LEARNING BASED WEB QUERY

PROCESSING

BY

YANLEI DIAO

A Thesis Presented to
The Hong Kong University of Science and Technology
in Partial Fulfilment
of the Requirements for
the Degree of Master of Philosophy
in Computer Science

Hong Kong, May, 2000

Copyright © by Yanlei Diao 2000
Authorization

I hereby declare that I am the sole author of this thesis.

I authorize the Hong Kong University of Science and Technology to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize the Hong Kong University of Science and Technology to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.
LEARNING BASED WEB QUERY

PROCESSING

BY

YANLEI DIAO

APPROVED:

DR. HONGJUN LU, SUPERVISOR

PROF. ROLAND T. CHIN, HEAD OF DEPARTMENT

Department of Computer Science

1 June, 2000
ACKNOWLEDGEMENTS

I would like to thank my supervisor Dr. Hongjun Lu, who introduced this topic to me and provided guidance to me all the way. I learned the methodology to conduct scientific research and academic writing skills as well as the knowledge from him. He was always available when I needed his opinions and provided invaluable feedback and assistance on my ideas, the system implementation and the thesis writing. Without his knowledge and support, this thesis could not exist.

I would also like to thank Prof. Frederick H. Lochovsksy, Dr. Kamalakar Karlapalem for being members of my thesis committee. They have given me a lot of comments on my thesis. In addition, I owe a special thanks to Dr. Bertram Shi in Electrical and Electronic Engineering Department who suggested a good design of classification knowledge to perform learning based Web query processing.

The system discussed in the thesis could not be realized without the hard work of Zengping Tian, Songting Chen and Ye Fan from Fudan University. As a team led by Dr. Lu, we have worked day and night in the past one year and published two papers, one demonstration paper at SIGMOD 2000 and one research paper at VLDB 2000, which is a great honor to the whole team. Hereby, I would like to express a great deal of gratitude to them.

Finally, I want to thank my friends in the Database laboratory. They have spent hours discussing this topic and debugging the program with me. Working in this laboratory was a valuable experience and a lot of fun.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TITLE PAGE</td>
<td>I</td>
</tr>
<tr>
<td>AUTHORIZATION</td>
<td>II</td>
</tr>
<tr>
<td>SIGNATURE PAGE</td>
<td>III</td>
</tr>
<tr>
<td>ACKNOWLEDGEMENTS</td>
<td>IV</td>
</tr>
<tr>
<td>TABLE OF CONTENTS</td>
<td>V</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>VII</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>VIII</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>IX</td>
</tr>
<tr>
<td>CHAPTER 1 INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1 THE PROBLEM</td>
<td>1</td>
</tr>
<tr>
<td>1.2 EXISTING APPROACHES</td>
<td>2</td>
</tr>
<tr>
<td>1.3 OUR CONTRIBUTION</td>
<td>3</td>
</tr>
<tr>
<td>1.4 ORGANIZATION OF THE THESIS</td>
<td>6</td>
</tr>
<tr>
<td>CHAPTER 2 RELATED WORK</td>
<td>8</td>
</tr>
<tr>
<td>2.1 INFORMATION RETRIEVAL</td>
<td>8</td>
</tr>
<tr>
<td>2.2 INTELLIGENT AGENTS</td>
<td>13</td>
</tr>
<tr>
<td>2.3 QUERY LANGUAGES AND WRAPPER GENERATION</td>
<td>16</td>
</tr>
<tr>
<td>CHAPTER 3 A LEARNING BASED WEB QUERY PROCESSING APPROACH</td>
<td>23</td>
</tr>
<tr>
<td>3.1 A LEARNING-BASED WEB QUERY PROCESSING SYSTEM</td>
<td>23</td>
</tr>
<tr>
<td>3.2 A QUERY SESSION</td>
<td>25</td>
</tr>
<tr>
<td>CHAPTER 4 WEB DATA MODELING</td>
<td>29</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

Figure 1: A Learning Based Web Query Processing System ..................24
Figure 2: A Query Session ................................................................25
Figure 3: An HTML Document and the Corresponding Segment Tree ....30
Figure 4: A Segment Graph ..............................................................33
Figure 5: Algorithm for Locating a Queried Segment on a Web Page ....44
Figure 6: A Sample Output .................................................................51
Figure 7: Accuracy of Training Strategies ..........................................58
Figure 8: Efficiency of Training Strategies .........................................59
Figure 9: Effects of Training Strategies with Enlarged Training Sizes ....60
Figure 10: Sample Results for "Data Mining Researchers" ....................64
LIST OF TABLES

Table 1: Terminology for Definitions of Evaluation Metrics .........................49
Table 2: URLs Used in Query Processing ...........................................52
Table 3: Basic Performance of the System ........................................53
Table 4: Effects of Systems Using Different Types of Knowledge ..............55
Table 5: Analysis of Systems Using Different Types of Knowledge ..........56
Table 6: Query Requirements and Keyword Queries .............................63
Table 7: Results of Different Queries ...............................................65
LEARNING BASED WEB QUERY

PROCESSING

BY

YANLEI DIAO

A Thesis Presented to
the Hong Kong University of Science and Technology
In Partial Fulfilment
of the Requirements for
the Degree of Master of Philosophy
in Computer Science
Hong Kong, May, 2000

ABSTRACT

Unlike relational database systems that return exact data items for a query, most current Web search and query systems return URLs of pages that might contain the required information. Processing the URLs to dig out the required information becomes a task of users, which is both ineffective and costly. In this work, we propose a novel Web query processing approach with learning capabilities. Under this approach, user queries are posted in the form of keywords that may not precisely describe the query requirements. A general-purpose search engine such as Yahoo! is employed to obtain candidate query results in the form of URLs. The first few URLs with high ranking are returned to a user for browsing. Meanwhile, the query processor learns how the user navigates through hyperlinks and locates the query
result within a Web page. With the learned knowledge, the query processor processes the rest URLs to produce precise query results without user involvement. The preliminary experimental results indicate that the approach can process a range of Web queries with satisfactory performance. The architecture of such a query processor, techniques of modeling HTML pages, and knowledge for navigation and queried segment identification are discussed. Experiments on the effectiveness of the approach, the required knowledge, and the training strategies are presented.
CHAPTER 1

INTRODUCTION

1.1 The Problem

The Internet and the Web have changed everything. It is estimated that the publicly
indexable Web now contains about 600 million pages, encompassing approximately
6 terabytes of text data [41]. The Web has become everyone's information source.
Each day, a huge number of people spend a lot of time on the Web searching for
information of interest, such as news, prices of goods, research papers, etc. With the
excitement on electronic commerce growing, the Internet will also become a
common platform for conducting business. The usage of the Web therefore will
increase more dramatically.

Search engines [8] are widely used to locate information across the Web. After a key
word query is issued, search engines match the input keywords with
indices they maintain to find relevant Web sites/pages. The URLs of those sites or
pages are returned to users in the order of certain ranking. Finally users browse the
Web pages and locate the information they are looking for. For users who are used
to retrieving information from database systems, searching from the Web is
sometimes frustrating. For example, if they would like to find the lowest price for a
certain part in a database, a simple SQL statement will do the job. However, it may
cost hours to search for the lowest price from the Web, if they have the stamina to
find it. One problem of search on the Web is that search engines return very large hit
lists with low precision. Users need to sift relevant documents from irrelevant ones
by manually fetching and browsing pages. Due to the limited human browsing capability, they often read the first few pages and discard the rest part of the hit list. A lot of information is lost in this way. Another discouraging aspect of current Web search is that URLs (or whole pages) are returned as search results. It is very likely that the answer to a user query is only part of the page (like one field in a relation). Retrieving the whole page actually leaves the task of search inside a page to Web users. With these two aspects remaining unchanged, Web users will not be freed from the heavy burden of browsing pages and locating required information, and information obtained from one search will be inherently limited.

1.2 Existing Approaches

While the dissimilarity between querying the Web and querying a database is caused by the fundamental differences between the Web and a database system, which will most likely remain, researchers from different disciplines have been trying to improve the situation.

A wide range of research work has been reported in Information Retrieval to improve the easiness and effectiveness of querying the Web, including developing better classification mechanisms, building more effective indices and using better searching strategies and ranking functions [9, 30, 37].

Using intelligent agent to help users is originated from the Artificial Intelligence community. In the context of querying the Web, such agents can learn user profiles or models from user search behaviors, and then employ the learned knowledge to predicate URLs that may have interesting information, thus providing suggestions to users [2, 6, 10, 35, 45, 46, 54].
Researchers from the database community take another approach. Since the Web can be viewed as a large distributed database system, it is reasonable to expect that the database technologies developed during past decades can be applied to Web queries. The related efforts include Web query language design [5, 22, 23, 34, 38, 43, 48] and wrapper generation [3, 4, 12, 21, 31, 39, 42, 47, 52, 56].

1.3 Our Contribution

While there are various issues in Web query processing and different approaches to tackling these issues, we describe in this thesis our efforts to build an on-line query processing system that enables users to query the Web and obtain the results in a database-like fashion. A shorter version of the thesis is to be published [19]. By our proposed approach, a user first issues a key word query (probably not very precise) and it is passed to a general search engine such as Yahoo!. The search engine returns URLs of Web pages that might contain requested information. At the beginning, these Web pages are retrieved and presented to the user for browsing. During browsing, the system records down the section of a Web page that contains the query result and the sequence of hyperlinks through which the user navigates to find it. Query results and user actions are analysed. After the user browses a few pages, the system knows what the user exactly wants and becomes capable of scanning Web pages and following links, if necessary, to locate the query results. Finally the system presents to the user segments of Web pages instead of the original Web pages. A segment can be a paragraph in text, a table or a list.

To our knowledge, it is the first query processing system that processes ad hoc queries on HTML pages and automatically extracts segments of pages as query
results. Despite the superficial similarity with a large body of related work, this system is unique at the following aspects:

1. Unlike information retrieval systems or intelligent agents that return URLs or Web pages as query results, our system tries to return the query answer as precise as possible (here "precise" means succinct and meeting query requirements well). Correspondingly in relational databases if a field is the answer, the field but not the whole table is returned. To avoid any confusion of terms, we would like to call the URLs or pages returned from some systems recommendations. In contrast, precise results inside pages returned by our system are called query results.

2. The system does not require a prior knowledge about users such as user profiles. Moreover, it does not require pre-processing of Web pages such as generating wrappers either. As a consequence, the system is well suited to processing ad hoc queries that can hardly be handled by static hypertext analysis [12, 13, 57], agents or system using wrappers.

3. The system exploits the page formatting and the linkage information simultaneously to automate query processing. Recently there has been a surge of research work either on hyperlinks to help Web search [2, 8, 26, 45] or on internal structure of HTML pages in wrapper generation and Information Extraction related applications [18, 21]. However, combining these two types of information in the context of a query processing system poses great challenges, which will be discussed later in detail.
We call our approach a learning-based approach because the system learns about the exact query requirements and the efficient way to locate the information during a query process. As a result of the learning, the system is able to deliver to users the results that better match users' needs in a more concise form. The learning approach brings the following advantages to our query processing system.

1. Users can still express their queries in keywords, which is the easiest way for casual Web users. If a user is not familiar with the vocabulary of information suppliers, the query specification may even be vague. To bridge the gap between the issued keywords and the real query requirements, the system learns the precise requirements from users. Consequently, unlike the traditional information retrieval approaches based on keyword matching, our system can return results that do not contain the specified keywords but contain the required information.

2. The system has the capability of navigating in the neighborhood of the page where the specified keywords occur. Often the occurrences of keywords do not guarantee the discovery of queried information. However, they indicate the current page is somewhat relevant to the result that may reside on a page one link or two away from it. This is especially true when the specified keywords are not very precise. Learning to navigate makes the system dependent more on user behaviors and capable of finding query results that keyword search fails to find.

3. Although the learned knowledge about query requirements and navigation heuristics is useful to one query, it helps to make 100% use of the hit list returned by a search engine. Users are relieved from browsing dozens or hundreds of Web pages in order to obtain all available information. They only
need to browse a small number of them to train the system. Moreover, for queries expected to be issued repeatedly, the system will manage the learned knowledge and apply it to the same query automatically. The learning process is required at most periodically.

4. The learning process is nearly imperceptible to users and minimal effort is required from them. Web pages are presented by the system interface and users browse them as they do with a popular browser. Simple actions of clicking hyperlinks and marking segments are captured by a background process from which the system completes the learning process and automatically processes other pages.

A prototype system has been implemented using the approach. The preliminary results are encouraging. User query requirements and navigation heuristics can be reasonably well captured and stored in a rather simple form. Given a set of about 100 URLs, users need to browse no more than 10 of them to make the system capable of locating the queried segments or denying the Web sites with the correctness rate higher than 80%.

1.4 Organization of the Thesis

The remainder of the thesis is organized as follows. Section 2 surveys related work. Section 3 describes the learning based Web query processing approach in detail. The data modeling is presented in Section 4. Knowledge to be learned, its representation and the acquisition process are described in Section 5. Section 6 describes how a user query is processed using the learned knowledge. Experiments
conducted to evaluate the approach are presented in Section 7. Section 8 concludes the proposal with discussions on future work.
CHAPTER 2

RELATED WORK

In this chapter we discuss three areas of research related to our work: 1) information retrieval, 2) intelligent agents and 3) Web query languages and wrapper generation.

2.1 Information Retrieval

The primary goal of Information Retrieval was indexing text and searching for useful documents in a collection. With its development over years, it is currently characterized by the following features. One is the document model. Dealing with unformatted data, it views a document as a long string of characters. After important words are extracted from the character string, indexes are created and queries are answered using some variation of the vector-space model of documents. Other features concern the searching and browsing capabilities. The search capabilities allow for a mapping between a user's query statement and the items in the information database that will answer that query. The query statement can consist of natural language text in composition style and/or query terms with Boolean logic indicators between them. IR systems will match the terms in the query statement with the precompiled indexes they maintain to find the documents that contain these terms. Once the search is complete, browse capabilities enable users to determine which items are of interest and select those to be displayed with a summary in one of the two ways: line item status and data visualization. Since searches tend to return many irrelevant items, browse capabilities, such as ranking, zoning and highlighting,
can assist users in focusing on items that have the highest likelihood in meeting their needs [37].

IR was viewed as a narrow area of interest mainly to librarians and information experts for some years. In the beginning of the 1990s, a single fact changed the perception -- the introduction of the World Wide Web. The Web is becoming a universal repository of human knowledge and culture which has allowed unprecedented sharing of ideas and information in a scale never seen before. Despite such success, the Web has introduced new problems of its own. Due to the huge amount of unstructured data and the diversified user backgrounds, finding useful information on the Web is frequently a tedious and difficult task. Information Retrieval on the Web has been trying to solve these problems. Generally speaking, it covers the following issues: Web navigation, indexing, query languages, query-document matching, ranking query output, and user-relevance feedback [30]. Recently great improvement has been made by the following efforts.

One successful paradigm for making the mass of information on the Internet searchable and comprehensible is by classifying them into topics of hierarchical specificity. Hierarchical classification of this kind has been used in collections of IBM's patent documents (http://www.ibm.com/patents), Library of Congress Catalog, and Internet search engines such as Yahoo! (http://www.yahoo.com/) and Infoseek (http://infoseek.go.com) that categorise the content of the Web. Some hierarchical organizations were constructed manually. However, the exponential growth in the volume of on-line textual information makes it a pressing need to maintain such organization automatically. The work of [11] constructs a classifier at each split of the class hierarchy. This approach is local in that the construction at
each split is based on the local information at that split. To improve on the limitations local classifiers bring, [61] casts the hierarchical classification as a flat classification with the proximity modeling the closeness of classes. The approach is global in that only a single classifier is built.

To better present the query results, three techniques, namely ranking, clustering and hierarchical organization, have been intensively studied. Relevance-ranking systems create an ordered list of search results [37]. Document-clustering systems create group of documents based on the associations among the documents [37]. Even if the grouping is meaningful, it may not correspond well to the user's query because clustering algorithms usually do not use information about the user's query in forming the clusters. *DynaCat* [53] uses a knowledge-based approach to organising retrieved documents. It dynamically categorises search results into a hierarchical organization that correspond to a user's query by using personalized knowledge of important kinds of queries, i.e. about the query types and category types, and a model of the domain terminology that is already existing.

Recently there has been a surge of research work at hyperlink topology in addition to text content of documents to enhance relevancy of retrieved documents. *Google* [51], a prototype of a large-scale search engine, makes heavy use of the structure present in hypertext to crawl and index the Web efficiently and produce much more satisfying search results than existing system. Its novel features include *PageRank*, which utilises the link structure of the Web to calculate a quality ranking for each Web page, and the association of anchor text with the pointed page to improve search results. *Information Management* group at *IBM Almadan* has also proposed various techniques to extract documents in the hypertext environment for
relevance and quality. Two pieces of work [15, 26] developed a notion of hyperlinked communities on the WWW through an analysis of the link topology. They view the communities as containing a core of central, authoritative pages linked together by "hub pages". Their algorithm finds such communities and returns pages with high authority value as the core of a community. **HyPursuit** [60], a hierarchical network search engine, uses both document contents and hyperlinks to cluster information servers and provides cluster-based information browsing, scalable query refinement, result set expansion, query routing and result set clustering. Other work directed at the integration of textual content and link information for the purpose of search and retrieval has been reported from [16, 27, 44].

As there are numerous heterogeneous sources on the Internet, **MetaSearchers** have been proposed and constructed to access multiple sources and provide users with a virtual integrated view. **STARTS** [25], **Stanford Protocol Proposal for Internet Retrieval and Search**, is an emerging protocol to facilitate three metasearching tasks, namely, source selection, query transformation and result merging. Luis Cravano et. al. gives good explanation of these tasks and surveys current work in [28]. Several metaseachers already exist on the Internet for querying multiple Web indexes. One is **MetaCrawler** (http://www.metacrawler.com) [58]. Given a query, it submits the query to every search service it knows in parallel. Upon receiving the hits from each service, it collates the results by merging all hits returned. To mitigate the side effect of returning more irrelevant hits, it uses both a powerful query syntax that specifies required and non-desired words, and expert options that allow users to rank hits by physical location, e.g. the user's country and logical locality, e.g. their Internet
domain. Another popular metasearcher is SavvySearch (http://www.savvysearch.com) [32]. It is designed to efficiently query other search engines by carefully selecting those search engines likely to return useful results and by responding to fluctuating load demands on the Web. Given a query, it makes decisions on how many search engines to contact simultaneously by resource reasoning and on the order to contact search engines by ranking them based on learned associations between search engines and query terms as well as recent data on search engine performance. To tackle the limited precision and vulnerability to keyword spamming of the above two metasearchers, NEC Research Institute developed a metasearch engine that is better in efficiency and precision by downloading and analysing each document and then displaying results that show the query terms in context [40].

To our understanding, there are some essential differences between current Web Information Retrieval and our proposed work.

Web Information Retrieval retrieves either documents or URLs of documents that are supposed to be relevant. When the returned hit list is of the magnitude of hundreds or thousands, it is very hard, if not impossible, for Web users to browse all of them and sift relevant documents out from all. On the other hand, our system returns precise query results so that users do not need to browse the long hit list and search inside pages to extract them out.

Techniques in IR require specific query requirements from users. On one hand, Web users have diverse backgrounds and it is impractical to expect from them much domain knowledge, e.g., the vocabulary of information suppliers to achieve high relevancy. On the other hand, keyword matching is inherently problematic in
that word sense is ambiguous and occurrence of a word can not determine the content. IR has been trying to improve precision by taking user feedback and reformulating queries or by improving the browse capability to better help users choose. However, the improvement is modest if we consider how many efforts casual users are willing and able to exert. Our observation is that even though users do not specify clearly what they want, their behaviors in browsing manifest the desired information. By learning from these behaviors, our system acquires the exact query requirements that can or can not be posted in keywords, which will probably improve the precision of queried results.

Another critical problem in IR is absence of a well-defined underlying data model for the Web, which implies that information definition and structure is frequently of low quality. Currently most systems are using the vector-space model that ignores rich information conveyed by hypertext. Hypertext is being intensively studied but there is little consistent vocabulary in the literature. Our system uses Segment Graph to capture the intra-document information of organization structure as well as text content and the inter-document linkage topology.

2.2 Intelligent Agents

A large number of intelligent agents have been developed and deployed in recent years to help users manage the growing amount of information on the Web. Most intelligent agents fall into one of the three categories.

Some agents rely on search engines and heuristics to find a specific type of targets, e.g. home pages or papers, for Web users. CiteSeer [7] is an autonomous Web agent for automatic retrieval and identification of interesting publications.
Given a set of keywords, the agent uses Web search engines and heuristics about publications to locate and download Postscript papers. After transformed to text files, the papers are parsed in order to extract information features such as the abstract and individually identified citations. Both features and citations are placed into an SQL database. Then the user queries the database to find relevant papers using keyword searches or by navigating the links between papers formed by the citations. In addition, the agent is capable of finding papers similar to a given paper. 

Ahoy! The homepage Finder [59] is a fielded Web service that embodies Dynamic Reference Sifting for the domain of personal homepages. Given a person’s name and institution, Ahoy! filters the output of multiple Web indexes (actually MetaCrawler) to extract one or two references that are most likely to point to the person’s homepage. The filtering is comprised of cross filtering based on a target person’s institution and e-mail address and heuristics-based filtering based on heuristics that deal with people’s names and the way most homepage are constructed.

Some other assistant agents help users in an interactive mode. They learn user profiles, get user feedback, and suggest interesting hyperlinks interactively while users are browsing the Web. WebWatcher [2, 35], acts as a learning apprentice. It provides interactive advice to the mosaic user regarding which hyperlinks to follow next, then learns by observing the user’s reaction to this advice as well as the eventual success or failure of the user’s actions. WebMate [17], another software agent that helps users to effectively browse and search the Web, extends the state of the art of Web information retrieval in several ways. First, it uses multiple TF-IDF vectors to keep track of user interests in different domains and learns automatically from each positive sample marked by the user. Second, it uses
the *Trigger Pair Model* to automatically extract keywords for refining document search. *Syskill & Webert* [54], a software agent that learns a profile of a user’s interest and uses this profile to identify interesting Web pages, is also worth mentioning though it clearly separates the learning and prediction processes. It compares six different algorithms from machine learning and information retrieval and finds that no one algorithm is clearly superior and naïve *Bayesian* classifier offers several advantages over other learning algorithm on this task.

The third category concerns adaptive agents that can work autonomously to find interesting pages. They employ some heuristics to spider over the Web, select some pages to present to users, and take their feedback to improve the heuristics used. Mark Balabanovic, et. al. proposed an adaptive agent that automates Web browsing to keep users abreast of new and interesting information [6, 10]. It does not require specific search requests. Instead, it presents users with a selection of documents it thinks they will find interesting. Then users evaluate each document, and the system adjusts its parameters in order to try to improve its performance. *InfoSpider* [20, 45], is a multi-agent system for online, dynamic information search on the Web. The user is initially asked to provide a list of keywords and a list of starting points. Each agent estimates the relevance of all neighboring documents by analyzing the text of document where it is currently situated. Based on these link relevance estimates, the agent follows one link to another document. An agent is made dead when the relevance of a document is lower than a threshold. Users can assess any visited document with relevant feedback that helps construct a feedback list of words and guides later search on unvisited documents. The output is a flux of
link to documents ranked according to some relevance estimate. A similar system, called Amalthea, is reported in [50].

Again there are some fundamental differences between intelligent agents and our proposed work. The first one is that they recommend interesting pages to a user while our system tries to extract succinct segments of documents that answer user queries well. The second is that intelligent agents are still using vector space model or converting HTML pages to text documents. Our system, however, is designed to make good use of the rich information conveyed by HTML. A sophisticated Segment Graph model is proposed to serve the purpose of query results retrieval. Last, intelligent agents often require a prior knowledge such as user profiles or make heavy use of heuristics for a specific domain. Given an ad hoc query from a casual user, they may not be adequate to retrieving relevant documents efficiently. Our approach does not require any a prior knowledge or heuristics for a particular domain.

2.3 Query Languages and Wrapper Generation

As we mentioned earlier, efforts from Database community on Web search and retrieval basically fall into two categories, namely Web query languages and wrapper generation.

Database people view the Web as a directed graph whose nodes are Web pages and whose edges are the links between pages. Web query languages formulate queries for retrieving certain pages in this paradigm. The queries can be based on the content of the desired pages and on the link structure connecting the pages. The simplest instance of this task, which is provided by search engines on the Web, is to
locate pages based on the words they contain. A simple generation of such a query is to apply more complex predicates on the contents of a page, e.g. concerning links and words. Finally queries may involve the structure of the pages. Query languages are classified into two generations [23].

The first generation, including W3QL [38], WebSQL [48] and WebLog [43], aimed to unify content-based queries and structure-based queries. They combine conditions on text patterns within documents with those on graph patterns describing link structure. WSQProposes to model the Web as a relational database composed of two (virtual) databases: Document and Anchor. The Document relation has one tuple for each document in the Web and the Anchor relation has one tuple for each anchor in each document in the Web. This relational abstraction of the Web allows the use of a query language similar to SQL to pose the queries. However, Document and Anchor relations are completely virtual and there is no way to enumerate them. Therefore, the WebSQL semantics depends on materialising portions of them by specifying documents of interest in the FROM clause of a query. A basic way of materialising a portion of the Web is by navigating from known URLs using path regular expressions. Another way is by content conditions of a keyword being mentioned in a document. The implementation uses search engines to generate candidate documents that satisfy the "mention" condition. W3QL is similar to WebSQL, with some notable differences: it uses external programs (similar to user defined functions) for specifying content conditions on files rather than building conditions into the language syntax and it provides mechanisms for handling forms encountered during navigation. WebLog differs from the above languages in using deductive rules instead of SQL-like syntax.
The second generation of Web query language, such as WebOQL [5], StruQL [22] and Florid [34], goes beyond the first generation in two significant ways. First it provides access to the structure of the Web objects that they manipulate. Second, it has the ability to create new complex structures as a result of a query. WebOQL proposes a data structure called Hypertree, which is an ordered arc-labeled tree with two types of arcs, internal and external. Internal arcs are used to represent structured objects and external arcs are used to represent references (typically hyperlinks) among objects. Arcs are labelled with records. Such a tree is build from an HTML file using a generic HTML wrapper. Sets of related hypertrees are collected into Webs. WebOQL is a functional language but takes the form of select-from-where clauses. It navigates hypertrees using regular expression. The navigation pattern is useful for extracting subtrees from trees whose structure is not known in detail and for iterating over trees connected by external arc. WebOQL also creates the query result by mapping a hypertree that satisfying query requirements to another hypertree. STRUQL is the query language of a Web site management system but is general enough for semistructured data based on a data model of labelled directed graphs. A STRUQL query is a set of possibly nested blocks consisting of where clause, create clause, link clause, etc. The where clause includes either membership conditions or conditions on pairs of nodes expressed using regular path expressions. It produces all bindings of node and arc variables to values in the input graph. The create clause constructs new pages and the link clause explicitly links newly constructed documents as users want, thus creating a complex data structure as the query result. FLORID differs from other query languages in that it uses deductive and object-oriented formalism F-logic.
Research work on wrapper generation tackles the fundamental difficulty in querying the Web. That is, Web pages are not well structured and there is no schema that describes the contents of Web pages. If the structures of Web pages can be constructed and the schemas are available, users can query them just like querying databases. The basic idea of wrapper generation is to exploit the formatting information on Web pages to hypothesize the underlying structure of a page. From this structure a wrapper that facilitates queries on the page is generated. One approach discussed in [3, 4] finds interesting tokens and the nesting hierarchy in a page using heuristics about font style, font size, indentation, etc. and generates a parser with the aid of Lex, Yacc that semi-automatically constructs a wrapper. A wrapper implementation toolkit is presented in [31, 52] for the Stanford TSIMMIS project. The toolkit contains a library for commonly used functions, such as for receiving queries from the application and packaging results, and a facility for translating queries into source specific commands and for translating results into a model useful to the application. It adopts a template based translation methodology. The wrapper implementor specifies a set of templates (rules) written in high level declarative language that describe the queries accepted by the wrapper as well as the objects it returns. Later application queries are matched with the templates to retrieve results. This method also extends the query capabilities of a source by making the unmatched predicates in an application query with the templates a filter query. A post-processing engine applies the filter query to results retrieved by the templates. Garlic wrappers [55], also use an object-based data model to construct wrappers that encapsulate legacy data sources and mediate between them and a middleware system to provide an integrated view of the data sources. W4F toolkit
[56] generates wrappers that consist of three independent layers: retrieving HTML content, extracting information using user specified extraction rules written in HEL (HTML Extraction Language), and mapping information to structures exported by the wrapper to upper-level applications.

In building wrappers, it is often needed to divide source documents into chunks of information that correspond to records, which is called the task of record identification. One approach to discover boundaries of records in a Web document uses a set of individual heuristics and combines these heuristics into a method to perform the task [21]. It first heuristically builds a "tag tree" based on the nested structure of start- and end-tags, and locates the subtree that contains the records of interest. It next applies five different heuristics that individually attempts to locate a separator tag among the candidate tags. Finally a way combines these individual heuristics to determine a consensus separator tag and hence discover the boundaries. Another piece of work, originated from recasting information extraction in the framework of classification, employs relational learning to identify boundaries of a target field [12, 24]. It generates two types of features, simple for mapping a token to a discrete value and relational for mapping a token to another token. It also defines a set of predicates that make up rules for boundary finding. To construct rules, the algorithm starts with all uncovered samples, and adds predicates greedily attempting to cover as many positive and as few negative samples as possible. It is similar to what FOIL does.

Wrappers often serve as components of a larger system, called mediator, for information integration and query processing on the Web. Mediators perform the similar tasks as metasearchers, namely source selection, query transformation and
query result merging. The only difference is that metasearchers work at unstructured
data while mediators deal with more structured data [28]. Ariadne [1] is a system for
building mediators that can gather and integrate information from multiple Internet
sources and retrieve information from them in a database-like manner. The system
capabilities include: modeling data by the LOOM knowledge presentation system so
that users post queries against this integrated view of data; query planning that
combines source selection and traditional query optimization; wrapper generation
for a particular domain of interest by a machine learning technique and object
identification across sources. TSIMMIS [29, 33], the Stanford -IBM Manager of
Multiple Information Sources, is another system for integrating information. It offers
a “lightweight” object model called OEM (Object-Exchange Model) that serves to
convey information among components and does not require strong typing of its
objects. OEM objects have the following four components, Object ID, Label, Type
and Value. The system also provides a high level language, MSL, which is an object-
oriented, logical query language targeted to the OEM data model. A query consists
of rules. Each rule consists of a head that describes objects made available by the
mediator, and a body that describes conditions satisfied by the source objects. To
integrate information from heterogeneous sources, the system also offers tools for
generating wrapper automatically as described earlier in this sub-chapter.

There are some limitations, especially in scalability, of the database approach
to Web query processing. First wrapper generation is only feasible for a certain
number of Web sites in a particular domain. We believe that the Web is expanding
much faster than wrappers can be generated for it. Therefore the majority of Web
sources still present their contents in an unstructured form, which is likely to remain
for many years. Besides, Web query languages require much knowledge about the content and the structure of a Web site, which is hard for casual Web users to have. Largely due to this reason, those languages are mostly used for Web site management instead of Web query processing in a very large scale. Unlike the database approaches, our proposed approach first utilizes keyword queries and global search engines to return a candidate set of relevant documents. This first step ensures good scalability (certainly it is restricted by the scalability of search engines but metasearchers can solve the problem). Then our system elaborately explores the content and structure information of these documents and searches their neighborhood to retrieve succinct but precise query results. The second step provides means to achieve high precision of returned query results.
CHAPTER 3

A LEARNING BASED WEB QUERY PROCESSING APPROACH

In this section, we describe the architecture of a learning-based Web query processing system and explain how a user query is processed.

3.1 A Learning-Based Web Query Processing System

Figure 1 depicts the reference architecture of a learning-based Web query processing system, a detailed description of which appears in [14]. It consists of seven major components: User Interface, Session Controller, Query Analyzer, Learner, Locator, Retriever & Parser, and Knowledge Base. The User Interface provides users with a friendly environment to work with the system. It accepts user queries and presents results to them. A browser with extended capability to capture user actions is also an important component of it. When a user is browsing Web pages, it records three types of user actions:

1. following a hyperlink to browse another page;
2. marking a segment that contains the required information; or
3. rejecting a site that does not contain the required information.

Query Analyser analyses a user query and converts it into a search condition according to the requirement of the search engine that is employed to return URLs from the Web. The set of URLs is passed to Session Controller as the input of the other components in the system.

23
As a learning-based system, the system can work in two different modes, the learning mode and the processing mode. When the system works in the learning mode, Learner is activated by Session Controller. From those captured actions and located query results, it generates knowledge about both the query requirements and the navigation heuristics to find a segment within a page that satisfies the requirements. The learned knowledge is stored in Knowledge Base. When the system works in the processing mode, Locator is activated. It applies the stored knowledge to Web pages reachable from a URL returned by the search engine (defined as a site later), and locates the segment that contains the required information. The two working modes are switched back and forth based on training strategies. The main task of Session Controller is to co-ordinate the interaction among various components of the system.
Since the result from the search engine is a set of URLs, a Retriever is integrated into the system to retrieve Web pages. The Parser parses each retrieved page and generates an internal data structure that will be used later for presentation, knowledge learning and information location.

3.2 A Query Session

![Diagram](image)

Figure 2: A Query Session

To have better understanding of how a query is processed, we describe a query session in detail. As described earlier, given a user query, the Query Analyser pre-processes it and sends the search requirement in the form of keywords to a general search engine. The returned result from the search engine is a set of the URLs of Web pages ranked in a certain order. After Session Controller receives the URLs, the system works in either the *learning mode* or the *processing mode*. Corresponding to these two modes, there are two types of processes: the *browsing process* and the *locating process*. Figure 2 depicts the details of the processes and
associated data flow (Query Analyser, and Retriever&Parser are omitted in this figure).

Session Controller activates the processes according to a training strategy stored in its *Training Strategies Module*. A training strategy defines when the learning process should be invoked and how the learning mode and processing mode are interleaved. Three strategies supported by the system are:

*Sequential training*. It partitions the URLs returned from the search engine into two sets in the original order. The first set is used for training. The system works in the learning mode until all the sites for training are browsed by the user. After training, the obtained knowledge is used to automatically process the second set of URLs. The system remains in the processing mode until the whole query session is completed.

*Random training*. Similar to sequential training, the system first works in the learning mode and then turns to the processing mode. The difference is that, instead of using the first group of Web sites for training, it randomly picks a number of sites from the returned URLs. The rationale is that the randomly picked sites may be more representative than those on the top of the returned list.

*Interleaved training*. Different from using the above two strategies, the system switches back and forth between the learning mode and the processing mode before a stopping criterion is met when interleaved training is used. It works as follows. At the beginning of a query session, as no knowledge is available, the system is in the learning mode. After a few sites are browsed, the system accumulates some knowledge and then works in the processing mode. As long as the
user confirms the results located by the system are correct, it remains in the processing mode. When the user finds an incorrect decision made by it, the system will switch to the learning mode to learn from the incorrectly processed site. This process goes until a stopping criterion for interleaved training is met. Such a criterion may be a certain number of browsed sites or an accuracy threshold specified by the user. After that, the system remains in the processing mode until all sites are processed.

During a browsing process, given a URL, Session Controller first asks Retriever & Parser to retrieve the page and transform it to a segment tree. It adds the tree to the segment graph, an internal data structure maintained by the Segment Graph Module (segment tree and segment graph are defined in the next section). Then the controller sends the tree to Browser where the tree is presented for browsing. If the user chooses a link, the system goes to process a new page. The process is repeated until the user marks a query segment or rejects the site. User behaviors, either choosing a link or marking a segment, are recorded on the segment graph. For a successfully located site, the controller generates intermediate files, knowledge scripts, from the segment graph. The scripts are finally sent to Learner for knowledge generation.

In a locating process, given a URL, Session Controller receives a segment tree from Retriever & Parser and adds the tree to the segment graph. Then it sends the tree to Locator. Locator returns a decision of choosing a link, finding a segment or rejecting the site. If a link is chosen, the system goes to process the new page and asks Locator to make another decision. The process ends when Locator finds a query segment or rejects the site. The located segment is sent to the Result Buffer Module.
that communicates with Query Result Presenter in the interface to present the result according to a training strategy. When Interleaved training is used and the stopping criteria for training is not met, Query Result Presenter asks the user to check results. If she/he finds the system returned a wrong result, a browsing process is activated for the current site. The only difference from a normal browsing process is that some pages can be fetched directly from the segment graph.

Therefore, the URLs returned from the search engine are either browsed by the user during the learning process or processed by Locator during the locating process. A query session terminates when all the Web sites are either browsed or processed. If there are too many sites returned, certain heuristics can be used to determine when a session should be completed.
CHAPTER 4

WEB DATA MODELING

Our objective is to enable users to search the Web in the similar way as querying a database. To this end, the data on the Web should be properly modeled so that the system can identify the required information.

The Web consists of a number of Web pages connected by hyperlinks. In order to obtain queried information from a large number of Web pages, both the internal structure of a Web page and the linkage between Web pages need to be captured. We begin with the modeling of a single Web page.

4.1 Modeling A Web Page

Usually a Web page is an HTML document that contains a sequence of tag delimited data elements. As an atomic element, one that does not contain other elements in it, may not contain enough information to meet the query requirements, segment that is a group of elements is used as the unit in our model. In other words, we partition documents into segments each of which serves as a candidate of the answer to a query. Four major segment types are paragraph, table, list and heading. Detailed definition of segments is presented in Appendix A. Segments can be nested, that is, a segment can include a number of sub-segments. An HTML document is the largest segment. Each segment has one attribute, content, which consists of all textual data in the scope delimited by the start tag and the end tag of the segment, thus including the content of its sub-segments. Content is used to check if a segment meets the query requirements. To facilitate navigation, another attribute, description is
designed for each segment (the use of this attribute will be discussed in §5.1). It is summarized from the content using certain heuristics. For example, the description of a table segment can be the table caption or the title row. Heuristics for generating segments is also listed in Appendix A. Hyperlinks, a special type of elements in HTML pages, can be represented in the same way as segments. The anchor is its description and the URL is its content. To take advantage of the hierarchical structure, each hyperlink is associated with its parent segment, i.e. the smallest segment that contains it.

Figure 3: An HTML Document and the Corresponding Segment Tree

With the notation of segments, a Web page can be modeled as a segment tree: The root is the Web page itself; the internal nodes are segments that contain sub-segments; the leaves are atomic segments, the minimal units in this model. Each node has two attributes and is associated with hyperlinks it contains. An example of

---

1 Whether anchor should be included in the parent segment is a technical issue. Currently we include it in if the parent segment contains other text.
HTML page and the corresponding segment tree is shown in Figure 3 (the attribute of description is omitted in this graph).

Segment tree is an internal data structure used to represent each Web page. The browser in our system transforms the segment tree to a text document and displays it in the form of a string of characters with some formatting as other popular browsers do. When the user marks a segment as a query result, he actually chooses a string of characters that may correspond to a segment or segments in the tree. It is the system’s responsibility to find segments that best matches the user-marked area. To do so, we must find a mapping between the marked text content in an HTML page and segments in a segment tree. The algorithm, adapted from 1-dimensional Orthogonal Range Searching and Intersection [64], is described as follows.

**Step 1: Parse an HTML document and generate the Segment Tree.** To facilitate range queries on the segment tree, we create temporary leaf nodes for text that does not belong to any leaf segments. As a result, the leaf nodes of the constructed segment tree make up a linear partition of the HTML document. All leaf nodes in the tree are sorted by their starting positions. Suppose there are \( N \) leaf nodes and their starting positions are \( P_1, P_2, \ldots, P_N \). Each leaf node is then represented by \( (P_i, P_{i+1}) \), where \( i = 1, 2, \ldots, N+1 \) and \( N+1 \) is the end position of the last leaf node. Each internal leaf node is represented by \( (P_i, P_{i+j}) \), where \( j \geq 2 \) and \( j \leq N+1-i \).

**Step 2: Search a Segment Tree with Range R.** When a user marks an area in an HTML document, the start position \( a \) and end position \( b \) of the marked area in the document can be returned by the browser. This step is to find a segment or segments in the segment tree that correspond to the range \( R(a, b) \). Recall that our goal is to
find relevant segments for training. Good navigation knowledge requires that the largest relevant segments instead of a large number of small relevant segments be reported because many small segments bring many words from their description to the navigation knowledge. To an extent, it is similar to the problem of finding the minimal cover in 1-dimensional orthogonal range search. Also consider that a marked area probably does not intersect with any segment properly. Consequently segments that intersect "well enough" with range $R$ should be reported. In short, given a query range $R(a, b)$, largest segments that intersect "well" with the range are to be reported. This task is accomplished by two sub-steps: First it finds the leaf nodes that positions $a$ and $b$ stab and then it traverses up along the stabbing paths to find largest segments that intersect "well" with $R$ (see Appendix B for the pseudo-codes). A final note is that if a temporary leaf node for text is reported by the range search, either its parent is finally chosen or it is discarded. It depends on how relevant the parent segment is.

4.2 Modeling a Web Site

Externally, Web pages are connected through hyperlinks. If we view a Web page as a node in a graph and a hyperlink as a directed edge from the page containing it to the pointed page, then Web pages in a site can be represented by a directed graph. If we further ignore backward links, links pointing to one part of the same page, and links pointing to pages outside the current site, a Web site can be modeled as hyperlink-connected segment trees, namely a \textit{Directed Segment Graph} or \textit{Segment Graph} in short. The entering page, one pointed by a URL returned form the search engine, is the \textit{root} of the graph. With such a model, \textit{site}, which will be used very
often later in this paper, refers to the collection of Web pages that are reachable from the root and of the same base URL as the root. We define the depth of a segment is the number of hyperlinks followed to reach the Web page that contains the segment. Note that segments on the same Web page may have different depths along different paths. Segments on the root page have depth 1. The Level of a segment is the minimal depth among all hyperlink paths in the segment graph.

We would like to emphasize that it is not our intention to provide a complete and sound model for Web pages and the Web. The sole objective of the above model is to facilitate the retrieval of meaningful query results in the form of segment that is small in size but carries sufficient semantics. The segment graph that combines the intra-document structure with the inter-document linkage can well serve the purpose. An example of such a graph is presented in Figure 4.

![Figure 4: A Segment Graph](image-url)
CHAPTER 5

LEARNING FROM THE USER FOR QUERY PROCESSING

To perform query processing in a paradigm described in the previous section, the system should have the knowledge about how to navigate through the Web and how to identify the required segment. The most efficient way to obtain such knowledge is to learn from users. In this section, we discuss what kinds of knowledge are needed and propose our approaches to generating the knowledge.

5.1 Knowledge for Locating Queried Segments

To perform the task of Web query processing, the system is doing something new. Hyperlinks and internal page structure have been extensively studied by others in previous work. However, they made a decision either among all links [2, 45] or among all segments [18, 57] but not from the mixture of links and segments. If our system could exhaustively search the segment graph and choose the most relevant segment from all in the graph, the problem would be simplified as hypertext classification. Unfortunately it is not feasible for an on-line query processor considering how large a segment graph can be. To restrict the search scope, a navigational path from the root should be terminated if the system locates a segment on a page that meets query requirements well enough to be the query result, or concludes the page is not relevant and will not lead to a relevant document. In other words, on each page, a decision of choosing a link, finding a segment, or giving up
this page should be made. A decision made between links and segments requires these two types of data structures are comparable.

Another observation of Web queries concerns the conventions of composing hypertext documents. Hyperlinks usually convey descriptive information about the query results. To be accurate, it describes the information of the pointed document but it is not the query result itself. In contrast, a segment that meets the query requirements must contain both the descriptive information and the query result itself. For example, we would like to retrieve admission requirements of graduate applicants. The anchor "Admissions" only tells the link points to a page related to admission requirements while the queried segment contains both the descriptive information of admissions and the concrete requirements such as GPA or test scores. Links and segments, two structures presenting different information are hardly comparable by one mechanism.

To make the query system workable, two types of knowledge are in need. One is Navigation Knowledge that only concerns descriptive information and helps find a path from a given URL to the queried segment on a Web page. The other type of knowledge assists to examine whether a segment meets the query requirements of both the descriptive information and the result. It is referred to as Classification Knowledge because it is in fact used to classify a segment into one of the two classes, containing or not containing the query result.

Attributes of content and description are designed for navigation knowledge and classification knowledge, respectively. Note that lengths of links and segments may differ remarkably. Assuming the description of a segment can summarize the semantics of the segment, we use it in navigation to avoid bias in learning that
element lengths would bring. Another rationale of using description is that in self-describing languages like Extensible Markup Language [62], element names serve naturally as element descriptions so that our model carries over directly to them.

5.2 Navigation Knowledge

Navigation knowledge is generated from user actions of following hyperlinks to locate queried segments. A path that starts from the entering page of a site and ends at the queried segment is called a navigational path, represented as \((link\rightarrow)\)* segment, where \(*\) means any number of occurrences. For example, if segment \(S_{41}\) in Figure 4 is a queried segment, one possible navigational path is \(L_2\rightarrow L_4\rightarrow S_{41}\). A hyperlink usually occurs in some segments in a document. Information of those segments also helps determine whether a link should be followed. To capture such information, we extend the navigational path with all segments that contain the links on the path, which is called extended navigational path. In our example, the extended navigational path to locate \(S_{41}\) is \((S_1\rightarrow S_{11}\rightarrow L_2) \rightarrow (S_3\rightarrow S_{31}\rightarrow L_4) \rightarrow (S_4\rightarrow S_{41})\). A segment or a link appearing on the extended navigational path is called a component of it, e.g. \(S_{11}, L_4, S_{41}\), etc. Extended navigational paths can be easily obtained from segment graphs in browsing processes.

To generate navigation knowledge from an extended navigational path, the first step is to assign a weight, denoted as \(W(\text{component})\), to each component on the path. This weight tells how closely a component is related to the query result. One intuition is that the closer to the queried segment, the higher weight the component gets. Then the issue is at what rate the weight of a component decays along the extended navigational path. Instead of hypothesizing the rate, we assume the queried
segment is the most closely related to itself (its weight is 1) and let the path length determine the rate. Suppose \( D \) to be the depth of a queried segment. On the \( i^{th} \) page along the path, \( N_i \) is the number of components appearing on the path. The weight of \( j^{th} \) component \((j \leq N_i)\) on the \( i^{th} \) page is given by:

\[
W(\text{component}_j) = \frac{(i-1)}{D} + 1/D * j/N_i .
\] (1)

The weighting scheme guarantees \((i-1)/D < i/D\), i.e. the weight of a component on the \(i-1^{th}\) page is less than that on the \(i^{th}\) page. The second term of the formula, \(1/D * j/N_i\), ensures with the same depth the more specific information a segment conveys, the more weight it gets. In other words, a child segment gets more weight than its parent. A link gets more weight than all segments containing it. Continue with the example in Figure 4. Some components on the path are assigned weights as follows:

<table>
<thead>
<tr>
<th>Depth:</th>
<th>Depth 1</th>
<th>Depth 2</th>
<th>Depth 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Path:</td>
<td>( S_1\rightarrow S_{11} \rightarrow L_2 )</td>
<td>( S_3\rightarrow S_{31} \rightarrow L_4 )</td>
<td>( S_4\rightarrow S_{41} )</td>
</tr>
<tr>
<td>( W(S_{11}) = 0/3 + 1/3 * 2/3 = 2/9 )</td>
<td>the 2(^{nd}) component at depth 1</td>
<td>( W(S_{41}) = 2/3 + 1/3 * 2/2 = 1 )</td>
<td>the 2(^{nd}) component at depth 3</td>
</tr>
<tr>
<td>( W(L_4) = 1/3 + 1/3 * 3/3 = 2/3 )</td>
<td>the 3(^{rd}) component at depth 2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The next step is to assign weights to terms that describe a component on the path. Since the attribute, \textit{description}, provides such descriptive information, we choose it to represent a component. Then for each component, only words that are in the description and consist of alphabetic letters are selected. A stop list and stemming are further applied to them. The derived words are called terms. In our algorithm, each term in the description of a component is assigned a weight,
\( w(\text{term}) \), which is equal to the weight of the component divided by the number of terms in the description:

\[
w(\text{term}) = \frac{W(\text{component})}{\# \text{terms in the description}}
\]  \hspace{1cm} (2)

It can be seen that the weight of a term is determined by the position of the component in which it appears as well as the number of terms in the description that describe the component together. Even if a term appears twice on the same path, the assigned weights will be different due to the different contributions. Term weight is accumulated through all browsing processes. Finally the navigation knowledge, represented as a set of \((\text{term}, \text{weight})\) pairs, is stored into the navigation knowledge base.

A note is that this method of learning navigation knowledge only employs learning for the positive class because it only learns from paths that lead to queried segments. Terms not on navigational paths get weight 0, which means they will not contribute to query result location. Ignoring negative class speeds up the learning process. The effectiveness of the learned knowledge using this method will be evaluated in experiments.

5.3 Classification Knowledge

The task of examining whether a segment meets query requirements is cast in the Bayesian learning framework because it has provided good performance in text and hypertext applications [12, 13]. Two different models, the multi-variate Bernoulli model and the multinomial model in this framework are reported in [49]. From their report, the multinomial model usually outperforms the multi-variate Bernoulli model
at large vocabulary sizes, and almost always beats the multi-variate Bernoulli model when vocabulary size is chosen optimally for both. We therefore adopt the multinomial model. The \textit{classification knowledge} is the knowledge that will be used by the Bayesian classifier.

Classification knowledge takes the form of a set of triplets, \((\text{feature}_i, N_{ii}, N_{i2})\), where \(N_{ii}\) is the number of occurrences of \textit{feature}_i in the content of queried segments, and \(N_{i2}\) is the number of occurrences of \textit{feature}_i in the content of segments that do not meet query requirements. Knowledge generation involves two basic issues, feature generation and selection of training samples.

This knowledge is generated from attribute \textit{content} of segments. For each segment, a set of \textit{features} is extracted from the raw text data of content, where a feature is the minimal piece of significant data like English words, integers, etc. To perform Web query processing, the system should not only deal with words as information retrieval system do, but also handles values and complex data types as database systems do. To this end, we define five basic data types of features: float, integer, English word (consisting of alphabetic letters), special word (consisting of alphanumeric letters) and special character. Some combanitions of the basic types are so frequently used that they have actually become conventions. To facilitate queries on these data, we define another four complex data types: date, time, email address, and telephone number. A lexical program using regular expressions extracts all these features.

To train the classifier, the queried segment is naturally the positive sample. As for negative samples, one intuition is to use all visited segments other than the queried one. A possible consequence is that the classifier is overwhelmed with
negative samples. By the observation that segments other than the queried one on the same page pose most challenges to classification, we only choose them as negative samples to train the classifier. Then the generation of classification knowledge is straightforward. During the browsing process, when the user marks a queried segment, the system collects feature\(_i\) in its content together with the number of occurrences \(N_{ii}\), and feature\(_j\) in other segments on the same page together with the number of occurrences \(N_{jj}\). For each feature, numbers of occurrences in both classes are accumulative through all browsing processes.

Since both types of knowledge involve terms, they are organized by Tries for efficient access.
CHAPTER 6

QUERY PROCESSING USING LEARNED KNOWLEDGE

In this section, we describe in detail how the learned knowledge is used to locate queried segments.

6.1 Algorithm for Locating Queried Segments

The system uses navigation knowledge and classification knowledge generated in browsing processes to locate queried segments without user involvement in the locating processes. Each locating process takes one URL returned from the employed search engine as the entrance to a site. Then it traverses the segment graph built on the fly. As a Web query processor, however, the system will perform badly if general graph searching approaches like breadth-first or depth-first search are used. Considering hyperlinks and segments on a page simultaneously further complicates the search process. The learned knowledge helps the system locate queried segments more efficiently and effectively.

By our approach, the choice between hyperlinks and segments on each page determines the navigation in a site. If a hyperlink is chosen, the locating process goes to the pointed page. Failure in finding the queried segment by following the link causes the locating process to make another choice between unvisited hyperlinks and not processed segments. If a segment is chosen, classification knowledge is applied to check if it meets the query requirements. If it does, it is returned as the query result and navigation terminates. Otherwise another choice is made between unvisited hyperlinks and not processed segments. If no result is found
after all links and segments are processed, the locating process backtracks to the page it last visited. The process is running in a recursive fashion.

The key issue is how to make a choice between hyperlinks and segments on a Web page. Navigation knowledge is used. It analyzes the descriptive information of links and segments on the same Web page, and assigns a weight to each of them. This weight uniformly $w(\text{term}) = W(\text{component}) / \#\text{terms in the description}$ tells how closely one element, either a link or a segment, is related to the query result. Then links and segments, the two different types of data, are sorted by the assigned weights. One basic idea is that a choice between them should be made by choosing the element with the highest weight.

Figure 1 presents the locating algorithm (it is a pseudo code and involves several components in the system). The input is a URL of the Web page to be processed and the output is a pointer to the returned segment. A NIL pointer means the result is not found. We use two stacks in our implementation, one for segments and the other for hyperlinks. Each stack element consists of two fields, a pointer to a segment or a hyperlink and an assigned weight.

In the algorithm, the URL is first passed to $\text{Retriever}\&\text{Parser}$ that retrieves the Web page and parses it into a segment tree (line 7). A $\text{Separate}$ function separates links from segments and store them in $\text{LinkStack}$ and $\text{SegmentStack}$, respectively (line 8). Note that segments are nested. This function pushes both atomic segments and segments that contain other segments into the stack. Because atomic segments may not carry sufficient information to be the query result, it is better to let classifier decide how large a segment answers the query. The $\text{ApplyNavigation}$ function assigns weights to segments and links using the
navigation knowledge and sorts them in a decreasing order of the weights in SegmentStack and LinkStack, respectively (line 11).

Each pass inside the while loop makes a decision between the link with the highest weight and the segment with the highest weight (line 13-15). If the weight of the segment is higher, function ApplyClassification applies classification knowledge to all segments on this page and determines if one of them meets the query requirements (line 16). Note that crawling a new page on the Internet takes far more time than processing a number of segments on the local machine. Besides, it is reasonable to assume if one segment provides best information about the query result among all links and segments on a page, it is more likely to find the result on this page instead of following a hyperlink. Classifying all segments instead of only the one with the highest weight makes location more efficiently and accurately. If the weight of the link is higher, the link is chosen. For the link pointed to an unvisited page, the algorithm calls itself to process the new page (line 20-21). The while loop goes until either a query result is found or both stacks are empty. In the former case, the result is returned immediately. In the latter case, the locating process backtracks. Function StopNavigation terminates the navigation if the locating process has already visited a certain number of pages and is still trying to visit more. It operates on a global counter of visited pages and returns value TRUE when the counter reaches a pre-defined number.
Algorithm LocatingProcess (URL: the URL of a page, QueryResult: returned value of the query)

begin

Stack SegmentStack, LinkStack;
Float SegmentMax, LinkMax;
Tree SegmentTree;

QueryResult := NIL;
SegmentTree := Crawl&Parser(URL);
Separate(SegmentTree, SegmentStack, LinkStack);
if (StopNavigation() == TRUE)
    PopAll(LinkStack);
ApplyNavigation(SegmentStack, LinkStack);
while ((QueryResult == NIL) AND (SegmentStack != NIL OR LinkStack != NIL))

    do
        SegmentMax := GetMaxScore(segmentStack);
        LinkMax := GetMaxScore(linkStack);
        if (SegmentMax >= LinkMax)
            ApplyClassification(SegmentStack, QueryResult);
            PopAll(SegmentStack);
        else
            URL := Pop(LinkStack);
            if (Unvisited(URL) == TRUE AND StopNavigation() == FALSE)
                LocatingProcess(URL, QueryResult);
        end if
    end do
end while
end.

Figure 5: Algorithm for Locating a Queried Segment on a Web Page

6.2 Application of Navigation Knowledge

Function ApplyNavigation assigns weights to segments and links and sorts them by the weights in a descending order. A critical problem in the locating problem is the
weight assignment. Navigation knowledge, which is a set of term-weight pairs, is a priori knowledge that is used to compute the weights of segments and links.

First, for a segment or a link we select terms from its description in the way as mentioned in § 5.2. The next step is to compute a weight from the selected terms represented as $term_1, term_2 \ldots term_n$. Again we use $W(component)$ to represent the weight of a segment or a link and $w(term)$ to represent the weight of a selected term. $W(component)$ is computed using the following functions:

$$W(component) = \max(topJ(w(term_1), w(term_2), \cdots, w(term_n))) .$$  \hspace{1cm} (3)

Function $topJ$ is designed to filter segments and links that are remotely related to the query result. As we know, during the training process, every term in a description obtains part of the weight of the component. All terms with weight higher than zero make up the vocabulary of navigation knowledge. As predicted by Zipf’s Law [63], a large number of terms have low weights and will be noise in the locating process. If a segment or a link is made up of such low weight terms, it is unlikely to be relevant to the query result and thus should be excluded from further consideration. To do so, we maintain a list of term weights that are top $J\%$ highest in the navigation knowledge base. The lowest weight in this list is a threshold. Function $topJ$ returns a set of term weights that are higher than the threshold. $J$ is set relative low so that $topJ$ usually returns an empty set for irrelevant segments and links. Currently $J$ is equal to 30. With an empty weight set, $W(component)$ is equal to 0. Segments or links with weight 0 will be removed from the stacks later in the sorting process.
The outer function $max$ is used to keep the most descriptive information in segments and links. Rather than using an average or other expressions of the weights returned by $topJ$, we choose the max function because the relevance of a segment or a link is often conveyed by very few words. To be accurate, the convention of HTML pages is using short informative sentence fragments [18], which is especially true to list segment, heading segment, hyperlinks, etc. As a result of this function, each segment or link is assigned a weight that best represents its relevance.

6.3 Application of Classification Knowledge

Function $ApplyClassification$ calls the naïve Bayesian classifier to apply classification knowledge to one or multiple segments. We begin with classification of one segment.

For each segment, feature generation mentioned in §5.3 is applied to the data of attribute content. A Bayesian classifier views each segment as a bag of features and assumes the occurrences of these features are independent. To perform classification, the following formulae (4) to (6) are adopted from [49] with a little simplification. The class label $C$ (queried segment, or not in this application) of a segment $D'$, is given by:

$$C = \arg \max \ P(C_k \mid D') = \arg \max \ P(D' \mid C_k) P(C_k)$$

$$= \arg \max \ P(F_1 \mid C_k) P(F_2 \mid C_k) \ldots P(F_n \mid C_k) P(C_k), \quad (4)$$

where $C_k$ is a class label ($k = 2$ in this application) and $F_i$ is a feature in the segment.
The estimation of the probability of feature \( F_i \) on condition of class \( k \) and each class prior are computed using the classification knowledge as follows:

\[
P(F_i \mid C_k) = \frac{1 + N_{ik}}{|V| + \sum_{i=1}^{[V]} N_{ik}}, \tag{5}
\]

\[
P(C_k) = \frac{\sum_{i=1}^{[V]} N_{ik}}{\sum_{k=1}^{K} \sum_{i=1}^{[V]} N_{ik}}, \tag{6}
\]

where \( N_{ik} \) comes from the triplet \((F_i, N_{i1}, N_{i2})\) with \( k=1,2 \) and \( \sum_{i=1}^{[V]} P(F_i \mid C_k) = 1 \).

To handle the probability of features that do not occur in training samples, smoothing of add-by-one is used. \(|V|\) is the vocabulary size of the classification knowledge.

Then we discuss how function ApplyClassification finds the query result. All segments on a page (both atomic and segments containing other segments) are sent to this function so that it is expected to find the segment that best meets the query requirements. In other words, in addition to the class label of a segment, the classifier should tell how confident the class label is assigned to it. To serve the purpose, we introduce to formula (4) one design parameter, confidence \( \alpha_k \) for each class label \( k \) with \( \sum_k \alpha_k = 1 \). Let \( E_k \) denote \( P(F_1|C_k)P(F_2|C_k)\ldots P(F_n|C_k)P(C_k) \).

For a certain segment, \( \alpha_k \) (\( k=1,2,\ldots,K \)) is calculated from the following equations:

\[
\frac{E_1}{\alpha_1} = \frac{E_2}{\alpha_2} = \ldots = \frac{E_K}{\alpha_K}, \quad \text{and} \quad \sum_{k=1}^{K} \alpha_k = 1. \tag{7}
\]
In the application of a query processor, suppose $C_i$ denotes the class of queried segment. Given a set of segments $D_1, D_2, \ldots, D_m$, function $ApplyClassification$ filters segments whose classification confidence $\alpha_j$ is lower than a threshold and chooses a segment $D$ with the largest $\alpha_j$ from all kept segments:

$$D = \arg \max_j \{ \alpha_j \mid \alpha_j > \text{Threshold}, j = 1, 2, \ldots, m \}, \quad (8)$$

where $\alpha_j$ is the classification confidence $\alpha_i$ with which class label $C_i$ is assigned to segment $D_j$. If all segments have $\alpha_i$ lower than the threshold, $ApplyClassification$ does not return any query result and the locating process goes on.
CHAPTER 7

PERFORMANCE EVALUATION

A prototype of the system has been implemented based on the proposed approach. The system is implemented in Visual C++ on a Pentium PC running Windows NT. Yahoo! is used as the external engine. The URLs of Web page matches (not site matches) are used in processing. A series of experiments were conducted to evaluate the proposed approach and study the related issues. In this section, we describe these tests and discuss the results.

7.1 Evaluation Metrics

For a given query and a URL returned from the search engine, the system either returns a segment or a message indicating that no result is found from the related Web site. We label the first case as *Found* and the latter one *Not Found*. For both cases, we use *Right* or *Wrong* to indicate whether the system makes a correct decision. Using the four terms, a query result belongs to one of the four categories defined in Table 1, where a queried segment is a segment that satisfies user query requirements.

<table>
<thead>
<tr>
<th>Right Found:</th>
<th>The queried segment is found.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wrong Found:</td>
<td>A segment other than the queried segment or from an irrelevant site is returned.</td>
</tr>
<tr>
<td>Right Not Found:</td>
<td>No segment is returned from an irrelevant site.</td>
</tr>
<tr>
<td>Wrong Not Found:</td>
<td>The system fails to locate the queried segment that the site contains.</td>
</tr>
</tbody>
</table>

Table 1: Terminology for Definitions of Evaluation Metrics
To evaluate effectiveness, the following metrics are defined:

\[
\text{Precision} = \frac{\text{# Right Found}}{\text{# Found}},
\]

\[
\text{Recall} = \frac{\text{# Right Found}}{\text{# Sites Containing Queried Segments}},
\]

\[
\text{Correctness} = \frac{\text{# Right Sites}}{\text{# Sites Processed}}.
\]

For correctness, \# right sites is the number of the sites for which the system makes correct decisions. That is, it either locates a queried segment, or indicates correctly that a site does not contain a queried segment.

To evaluate efficiency, we take the crawled pages as the measure because the time of processing a page is insignificant compared to retrieving the page through the Internet. Two measurements are defined. The absolute path length is the number of crawled pages to locate a queried segment or to conclude that no queried segment can be found for the site. The relative path length to locate a queried segment is the ratio between the absolute path length and the level of the queried segment (i.e. the length of the shortest path to locate this segment). The two metrics are presented as:

\[
\text{Absolute Path length} = \# \text{Crawled pages},
\]

\[
\text{Relative Path Length} = \# \text{Crawled pages} / \text{Level of the Queried Segment}.
\]

7.2 System Capability

Before quantitative analysis of the system performance, we first present sample query results that indicate the capability of the system. A user posted a query consisting of 3 words, Hong Kong hotel, with the intention of finding hotel room rates in Hong Kong. The query was passed to Yahoo! and a set of URLs was returned, which is shown in the left frame of Figure 6, a snapshot of the system
output. The right frame shows the query results after seven successful browsing processes during which the system learned the knowledge about the query. Currently the right frame presents the results located for Site 34 and 35. From the results we can see some of the novel features of the system.

![Image of query results]

Figure 6: A Sample Output

1. In addition to URLs and page titles that ordinary search engines can return, our system returns segments of the Web pages that contain queried information. URLs, short description of the page and the queried segment make up a complete answer to a Web query.

2. The query results contain exactly the information that the user is querying for. Note that the segments from site 34 and 35 do not contain any input keyword and meet the requirement of room rates that are not specified in the keyword query. It indicates that the system learned the query requirement from the user.
3. Both segments are from pages whose URLs are not directly returned by *Yahoo!*. It indicates that the system learned how to follow the hyperlinks to the page that contains a queried segment.

7.3 **Effectiveness of the System**

In this subsection, we present the results of two queries to illustrate the effectiveness of the system. Two queries used are:

- Q1: Hong Kong hotel room rate,
- Q2: Hong Kong hotel.

<table>
<thead>
<tr>
<th>Query</th>
<th>URL Returned</th>
<th>URL Selected</th>
<th>URL Used</th>
<th>Training</th>
<th>Testing</th>
<th>Irrelevant Sites</th>
<th>Relevant Sites</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1. &quot;Hong Kong hotel room rate&quot;</td>
<td>424</td>
<td>100</td>
<td>69</td>
<td>9</td>
<td>60</td>
<td>31</td>
<td>29</td>
</tr>
<tr>
<td>Q2. &quot;Hong Kong hotel&quot;</td>
<td>69533</td>
<td>100</td>
<td>71</td>
<td>9</td>
<td>62</td>
<td>24</td>
<td>38</td>
</tr>
</tbody>
</table>

Table 2: URLs Used in Query Processing

The intention of the user was to locate the room rates of hotels in Hong Kong. Processing and usage of the URLs returned from *Yahoo!* is summarized in Table 2. Only 100 top-ranked URLs were chosen for processing. Among them, some URLs were removed for a fair evaluation. Examples of removed URLs include non-accessible ones, duplicates, URLs pointing to non-HTML documents, URLs pointing to non-English documents, etc. The sequential training strategy was used

---

2 Processing dynamic Web pages is still under development.
and the number of URLs used for training is shown in column *Training*. The relevancy of a site was determined by examining page contents manually.

<table>
<thead>
<tr>
<th>Query</th>
<th>Found</th>
<th>Not Found</th>
<th>Effectiveness Measure (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Right</td>
<td>Wrong</td>
<td>Correctness</td>
</tr>
<tr>
<td>Q1. “Hong Kong hotel room rate”</td>
<td>23</td>
<td>7</td>
<td>26</td>
</tr>
<tr>
<td>Q2. “Hong Kong hotel”</td>
<td>28</td>
<td>4</td>
<td>21</td>
</tr>
</tbody>
</table>

Table 3: Basic Performance of the System

The results of the experiments are summarized in Table 3. It can be observed that, with sequential training, the correctness for both queries reaches about 80%. Considering the big discrepancy between the keywords expressed in a user query and the exact requirement, the results are really encouraging. It justifies the basic approach described in the previous sections.

To understand why the system failed to locate some queried segments, we examined the cases where the system chose segments that did not meet query requirements. The major reason is that those returned segments contain much noise, namely the words that are very close to those used to locate queried segments. One returned segment is as follows:

"**QUEEN ELIZABETH 2. Standby fares for the six day transatlantic crossings on this famous ship are available on Nov. 21 (New York to Southampton) for $1,199 per person, double occupancy and on the Dec. 14 sailing (Southampton to New York) for $1,099. The single supplement is $350.""

53
This segment contains words “single” and “double”, sign $ and numbers that are important clues to locate segments for room rates of hotels.

There are a number of reasons that the system failed to locate a queried segment. Among four such cases of query 1, three of them were caused by the low confidence of being classified as queried segments, which indicates there is deficiency in the generated classification knowledge. Another such case resulted from deprecated use of HTML tags in the original document: A hyperlink uses an image element as the content but leaves the attribute ALT of tag IMG blank. As a result, the system could not find any descriptive information about the link and thus ignored the path that leads to the queried segment.

7.4 Effectiveness of the Knowledge

To verify the effectiveness of using the two types of knowledge, navigation and classification, we modified the system to apply only one type of knowledge in the process of learning and locating.

The naïve Bayesian model was chosen to build a classifier that only used classification knowledge to locate queried segments. The training samples of the classifier are obtained as follows: during the browsing process, if a link is chosen, anchor of this link is a positive sample and other links on the page are negative samples. If a segment is marked as the query result, the segment is a positive sample while other segments on the page are negative ones. When a page is processed, the classifier calculates the classification confidence $\alpha_i$ for all segments and links. Only those segments and links whose confidence is greater than the filtering threshold (see formula (8)) are kept. Among them, the one with the highest confidence is
selected. If a segment is chosen, it is returned as the queried segment. If a link is chosen, then the system follows the link to process a new page. If it fails to find a result by following the link, it chooses the one with the next highest confidence. If all kept links and segments are processed without any finding, the system backtracks to a previously visited page.

An alternative system only used navigation knowledge in query processing. During the browsing process only the navigational path is captured to generate navigation knowledge. Locating algorithm is modified correspondingly. It first applies navigational knowledge to all links and segments so that each of them is assigned a weight as described in §6.2. Among the segments and links with weight higher than zero, the one with the largest weight is chosen. If a segment is selected, the system returns the segment. If a link is chosen, the system follows this link and processes a new page. If it fails to find a result by following the link, it chooses a segment or a link with the next highest weight. If all kept links and segments are processed without any finding, the system backtracks.

<table>
<thead>
<tr>
<th>Accuracy</th>
<th>Query 1</th>
<th></th>
<th>Query 2</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Correctness</td>
<td>Precision</td>
<td>Recall</td>
<td>Correctness</td>
</tr>
<tr>
<td>Both Types of Knowledge</td>
<td>81.7%</td>
<td>76.7%</td>
<td>79.3%</td>
<td>79.0%</td>
</tr>
<tr>
<td>Bayesian Only</td>
<td>58.3%</td>
<td>51.2%</td>
<td>69.0%</td>
<td>38.7%</td>
</tr>
<tr>
<td>Navigation Only</td>
<td>36.7%</td>
<td>28.8%</td>
<td>55.6%</td>
<td>29.0%</td>
</tr>
</tbody>
</table>

Table 4: Effects of Systems Using Different Types of Knowledge
Two queries and the testing environment are the same as shown in Table 2. The results are summarized in Table 4. The results of using both types of knowledge are also included for easy comparison.

<table>
<thead>
<tr>
<th>Correctness</th>
<th>Query 1</th>
<th></th>
<th>Query 2</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Irrelevant</td>
<td>Level=1</td>
<td>Irrelevant</td>
<td>Level=1</td>
</tr>
<tr>
<td></td>
<td>31/60</td>
<td>27/60</td>
<td>24/62</td>
<td>12/62</td>
</tr>
<tr>
<td>Both Types of Knowledge</td>
<td>83.8%</td>
<td>81.5%</td>
<td>50%</td>
<td>87.5%</td>
</tr>
<tr>
<td>Bayesian Only</td>
<td>48.4%</td>
<td>70.4%</td>
<td>50%</td>
<td>33.3%</td>
</tr>
<tr>
<td>Navigation Only</td>
<td>21.2%</td>
<td>60%</td>
<td>0%</td>
<td>12.5%</td>
</tr>
</tbody>
</table>

Table 5: Analysis of Systems Using Different Types of Knowledge

It can be clearly observed that, the system employing both types of knowledge performs much better than those that employ only one type of knowledge. To analyze the reason for the poor performance of using only one type of knowledge, we classified sites by the level of the queried segment. In Table 5, the title cells specifies the level distribution and the fraction stands for among all test sites, how many of them belong to the level. Usually the coarser the query, the more sites belong to the level above one. The correctness of each group is reported in table cells.

One finding is that the system with one type of knowledge works reasonably only when the queried segment occurs on the first page. Its ability to filter irrelevant sites and to crawl through hyperlinks is very limited. In contrast, employing two types of knowledge manifests good performance of irrelevance filtering and navigation. Even if the queried segment is on the first page, using one type of knowledge is still less accurate then using two types of knowledge. The co-
ordination of navigation knowledge and classification knowledge provides a good way to process Web queries.

7.5 **Effects of Training Strategies**

Our system supports three training strategies, *sequential training*, *random training* and *interleaved training*. This group of experiments was designed to find out which training strategy best served the purpose of query result location in terms of both accuracy and efficiency. Besides the training strategies, the training size was another focus of this part. In a query system, asking users to browse many sites is impractical and may not guarantee the desired performance. In our tests, we varied the training size from 3 to 10 for each training method. For interleaved training, the stopping criterion of training was a pre-defined number of browsed sites, i.e. the training size. The two queries of hotel room rates were used again. Figures 7 shows the performance in terms of accuracy\(^3\).

Experimental results of query 1 and query 2 exhibit consistency in the following aspects.

1. Random training performs badly, especially low in recall. With a low recall, precision is not very significant even if 80% can occasionally be achieved. Assumption that randomly picked sites are more representative than those on the top of the returned list is not true. Because URLs returned by a research engine are ranked by scores of relevance, sites ranked at the top are more typical of the

---

\(^3\) Points on the curves of sequential training show values a little different from those in previous section due to the connection problem of some sites at the different running time
query requirements. Using randomly picked sites makes it slow to accumulate knowledge.

Figure 7: Accuracy of Training Strategies

2. With a relatively small training size, the difference between sequential training and interleaved training is not obvious, because they both choose the first few sites for browsing. As the training size increases, the difference is enlarged. With 10 training sites, interleaved training beats sequential training by 10% in all three metrics. We own the better performance of interleaved training to its way of updating knowledge. With modest knowledge, the system makes a mistake when
the site presents something new to it. At this time new knowledge needs to be added. Interleaved training does it. Sequential training, however, blindly adds knowledge from the first few sites. Even with a fairly large number of training size some necessary knowledge is still missing.

3. In this experiment, best accuracy reaches or exceeds 90% in all metrics when the interleaved training strategy is used.

One metric that evaluated efficiency of the system was relative path length for a right “Found”. When sequential training and interleaved training were used, relative path length for a right “Found” segment was always in the range of 1.0 to around 1.2, which shows the system is quite accurate at finding the navigational path and thus efficient in extracting the segment. Using random training the system often crawled far more than necessary. Graphs are omitted in the interest of space.

![Graph 1: Query 1 - For a Right “Not Found” (3-10)](image1)

![Graph 2: Query 2 - For a Right “Not Found” (3-10)](image2)

Figure 8: Efficiency of Training Strategies

Figure 8 shows the system performance with respect to the other metric: absolute path length for a right “Not Found”. As the training size grows, to conclude for an irrelevant site, the system crawls more pages using any training strategy. That is caused by increased navigational knowledge. When the knowledge is limited, the system only checks the first page and then gives this site up. With more navigation knowledge, the system tries more pages before it identifies the
irrelevance. In the process of enlarging training size, however, only the system using interleaved training maintains the absolute path length in a range between 1 and 1.5 pages.

![Graphs showing the effects of training strategies with enlarged training sizes.]

Figure 9: Effects of Training Strategies with Enlarged Training Sizes

We further investigated if enlarging the training size could improve the performance of sequential training and random training. The training size was increased from 10 to 20. Experimental results are presented in Figure 9. From the two graphs concerning correctness, the system using random training improves correctness whereas that using sequential training worsens it by 10%. By our previous explanation, at training size 10, random training does not accumulate...
enough knowledge. More training sites increase the knowledge and enhance accuracy. Performance deterioration of sequential training is caused mostly by the decreasing recall as depicted by the graphs concerning recall. It results from too many features involved in the Bayesian classifier. Feature selection from the vocabulary is one solution if such large training size for one query is needed. The last two graphs of Figure 9 show that the absolute path length for a right "Not Found" goes up remarkably when the training size increases from 10 to 20.

The results for this set of experiments can be summarized as follows.

1. In terms of accuracy the best performance is achieved by interleaved training. The optimum includes over 90% in correctness and approximately 90% in both precision and recall.

2. The system almost always navigates through the shortest path to locate a queried segment using interleaved training. To identify an irrelevant site, it uses not more than 1.5 pages.

3. Enlarging the training size for random and sequential training is not effective. Their accuracy is still 10% to 20% below the optimum and sequential training even worsens the performance. Large training size also causes more page crawling to identify an irrelevant site.

A final note is that, when interleaved training strategy was used with 10 training sites, the number of wrong decisions of query 1 was reduced from 11 to 4 (refer to Table 3). Two wrong "Found"s were caused by noisy words and prices in the segments. One wrong "Not Found" resulted from the low confidence gained from the classifier and the other wrong "Not Found" from the use of an image as
mentioned earlier. Obviously the knowledge learned by using interleaved training improved greatly.

7.6 Experiments on a Range of Queries

We made some observations of issues related to implementing a Web query processor from two queries concerning room rates of hotels. To better evaluate the effectiveness and efficiency of such a processor, we tested a range of queries on it. Query results from the Web would be domain specific and vary with Web site organizations. However, aimed at processing ad hoc queries, the system was expected to be workable and provide reasonable performance in such a challenging environment.

Three query requirements with distinct characteristics were selected.

Query 1: room rates of Hong Kong hotels (included for comparison). It targets at prices, which is well defined and easy to identify by a user during the browsing process. The system is expected to extract the price information during the locating process.

Query 2: admission requirements. A user may not know the exact query requirements when she/he issues the keyword query. During the browsing process, she/he makes it clear that the requirements concern concrete items such as degree, GPA, GRE, TOEFL, etc. The system is expected to extract these items as query results. Compared to Query 1, the query results have larger variance because they may contain different sets of items as the need is.
Query 3: data mining researcher. The query target is in fact a concept. The user wants to extract segments of pages as evidence that a person is a data mining researcher. It is even hard for human readers to tell what these segments should contain and such decision is very subjective. During the browsing process, the user gets to know a data mining researcher can be reflected by research interests, research projects, professional activity, etc. The system is expected to identify the pieces of the evidence, associate them with data mining and return the correct segments.

For the first two query requirements, we issued two keyword queries with different precision. All five keyword queries are listed in Table 6. For each keyword query, we collected the top 100 URLs returned by Yahoo! and cleaned them for later tests\(^4\). The interleaved training strategy was used with training size of 10.

<table>
<thead>
<tr>
<th>Query Requirement (QR)</th>
<th>Keyword Query KQ1</th>
<th>Keyword Query KQ2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: room rates of Hong Kong hotels</td>
<td>“Hong Kong hotel room rate”</td>
<td>“Hong Kong hotel”</td>
</tr>
<tr>
<td>2: admission requirements on graduate applicants</td>
<td>“requirements graduate applicant”</td>
<td>“graduate applicant”</td>
</tr>
<tr>
<td>3: data mining researcher</td>
<td>“data mining researcher”</td>
<td></td>
</tr>
</tbody>
</table>

Table 6: Query Requirements and Keyword Queries

Intuitively the third query is the most difficult one. Some samples of the returned segments are shown in Figure 10. For sample one, the system figured out the evidence was research interests and the interests included data mining. For the

\(^4\) Manual check of each site in the test prevented us from enlarging the URL list to a size comparable to that of a database. We assume the system will process other URLs with the same performance as it does with the first 100.
second sample, both research interests and projects were captured. For sample three, the evidence of a data mining researcher was professional activity. The system also filtered a number of irrelevant sites that contain "data mining" but have nothing to do with a researcher. It is shown that the system is able to learn and apply knowledge even for such a challenging query.

Sample 1:
Page Title: Rakesh Agrawal's Home Page

Content:
I'm Rakesh Agrawal, a researcher at the IBM Almaden Research Center in San Jose, California. My current research interests include data mining, text and web mining, OLAP, and electronic commerce. The technology we developed in our Quest project forms the core of the IBM data mining product, IBM Intelligent Miner.

Sample 2:
Page Title: Pirjo Moen

Content:

<table>
<thead>
<tr>
<th>Research interests</th>
<th>Projects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data mining</td>
<td>FDK Data mining</td>
</tr>
</tbody>
</table>

Databases Knowledge Bases Hypertext

Sample 3:
Page Title: Microsoft Research: Researcher Profiles

Content:
In 1994 and 1995, Usama was program co-chair of the International Conference on Knowledge Discovery and Data Mining (KDD). He served as general chair of the KDD-96 conference, is an editor-in-chief of the new technical journal Data Mining and Knowledge Discovery (http://www.research.microsoft.com/datamine), and co-edited the recent MIT Press book: "Advances in Knowledge Discovery and Data Mining."

Figure 10: Sample Results for "Data Mining Researchers"

The quantitative results are presented in Table 7. Our system works well for the first 4 queries. Accuracy is above 80% and in some queries it reaches 90%. For the last query, precision and recall are not as high as those are in other queries. Fortunately the system is still capable of filtering out irrelevant sites, thus to make
the correctness reasonably good. The relative path length to locate a queried segment is close to 1. The absolute path length in an irrelevant site is no more than 2.5 pages.

Another observation is that the performance of our system is not affected much by how precise the keyword query is. Thus users do not need to worry about the exact words when they issue the queries. In the browsing process they can specify the requirements by their actions. The system will learn them and enforce them in the locating process.

<table>
<thead>
<tr>
<th></th>
<th>QR1</th>
<th>QR2</th>
<th>QR3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>KQ11</td>
<td>KQ12</td>
<td>KQ21</td>
</tr>
<tr>
<td>Correctness</td>
<td>0.93</td>
<td>0.90</td>
<td>0.84</td>
</tr>
<tr>
<td>Precision</td>
<td>0.92</td>
<td>0.92</td>
<td>0.85</td>
</tr>
<tr>
<td>Recall</td>
<td>0.88</td>
<td>0.94</td>
<td>0.94</td>
</tr>
<tr>
<td>Relative Path Length (Found)</td>
<td>1.00</td>
<td>1.21</td>
<td>1.08</td>
</tr>
<tr>
<td>Absolute Path Length (Not Found)</td>
<td>1.30</td>
<td>1.57</td>
<td>2.50</td>
</tr>
</tbody>
</table>

Table 7: Results of Different Queries
CHAPTER 8

CONCLUSION

In this paper, we proposed a novel approach for processing queries on the Web. Taking such approach, a user can issue queries in free text sentences and get the results in the form of a list of segments containing the required information. Minimum involvement is required from the user to train the system. To process a query, a general-purpose search engine like Yahoo! is employed to get the initial relevant URLs, which are taken as the input of a series of browsing and locating processes. During the browsing processes, the system learns user requirements and the way she/he navigates through the hyperlinks to locate the segments that meet query requirements. During the locating processes, such learned knowledge is applied to locate the queried segments from a large number of Web pages without interaction of the user. The query results are returned to the user automatically after a query session terminates. Our preliminary experiments show encouraging results, which provides concrete evidence that the proposed approach is able to tackle the difficult problem of queries on the Web.

A prototype system has been developed based on the proposed approach. More comprehensive experiments are being conducted. Our future work also concerns knowledge representation and development of more sophisticated algorithms for learning and applying knowledge. To process a wide range of Web queries over diverse Web pages, page modeling is another issue that deserves further study.
REFERENCES


72
APPENDIX A: NOTES ON A SEGMENT TREE

In this work, each HTML page is modeled as a segment tree. A segment is a group of HTML elements and is used as the unit in query result location. Four major segment types, i.e. paragraph, table, list and heading, are derived from HTML 4.0 specification as follows:

<table>
<thead>
<tr>
<th>Segment</th>
<th>Start Tags</th>
<th>End Tags</th>
</tr>
</thead>
<tbody>
<tr>
<td>Paragraph</td>
<td>ADDRESS, BLOCKQUOTE, DIV, PRE, P</td>
<td>/ADDRESS, /BLOCKQUOTE, /DIV, /PRE, /P or %Segment_Start_Tag</td>
</tr>
<tr>
<td>Table</td>
<td>TABLE</td>
<td>/ TABLE</td>
</tr>
<tr>
<td>List</td>
<td>DIR, OL, DL, UL, MENU</td>
<td>/DIR, /OL, /DL, /UL, /MENU</td>
</tr>
<tr>
<td>Heading</td>
<td>Hi (i=1...6)</td>
<td>H1, H2, ..., Hi (i=1...6)</td>
</tr>
</tbody>
</table>

There may be text data in a document that do not belong to any segment. In this case we manually add a tag <P> before the text to make it a paragraph segment. For example, <TABLE>...</TABLE> ABC <UL>...</UL> is transformed to <TABLE>...</TABLE> <P> ABC <UL>...</UL>.

Segments can be nested, that is, a segment can include a number of sub-segments. The smallest segments are atomic ones that do not contain other segments in them. An HTML page is the largest segment. All segments obtained from a Web page make up a segment tree, where atomic segments are leaves, segments containing sub-segments are internal nodes and the page itself is the tree root.

Each segment has two attributes, content and description. Content consists of all textual data in the scope delimited by the start tag and the end tag of the segment.
Description summarizes the data in content and provides descriptive information for each segment. The heuristics for generating *description* comes from the observation of linguistic conventions used in HTML pages. Each segment type has its own heuristics presented as follows:

<table>
<thead>
<tr>
<th>Segment</th>
<th>Heuristics for Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Paragraph</td>
<td>The first and the last sentence in it</td>
</tr>
<tr>
<td>Table</td>
<td>Contents of CAPTION element and TH element. If they are not available, use text between the first pair of &lt;TR&gt;&lt;/TR&gt; and text between the first pair of &lt;TD&gt;&lt;/TD&gt; in each row.</td>
</tr>
</tbody>
</table>
| List      | DIR, OL, UL, MENU: text immediately ahead  
            | DL: text immediately ahead and contents of DT elements in it |
| Heading   | Content of the heading element, i.e. between &lt;Hi&gt; and &lt;/Hi&gt; |

In heuristics for generating the description of a list segment, the *text immediately head* is the text before the list segment that meets the following pattern:

```
{ Segment_Start_Tag|Segment_End_Tag|TR|TD|DT|LI } Text
```

*Immedately Ahead* {None|P|END_P|END_H} { %List }.

*Hyperlinks*, a special type of elements in HTML pages, are represented in the same way as segments. The anchor is its description and the URL is its content. To take advantage of the hierarchical structure, each hyperlink is associated with its parent segment, i.e. the smallest segment that contains it.
APPENDIX B: PSEUDO-CODES FOR RANGE SEARCH

ON A SEGMENT TREE

Sub-step 1: Find leaf nodes that position a and b of query range R stab.

//Suppose the current node is (c, d). Stabbed leaf nodes are found
//by calling the following functions recursively from the root node.

//Find the leaf node a stab
SegmentNo AStabs(c, d) {
    SegmentNo Found=-1;
    If (a>=d) return -1;
    If (c,d) is a leaf node {
        Found=GetSegmentNo(c,d);
        return Found;
    } else
        while (Found==-1 && there is unvisited child)
            Found=AStabs( the leftmost unvisited child);
    return Found;
}

//Find the leaf node b stab
SegmentNo BStabs(c, d) {
    SegmentNo Found=-1;
    If (b<=c) return -1;
    If (c,d) is a leaf node {
        Found=GetSegmentNo(c,d);
        return Found;
    } else
        while (Found==-1 && there is unvisited child)
            Found=BStabs( the rightmost unvisited child);
    return Found;
}
Sub-step 2: Find the largest segments that intersect well with \( R(a,b) \)

//Suppose \((c,d)\) is any tree node
//GoodSegment checks if a segment intersects well with \( R(a,b) \)
float OverlappingThreshold = 0.7
bool GoodSegment(c, d) { 
  int start, end;
  start = max(a, c);
  end = min(b, d);
  if ((end - start) / (d - c) > OverlappingThreshold)
    return true;
  else return false;
}

//Largest segments intersected well are reported by calling
//GroupA recursively from the leaf node stabbed by a and
//GroupB recursively from the leaf node stabbed by b.
//Suppose \((c,d)\) is any tree node
void GroupA(c,d){
  if (c<=a && d>=b) { // (c,d) covers \((a,b)\)
    if (GoodSegment(c,d)==true) {
      if (GoodSegment(Parent(c,d))==false) Report(c,d);
      else GroupA(Parent(c,d));
    } else {
      // (c,d) and (a,b) partially intersect
      if (GoodSegment(c,d)==true) {
        if (GoodSegment(Parent(c,d))==false) {
          Report(c,d);
          Report(RightSiblings(c,d));
        } /*1
      } else
        Report(RightSiblings(c,d));
    } /*1
    GroupA(Parent(c,d));
  } /*2
}
//GroupB(c,d) calls Report(LeftSiblings(c,d)) in lines *1.
//GroupB(c,d) calls GroupB(Parent(c,d)) in line *2.
//Report(RightSiblings(c,d)) reports only nodes left to b
//Report(LeftSiblings(c,d)) reports only nodes right to a