Omini: A Fully Automated Object Extraction System for the World Wide Web

David Buttler &**Ling Liu** &**Calton Pu**
Georgia Institute of Technology
College of Computing
Atlanta, GA 30332, U.S.A.
{buttler, lingliu, calton}@cc.gatech.edu

Abstract

This paper presents a fully automated object extraction system – Omini. A distinct feature of Omini is the suite of algorithms and the automatically learned information extraction rules for discovering and extracting objects from dynamic Web pages or static Web pages that contain multiple object instances. We evaluated the system using more than 2,000 Web pages over 40 sites. It achieves 100% precision (returns only correct objects) and excellent recall (between 93% and 98%, with very few significant objects left out). The object boundary identification algorithms are fast, about 0.1 second per page with a simple optimization.

1 Introduction

The amount of information on the web is growing at an astonishing speed. Search engines and browsers have become ubiquitous tools for accessing and finding information on the Web. Not surprisingly, the explosive growth of the Web has made information search and extraction a harder problem than ever. As of February 1999 [15], the publicly indexable web (static pages) contains about 800 million pages. No search engine indexes more than one sixth of the indexable web. Furthermore, search engines may not index new or modified (static) pages for months. To make the matter even worse, not only the number of static web pages increases approximately 15% per month, the number of dynamic pages generated by programs (i.e., the web pages behind the forms) has been growing exponentially. The huge and rapidly growing number of dynamic pages forms an invisible Web, out of the reach of search engines.

To address the search problem over dynamic pages, several domain-specific information integration portal services have emerged, such as excite's jango and cnet.com. These integration services offer an uniformed access to heterogeneous collections of dynamic pages using the wrapper technology [5]. A wrapper is an end-to-end computer program that performs two tasks. First it transforms a search request at the aggregation server to a search request at the remote information source provided by a content provider. Second, it converts the search result returned by the content provider into a normalized format for summarization and aggregation processing at the integration server. Most wrappers so far have been developed and maintained by semi-automatic wrapper generation systems [1, 2, 5, 14, 16, 10, 18]. Common techniques used for constructing wrapper programs often require embedding programmers' understanding of the specific presentation layout or specific contents of the Web pages. This turns out to be labor intensive and error-prone, especially for the web sites that are frequently changing their information presence on the Web. As a result, most of the information integration services do not scale. They have a hard time to effectively incorporate additional or new content providers into their existing integration access framework.

This paper presents a fully automated object extraction system – Omini. A distinct feature of Omini is the suite of algorithms and the automatically learned information extraction rules for discovering and extracting objects from dynamic Web pages or static Web pages that contain multiple data objects. The Omini system has been tested over more than 150 web sites by both end users and a wrapper generation system XWRAPElite [20], developed at Georgia Tech. Our algorithms for automatically learning object extraction rules are fast. The entire process is O(n), where n is the size (length in characters) of an input Web page. Our approach for extracting objects from Web pages using the automatically learned extraction rules is effective. We conducted a series of experiments over 2000 Web pages from 50 web sites, the results were consistent and satisfying, attaining recall ratio in the range of 93% to 98% and precision ratios 100% on all the sites we examined.

There are several research projects that have addressed the problem of information extraction from Web documents. To our knowledge, all approaches proposed so far discover and extract objects using either a manual approach or a semi-automatic approach. For example, [3, 10] discover object boundaries

manually. They first examine the documents and find the HTML tags that separate the objects of interest, and then write a program to separate the object regions. [1, 2, 5, 7, 12, 13, 14, 18, 19] separate object regions with some degree of automation. These approaches rely primarily on the use of syntactic knowledge, such as specific HTML tags, to identify object boundaries. They differ from each other in the degree of automation introduced in the data extraction process. In comparison, the approach developed Embley and his colleagues at BYU [7] has relatively higher degree of automation, although two out of five heuristics (ontology heuristic and identifiable tag heuristic) are based on pre-determined knowledge about the Web pages being extracted. As reported in [7], the ontology heuristic plays a critical role in achieving the high accuracy of their extraction approach, but it relies completely on human knowledge about the Web site and costly to develop (it takes about 2-man weeks to develop an ontology heuristic for a given web site [7]). Furthermore, they are error-prone with respect to changes in the content and presentation of the Web sites.

The Omini approach differs from these existing proposals in two distinct ways. First, Omini performs object extraction in two consecutive processes, object-rich subtree extraction and object separator extraction. Each process considerably reduces the number of possibilities considered in the next process. The goal of the object-rich subtree extraction is to locate the objects of interest in a page, while the goal of the object separator extraction is to find the object separator tag that can effectively separate objects. Second and most importantly, both extraction processes are fully automated. A set of heuristics and a mechanism to combine them have been developed for both subtree extraction and object separator extraction process.

Before explaining the details of our approach, we would like to note that fully automated approach to information extraction from Web pages is just one of the challenges in building a scalable and reliable information search and aggregation service for the Web. Other important problems include resolving semantic heterogeneity among different information content providers, efficient query planning and fusion for gathering and integrating the requested information from different Web sites, and intelligent caching of retrieved data. The focus of this paper is solely on automated information extraction.

The rest of the paper proceeds as follows. We briefly review a set of preliminary concepts in Section 2. We introduce the Omini system architecture in Section 3. Then we describe the object-rich subtree extraction algorithms in Section 4 and the object separator extraction algorithms in Section 5. Section 6 describes a combined algorithm that unifies the five independent object extraction algorithms. Section 6.2 reports the experiments and demonstrates the effectiveness of our object extraction approach through an analysis of our experimental results. We conclude the paper with a summary and an outline of future work in Section 7.

2 Preliminaries

2.1 Well-Formed Web Document

The web documents considered in this paper are HTML or XML documents. A web document consists of text and tags. A *tag* in a web document is marked by a tag name and an optional list of tag attributes enclosed in a pair of opening and closing brackets "<" and ">". *Text* is a sequence of characters in between two tags. By HTML [8] and XML [21] specification standard, most of tags in a Web document appear in pairs. A tag whose name does not start with a forward slash (i.e., "/") is called a *start tag*; otherwise it is called an *end tag* and the name of an end tag is the name of its corresponding start tag proceeded by "/". A web document is said to be **well-formed** if it satisfies the following conditions:

- There are no opening or closing brackets, < and >, in the text of the document that are not tags. Instead, these characters, when used in the text of a document, are encoded as < and >.
- All tags must be paired; namely every start tag has a corresponding end tag.
- All attribute values in a tag must be quoted (e.g. <a href="www.w3c.org">).
- All tags which do not normally have end-tags (such as <*IMG*>, <*HR*>, and <*BR*>) are immediately followed by a corresponding end tag. For example: <*BR*> will be denoted by <*BR*></*BR*>.
- Pairs of tags must be nested inside one another without overlapping. For example, the document fragment "<*a*> ... <*b*>...</*a*> ... </*b*>" is not well formed. The correct nesting for this example fragment is "<*a*> ... <*b*>...</*b*> ... </*a*>".

Documents that are not well formed can be converted to well-formed documents. We refer to such a transformation as document normalization. HTML Tidy [17] is a well-known Internet tool for transforming an arbitrary HTML document into a well formed one.

2.2 Tree Representation of Web Documents

A well-formed web document can be modeled as a tag tree. All the internal nodes of a tag tree are tag nodes and all leaf nodes are content nodes (numbers, strings, or other data types such as encoded MIME types). A *tag node* denotes the part of the web document identified by a start tag and its corresponding end tag and all characters in-between. A tag node is labeled by the name of the start tag. A *leaf node* denotes the content data (text) between a start tag and its corresponding end tag or between an end tag and the next start tag in a web document. A leaf node is labeled by its content. An example tag node in an HTML document is <*Title*> Home Page </*Title*>, where <*Title*> is the name of the tag node and the text string Home Page is a leaf node.

Definition 1 (Tag Tree)
A tag tree of a document D is defined as a directed tree $\mathcal{T} = (V, E)$ *where* $V = V_T \cup V_C$, V_T *is a finite*

set of tag nodes and V_C is a finite set of content nodes; $E \subset (V \times V)$, representing the directed edges. $\mathcal{T}$ satisfies the following conditions: $\forall (u,v) \in E, (v,u) \notin E$; $\forall u \in V, (u,u) \notin E$; and $\forall u \in V_C$, $\nexists v \in V$ such that $(u,v) \in E$.

For any node $u \in V$, we use the predicate $parent(u)$ to refer to the parent node of u. $parent(u) = \{w | w \in V, (w,u) \in E\}$. The root node of a tree $\mathcal{T}$ is the only node which does not have a parent node. Similarly, for any node $u \in V$, we use $children(u)$ to refer to the set of child nodes of u. $children(u) = \{w | w \in V, (u,w) \in E\}$. This definition says that a node w is a child node of u if and only if there exists an edge $(u,w) \in E$.

Definition 2 (path: $\Longrightarrow^*$)
Let $\mathcal{T} = (V,E)$ be the tag tree for a web document D. There is a path from node $u \in V$ to node $v \in V$, denoted by $u \Longrightarrow^* v$, if and only if one of the following conditions is satisfied:

(i) $u = v$
(ii) $(u,v) \in E$
(iii) $\exists u' \in V$, $u' \neq u$ *and* $u' \neq v$, *s.t.* $u \Longrightarrow^* u'$ *and* $u' \Longrightarrow^* v$.

If $u \Longrightarrow^* v$, then u is called an ancestor of v and we say that node v is *reachable* from node u.

There is a path from the root node to every other node in the tree. For a given node, the path expression from the root of the tree to the node can uniquely identify the node. Therefore, in subsequent sections we sometimes use such a path expression to refer to the node.

Consider Figure 1. The path from the root node *HTML* to the *Title* node goes through the *Head* node. It can be expressed as $HTML \Longrightarrow^* Title$. An alternative method to represent a path is to use the dot notation. For example, the expression HTML[1].Head[1].Title[1] can also be used to describe the path from the *HTML* node to the *Title* node in Figure 1. The numbers in the brackets after each node denotes the order of the child in the tag tree. Similarly, the path from *HTML* to *Body* is HTML[1].Body[2].

Definition 3 (Subtree)
Let $\mathcal{T} = (V,E)$ be the tag tree for a web document D, and $\mathcal{T}' = (V',E')$ is called a subtree of $\mathcal{T}$ anchored at node u, denoted as subtree(u), *if and only if the following conditions hold:*

- $V' \subseteq V$, *and* $\forall v \in V, v \neq u$, *if* $u \Longrightarrow^* v$ *then* $v \in V'$;
- $E' \subseteq E$, *and* $\forall v \in V', v \neq u, v \notin V_C, \exists w \in V', w \neq v$, *and* $(v,w) \in E'$

For a tag tree $\mathcal{T} = (V,E)$, the total number of subtrees is $|V| - 1$. We call a subtree anchored at node u a minimal subtree with property P, if it is the smallest subtree that has the property P, namely there is no other subtree, say $subtree(w), w \in V$, which satisfies both the property P and the condition $u \Longrightarrow^* w$ (u is an ancestor of w).

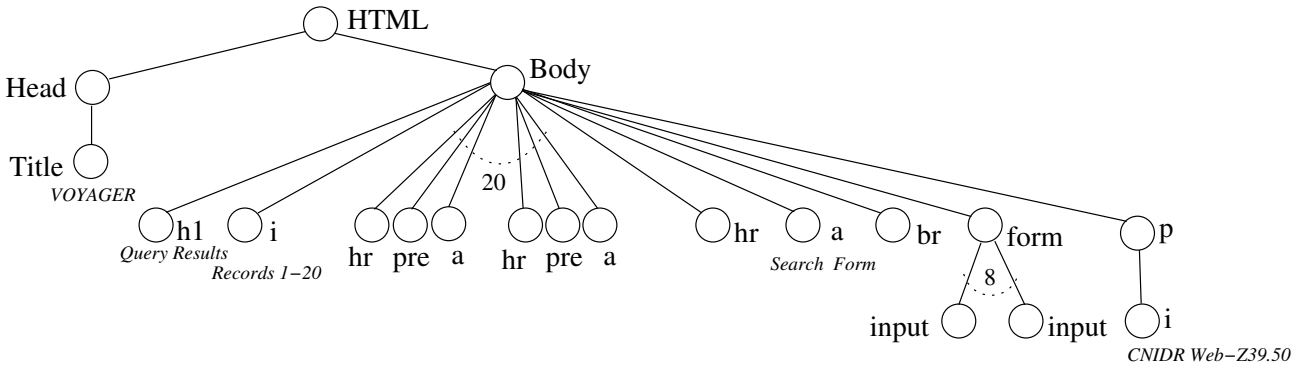

Figure 1: Tree Representation for Library Of Congress search results page.

Definition 4 (Minimal Subtree with Property P)
Let $\mathcal{T} = (V, E)$ be the tag tree for a web document, and subtree$(u) = (V', E')$ *be a subtree of $\mathcal{T}$ anchored at node u. We call* subtree(u) *a minimal subtree with property P, denoted as* subtree(u, P)*, if and only if $\forall v \in V, v \neq u$, if* subtree$(v)$ *has the property P, then $v \Longrightarrow^* u$ holds.*

Consider Figure 1, there are two subtrees that contain all of the *hr* nodes, the subtree anchored at *HTML* and the subtree anchored at *Body*. The subtree anchored at *Body*, as shown in Figure 2, is the minimal subtree that contains all of the *hr* nodes.

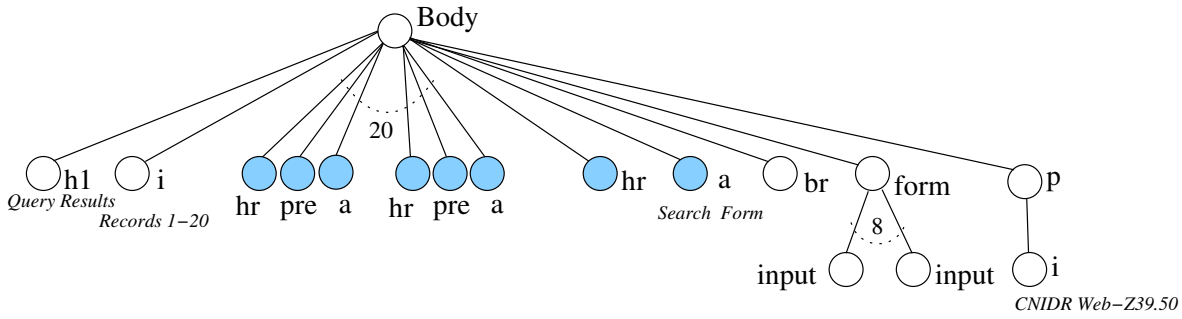

Figure 2: Minimal Subtree from Figure 1

In addition to the notion of subtree and minimal subtree, the following concepts are used frequently in the subsequent sections to describe the Omini object extraction algorithms.

- $fanout(u)$: For any node $u \in V$, we use $fanout(u)$ to denote the cardinality of the set of children of u. $fanout(u) = ||children(u)||$ if $u \in V_T$ and $fanout(u) = \emptyset$ if $u \in V_C$.
- $nodeSize(u)$: For any node $u \in V$, if $u \in V_C$, i.e., u is a leaf node, then $nodeSize(u)$ denotes the content size in bytes of node u. Otherwise, u is a tag node, i.e., $u \in V_T$ and $fanout(u) > 0$. We

define $nodeSize(u)$ to be the sum of the node sizes of all the leaf nodes reachable from node u, i.e. $nodeSize(u) = \sum_{v_i \in children(u)}(nodeSize(v_i))$.

- $subtreeSize(u)$: For any node $u \in V$, we define the size of the subtree anchored at node u to be the node size of u. I.e., $subtreeSize(u) = nodeSize(u)$.
- $tagCount(u)$: For any node $u \in V$, if $u \in V_C$ is a leaf node, then $tagCount(u) = 1$. Otherwise, $u \in V_T$ is a tag node and $tagCount(u) = 1 + \sum_{v_i \in children(u)}(tagCount(v_i))$. $tagCount(u)$ refers to the total number of tag nodes of which u is an ancestor.

3 System Architecture

Figure 3 shows the Omini system architecture. A user or an application may submit a URL to the Omini system to initiate the object extraction process. The results returned by the Omini is a list of objects extracted from the given web page. The Omini object extraction process consists of three phases.

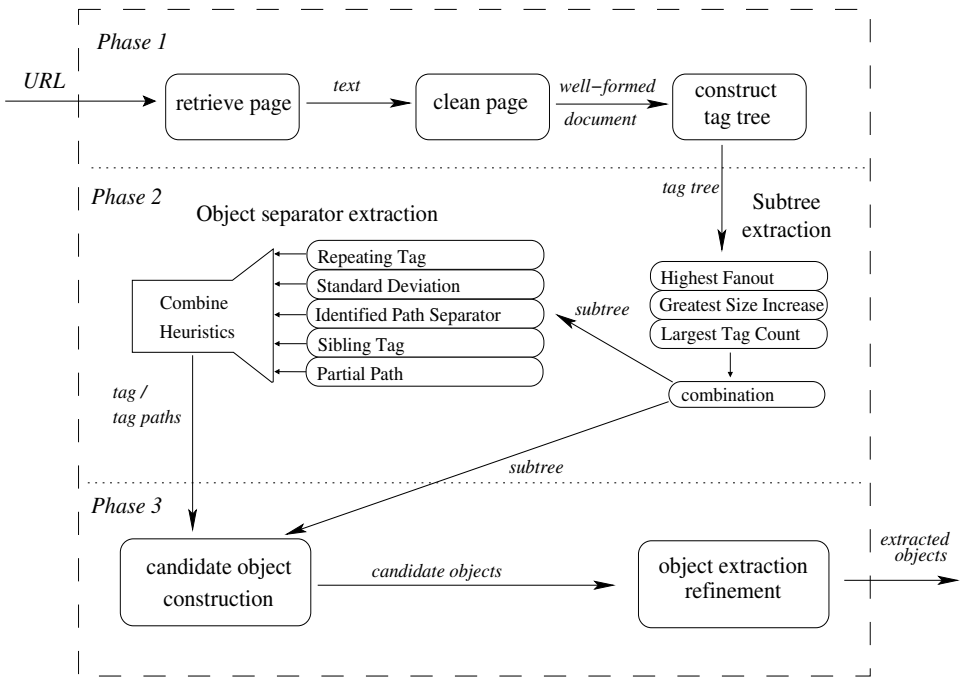

Figure 3: Omini System Architecture

Phase 1: Preparing a web document for extraction

The phase one is dedicated to preparing the web document for extraction. It takes a URL from an end-user or an application, and performs the following three tasks: First, the Web page specified by the URL will be fetched from a remote site. Second, the fetched page will be cleaned using the syntactic

normalization algorithm, which transforms the given web page into a well-formed web document. Third, the well-formed web document will be converted into a tag tree representation based on the nested structure of start and end tags. Due to the space restriction, the tag tree construction algorithm is omitted here. Readers may refer to our technical report [4] for detail.

Phase 2: Locating objects of interest in a Web page

The phase 2 process is divided into two consecutive steps: object-rich subtree extraction and object separator extraction. The former locates the minimal subtree that contains all the objects of interest in a page. The latter finds the object separator tags that can effectively separate objects.

Step 1: Object-rich subtree extraction

Web pages are designed for human browsing. In addition to the primary content regions, many web pages often contain other information such as advertisements, navigation links, and so on. Therefore, given a web document D, the first task in the object discovery phase is to identify which part of the document is the primary content region. Let T be the tag tree of the document D, then the task of locating the primary content region is reduced to the problem of locating the subtree of T which contains all the objects of interest. We call this task the *Object-rich subtree Discovery*. Obviously, there may be more than one subtree that contain all the objects of interest. The main goal of the object-rich subtree discovery is to locate the *minimal* subtree of T which contains all the objects of interest.In the first prototype of Omini, three individual subtree discovery rules and a method to combine them have been implemented. To choose the correct subtree we compare the fanout, the content size, or the tag count of all the subtrees in a given Web document (see Section 4 for further detail).

Step 2: Object separator extraction

Once the primary content region is found, the next task is to decide how to separate data objects from each other, and from any other information in the web page. We refer to this task as the *Object separator discovery*. One goal of the object separator discovery is to develop a method that can fully automate the process of discovering the correct object separator, which will effectively separate objects in the primary content region and extract the objects of interest. To achieve this objective, we develop a set of individual algorithms, each of which can independently identify a ranked list of object separators, and provide a mechanism to combine these independent algorithms into a methodical approach for the object separator discovery.

Phase 3: Extracting objects of interest in a page

The phase 3 consists of two tasks: Candidate Object Construction and Object Extraction Refinement.

Candidate object construction is the process of extracting objects from the raw text data of the web document using the object separator tag identified in Phase 2. After the object separator tag is chosen the objects need to be extracted from the components of the chosen subtree. Sometimes the separator tag sits between objects, and other times it is the root of the object or a part of the object. Occasionally, an object may be broken down into two or more pieces by the chosen separator.

Object Extraction Refinement is the process of eliminating candidate objects that do not conform to the set of minimum criteria, which are derived by the object extraction process and satisfied by most of extracted objects. More concretely, in the process of constructing the objects, extraneous objects such as list headers or footers may be extracted occasionally. The object extraction refinement step will remove those objects that are structurally not of the same type as the majority of objects, such as an object that is missing a common set of tags, or that has too many unique tags. Also if the object is too small or too large it will be removed as well.

4 Heuristics for Object-rich Subtree Extraction

Finding the minimal subtree that contains all the objects of similar structure is critical to the accuracy of the object separator heuristics. It is observed that Web documents have many different types and different layout representations. It is impossible to find a single minimal subtree algorithm that works for all kinds of web documents. In the first prototype of the Omini, we implement three minimal subtree algorithms: the highest fanout, the greatest size and size increase, and the highest tag count. We also provide a mechanism to combine them into a compound method for extracting the minimal object-rich subtree.

4.1 The Highest Fan-out Subtrees (HF)

This heuristic ranks all subtrees of a given document by their fan-outs and choose the highest fan-out subtree as the minimal subtree. The HF heuristic was introduced in [7]. The entire information extraction process described in [7] relies on the assumption that "*in a Web document of multiple records of interest, the subtree of T whose root has the highest fan-out should contain the records*". This heuristic works well for web pages that have almost no advertisement or designated navigational region. Unfortunately, many useful web sites today provide more than just the search result objects. For example, most e-commerce sites want to provide brand-recognition as well as a consistent and highly evolved look-and-feel for their web sites. These web pages are likely to contain a lot of navigational aids and other page elements that are not directly related with the content of the query results. In such cases, the highest count heuristics does poorly. This is particularly true when the number of navigational links is larger than the maximum number of query results displayed on a single page.

4.2 The Greatest Size Increase (GSI) Subtrees

The GSI heuristic ranks all of the subtrees by examining the content size of different subtrees. Obviously, any ancestor of a node is either equal or larger than the node itself. Thus, when one subtree is the ancestor of the other subtree, we rank them by the increase in size from the average size of the child nodes of the subtree to the size of the subtree. This is calculated by dividing the node size by the node fanout

and subtracting the result from the original node size. The GSI heuristic is motivated by the following observations. First, when the highest fan-out heuristic fails, it usually fails on navigation menus in web pages; navigation menus usually contain only links and the descriptive link names. Relatively speaking, the data objects returned from a search are much larger than navigation links. Second, a single data object will be relatively smaller in size than the complete set of objects. Third, a subtree that contains the set of data objects of interest may not have the highest fanout but will have a much larger size or size increase relative to its children than most subtrees.

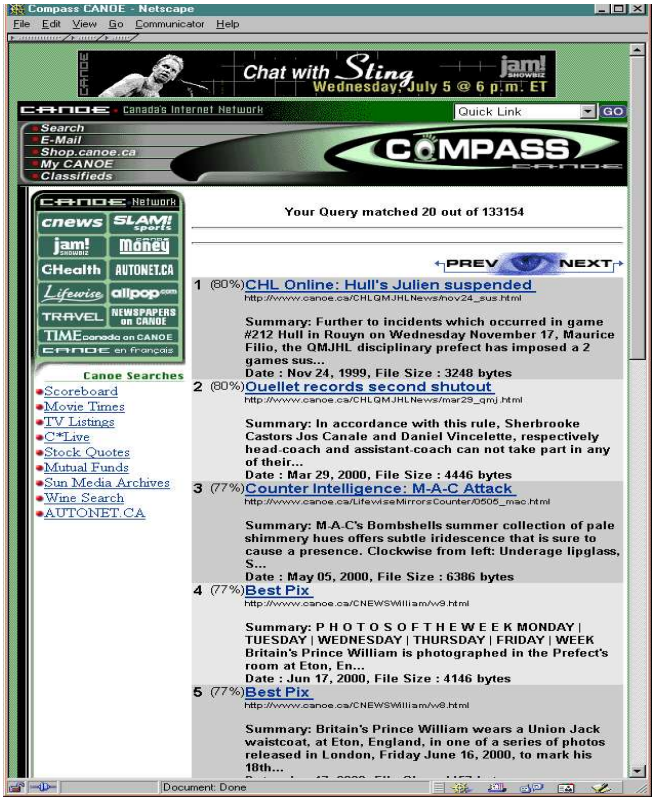

Figure 4: Canoe.com Search Result http://www.canoe.com – on July 2, 2000

Consider the web document in Figure 4 and its tag tree in Figure 5. The objects that we are interested in extracting from this example Web document is obviously the search results, namely the twelve news items marked by the twelve tables at the right side of the tree. By applying the GSI heuristic, the highest ranked subtree is the subtree anchored at the tag node $HTML[1].body[2].form[4]$. Obviously it is the minimal subtree that contains all the news objects of interest. However, by applying the HF heuristic, the subtree anchored at the tag node $HTML[1].body[2].form[4].table[5].tr[1].td[2].font[1]$ will be the chosen minimal subtree even though it does not contain any news objects of interest.

The GSI heuristic captures those subtrees that may not have the highest fanout, but have larger size and larger difference in size between the root node of the subtree and each of its child nodes.

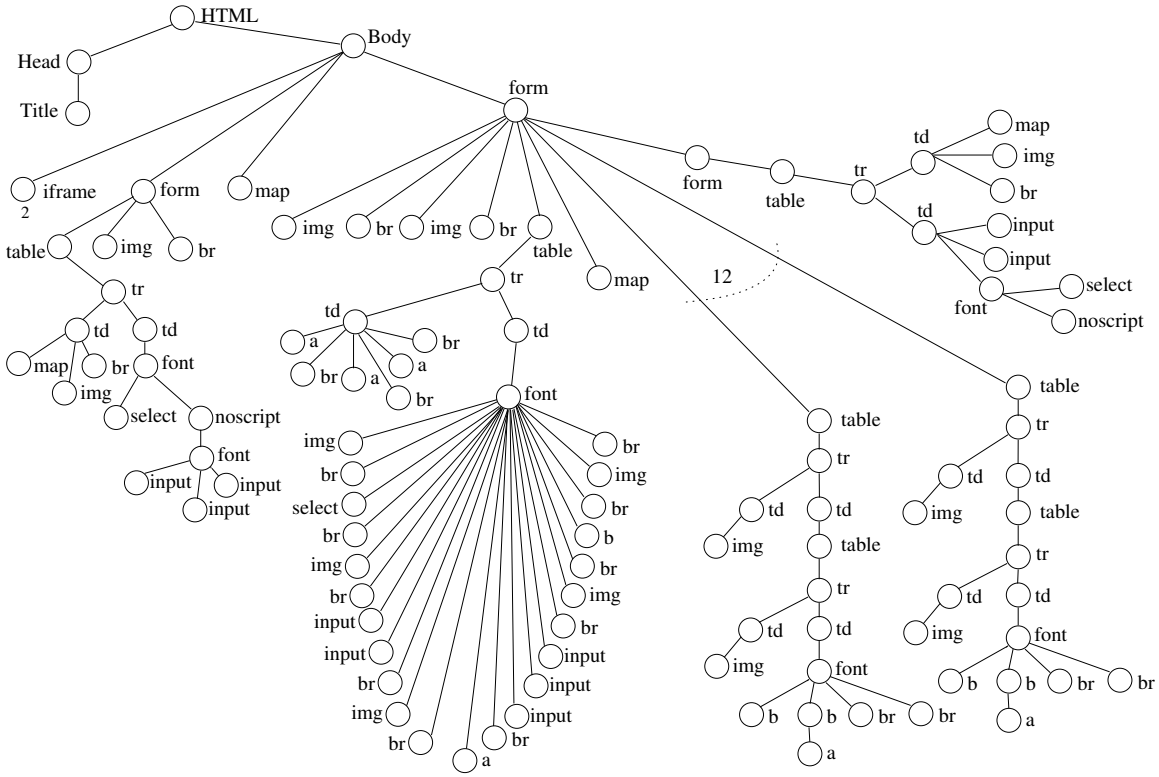

Figure 5: Tag Tree for Canoe.com, Figure 4

4.3 The Largest Tag Count Subtrees (LTC)

The LTC heuristic is motivated by the observation that data objects typically contain several mark-up tags and a subtree of the highest fan-out may not necessarily have the highest tag counts (see the example for HF in Figure 5). This heuristic implies that the more tags that are in a particular subtree, the more likely it will contain the data objects. However, there is one exception: when comparing a subtree anchored at node u with another subtree anchored at an ancestor of u, the ancestor will always have more tags. In that case we look at all the child nodes of the two subtrees. The subtree that has the highest appearance count of a child node tag will be ranked higher than the other subtree. For example, comparing the two subtrees $HTML[1].Body[2]$ and $HTML[1].Body[2].form[4]$ in Figure 5. The child tag $form$ in the subtree $HTML[1].Body[2]$ has the highest appearance count of 2. The child tag *table* in the subtree $HTML[1].Body[2].form[4]$ has the highest appearance count of 13. Thus, the LTC algorithm ranks the subtree $HTML[1].Body[2].form[4]$ higher.

The LTC algorithm works in two steps: For a given tag tree, in the first step we ranks all the subtrees in ascending order by the total number of tags they have. In the second step we walk down the ranked list, and re-examine those subtrees that have ancestor relationship. For each subtree in the ranked list, say

T_i, we compare it with every other subtree, say T_j, in the list. If $T_i \longrightarrow T_j$, i.e., they have an ancestor relationship, then we find the highest appearance count of the child node for both T_i and T_j. If the highest appearance count of the child node from T_j is greater than the highest appearance count of the child node from T_i, then T_i and T_j will exchange their ranking positions in the ranked list. Otherwise T_i will be compared with the next subtree after T_j in the ranked list. The process continues until all the subtrees are re-examined.

Recall the web document in Figure 4 and its tag tree in Figure 5. Table 1 lists the top five ranked subtrees obtained by applying HF, GSI, and LTC separately to the tag tree (Figure 5) of this example document. The GSI and LTC heuristics ranks the correct minimal subtree as its number one choice, whereas the subtree that the HF heuristic ranks the highest does not contain the news objects of interest even though it has the highest fanout.

Rank	Subtrees by HF	GSI	LTC
1	$HTML[1].body[2].form[4].$ $table[5].tr[1].td[2].font[1]$	$HTML[1].body[2].form[4]$	$HTML[1].body[2].form[4]$
2	$HTML[1].body[2].form[4]$	$HTML[1].body[2]$	$HTML[1].body[2].form[4].$ $table[5].tr[1].td[2].font[1]$
3	$HTML[1].body[2]$	$HTML[1].body[2].form[2].$ $table[1].tr[1].td[2]$	$HTML[1].body[2].form[4].$ $table[5].tr[1]$
4	$HTML[1].body[2].form[4].$ $table[5].tr[1].td[1]$	$HTML[1].body[2].form[4].$ $form[19].table[1].tr[1].td[2]$	$HTML[1].body[2]$
5	$HTML[1].body[2].form[4].table[10].$ $tr[1].td[2].table[1].tr[1].td[2].font[1]$	$HTML[1].body[2].form[2]$	$HTML[1].body[2].form[2]$

Table 1: Comparing HF, GSI, and LTC on canoe.com tag tree in Figure 5

4.4 The Compound Algorithm for Locating the minimal subtree

We have discussed three individual subtree algorithms. Each of the individual heuristics ranks subtrees independently by a single metric: fanout, size increase, or tag count. The idea of combining these three independent algorithms is to treat each metric as a separate dimension in a multi-dimensional space, and ranks subtrees by their multi-dimensional volume. Such a combination has several advantages. Subtrees with a low fanout, few tags, or a small size will have a smaller volume comparing to those that have more tags and a larger size. Similarly, the higher fanout subtree is ranked higher only when it also has a relatively larger size and a higher tag count. For example, subtrees which have a large number of navigation links but no content, such as a navigation menu, will be ranked low, while subtrees that have a few objects containing several tags and text, such as product descriptions, will obtain a higher ranking. Due to the space limitation, we omit the detailed algorithm for subtree extraction and the experimental results in this paper. Readers may refer to our technical report [4] for further detail.

5 Heuristics for Object Separator Extraction

After the object-rich subtree extraction process, the problem of extracting the object separator tag in a web page is reduced to the problem of finding the right object separator tag in the chosen minimal subtree. The problem can be addressed in two steps. First we need to decide which tags in the chosen minimal subtree should be considered as candidate object separator tags. Second, we need a method to identify the right object separator tag from the set of candidate tags, which will effectively separate all the objects.

There are several ways of choosing the object separator tags. One may consider every node in the chosen subtree as a candidate tag or just the child nodes of the chosen subtree as the candidate tags. Based on the semantics of the minimal object-rich subtree, it is sufficient to consider only the child nodes in the chosen subtree as the candidate separator tags.

In the first prototype of Omini, five separator tag identification heuristics are supported, covering a wide range of possible mechanisms for discovering object separators. Each of the five heuristics independently ranks the candidate tags. The standard deviation heuristic (SD) and the repeating pattern heuristic (RP) were first proposed in [7]. The SD heuristic ranks the candidate tags based on the standard deviation of sizes between two tags. The RP heuristic ranks the candidate tags based on the difference between the counts of a pair of tags and the counts of a single tag. The partial path heuristic (PP) and the sibling tag heuristic (SB) are introduced in Omini. The former is motivated by the observation that the multiple instances of the same object type often have the same tag structure. The latter is based on the observation that the objects identified by highest count sibling pairs are more likely to be of the same object type than the highest count single tags. The identifiable path separator heuristic (IPS) is an extension of the IT (Identifiable Tag) heuristic proposed in [7]. Instead of using the same list of pre-determined and ranked candidate tags for every tag tree, a different list is used based on the subtree that is chosen.

In the subsequent sections we describe each of the five individual heuristics first and then we discuss the method to best combine the rankings of these five heuristics for selecting the correct object separator tag.

5.1 Standard Deviation Heuristic (SD)

The SD heuristic measures the standard deviation in the distance (in terms of the number of characters) between two consecutive occurrences of a candidate tag, and then ranks the list of candidate tags in ascending order by their standard deviation. It is motivated by the observation that the multiple instances of the same object type in a web document are typically about the same size.

Consider the tag tree for the Library of Congress web page shown in Figure 1. From the subtree extraction step, the subtree anchored at the node HTML[1].Body[2] is the chosen minimal subtree as shown

in Figure 2. Among the set of child node tags, some tags have much higher counts than the others do. For example, the tag *hr* appears twenty-one times, the tag *a* appears twenty-one times, and the tag *pre* occurs twenty times. We refer to these tags as the highest count tags. The standard deviation in distance is calculated between two consecutive occurrences of hr tag, between two consecutive occurrences of pre tag, between two consecutive occurrences of a tag, and so on.

Given a candidate tag T, the standard deviation for T is calculated as $\sigma(T) = \sqrt{\frac{\sum_{i=1}^{n}(t_i-\mu)^2}{n}}$ [11], where n is the number of occurrences of the tag T (i.e., the appearance count of the tag T), t_i is the size of the subtree anchored at the i^{th} appearance of the tag T, and $\mu = \frac{\sum_{i=1}^{n} t_i}{n}$ is the average distance between any two consecutive occurrences of the tag T. The SD algorithm first computes the average distance for each of the candidate tags that appeared more than one in terms of their subtree sizes. Then the square of the difference between the average distance and each appearance of the tag is summed. The variance is calculated by dividing the sum by the number of appearances of the given candidate tag as child node. The standard deviation is determined by taking the square root of the variance. After obtaining the standard deviation for all the candidate tags, a ranking is produced with the lowest SD tag at the top of the list and the highest DD tag at the lowest end of the list.

Table 2 shows the top three candidate tags by applying the SD algorithm to the library of congress example in Figure 2. It ranks the candidate tags in ascending order by the standard deviation in distance, with the smallest standard deviation first.

The algorithm for ranking the candidate tags using the SD heuristic is omitted here due to space restriction. Readers who are interested in further detail may refer to our technical report [4].

Rank	Tag	Standard Deviation
1	hr	114
2	pre	117
3	a	122

Table 2: Standard Deviation for tags from the minimal subtree in Figure 2

5.2 Repeating Pattern Heuristic (RP)

The RP heuristic chooses the object separators by counting the number of occurrences of all pairs of candidate tags that have no text in between. It computes the absolute value of the difference between the count for a pair of tags and the count for each of the two paired tags alone and then ranks the candidate tags in ascending order by this absolute value. The intuition behind this heuristic is that a single tag may be used to mean many things, but a pattern of two or more tags is more likely to mean just one thing. When there is no such pairs of tags in the chosen subtree, the RP heuristic produces an empty list of candidate tags. It simply means that the RP heuristic has no answer about which of the candidate tags is the object separator tag.

Tag Pair	Pair Count	Difference
table, tr	13	0
img, br	2	0
map, table	1	0
form, table	1	0
br, img	1	1
br, table	1	1

Table 3: Repeating tag ranking for minimal subtree from Figure 5

Consider the subtree $HTML[1].Body[2].form[4]$ in Figure 5. Table 3 shows a ranked list of all tag pairs in descending order by the pair counts and the difference between the pair count and the tag count.

5.3 Identifiable Path Separator Tag heuristic (IPS)

The IPS heuristic ranks the candidate tags of the chosen subtree according to the list of system-supplied IPS tags. The IPS tags are those tags that are identified by the system as the most commonly used object separator tags for different types of subtrees in Web documents. The idea behind this heuristic is the following. First we observe that Web documents, generated either by hand or by authoring tools or by server programs, often consist of multiple presentation layouts within a single page, each is defined by some specific type of HTML tags. For example, a web page may contain a table marked by table tag table, a list marked by the list tag ul or ol, and a paragraph marked by the tag p. Second, each such a presentation layout tends to use regular structure. For example, a table tends to use the row tag tr and the column tag td to define rows and columns of the table; and a list tends to use the list item tag li to define the list structure. Therefore, for each presentation layout (i.e., a subtree type), there are a few tags that are used consistently for separating objects within the subtree.

Based on these observations and the Web documents we have tested, we create a list of object separator tags for each type of subtrees as shown in Table 4.The full list of object separators is composed of all the identified tags for each type of subtree listed in Table 4, with duplicates removed.

Subtree	Tag List
body	table,p,hr,ul,li,blockquote,div,pre,b,a
table	tr,b
form	table,p,dl
td	table,hr,dt,li,p,tr,font
dl	dt,dd
form	table,p,dl
ol	li
ul	li
blockquote	p

Table 4: A Table of Object Separator Tags

Tag	% of time used as object separator
tr	34
table	18
p	10
li	8
hr	6
dt	6
ul	2
pre	2
font	2
dl	2
div	2
dd	2
blockquote	2
b	2
a	2

Table 5: Object separator probability

The next step is to determine the rankings of these commonly used object separator tags. Table 5 lists the distribution of all object separator tags we observed in our tests of 50 web sites with over 2000 web pages. Based on the experimental results over the testing web sites, we produce the following ranking of the full list of IPS tags. We refer to such an ordered list as $IPSList$:
$\{tr,table,p,li,hr,dt,ul,pre,font,dl,div,dd,blockquote,b,a,span,td,br,h4,h3,h2,h1,strong,em,i\}$.
For tags of the same rank, the order is arbitrary.

5.4 Sibling Tag Heuristic (SB)

The SB heuristic counts pairs of tags that are immediate siblings in the minimal subtree, and ranks all pairs of tags in descending order by the number of occurrences of the pair. For those pairs of tags that have equal occurrence, the ranking follows the order of their appearances in the Web document. For example, in the fragment <a> ... ... <c> ... </c> , the sibling pairs are (a,b) and (b,c) and each occurs only once. Table 6 ranks sibling pairs from the Library of Congress tag tree in Figure 1 and Canoe.com tag tree in Figure 5. The first tag of the highest ranked pair is chosen as the object separator. In Figure 1 the (hr,pre) tag pair appears before the (pre,a), and thus it ranks higher.

The sibling tag heuristic is motivated by the observation that given a minimal subtree, the object separator tag is expected to appear as many times as there are objects. However, in some cases there are other tags that occur more often than any object separator tag. In these web pages the highest count tag may not be the correct object separator, especially when the highest count tag appears irregularly. Instead, the SB heuristic looks at sibling tags for repetition of pairs of tags, such as the <*hr*><*pre*>

	Canoe.com		Library of Congress	
Rank	**pair**	**count**	**pair**	**count**
1	table,table	11	hr,pre	20
2	img,br	2	pre,a	20
3	br,img	1	a,hr	20
4	br,table	1	h1,i	1
5	table,map	1	i,hr	1
6	map,table	1	hr,a	1
7	table,form	1	a, br	1
8			br,form	1
9			form,p	1

Table 6: Tag ranking for SB heuristic on Figure 5 and 1

pattern in the Library of Congress search results in Figure 1 or the <*table*><*table*> pattern in the canoe.com page in Figure 5. In both cases, patterns, which contain the object separator tag, should appear much more frequently than any other patterns.

5.5 Partial Path Heuristic (PP)

The PP heuristic lists all paths from a candidate node to any other node, which is reachable from this candidate node, in the chosen subtree, and counts the number of occurrences of each identical path. The list of candidate tags is ranked in descending order first by the count of all the identified paths and then the length of the paths. If two paths have an equal count, then the longer path will rank higher than the shorter one because it indicates more structure. Interesting to note is that if there are no paths with a length more than one, such as in Figure 1, this heuristic reduces to simply choosing the tag with the highest count. The main idea behind this heuristic is motivated by the observation that the multiple instances of the same object type often have the same tag structure. Table 7 lists all of the partial paths for the example web document in Figure 5. The PP rankings for this example web page and the Library of Congress page in Figure 1 are given in Table 8.

Path	**count**	**Path**	**count**
table.tr.td	26	table.tr.td.table.tr	12
table.tr.td.table.tr.td.font.b	24	table.tr.td.table	12
table.tr.td.table.tr.td.font.br	24	table.tr.td.img	12
table.tr.td.table.tr.td.	24	table.tr.td.br	3
table.tr	13	table.tr.td.a	3
table	13	form.table.tr.td.input	2
table.tr.td.table.tr.td.font.b.a	12	form.table.tr.td	2
table.tr.td.table.tr.td.font	12	img	2
table.tr.td.table.tr.td.img	12	br	2

Table 7: Partial paths and their count from the minimal subtree in Figure 5

	Canoe.com		Library of Congress	
Rank	**tag**	**count**	**tag**	**count**
1	table	26	hr	21
2	form	2	a	21
3	img	2	pre	20
4	br	2	form	8

Table 8: Tag ranking from partial path rankings for Figure 5 and Figure 1

6 Determining the Correct Object Separator Tag: The Combined Algorithm

6.1 Performance Measures of Individual Heuristics

We have discussed each individual heuristic and its algorithm to produce a ranked list of candidate tags. Each of the five individual algorithms works independently towards the same goal – finding the right object separator from the set of candidate tags. As a result, each heuristic chooses the highest ranked tag as the object "correct" separator. However, as we observed from the discussions in the previous sections, these five heuristics may not always agree on their highest ranked choice. To understand the performance of these individual heuristics on different types of web pages, we conducted a series of experiments over 500 web pages from 15 different web sites (see Table 9). An empirical probability distribution for the success rate of each individual heuristic is listed in Table 10. For each heuristic, we first calculated the number of times the heuristic chose a correct object separator tag at a particular rank. Then for each web site, we calculated the success rate of the given heuristic by normalizing the number of times the correct separator tag is chosen by the number of pages tested over a particular web site. The success rate for each heuristic over the 15 web sites (shown in Table 10) was calculated by averaging the normalized numbers from each web site.

6.2 The Combined Algorithm and Its Performance

An obvious way to improve the accuracy of finding a correct object separator in a web document is to consider the best way of combining these independent heuristics. A well-known approach for combining evidences from two or more independent observations is to use the basic laws of probability [11]: Let $P(A)$ be the probability associated with the result of applying heuristic A over a web document, and $P(B)$ be the probability associated with the result of applying heuristic B over the same web document. The formula $P(A \cup B) = P(A) + P(B) - P(A \cap B)$ will produce the compound probability $P(A \cup B)$ for locating a correct object separator tag in this web document. For example, if the probability factors that a tag **tr** is an object separator in a web document are 78%, 63%, and 85%, then the compound probability for tag **tr** is 89% ($78\% + 63\% + 85\% - 78\% \times 63\% - 78\% \times 85\% - 63\% \times 85\% + 78\% \times 63\% \times 85\% = 89\%$).

Website	Date
agents.umbc.edu	July 2000
www.alphabetstreet.infront.co.uk	March 2000
www.alphaworks.ibm.com/	March 2000
www.amazon.com	December 1999
www.aw.com	December 1999
www.bookpool.com	March 2000
cbc.ca/consumers/	March 2000
www.chapters.com	March 2000
www.google.com	March 2000
www.hotbot.com	March 2000
www.ibm.com/developer/java	March 2000
www.kingbooks.com	March 2000
www.loc.gov	March 2000
www.rubylane.com	March 2000
www.signpost.org	March 2000

Table 9: Test Web Site List

Heuristic	Rank 1	2	3	4	5
SD	0.78	0.18	0.10	0.00	0.00
RP	0.73	0.13	0.00	0.00	0.00
IPS	0.40	0.46	0.13	0.07	0.00
PP	0.85	0.06	0.02	0.00	0.00
SB	0.63	0.17	0.12	0.06	0.03

Table 10: Probability rankings for object separator heuristics

To combine the five individual heuristics, there are 26 possible combinations ($\sum_{i=0}^{5} C(5,i) - 6 = 26$) in addition to the trivial case of none and the five individual heuristics. We take two steps to determine which of the combinations produces the best overall result: First, for each combination, we computed the compound probability for each candidate tag in every web document from our test set, based on the probability distribution of each heuristic. Then we determine the success rate for each of the 26 combinations. Given a combination C_i ($i = 1, ..., 26$), let n be the number of Web documents in our experiments and D_j ($j = 1, ..., n$) denote the jth document being tested. For each document D_j, if there are M tags that have the high compound probability and only H of the M tags are correct object separator tags, then the success rate of the combination C_i for D_j, denoted by $success(C_i, D_(j))$, is H/M. Thus, the success rate for the combination C_i over all the experimental web documents can be calculated by the formula $(\sum_{j=1}^{n} success(C_i, D_j))/n$. Table 11 lists the results of all combinations. To conveniently represent a combination, each heuristic is abbreviated by a one letter acronym: **HC** by **H**, **SD** by **S**, **RP** by **R**, **IPS** by **I**, **PP** by **P**, and **SB** by **B**. Thus, **RSIPB** stands for the combination of the **RP**, **SD**, **IPS**, **PP**, and **SB** heuristics. Based on the set of testing web documents, the combination of all five heuristics performs the best. Our algorithm for object separator tag extraction is developed based on this observation. Due to the space restriction, readers who are interested in algorithmic details

may refer to our technical report [4].

Combo	Success	Combo	Success	Combo	Success
IB	0.61	RB	0.73	RI	0.75
RS	0.78	SI	0.78	SB	0.78
RIB	0.80	RSB	0.84	SIB	0.84
RPB	0.85	RP	0.85	SP	0.85
IPB	0.85	IP	0.85	PB	0.85
RSI	0.86	RIPB	0.86	RIP	0.86
RSP	0.88	SIPB	0.89	SIP	0.89
SPB	0.89	RSIP	0.92	RSIB	0.92
RSPB	0.92	RSIPB	0.98		

Table 11: Success rates for heuristic combinations on test data

In this section we report our experiment setup and the experimental results for validation of our approach. We also report the experimental results conducted for a comparison of our approach with the BYU's approach [7].

6.3 Experimental Setup

To run our experiments, we downloaded and cached web pages from 50 different Web sites.To automatically retrieve the pages we first generated a random list of 100 words from the standard Unix dictionary. Then we fed each word into a search form at each of the 50 web sites. After retrieving the pages we discarded those pages which returned no results. For the static web pages which do not have a search interface, such as agents.umbc.edu, a manual approach is used to load the pages to the local storage. All experiments were carried out on the local version of the pages so as not to overload web sites and to be able to obtain consistent results over time.

For each web site, example pages were manually examined to determine the path of the minimal subtree as well as all possible separator tags. The results of the algorithms were compared with the actual separator tags; the rank that the algorithms choose for a particular separator is recorded for each web page.

The success rate of an algorithm is calculated in two steps. First, for each web site we calculate the percentage of the downloaded pages in which the highest ranked tag of the algorithm is the correct separator tag. These percentages are then averaged over the collection of web sites to determine the success rate for individual heuristics and their combinations.

6.4 Validation Tests

We have discussed the combined algorithm for object extraction and our experimental approach to determining the best combination in Section 6. To further validate the effectiveness and quality of

the Omini approach to object extraction, we ran the algorithms over 1,500 web pages from 25 web sites listed in Table 12. These 25 web sites cover a broad range of application domains and a good variety of web documents. For each of the 1,500 documents, we applied the five heuristics and the combination of the five **RSIPB**. Table 13 lists the experimental results. Even though four out of five individual heuristics did not reach a success rate higher than 90%, the combination of the five attained 94% success rate. The experimental results indicate that, while the heuristics are not extremely stable, it is beneficial to combine them.

Website	**Date**
www.amazon.com	March 2000
www.amazon.com (ZShops)	March 2000
www.bn.com	March 2000
www.bookbuyer.com	March 2000
www.borders.com	March 2000
www.canoe.com	March 2000
www.codysbooks.com	March 2000
www.ebay.com	March 2000
www.etoys.com	March 2000
www.excite.com	March 2000
www.fatbrain.com	March 2000
www.gameCenter.com	March 2000
www.gamelan.com	March 2000
www.goto.com	March 2000
www.ibm.com	March 2000
www.ibm.com/developer/xml	March 2000
www.msn.com/auctions	March 2000
www.powells.com	March 2000
www.quote.com	March 2000
www.thestar.org	March 2000
www.vancouversun.com	March 2000
www.vnunet.com	March 2000
www.wine.com	March 2000
www.yahoo.com	March 2000
www.yahoo.com/auctions	March 2000

Table 12: Experimental Web Site List

Heuristic	**Rank 1**	**2**	**3**	**4**	**5**
SD	0.77	0.15	0.06	0.05	0.00
RP	0.77	0.10	0.07	0.02	0.00
IPS	0.88	0.08	0.07	0.00	0.00
PP	0.93	0.05	0.00	0.00	0.00
SB	0.71	0.13	0.06	0.07	0.04
RSIPB	0.94	0.05	0.02	0.01	0.00

Table 13: Probability rankings for object separator heuristics on experimental data

6.5 Recall and Precision

Success rate is not the only measure of quality for object separator identification algorithms. We also use recall and precision to judge how well our algorithms perform. Recall is the percentage of positive instances of the target concept (in our case the object separator tag) that are correctly identified. Precision is the percentage of extractions made that are correct. Both of these numbers are defined in terms of false positives (FP), false negatives (FN), and true positives (TP). $Recall = \frac{TP}{TP+FN}$ and $Precision = \frac{TP}{TP+FP}$. A true positive is an instance where an object separator exists and it is correctly identified by the algorithms. A false negative is an instance where the object separator exists but is not found by the algorithms. A false positive is an instance where the object separator does not exist, but a tag is mistakenly identified as an object separator.

Heuristic	Success	Precision	Recall
SD	0.78	1.00	0.78
RP	0.73	0.84	0.73
IPS	0.71	0.82	0.71
PP	0.85	0.92	0.85
SB	0.62	0.89	0.62
RSIPB	0.98	1.00	0.98

Table 14: Probability rankings for object separator heuristics on test data

Heuristic	Success	Precision	Recall
SD	0.77	1.00	0.77
RP	0.77	0.97	0.77
IPS	0.88	0.94	0.88
PP	0.93	1.00	0.93
SB	0.71	0.97	0.71
RSIPB	0.94	1.00	0.94

Table 15: Probability rankings for object separator heuristics on experimental data

A careful examination of the algorithms shows that not every page will have an object separator chosen. For example, both RP and IPS reject tags that occur below a given threshold. If the only tags that exist in the chosen subtree occur fewer times than the threshold, then neither heuristic will be able to choose a tag.

6.6 Performance

We have collected performance data on how quickly the algorithms are able to extract data objects from target web pages. We have measured the execution time of the algorithms over the fifteen test web sites, as well as the twenty-five validation web sites. For each web page the algorithms were run

ten times over the page. Tables 16 and 17 show the average of the resulting times.

We measured performance from two different methods of extracting objects. In the first method, we used the algorithms described above to discover the minimal subtree and the object separator. Since the structure of websites does not change often, it may be worthwhile to store rules that allow the subtree and object separator to be immediately chosen, rather than discovering them every time data is extracted from a page at the web site. In the second method, we used cached rules that were discovered for each web-site to find the minimal subtree and object separator tag.

Web Site	Read File	Parse Page	Choose Subtree	Object Separator	Combine Heuristics	Construct Objects	Total
Test	8.54	95.85	32.77	64.85	0.31	0.08	203.38
Experimental	13.21	130.96	46.21	58.08	0.25	0.21	248.96
Combined	11.57	118.62	41.49	60.46	0.27	0.16	242.59

Table 16: Execution time in milliseconds for object extraction from web sites

Web Site	Read File	Parse Page	Choose Subtree	Construct Objects	Total
Test	9.38	92.54	7.77	3.15	112.31
Experimental	12.84	121.72	6.76	3.76	144.68
Combined	11.66	111.74	7.11	3.55	133.61

Table 17: Execution time in milliseconds for object extraction with rules from various web sites

Tables 16 and 17 show that by remembering simple rules, the execution time can be nearly halved. While the reading in and parsing the data still takes roughly similar amounts of time, the phases of choosing the minimal subtree, choosing the object separator and constructing the objects is an order of magnitude faster. When the rules are used, the time taken to extract the objects is essentially the same as the time it takes to read and parse the web page.

6.7 Experimental Comparison with the BYU's approach

The Omini object extraction approach was motivated by several research results reported in the literature [16, 9, 7]. Among these reported research results, the work done by Embley and his colleagues at BYU [7] inspires us the most. For example, we adopt two of the five data extraction heuristics from [7], namely the SD and RP heuristics, without any change. Our IPS heuristic is an evolution of their identifiable tag (IT) heuristic. IT chooses tags based on a predefined list of common object separators. We found this to be inflexible when a larger variety of web sites are considered. Therefore, instead of using the same list of candidate separators for all kinds of web sites, we use different lists depending on the type of the tag node at which the minimal subtree is anchored. This allows us, for example,

to list the tag *tr* first for tables, *li* first for lists, *table* for *body* tags, and so on. Our IPS heuristic has much higher extensibility and scalability with respect to the fast evolution of the Web. We did not include the highest count (HC) heuristic, which ranks tags based the number of times they appear. Through several experiments we found that the success rate for the HC heuristic is not as desirable as one would expect. First, the HC this heuristic was not a part of any of the most successful heuristic combinations; Second, those combinations that include the HC heuristics were often less successful in choosing a correct object separator than the same combination without the HC heuristic. Finally, the PP heuristic is extension of the HC heuristic that behaves exactly the same as HC on certain classes of web sites (such as the Library of Congress page in Figure 1). We also rejected their proposed ontology match heuristic. This heuristic relies on knowing about the domain of a web site and having a detailed ontology for that domain. An example of such ontology is to use a specific word or phrase that is known to appear only once in every object of interest. As reported in [7], developing the ontology for a domain takes about 2 man-weeks of work. The goal of the Omini project is to develop a fully automated object extraction system. We believe that a fully automated approach is the best way to develop a scalable system as the Web grows, and especially as the number of domains and the specificity of each domain increases. While the OM heuristic reportedly works well in [7, 6], the required human intervention makes this heuristic unsuited to our approach.

For the sake of performance comparison, we have implemented all of the heuristics in [7] except for the ontology based heuristic. We conducted a set of experiments over a wide variety of web pages. There are some cases where the approach in [7] performs well. In those cases our heuristics perform at least as well, if not better. However, there are some cases where the four heuristics in [7] perform poorly; for example, on the sites listed in Table 18, their heuristics only achieved a success rate of only 59%. In these cases our heuristics still perform very well, with a 93% success rate.

Website	Date
www.bookpool.com	March 2000
www.ebay.com	March 2000
www.goto.com	March 2000
www.powells.com	March 2000
www.signpost.org	March 2000

Table 18: Example Web Sites

For the web sites listed in Table 9, which we used for testing and training in Section 6, the BYU's approach [7] only works well for some web sites. As a result, their algorithms are less accurate than the Omini approach over the same collection of the web documents. Table 20 lists the success rates for their individual heuristics and the combinations of their heuristics over our test data. The heuristics HC, IT, RP, SD are abbreviated **H**, **T**, **R**, and **S** respectively. In comparison, the combined heuristic in [7] attained a success rate of 86% while the combined algorithm in Omini achieves 98% (recall Table 11).

Embley		Extended	
Heuristic	**Success**	**Heuristic**	**Success**
RP	19	RP	19
SD	23	SD	23
IT	40	IPS	76
HC	40	SB	56
		PP	78
HTRS	59	RSIPB	93

Table 19: Success rates for different heuristics and combinations on comparison web sites in Table 18

Heuristic	**Rank 1**	**2**	**3**	**4**	**5**
HC	0.79	0.13	0.14	0.00	0.00
IT	0.46	0.33	0.20	0.06	0.00
RP	0.73	0.13	0.00	0.00	0.00
SD	0.78	0.18	0.10	0.00	0.00
Combination	**Rank 1**	**2**	**3**	**4**	**5**
HT	0.79	0.00	0.20	0.07	0.00
HR	0.79	0.20	0.07	0.00	0.00
HS	0.79	0.07	0.20	0.00	0.00
TR	0.85	0.01	0.20	0.00	0.00
TS	0.78	0.01	0.27	0.00	0.00
RS	0.78	0.15	0.13	0.00	0.00
HTR	0.86	0.00	0.20	0.00	0.00
HTS	0.84	0.01	0.09	0.11	0.00
HRS	0.86	0.00	0.20	0.00	0.00
TRS	0.84	0.08	0.13	0.00	0.00
HTRS	0.86	0.04	0.15	0.01	0.00

Table 20: Experimental Results for the heuristics and their combination proposed in [7]

7 Conclusion

We have presented Omini, a fully automated object extraction system for Web pages. Omini performs object extraction in three phases. First, Omini parses and normalizes a web page into a tree structure. Second, Omini employs a set of subtree extraction rules to extract the minimal subtree that contains all the objects of interest. Third, the concrete object boundaries are identified and then objects themselves are extracted from the subtree. The main contribution of the paper is the fully automated object boundary identification algorithms described in Sections 5 and Section 6.

We tested and evaluated Omini in a series of experiments (Section 6.2) using more than 1,500 documents from 25 Web sites (primarily electronic commerce sites). The experimental evaluation consists of three parts. First, the result of Omini analysis is visually inspected for the identification of false negatives (correct objects that were left out) and false positives (incorrect objects included in the result). Omini results show an accumulated precision of 100% (only correct objects are returned) and an accumulated

recall between 93% and 98% (very few actual objects left out). Second, we replicated BYU's system [7] without the ontology heuristic (the human-dependent component). Omini results compare favorably to the BYU information extraction system. Finally, the Omini execution overhead is measured and the typical web page processing takes about 0.2 seconds. Furthermore, if the subtree and object separator discovery rules are cached, then the overhead is dominated by the fetching and parsing of web pages (about 0.1 seconds).

Fully automated object extraction is an important and necessary component in the construction of scalable and reliable next-generation information search and aggregation services on the Web. We plan to demonstrate the usefulness of Omini by combining it with a wrapper generation system (e.g., the XWRAP Elite [20]) to automate the wrapper generation and evolution process. Other potential future research directions include the automation of evaluation process and incorporation of feedback-based refinement of object extraction, as well as the integration with query optimization and semantic interoperability software systems.

References

[1] B. Adelberg. Nodose - a tool for semi-automatically extracting structured and semi-structured data from text documents. *ACM SIGMOD*, 1998.

[2] N. Ashish and C. A. Knoblock. Semi-automatic wrapper generation for internet information sources. In *Proceedings of Coopis Conference*, 1997.

[3] P. Atzeni and G. Mecca. Cut and paste. *Proceedings of 16th ACM SIGMOD Symposion on Principles of Database Systems*, 1997.

[4] D. Buttler, L. Liu, and C. Pu. Omini: An Object Mining and Extraction System for the Web. *Technical Report, Sept. 2000. Georgia Tech, College of Computing.*

[5] R. Doorenbos, O. Etzioni, and D. Weld. A scalable comparison-shopping agent for the world wide web. *Proceedings of Autonomous Agents*, pages 39–48, 1997.

[6] D. W. Embley, D. Campbell, Y. Jiang, Y. Ng, R. Smith, S. Liddle, and D. Quass. A conceptual-modeling approach to extracting data from the web. In *Proceedings of the 17th International Conference on Conceptual Modeling (ER'98)*, Singapore, November 1998.

[7] D. W. Embley, Y. Jiang, and Y.-K. Ng. Record-boundary discovery in web-documents. In *Proceedings of the 1999 ACM SIGMOD*, Philadelphia, Pennsylvania, USA, June 1999.

[8] S. P. et al. XHTML 1.0: The extensible hypertext markup language. Technical report, W3C, 2000.

[9] O. Etzioni and D. Weld. A softbot-based interface to the internet. *Communications of the ACM*, 37(7):72–76, 1994.

[10] J. Hammer, H. Garcia-Molina, J. Cho, R. Aranha, and A. Crespo. Extracting semi-structured data from the web. *Proceedings of Workshop on Management of Semi-structured Data*, pages 18–25, 1997.

[11] J. J. Higgins and S. Keller-McNulty. *Concepts in Probability and Stochastic Modeling.* Duxbury, 1995.

[12] C. A. Knoblock, S. Minton, J. L. Ambite, N. Ashish, P. J. Modi, I. Muslea, A. Philpot, and S. Tejada. Modeling web sources for information integration. In *Proceedings of AAAI Conference*, 1998.

[13] N. Kushmerick. Wrapper induction for information extraction. *PhD Thesis, Dept. of Computer Science, U. of Washington, TR UW-CSE-97-11-04*, 1997.

[14] N. Kushmerick, D. Weil, and R. Doorenbos. Wrapper induction for information extraction. In *Proceedings of Int. Joint Conference on Artifical Intelligence (IJCAI)*, 1997.

[15] S. Lawrence. Search the web: Fundamental limitations, new techniques, and future direcitions. *NEC Symposium*, June 2000.

[16] L. Liu, C. Pu, and W. Han. XWrap: An XML-enabled Wrapper Construction System for Web Information Sources. *Proceedings of the International Conference on Data Engineering*, 2000.

[17] D. Raggett. Clean up your web pages with HTML tidy. http://www.w3c.org/People/Raggett/tidy/, 2000.

[18] A. Sahuguet and F. Azavant. WysiWyg Web Wrapper Factory (W4F). *Proceedings of WWW Conference*, 1999.

[19] S. Soderland. Learning to extract text-based information from the world wide web. *Proceedings of Knowledge Discovery and Data Mining*, 1997.

[20] G. Tech. XWRAP Elite Project. *http://www.cc.gatech.edu/projects/disl/XWRAPElite*, May 2000.

[21] Extensible markup language (XML) 1.0. Technical report, W3C, 1998.

A Web Site List

The list of websites used in testing and verifying the object separator discovery algorithms presented in this paper are in Tables 9 and 12. Cached pages from these websites are available from the authors for validation and other development work.

B Full web site list

Website	Date
agents.umbc.edu	July 2000
www.alphabetstreet.infront.co.uk	March 2000
www.alphaworks.ibm.com/	March 2000
www.amazon.com	December 1999
www.aw.com	December 1999
www.bookpool.com	March 2000
cbc.ca/consumers/	March 2000
www.chapters.com	March 2000
www.google.com	March 2000
www.hotbot.com	March 2000
www.ibm.com/developer/java	March 2000
www.kingbooks.com	March 2000
www.loc.gov	March 2000
www.rubylane.com	March 2000
www.signpost.org	March 2000

Table 21: Test Web Site List

Website	Date
www.amazon.com	March 2000
www.amazon.com (ZShops)	March 2000
www.bn.com	March 2000
www.bookbuyer.com	March 2000
www.borders.com	March 2000
www.canoe.com	March 2000
www.codysbooks.com	March 2000
www.ebay.com	March 2000
www.etoys.com	March 2000
www.excite.com	March 2000
www.fatbrain.com	March 2000
www.gameCenter.com	March 2000
www.gamelan.com	March 2000
www.goto.com	March 2000
www.ibm.com	March 2000
www.ibm.com/developer/xml	March 2000
www.msn.com/auctions	March 2000
www.powells.com	March 2000
www.quote.com	March 2000
www.thestar.org	March 2000
www.vancouversun.com	March 2000
www.vnunet.com	March 2000
www.wine.com	March 2000
www.yahoo.com	March 2000
www.yahoo.com/auctions	March 2000

Table 22: Experimental Web Site List

Website	Date	Page Count
agents.umbc.edu	July 2000	1
www.alphabetstreet.infront.co.uk	March 2000	30
www.alphaworks.ibm.com	March 2000	30
www.amazon.com	December 1999	99
www.amazon.com	March 2000	73
www.amazon.com (ZShops)	March 2000	76
www.amazon.com (ZBooks)	March 2000	24
www.aw.com	March 2000	9
www.bn.com	March 2000	83
www.bookbuyer.com	March 2000	82
www.bookpool.com	March 2000	4
www.borders.com	March 2000	88
www.canoe.com	March 2000	100
www.canoe.com (web search)	March 2000	100
cbc.ca/consumers/	March 2000	43
www.chapters.com	March 2000	100
www.cnet.com (game search)	March 2000	99
www.cnet.com (articles)	March 2000	100
www.cnet.com (web search)	March 2000	100
www.codysbooks.com	March 2000	100
www.ebay.com	March 2000	93
www.etoys.com	March 2000	36
www.excite.com	March 2000	100
www.fatbrain.com	March 2000	71
www.gameCenter.com	March 2000	6
www.gamelan.com	March 2000	53
www.google.com	March 2000	100
www.goto.com	March 2000	100
www.hotbot.com	March 2000	27
www.ibm.com	March 2000	65
www.ibm.com/developer/xml	March 2000	72
www.ibm.com/developer/java	March 2000	34
www.redbooks.ibm.com	March 2000	41
www.kingbooks.com	March 2000	69
www.loc.gov	March 2000	84
www.lycos.com	March 2000	100
auctions.msn.com	March 2000	1
www.powells.com	March 2000	84
www.quote.com	March 2000	1
www.rubylane.com	March 2000	1
www.sfgate.com	March 2000	35
www.signpost.org	March 2000	55
www.thestar.org	March 2000	1
www.vancouversun.com	March 2000	18
www.vnunet.com	March 2000	81
www.wine.com	March 2000	20
auctions.yahoo.com	March 2000	1
www.yahoo.com	March 2000	96

Table 23: All cached Web Sites