: one contains information about the forms used to enter inputs to the site, and the other describes what information needs to be extracted from the output (see section 4.4).

From these two files, a wrapper is created which consists of four Java classes: Input, Output, Wrapper, and Stub. The name given to each of these four classes is the name of the website concatenated by the function of the class. For example, the wrapper for 411.com consists of `411_Input.java`, `411_Ouput.java`, `411_Wrapper.java`, and `411_Stub.java`.

Input

The Input class represents the form that is submitted. Information about each element in the form is extracted from the input XML file, and stored in an array. Each entry in this array is a tuple consisting of the name of that input element and its type.

Output

The Output class represents the response fields selected by the user, corresponding to the results that are derived from the submitted inputs. This class has a similar structure to the Input class. Each selection made by the user is extracted from the XML file, and added to an array. The array keeps track of the name of each output selection and its location on the page (represented using XPath as described in section 4.4.2).

Wrapper

The Wrapper class is responsible for submitting the inputs to the targeted deep web source, retrieving the response page, and parsing it to obtain the desired outputs. It takes in an instance of the Input class, which contains all the fields and their associated values. These are combined with attributes of the form, and a submission URL is generated. Then, a HTTP request is created and sent to this URL. One item of note

here is the difference between submitting via the "Get" or "Post" method. The "Get" method is used for instances where the number of inputs is few, so these input values are directly encoded into the URL. The "Post" method, on the other hand, stores input fields in the body of the request, and is appropriate for cases where there are many fields or large data size.

After submitting the request, the response page is retrieved. Before anything can be extracted from the page, it must first be converted to XML format. This is because HTML documents sometimes contain unclosed tags or is malformed in other ways. Converting to XML ensures these problems are fixed. The XPath query language is used on the resultant XML page to extract the output fields. These fields are subsequently passed to an instance of the Output class.

Stub

The Stub class is used to execute the wrapper. It contains some simple graphical user interface elements which are used to prompt the user for each field in the Input class. These input values are passed into an instance of the Wrapper class, which performs the web page submission, retrieval, and parsing. The resulting output fields are then stored in the Output class and also displayed on screen. Thus, the Stub class functions as a testing tool that can be used to check the functionality of the wrapper as a whole.

Chapter 5

User Experiment

To assess the effectiveness of the human guided approach to wrapper generation, we asked a group of potential users to find deep web sources which answer a list of provided questions. The idea is to evaluate the quality of the answers they arrive at compared to the reference answers that we found.

5.1 Method

Five university students (from MIT and the University of Washington) were asked to participate in this study; four of them were undergraduates and one was a graduate students. To insure that a diverse sample of users was chosen, the five students come from different majors. All of the users were familiar with the Firefox web browser, but none has had any previous experience with deep web searching or has used the Morpheus system.

Each user was given a list of ten open ended questions, and asked to use the clickstream recorder to help research and answer these questions. To ensure that no specialized knowledge is needed for these questions, all were chosen in the domain of travel and entertainment, topics that most people are familiar with. For example, one of the questions is: "What are some trails around the Portland, Oregon area that are at least 15 miles long and have a camp site?"

Users were asked to spend no more than five minutes on each question. At the

end of that time, they should submit their best answer, or indicate that no answer was found.

For each question, the answer returned by a user is compared to the reference answer that we found. These reference answers are derived after a lot of research, and are from sources that are considered to be the most relevant for each question. A score is given based on the quality of a user's answer: 1 if it matches well with the reference or is otherwise high quality, ½ indicates it's passable with some criteria in the question not satisfied, and 0 means no answer was returned or the answer is deficient in significant ways.

5.2 Results

Table 5.1 shows the results of this user study. The rows indicate the scores assigned to the answers given by each user, and the columns show how each of the five users performed on the questions. The "mean" gives an average score, with ½ counting as 0.5 in this computation. Aggregated over all the users, 84% of questions were answered with a rating of at least "½", and 56% of questions were rated "1". This means that a large majority of users were able to find websites that gave at least decent answers.

Question	User A	User B	User C	User D	User E	Mean
1	½	1	1	½	½	0.7
2	1	1	1	0	1	0.8
3	1	1	1	0	1	0.8
4	1	½	½	½	1	0.7
5	½	1	½	0	1	0.6
6	1	½	0	½	0	0.4
7	½	1	0	1	1	0.7
8	1	1	½	1	1	0.9
9	½	1	1	0	1	0.7
10	1	1	½	0	1	0.7
Mean	0.8	0.9	0.6	0.35	0.85	

Table 5.1: Results from the user study. A score of 1 indicates the user answered the question perfectly, ½ means that the answer was acceptable, and 0 is given when the answer was not close or not provided at all.

For the most part, each question had "great" or "acceptable" answers from most of the users. The only exception to this was question 6, which contains a lot of specific conditions and is probably quite difficult. Another interesting point to note is that every question had at least one "great" answer. This means that combining the answers from these five users would yield perfect results.

When looking at the performance of each of the users, it is clear that some were better at answering these questions than others. For example, User B produced consistently better results than User D. This was expected, since these users were selected from different academic backgrounds, and they have different amounts of experience in performing research using the internet.

This user study demonstrates that the human guided approach to finding deep web sites shows significant promise. It must be noted that these results are the result of a general system test and cannot be considered the product of a scientific experiment. For example, the users were selected in an ad hoc fashion, and the questions have not been screened for bias. Nevertheless, the results of this study indicate that the wrapper generation system implemented in this research is effective in locating relevant deep web sources.

Chapter 6

Contributions, Limitations, and Future Work

6.1 Contributions

In this research, I investigated the problem of automating wrapper construction for the Morpheus deep web search engine. After several design iterations and prototypes, I implemented a wrapper generation system which utilizes human expertise in finding meaningful deep web content. The system is made up of two components. The clickstream recorder helps users identify relevant deep web sites, and saves characteristic information about their inputs and outputs. The wrapper constructor tool then converts the saved information into wrappers for the Morpheus system.

6.2 Limitations

Because many different languages and techniques are used in constructing web pages, it is difficult to design the clickstream recorder so that it works in every situation.

Several types of web sites present problems for the clickstream recorder. Sites which use hierarchical form, where the result of one set of inputs leads to another set of inputs, can mislead the clickstream recorder into saving the wrong information. Similarly, pages which use Javascript to dynamically adjust forms can cause the clickstream

recorder to miss form elements during the saving process. Finally, pages which contain multiple forms, such as Orbitz.com, present difficulties in that the constructor tool cannot determine which form is the relevant one. In all of these cases, a human must manually adjust the saved data files to correct these mistakes.

6.3 Future Work

In the future, more effort could be spent in improving the methods used to parse forms in the clickstream recorder. To handle the case of hierarchical forms, a heuristic could be written to recognize the situation and handle it accordingly. For pages with Javascript, form information could be saved when the user submits the form, thus ensuring that no more changes to the structure of the pages can occur. But perhaps a better solution to these problems is to build a more general framework to let the user easily adjust the saved data. For example, the clickstream recorder could show the form information that is extracted to the user before saving, thus allowing adjustments to be made on the fly. This would be in the spirit of the human guided approach.

The wrapper constructor can also be improved to better handle unusual inputs. This can be done by creating better heuristics to the different types of information present in the saved data. The wrapper constructor tool can also give the user more freedom to adjust the form and selection information during the wrapper creation process.

Bibliography

[1] Zaenal Akbar and Laksana Tri Handoko. A simple mechanism for focused web-harvesting. *Proceeding of the International Conference on Advanced Computational Intelligence and Its Applications*, abs/0809.0723, 2008.

[2] Jesse Alpert and Nissan Hajaj. Office google blog: We knew the web was big. Website, 2008. `http://googleblog.blogspot.com/2008/07/we-knew-web-was-big.html`.

[3] Michael K. Bergman. The deep web: Surfacing hidden value. *The Journal of Electronic Publishing*, 7(1), August 2001.

[4] Michael K. Bergman. The murky depths of the 'deep web'. Website, February 2007. `http://www.mkbergman.com/?p=343`.

[5] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. In *Computer Networks and ISDN Systems*, pages 107–117. Elsevier Science Publishers B. V., 1998.

[6] Tao Cheng and Kevin C. Chang. Entity search engine: Towards agile best-effort information integration over the web. In *CIDR*, pages 108–113. Conference on Innovative Data Systems Research, 2007.

[7] Tao Cheng, Xifeng Yan, and Kevin Chen-Chuan Chang. Supporting entity search: a large-scale prototype search engine. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 1144–1146, New York, NY, USA, 2007. ACM.

[8] S. B. Davidson, J. Crabtree, B. P. Brunk, J. Schug, V. Tannen, G. C. Overton, and C. J. Stoeckert, Jr. K2/kleisli and gus: experiments in integrated access to genomic data sources. *IBM Systems Journal*, 40(2):512–531, 2001.

[9] Pedro DeRose, Warren Shen, Fei Chen, AnHai Doan, and Raghu Ramakrishnan. Building structured web community portals: a top-down, compositional, and incremental approach. In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pages 399–410. VLDB Endowment, 2007.

[10] Pedro Derose, Warren Shen, Fei Chen, Yoonkyong Lee, Douglas Burdick, AnHai Doan, and Raghu Ramakrishnan. Dblife: A community information management

platform for the database research community. In *CIDR*, pages 169–172. Conference on Innovative Data Systems Research, 2007.

[11] Pete Dobbins, Tiffany Dohzen, Christan Grant, Joachim Hammer, Malachi Jones, Dev Oliver, Mujde Pamuk, Jungmin Shin, and Mike Stonebraker. Morpheus 2.0: A data transformation management system. In *VLDB 2007 Third International Workshop on Database Interoperability*, Vienna, Austria, 2007.

[12] Tiffany Dohzen, Mujde Pamuk, Seok-Won Seong, Joachim Hammer, and Michael Stonebraker. Data integration through transform reuse in the morpheus project. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 736–738, New York, NY, USA, 2006. ACM.

[13] Joachim Hammer, Mike Stonebraker, and Oguzhan Topsakal. Thalia: Test harness for the assessment of legacy information integration approaches. In *ICDE '05: Proceedings of the 21st International Conference on Data Engineering*, pages 485–486, Washington, DC, USA, 2005. IEEE Computer Society.

[14] Paula J. Hane. Northern light launches mi analyst. Website, May 2007. `http://newsbreaks.infotoday.com/nbReader.asp?ArticleId=36109`.

[15] Péter Jascó. Thoughts about federated searching. *Information Today*, 21(9), October 2004.

[16] Javascript - mozilla developer center, February 2009. `https://developer.mozilla.org/en/javascript`.

[17] Jayant Madhavan, Shawn R. Jeffery, Shirley Cohen, Xin (Luna) Dong, David Ko, Cong Yu, and Alon Halevy. Web-scale data integration: You can afford to pay as you go. In *CIDR*, pages 342–350. Conference on Innovative Data Systems Research, 2007.

[18] Jayant Madhavan, David Ko, Lucja Kot, Vignesh Ganapathy, Alex Rasmussen, and Alon Halevy. Google's deep web crawl. *Proc. VLDB Endow.*, 1(2):1241–1252, 2008.

[19] Resource description framework (rdf) / w3c semantic web activity, March 2009. `http://www.w3.org/RDF/`.

[20] Science.gov: About, March 2008. `http://www.science.gov/about.html`.

[21] Jungmin Shin, Joachim Hammer, and Herman Lam. Rdf-based approach to data transform composition. In *ICIS '08: Proceedings of the Seventh IEEE/ACIS International Conference on Computer and Information Science (icis 2008)*, pages 645–648, Washington, DC, USA, 2008. IEEE Computer Society.

[22] Michael Stonebraker. Integrating the deep web with the shallow web. Project proposal for Morpheus 2.0, 2007.

[23] About stumbleupon, 2008. `http://www.stumbleupon.com/aboutus/`.

[24] Web service definition language (wsdl), March 2001. http://www.w3.org/TR/wsdl/.

[25] Xml path language (xpath) 2.0, January 2007. http://www.w3.org/TR/xpath20/.

[26] Xul reference - mozilla developer center, March 2009. https://developer.mozilla.org/En/XUL_Reference.