Tree Pattern Inference and Matching for Wrapper Induction on the World Wide Web

by

Andrew William Hogue

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2004

Author . . . .
Department of Electrical Engineering and Computer Science
May 13, 2004

Certified by . . . .
David R. Karger
Associate Professor
Thesis Supervisor

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

Tree Pattern Inference and Matching for Wrapper Induction on the World Wide Web

by

Andrew William Hogue

Submitted to the Department of Electrical Engineering and Computer Science
on May 13, 2004, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

We develop a method for learning patterns from a set of positive examples to retrieve semantic content from tree-structured data. Specifically, we focus on HTML documents on the World Wide Web, which contain a wealth of semantic information and have a useful underlying tree structure. A user provides examples of relevant data they wish to extract from a web site through a simple user interface in a web browser. To construct patterns, we use the notion of the *edit distance* between the subtrees represented by these examples to distill them into a more general pattern. This pattern may then be used to retrieve other instances of the selected data from the same page or other similar pages. By linking patterns and their components with semantic labels using RDF, we can create semantic "overlays" for Web information which are useful in such projects as the Semantic Web and the Haystack information management environment.

Thesis Supervisor: David R. Karger
Title: Associate Professor

Acknowledgments

I would first like to extend my deepest gratitude to my advisor, David Karger, without whose insights, advice, extensive feedback, and financial support this work would have been impossible.

The members of the Haystack research group have been an invaluable resource for ideas and assistance. In particular, Karun Bakshi, David Huynh, Dennis Quan, and Vineet Sinha deserve my gratitude for both their insights for and patience with someone learning the Haystack system.

My roommate and friend, Jaydeep Bardhan, deserves my thanks not only for the services of his whiteboard, but for being a sounding board for ideas and for redirecting me towards (or away from) the task at hand as necessary.

My family has always been there for me, supporting my interests no matter where they took me, and I have them to thank for setting me along this path in life. Mom, Dad, Paul, Greg, and John, thank you. My "second" family, as well, has always been there for me the last seven years, even when times got rough, and I would not have made it here with out them.

Finally, words cannot express the love and gratitude I have for my wife, Tiffany, although it doesn't stop me from trying. For supporting me this year and all the years before, for being my best friend and closest confidant, I thank you from the bottom of my heart. All of this is done for you.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The promise of the Semantic Web is to "bring structure to the meaningful content of Web pages, creating an environment where software agents roaming from page to page can readily carry out sophisticated tasks for users." [7] Information which is currently prepared only for humans to read will be richly labeled, classified, and indexed to allow intelligent agents to schedule our appointments, perform more accurate searches, and generally interact more effectively with the sea of data on the Web.

These advances, however, rely on the accurate semantic labeling of data that currently exists only in human-readable format on the World Wide Web. Normally, this labeling would be a task for content providers, as they have easy access to the relational data which makes up the pages, as well as the ability to alter the existing published content. Several tools, such as browsers and distributed search engines for ontologies, have been developed explicitly with the goal of making it easier for content providers to add semantic markup to their existing World Wide Web pages.

Unfortunately, providers often have little or no incentive to mark up their existing documents. In this work, then, we take a different approach. Our goal is to provide a tool which allows *end-users*, rather than content providers, to author and utilize their own semantic labels for existing content. In particular, we aim to make the extraction of semantic content accessible to non-technical users, modifying existing user interfaces to provide them with the ability to label web pages with semantic meaning. By giving these users control over semantic content, we hope to reduce the reliance of the Semantic Web on content providers and speed its adoption.

Our system allows users to create semantic patterns, also called *wrappers*, by simply browsing to a web site, highlighting its relevant semantic features, and providing semantic labels for them. From these positive examples, a flexible, reusable pattern is induced.

The patterns described here are created by a powerful algorithm which takes advantage of the inherent hierarchical structure of HTML. We utilize the technique of *tree edit distance* to find the *best mapping* between the given examples. This mapping allows us to highlight and extract only the structural elements that the examples have in common, discarding any instance-specific content. What is left is a generic pattern, capable of recognizing other instances of the same type.

Once a wrapper is created, the user may then give it semantic meaning by overlaying it with statements about the classes and properties it represents. These descriptions are created through a simple user interface, but are underlaid by statements in RDF, the language which is the framework for the Semantic Web. By drawing these classes and properties from an existing ontology appropriate for the page in question, the user gives the wrapper a general meaning compatible with other sites of the same type.

On subsequent visits to the same page, or to similar pages on the same site, the predefined wrapper is reevaluated. When matches are found, instances of the semantic classes represented by the pattern are created on the fly. The semantic predicates which the user has bound to the pattern are then used to "fill in" the properties of this instance with data from the page.

We then give the user the ability to interact with these new objects by "overlaying" them on the browser window. The area of the page which has been matched by the pattern now also has the contextual *meaning* of the matched object. This process allows us to reify the matched items into first-class objects in the user environment, rather than flat text.

By integrating this tool into the Haystack [24] information management environment, these semantic overlays create a rich environment which allows the user to interact with content on the Web. Semantic content on the web goes from being flat text to having full semantic meaning and context. For instance, a pattern defined on a page containing upcoming seminars would allow the user to right-click and add an interesting talk to their calendar. Patterns defined on news sites would allow the user to create, modify, and subscribe to their own RSS feeds.

Wrappers also provide a powerful means for importing, exporting, and manipulating the unstructured data on the Web. Once wrapped, the information is in a structured, relational format, RDF, that can be easily managed and queried. Disparate sources of similar data can be easily brought together. For instance, wrappers created on several news web sites can be integrated into a single RSS feed. Alternatively, wrappers of the same semantic type allow us to reformat and integrate data. A user could integrate all their news sites into a single page, formatted in whatever way is best for that user.

By creating wrappers, users are, in effect, creating a bridge between the *syntactic* structure and the *semantic* structure of the web page. In general, this parallel structure has always existed, abstractly, in the intentions of the page's creator and in the interpretations of the page's reader. In our system, however, the act of building a wrapper for this content makes the connection explicit on the user side. It is from this syntactic-semantic bridge that our wrappers get their power.

We begin by surveying related work in several fields in Chapter 2. Chapter 3 characterizes the types of semantic and syntactic structure we expect to see on the World Wide Web. We then describe the main hierarchical wrapper induction and matching algorithm in Chapter 4, and show how to apply semantic meaning to these wrappers in Chapter 5. The user interface for wrapper induction and matching is described in Chapter 6. We next give our experimental results in Chapter 7. Finally, we propose several avenues of future research and give our conclusions in Chapter 8. Three appendices describe the web sites and types of semantic content that were surveyed in creating our algorithms (Appendix A), the results of running our algorithm on several of those sites (Appendix B), and several details of our implementation (Appendix C), including both programming interfaces and issues which arose.

Chapter 2

Related Work

2.1 Information Extraction

Our approach to pattern induction and matching is one case of the larger task of *information extraction*. This field, especially in relation to documents on the World Wide Web, has received much attention in recent years. Information extraction covers the automated retrieval of data from both structured and unstructured documents. Various approaches, using both supervised and unsupervised learning, have been tried, with varying degrees of success.

The subfield of information extraction dealing with documents on the World Wide Web is called *wrapper induction*, defined by Kushmerick [20] as the task of learning a procedure for extracting tuples from a particular information source from examples provided by the user. Kushmerick defined the HLRT class of wrappers, implemented in the WIEN (Wrapper Induction ENvironment) system. These wrappers were restricted to locating information which is delimited by four types of flags, the "head," "left," "right," and "tail." Because of this limitation, this class was found to successfully wrap only 48% of relational data in HTML documents in a 1997 Web survey.

A related approach, the STALKER system [23], attempts to capture some of the hierarchical structure of HTML and its semantic data. The *embedded catalog* ($\mathcal{EC}$) formalism consists of lists of k-tuples, with each element of the k-tuple being either information of relevance to the user or another k-tuple. The $\mathcal{EC}$ description of a page is therefore a hierarchy similar to the subject-predicate-object description used by RDF for the Semantic Web. The $\mathcal{EC}$ of a page allows the STALKER system greater flexibility with fewer examples in locating information in a page than HLRT wrappers. However, despite the hierarchical nature of the $\mathcal{EC}$, STALKER's matching algorithm still treats the underlying HTML source of pages as a linear string, ignoring its hierarchical structure.

Several other approaches to information extraction utilize probabilistic models. Hidden Markov Models offer the opportunity to learn not only the the probabilities but the state structure to represent information in various types of documents [12, 27]. These approaches also treat the document as a linear set of objects, first parsing it into tokens such as HTML tags and text. The state structure of the HMM is either hand crafted, or is learned from the set of training examples using stochastic optimization. These models have been quite successful at extracting information from semi-structured documents, such as academic papers, but their usefulness on HTML documents is unproven.

Other models learn to classify data by assuming that "nearby" elements in a hierarchy should be classified similarly [28]. For instance, the document tree on a filesystem or web site is useful for inferring which documents are related. If most documents in a certain directory subtree have been classified in a certain way, new documents appearing in nearby subtrees are more likely to be similarly classified.

Probabilistic Context Free Grammars, or PCFGs, are a statistical technique for semantically tagging semi-structured data [9]. When used to extract semantic content from English sentences, a probabilistic model is learned by parsing tagged training examples and noting the frequency of occurrences of certain context-free rules among the training data. This model may then be used to tag new sentences by finding the most likely set of rules which could have created that sentence in the model. With the addition of "semantic" tags to the training examples, PCFGs may be used to label phrases with semantic meaning, the first step in higher-level information extraction. For example, PCFGs have been successfully used to extract the post, company, person entering and person leaving the post from newspaper texts describing corporate management successions.

One example of an interactive system for learning patterns on various types of documents is LAPIS [26]. Similar to our approach, this system provides an interactive interface where users may specify examples relevant to a pattern by highlighting them. Patterns are constructed using a language called *text constraints*, which includes operators such as *before*, *after*, *contains*, and *starts-with*. By using a pre-defined library of parsers which tokenize and label the document, users can create patterns of arbitrary complexity, or allow the system to infer them from examples. This inference is performed by constructing a dictionary of *region sets*, which describe areas of the document which match certain tokens from the parsers. By analyzing the overlaps and repetitions of these region sets, LAPIS extracts its structured text constraints patterns. The results of matching these patterns are then displayed for the user, allowing them to perform such tasks as simultaneous editing and outlier finding. While it currently has no ties to the Semantic Web, LAPIS is a powerful pattern induction system, and our system will take advantage of its parsing abilities in cases where our tree edit distance algorithm is not applicable.

2.2 The Semantic Web

As mentioned in Chapter 1, the Semantic Web is a tool which promises to "bring structure to the meaningful content of Web pages, creating an environment where software agents roaming from page to page can readily carry out sophisticated tasks for users." [7] These advances, however, rely on the accurate semantic labeling of data that currently exists only in human-readable format on the World Wide Web. Existing Semantic Web projects have tended to focus on enabling content-providers to produce this semantic content, and end-users to consume it, whereas few have had our goal of allowing *end-users* to create their own metadata.

The MIND SWAP project [13] has created a suite of tools which enable users to author, search, and browse Semantic Web documents. These include a program for converting tab- and comma-delimited files to RDF, an editor for creating HTML and

RDF documents at the same time, an interface for labeling non-text content with RDF, and an ontology manager for search and browsing. The most relevant tool to our work is the Web Scraper, which allows users to extract semantic information from structured HTML documents. To use the Web Scraper, a user must analyze the HTML source of a document and provide explicit delimiters for relevant information. Once the data has been extracted, an ontology browser is provided, allowing the user to choose appropriate semantic labels from existing Semantic Web ontologies. While the patterns created by the Web Scraper tend to be more powerful than those described here because of their explicit declaration, the interface for defining them is complex, requiring a knowledge of HTML and the low-level structure of a page. Our system has been designed with the non-technical user in mind, and allows pattern induction through a standard web browser interface, using operations such as highlighting and right clicking on a document.

The XPath [4] standard is another useful language for extracting data from hierarchical documents such as XML and HTML, and several tools such as xpath2rss [6] have been built with it. Similar to the Web Scraper, though, these require the user to have a detailed knowledge of the document and of the language for describing patterns, and few tools have been developed to allow intuitive induction of useful patterns.

The Annotea project [17] is a Web-based system which allows users to add descriptive metadata to individual pages. The system is implemented using current web standards, and allows users of an Annotea-enabled browser to create annotations and associate them with text within a given document, or with the page as a whole. Similarly, the concept of Sticky Notes for the Semantic Web [18] has been proposed, also allowing users to add annotations to existing documents. Both of these applications are interesting in that they give non-technical end users the ability to annotate and supply semantic information for any web document. However, these annotations must be provided on a page-by-page basis, as no underlying pattern scheme is learned to make the annotations more widely applicable. Our system aims to generalize these types of annotation, allowing a user to utilize them with every occurrence of the same semantic data on that web site.

Our work extends the Haystack [24] information management client. This system has strong ties to the Semantic Web, in that it is based on the RDF standard [3]. The main benefit to the Haystack interface is that every object is semantically "alive" to the user. This means that the system can provide relevant context menus for any element displayed on the screen. For instance, in the interface for composing an email message the "To" and "CC" fields not only provide context menus for adding additional recipients, but also provide menus to interact with the actual "Person" objects behind existing recipients. These semantically-driven context menus will be important to our implementation of semantic wrappers on the Web.

2.3 Tree Data Structures and Algorithms

Tree pattern matching algorithms have been well studied in fields such as compiler optimization and molecular biology [8, 11, 15]. Traditional approaches have focused on queries that are hand-crafted by users, requiring intimate knowledge of both the pattern matching semantics as well as the underlying structure of the data being queried. Most approaches focus on matching structures in unordered trees with either limited or no wildcards.

As mentioned earlier, the problem of locating nodes in a structured document tree by specifying the path from root to node is addressed by the XPath standard [4]. XPath allows users to specify path-structured queries which return sets of nodes from an XML document's tree. Once again, XPath is designed for technical users, and little work has been done on inducing XPath queries from examples provided by novice users.

A related problem is that of *subtree isomorphism* [16, 21], which attempts to find similarly *shaped* structures in trees. These algorithms generally treat the structure of the given tree without regard to node labels or sibling order. They are most often specializations of the more difficult task of *graph isomorphism* [22], whose complexity is unknown, but may lie between P and NP-complete [30]. Our approach attempts to avoid the complexity issues involved with these algorithms by making assumptions about the structure, ordering, and labeling of the pattern and document trees we are considering.

The problem of pattern matching on strings also has a long history. The edit distance between strings may be computed in polynomial time using dynamic programming [10]. This technique is used by the `agrep` program [31] to find approximate matches to a query string. The *tree edit distance* algorithm [32] upon which our approach is based is an extension of string edit distance. Both algorithms find the least-cost method for turning one object into another using operations such as insert, delete, and change.

Chapter 3

Pattern Characterization

To attack the problem of learning patterns for information extraction, we must first characterize the types of structures we expect to see when looking at semantic data on the web. These characterizations are the result of a survey of a wide variety of structured semantic content across several popular web sites.[1]

In the discussions below, we will denote semantic classes and properties using typewriter font. Classes are generally capitalized, while properties will be lowercase. For example, the class `Movie` has the property `director`.

3.1 Repeated Instances

The first general characteristic of interesting data on the web is that it tends to "repeat" itself. This is not to say that the *content* is the same, but that the *structure* that displays this content is the same or similar.

There is an underlying reason for this repetitive nature which is based in the way web pages are written. Page authors tasked with presenting relational data generally use CGI [1] scripts to automatically generate these web pages. These scripts retrieve relational data from a database and dynamically publish it to the web using HTML templates. Each template has slots for a certain subset of the data, and the script simply fills this template from its database query before sending the response to the

[1] See Appendix A for details of the sites and content surveyed.

user's browser.

These templates give web pages their *syntactic* structure. They provide elements for formatting, presentation, font, and other visual elements. The relational databases that provide the data for the templates give pages their *semantic* structure. The slots in the templates are filled with the same type of semantic data every time. In this work, we will attempt to "invert" this process, inferring the structure of these templates and the semantics of their contents from examples given by the user.

It is important to notice that this templating process manifests itself in two ways. The first is multiple occurrences of the same semantic type in the same page. In this case, several rows from the relational database are being displayed at once. Because they have the same semantic type, the page author has used the same template to format them. For example, when a user performs a search on Google,[2] they get back a page like that in Figure 3-1, containing several instances of the type `SearchResult`. Note that each instance is syntactically formatted in the same basic way, with a `link` to the result page, a `summary` of the result, the `hostname` and `size` of the page in green, and a link to the `cached` page and to `similarPages`.

The other way that templates are used is to format an entire page. In these cases, the full page generally represents a single semantic class, and the slots in the page contain its properties. This allows the user to consider a single object at a time along with all of its attributes. As an example, two pages from the Internet Movie Database[3] are shown in Figure 3-2. Here we see two examples of the type `Movie`, served on two different pages, which show the same underlying structure. Each has a `title`, a `year`, a `director` and `writers`. Other items such as the `plotOutline` and `rating` are also structurally similar.

The syntactically and semantically repetitive nature of relational data suggests a format for our patterns. Each pattern should form a "template" for the general class of data we are trying to match. This template should capture the elements that are common between different instances, but leave "slots" where instance-specific

[2] `http://www.google.com/`
[3] `http://imdb.com`

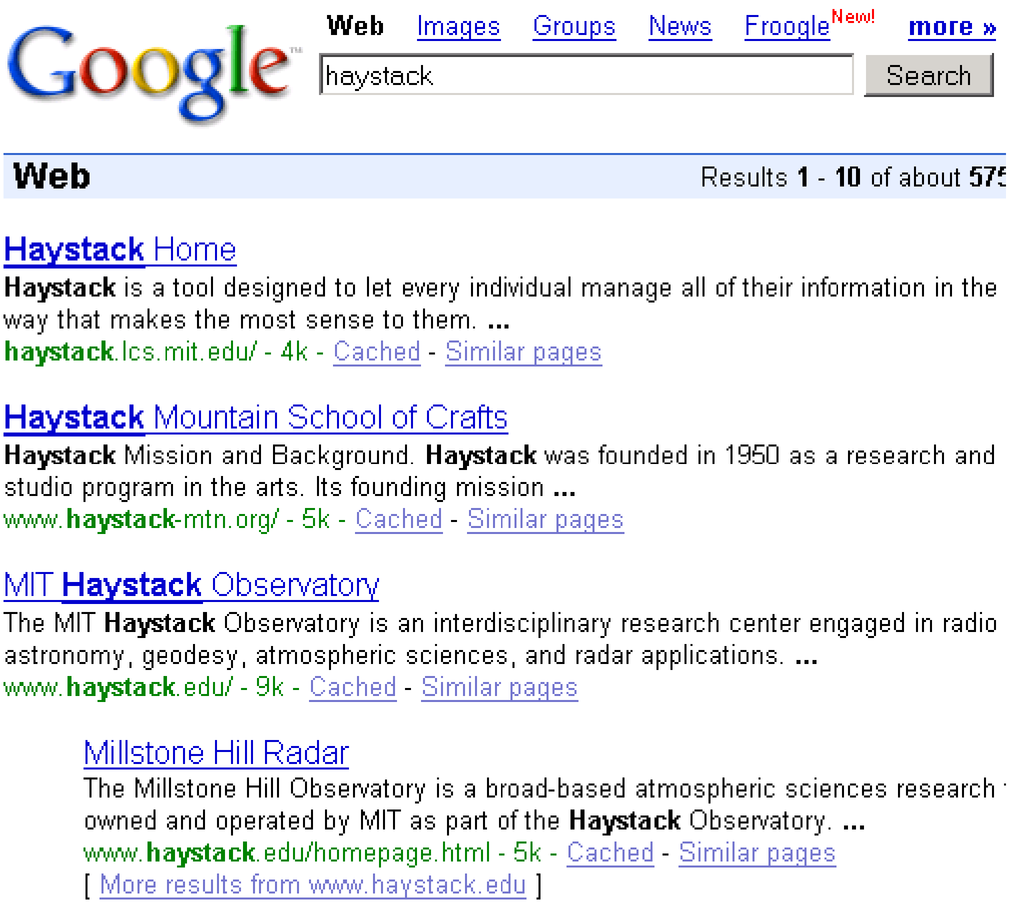


Figure 3-1: A sample search on the Google search engine (`http://google.com`).

properties are located. When we match a pattern, we will effectively “fill in” these slots, allowing us to use the semantic data for other applications.

3.2 Internal and Leaf Nodes

Data formatted in HTML has an underlying *tree* structure which describes how elements of the document are formatted. Working down from the document’s root, each node in the tree provides more specific information to the browser about how elements should be formatted. Many display elements, including text and images, may only appear at leaf nodes.

Because of the way HTML is rendered, nodes in the same subtree of the document are displayed in the same contiguous block in the browser. Importantly, this relation

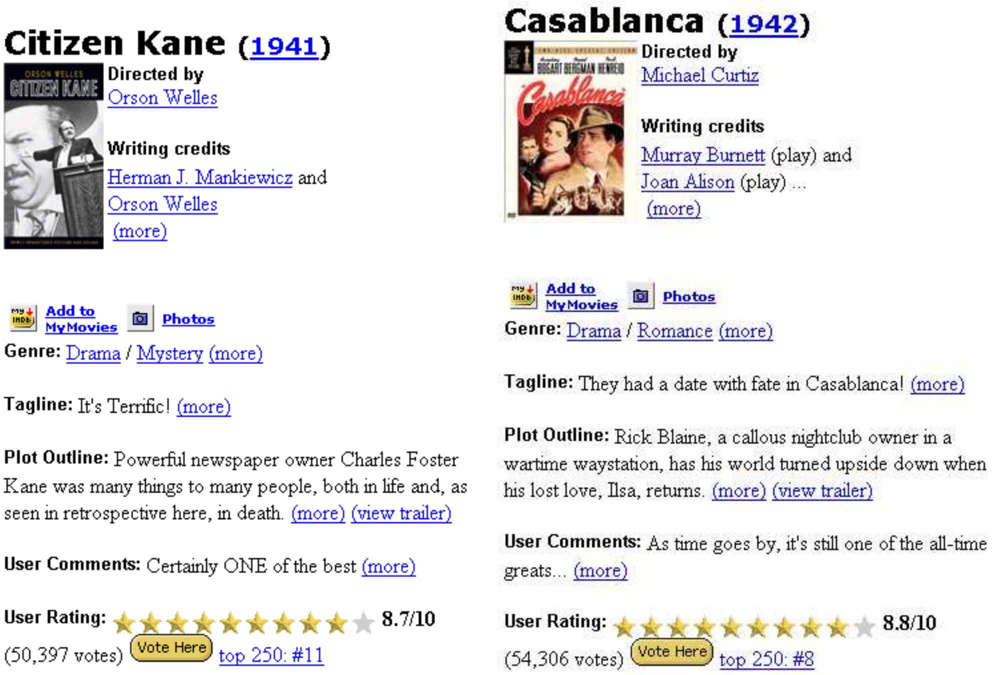


Figure 3-2: Two pages from the Internet Movie Database (`http://imdb.com`).

works both ways - neighboring elements in the browser are backed by nodes in the same subtree of the page's HTML. Thus, an instance of a semantic type displayed in one section of a page is represented by a subtree of nodes in the document tree. This relationship may also be thought of as a result of the templating process described in the previous section. Page authors using templates to format their data tend to group that data together structurally. This grouping makes it easier to "glue" templates together into a full page, as well as to reuse templates for similar data on separate pages.

Along with the general repetitive nature of semantic types, we also notice that there is a pattern to *which parts* of these subtrees actually change between examples. In almost all cases, the nodes that differ between examples are leaf nodes, or entire subtrees at the lowest level. The "upper" structural elements, or the internal nodes of the subtree, tend to be consistent among instances of the same type. For example, in the Google `SearchResult` instances depicted in Figure 3-1, only the text of each result changes. The internal nodes, which affect layout, font selection, and link structure, are identical between the instances.

This idea that the text of the semantic instances is different, but the internal structure is the same, suggests a way to formalize the "template/slot" idea mentioned in the previous section. We will take the common, internal structure of the instance subtrees as our template, but discard the instance-specific leaf nodes. These discarded nodes will later form the "slots" which bind to semantic data.

3.3 Subtree Structure

In this section, we attempt to classify the distinct types of syntactic structure we have observed in our survey of web pages containing relational semantic data. We have categorized these types of structure along two axes:

Spanning Elements The nodes that comprise a single example may either be grouped under a *single node*, or be spread across *multiple sibling nodes*.

Semantic Siblings Examples of the same class or property may either be *siblings*, sharing a parent node, or be located in *separate* parts of the document tree (or even on separate pages).

We now proceed to investigate these axes in more depth, and give examples of their occurrence in practice.

3.3.1 Single Siblings

As mentioned above in our discussion of HTML templates, one of the most common types of syntactic structure for semantic content is within a single subtree of the document. All semantic information is grouped syntactically under a single node. Formatting elements are children or descendants of this root node.

This structure is beneficial from both the author's and the user's perspectives, as HTML content in the same subtree is rendered together in the browser. This means that properties of the same instance of an object are kept together on the page, which is important from the standpoint of a simple and consistent user experience.

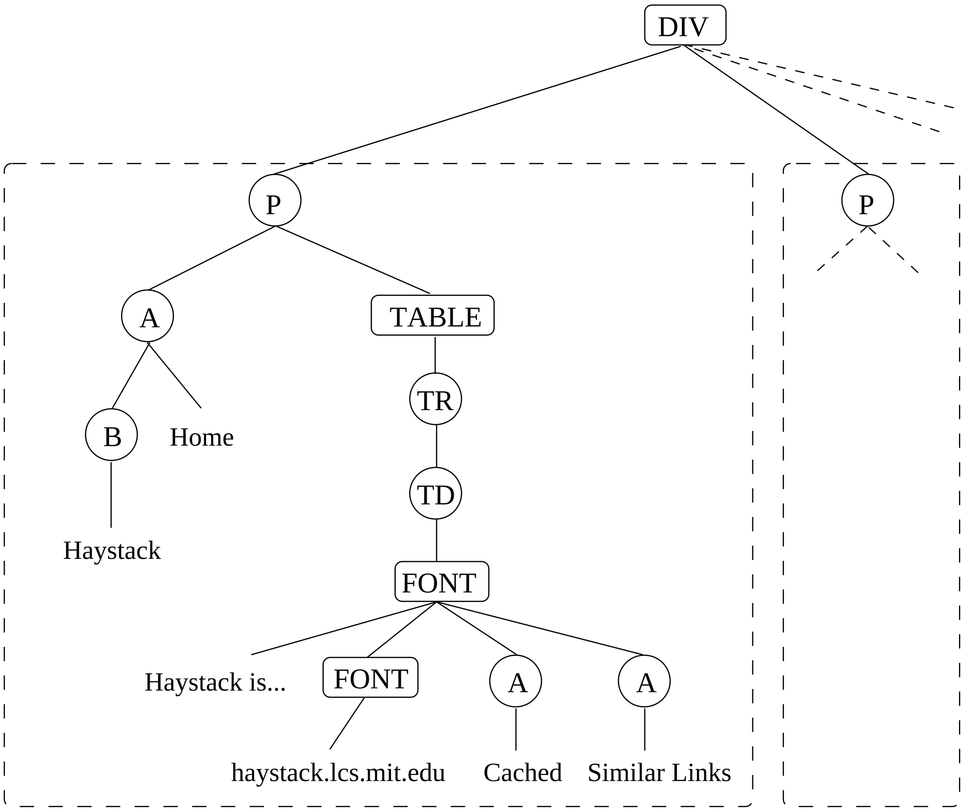


Figure 3-3: An example of semantic content grouped under a single node.

Many examples of this type exist. For example, a simplified version of a Google `SearchResult` is shown in Figure 3-3. In this figure, the dashed lines group instances of a single class. Note that all semantic properties for each `SearchResult` object are grouped under their own root node.

This type of structure will work well with our tree edit distance approach, as we can simply calculate the edit distance between the example subtrees.

3.3.2 Multiple Siblings

In other cases, the content for a single semantic class spans several neighboring subtrees. These instances often appear similar to the user when rendered in the browser because sibling subtrees are displayed next to each other, just as the contents of a single subtree are.

One example of this type of structure is the `NewsStory` type on the New York

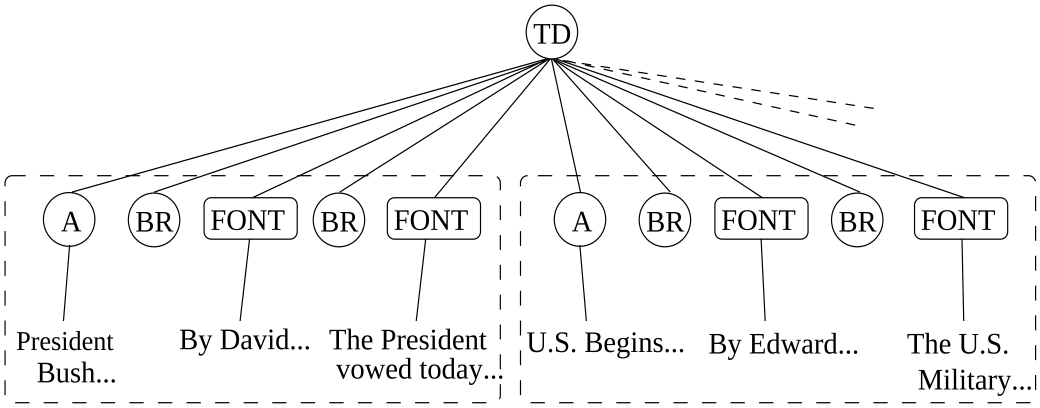


Figure 3-4: An example of semantic content grouped under several sibling nodes.

Times web site,[4] as shown in Figure 3-4. Once again, nodes which represent single `NewsStory` instances are outlined with dashed lines. Note that each `NewsStory` spans several nodes with the same parent, but that there are also other nodes with that parent that are part of *other* instances.

This type of structure will be more difficult to form patterns for, as the tree edit distance algorithm is not defined for forests.

3.3.3 Sibling Repeats

Another consideration when dealing with repetitive semantic content is where the repeated structure occurs. In one case, several repeated instances of the same content occur with the *same* parent node. These repeats will prove important when we attempt to generalize our patterns from a single user example. If we find that neighboring subtrees have similar structure, we can automatically generalize them in our pattern without the need for user interaction.

An example of this structure is the set of Google `SearchResult` instances shown in Figure 3-1, with the structure shown in Figure 3-3. Note that all `SearchResult`s have the same parent node in Figure 3-3. This means that if the user indicates one of these results, we can easily find other subtrees with the same type by simply looking at the siblings of the initial example.

[4] `http://nytimes.com`

3.3.4 Non-sibling Repeats

The other type of repetitive structure we must consider is that in which instances of the same semantic type do *not* share a common parent. This structure most often occurs *between* web pages. Because we have no way of heuristically finding additional examples of the same type of content, this type of structure will usually require at least two examples from the user to form a general pattern.

An example of this structure is the IMDB `Movie` class. Because each `Movie` is on its own page, the user must often specify an example from two different `Movie` instances to form a general pattern. For instance, to wrap the property `director`, the user would have to visit two different pages, highlighting the text of the property on each one.

Chapter 4

Wrapper Induction and Matching

In this chapter we build up a method for creating patterns, or *wrappers*, from users' examples. We start with basic types of patterns using root-to-node paths, and proceed to generalize them to handle more complex structures often seen on the web.

The common ground between all the pattern induction methods we present in this chapter is the desire to locate certain nodes in a document tree that have semantic meaning for the user. A user forms a pattern by indicating a set of nodes in the document as examples. This chapter is about how to form a pattern based on those nodes.

We will consider two main ways of capturing the properties of nodes in the document tree. First, we consider the root-to-node *path* of the node. This "extrinsic" property allows us to capture and compare the *location* of the node in the tree as compared with other nodes. Second, we will attempt to generalize the *structure* of the subtree rooted at the node. This "intrinsic" definition allows us to link the syntactic structure of the node with other nodes and with its semantic meaning.

As noted in Section 3.2, because of the way HTML is rendered in a browser, when a user selects a contiguous block of a web page, they are, in effect, selecting a subtree[1] from the page's DOM. Thus, by providing highlighted examples in the browser, the user is indicating that several subtrees in the DOM have both similar syntactic structure and similar semantic meaning. In the discussion below, then, we

[1]Or a set of sibling subtrees. See Section 4.4.3 for heuristics dealing with this case.

reference the selection and the selected subtree interchangeably.

4.1 Definitions

We will be working with trees T indexed in preorder as $T[i]$. The size of a tree is denoted $|T|$, and the size of the subtree rooted at node v is denoted $|v|$.

Each node v in a tree has a label, denoted $label(v)$. In the case of an HTML document, the label is either the tag name for a markup node, or the actual text of the node in the case of text nodes. We will also index each node in terms of its position among the children of its parent, $sibling\text{-}num(v)$. We define $sibling\text{-}num(T[0]) = 0$ for the root.

Finally, two nodes are considered equal if their labels are equal[2]. In talking about patterns we will create special *wildcard* nodes, denoted with the label "$*$". Wildcard nodes are considered to be equal to any other single node.

4.2 Path Labels

The most basic way to uniquely locate a node in a tree is by its labeled path from the root. We define the *path* of a node to be the sequence of nodes from the root of a tree to a node v. In particular, we define two "flavors" of path. The first is the *sibling* path, σ, containing $sibling\text{-}num(w)$ for each node w along the path to v. The second is the *tag* path τ, representing the sequence of tag names $label(w)$ for each node w along the path to v. As an example, in Figure 4-1, each node is labeled with both σ and τ.

Both types of paths have benefits for pattern matching. First, the sibling number path, σ, *uniquely identifies* a node within the tree, whereas τ does not. For example, there are two nodes with τ = `A.B.C` in Figure 4-1, each with a different σ (`0.0.0` and `0.1.0`).[3] This ability to uniquely identify a node's location within the document

[2]We currently do not consider DOM attributes, such as the `src` of an `IMG` tag or the `href` of an `A` tag, for the purposes of node equality. We discuss this enhancement in Section 8.2.1.

[3]We will find it convenient to refer to paths either as arrays (indexed starting from the root down

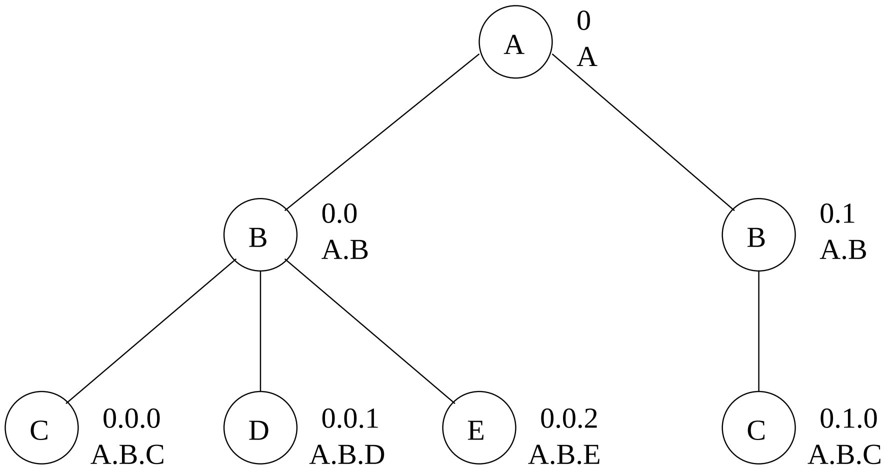


Figure 4-1: An example tree labeled with σ and τ, the two types of root-to-node paths.

tree makes σ useful for locating similar nodes between different documents, or for reidentifying the same node across multiple visits to the same page.

On the other hand, for the purposes of pattern matching on a single page, finding multiple nodes with the same τ will be beneficial. As we described in Section 3.1, similar semantic content is often found in similarly structured parts of a page. Nodes with identical formatting (for instance, the rows of an HTML `table`), have the same τ. Thus, finding all nodes in a page with the same τ is often a good heuristic for finding other matches for the same semantic content.

One extension to these definitions is important to handle user selections that span multiple sibling nodes. The last (most specific) element in σ or τ may be a *range* of elements, rather than a single element. For instance, the user might select both nodes `0.0.1` and `0.0.2` in Figure 4-1, resulting in σ = `0.0.[1-2]` and τ = `A.B.{D,E}`. Paths which contain ranges may still be used as described above to locate similar nodes both within and between pages.

Our most basic form of pattern induction, then, is to simply use the sibling and tag paths of the selection made by the user. We utilize two tactics, depending on whether the semantic content repeats many times on the same page or only occurs once on the page. When the user selects a node, we first look for semantically similar

to the leaf), or as strings, concatenated by "." characters.

nodes using the tag path, τ. If we find them, we store τ as the pattern. When the user returns to that page (or a similar one), we simply find all nodes in the page with that τ. If, on the other hand, we do not find other nodes with the same τ, we store the sibling path, σ, as the pattern. On future visits to this or similar pages, this will allow us to locate and match the same node in the document.

4.2.1 Path Functions

We now define two functions on node paths that will be useful in our subsequent discussions. The first is the *contains* relation, defined for sibling paths. Formally, a node w is contained by another node v if either $v = w$ (they are the same node), or if w is a descendant of v in the tree. In terms of paths, $contains(\sigma_v, \sigma_w)$ iff $\sigma_w[i] = \sigma_v[i]$ for all nodes in σ_v. A similar relation holds for the tag path, τ.

Another useful function will be the ability to convert an *absolute* path (from the root of the tree to a specific node) to a *relative* path, rooted at another ancestor of the specific node. To turn σ_v into a path relative to σ_w:

$$relative\text{-}path(\sigma_v, \sigma_w) = \{0, \sigma_v[|\sigma_w|..|\sigma_v| - 1]\}$$

For instance, *relative-path*(`0.1.2.3.4.5`, `0.1.2.3`) = `0.4.5`. As with *contains*, *relative-path* also holds for the tag path, τ.

4.3 Best Mapping

Locating nodes using the root-to-node path has several restrictions. For instance, while two nodes might have similar semantic meaning, one might be offset by an extra formatting element to emphasize it, thus changing its τ. Also, semantic content on similar pages may be located in slightly different locations (for instance, if it is offset because of advertising content), thus changing its σ. Only considering nodes' path to the root would make it very difficult to form consistent, flexible, reusable patterns. In addition, because paths do not capture any information about the structure of the *subtree* rooted at the selected node, they risk missing important features that may be important for pattern matching.

Instead, in this section, we consider utilizing the structure around and below the selected nodes in the tree to form patterns. By making the assumption that two nodes with similar semantic meaning also have similar syntactic structure, we may form patterns by finding their common syntactic elements. We may then use that syntactic pattern to attempt to find other nodes with the same semantic meaning.

To do this, we will consider the *best mapping* between two subtrees in the document. This mapping, a byproduct of finding the *edit distance* between two trees, will allow us to extract only those nodes which the two subtrees have in common. Nodes which are not in common between the two trees remain unmapped, and thus are not included in our patterns.

We first develop the notions of the edit distance and the best mapping between two trees, and then show how these can be used to form general patterns. We then extend these basic algorithms with other heuristics which allow us to form more general patterns, often from only a single example subtree.

4.3.1 Tree Edit Distance

The *tree edit distance* between two trees T_1 and T_2 is defined as the cost of a sequence of edit operations which transform T_1 into T_2. The possible operations include inserting a node, deleting a node, and changing one node into another, each of which have an associated cost, γ. The *best mapping*, M, is the lowest-cost set of edit operations to turn T_1 into T_2. M is defined as follows (based on Zhang's algorithm [32]):

- M is a set of pairs of integers (i, j) satisfying:

    1. $1 \leq i \leq |T_1|$, $1 \leq j \leq |T_2|$
    2. For any pair (i_1, j_1) and (i_2, j_2) in M,

        (a) $i_1 = i_2$ iff $j_1 = j_2$ (one-to-one),

        (b) $T_1[i_1]$ is to the left of $T_1[i_2]$ iff $T_2[j_1]$ is to the left of $T_2[j_2]$ (sibling order preserved),

(c) $T_1[i_1]$ is the parent of $T_1[i_2]$ iff $T_2[j_1]$ is the parent of $T_2[j_2]$ (parent-child relationships preserved)[4].

- If $T[i]$ is an ancestor of $T[j]$, and M contains the pair (i, Λ), where Λ is the empty node, then M also contains (j, Λ). Similarly, if M contains (Λ, i), it also contains (Λ, j). Intuitively, if we insert or delete a node, we must also insert or delete all nodes in its subtree.

- The *cost* $\gamma(M)$ of a mapping M from T_1 to T_2, is defined to be

$$\gamma(M) = \sum_{i,j \in M} \gamma(T_1[i] \to T_2[j]) + \sum_{i \notin M} \gamma(T_1[i] \to \Lambda) + \sum_{j \notin M} \gamma(\Lambda \to T_2[j])$$

  where Λ is the empty node.

For our problem, we define the cost of an individual operation to be

$$\gamma(v \to w) = \begin{cases} 1 & \text{if } v \neq w \\ 0 & \text{otherwise.} \end{cases}$$

Intuitively, the cost of inserting or deleting a node is the cost of inserting or deleting the entire subtree rooted at that node. Changing a node is equivalent to a delete and an insert. Note that, due to the second condition above, inserting or deleting a node costs as much as inserting or deleting the entire subtree rooted at the node.

Note also that this cost definition is symmetric for subtrees $T[i]$ and $T[j]$:

$$\gamma(v \to w) = \gamma(w \to v).$$

We can calculate the least-cost tree edit distance between v and w using a dynamic programming approach, similar to the calculation of the edit distance between strings. We first check whether $v = w$ (as per our definition of equality in Section 4.1). If

[4]Note that this is not the same definition as Zhang's algorithm, which only requires that *ancestor* relationships be preserved. Our restriction is stronger, and allows for a significant improvement in speed.

they are not equal, we add $delete(v)$ and $insert(w)$ to our mapping M, as well as the appropriate *inserts* and *deletes* for all nodes in their respective subtrees, and return.

If the two nodes are equal, we attempt to recursively find the best mapping between their child nodes. Let $\mathcal{M} = children(v)$ and $\mathcal{N} = children(w)$. Let $m = |\mathcal{M}|$ and $n = |\mathcal{N}|$. We create an $m+1$ by $n+1$ matrix c. $c[k][l]$ contains the least-cost mapping M_{kl} between $\mathcal{M}[k]$ and $\mathcal{N}[l]$, for $0 \leq k \leq m+1$ and $0 \leq l \leq n+1$. We can calculate the entries of c by noting that

$$c[k][l] = \min_{\gamma} \begin{cases} c[k][l-1] & + \quad delete(\mathcal{M}[k]) \\ c[k-1][l] & + \quad insert(\mathcal{N}[l]) \\ c[k-1][l-1] & + \quad best\text{-}mapping(\mathcal{M}[k], \mathcal{N}[l]). \end{cases}$$

$c[0][0]$ is defined to be an empty mapping. $c[m][0]$ and $c[0][n]$ represent mappings where all nodes of $\mathcal{M}$ are deleted and all nodes of $\mathcal{N}$ are inserted, respectively. *best-mapping*$(\mathcal{M}[k], \mathcal{N}[l])$ is calculated recursively.

Once the full matrix c has been calculated, we may simply read off and return $c[m][n]$ as the best mapping between $\mathcal{M}$ and $\mathcal{N}$.

The running time of this algorithm is $O(|v||w|)$ in the worst case, where both v and w have a depth of 2, all nodes are children of the roots, and all nodes have different labels. In practice, two items reduce this running time drastically. First, the algorithm is actually dependent on the branching factor at each node of the two trees, as the size of c is dependent on the number of child nodes being compared. Second, we only construct c for nodes which are equal (that is, which have the same label). If two nodes have different labels, we immediately return a mapping which deletes one and inserts the other. This greatly reduces the number of recursive calls required to populate c.

4.3.2 Skeleton Patterns

We now describe a method for generating tree-structured patterns given the best mapping M between two example trees T_1 and T_2. We begin by using T_1 as our template pattern P. We then remove any nodes from P which are *deleted* or *changed* in M, replacing them with wildcard nodes. If an entire subtree is deleted or changed, we replace it with a single wildcard. We term the remaining tree a "skeleton pattern," because it contains only the structural elements which the examples have in common, with none of the specific properties of either. Another perspective is that P is the largest subtree that "matches" both T_1 and T_2.

An example of this process is shown in Figure 4-2. Two subtrees, taken from a site containing upcoming talks and seminars,[5] are mapped. Looking at the generated pattern, we note that the structural nodes of the trees remain intact, while the specific text describing each talk is unmapped. When we generate the skeleton pattern, the text is removed, and only the structural elements remain.

Skeleton patterns prove to be very useful in finding semantic content similar to the original examples. To match, we attempt to align the pattern P with some subtree of the document. An alignment is essentially a mapping, M, as described above, which contains no *delete* or *change* operations. To find these alignments, we could, in theory, use the same edit distance algorithm we used above, or a tree matching algorithm such as that described by Cole, et. al. [8].

Instead, because of the size of the patterns being considered, we utilize a simple greedy algorithm for finding alignments. In essence, at each level of the pattern, we attempt to align each child node in the pattern with the leftmost possible child node in the document. We continue this process recursively until every node in the pattern is aligned with one node in the document, constituting a match, or no alignment is found.

In more detail, we begin by trying to align the root, $P[0]$, of the pattern with each node in the document to be matched. If we find a node v such that $P[0]$ and v match, we recurse, attempting to align the children of $P[0]$ with the children of v. We take the list of children of $P[0]$ and of v, and attempt to find an alignment. An alignment is a mapping from the children of $P[0]$ to the children of v where every child of $P[0]$ maps to at least one child of v, and if $P[q]$ maps to $T[r]$ and $P[s]$ maps to $T[t]$, and

[5] `http://www.csail.mit.edu/events/eventcalendar/calendar.php`

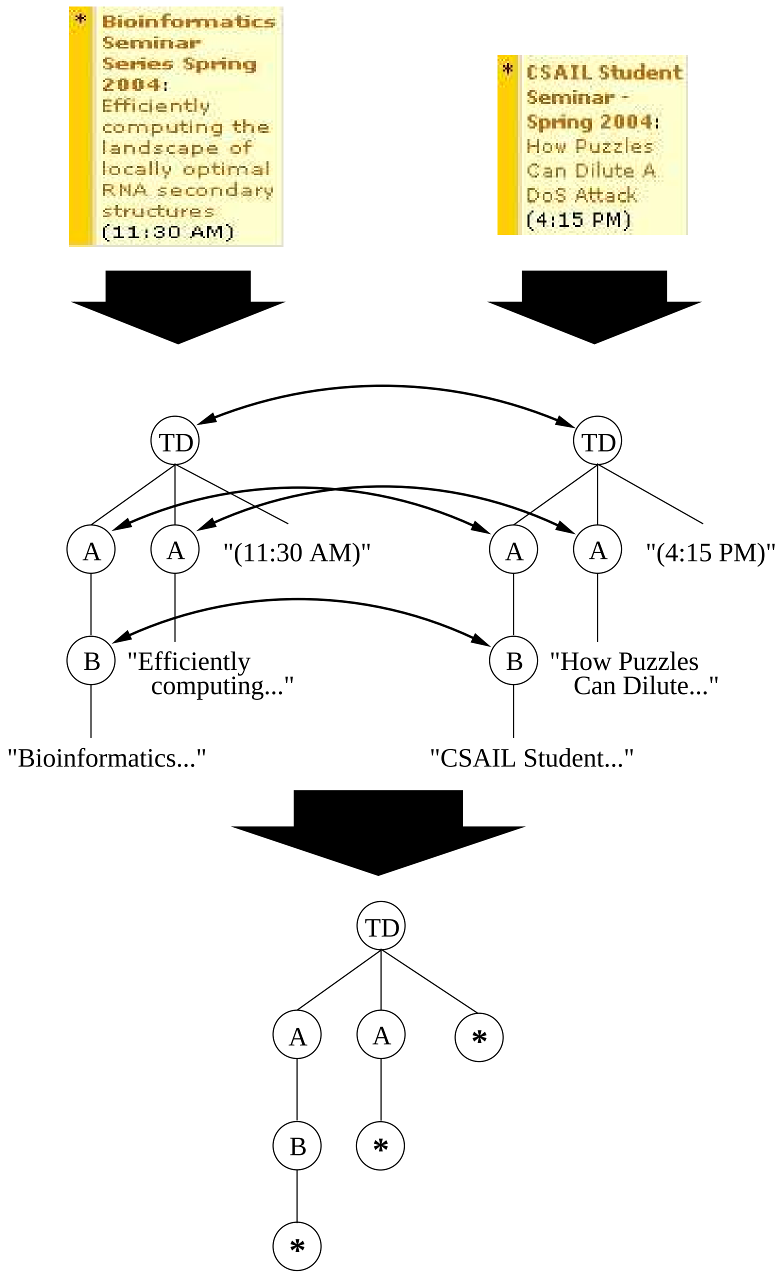


Figure 4-2: Generating a skeleton pattern with wildcards from the best mapping between two examples.

$P[q]$ is a left-sibling of $P[s]$, $T[r]$ is also a left-sibling of $T[t]$. That is, sibling order is preserved in the alignment.

Figure 4-3(a) shows a valid alignment between the a list of pattern children and a list of document children. Figure 4-3(b) shows a pattern-document pairing with no valid alignment. Note that the `‘‘Text’’` and `B` nodes do *not* align in Figure 4-3(b)

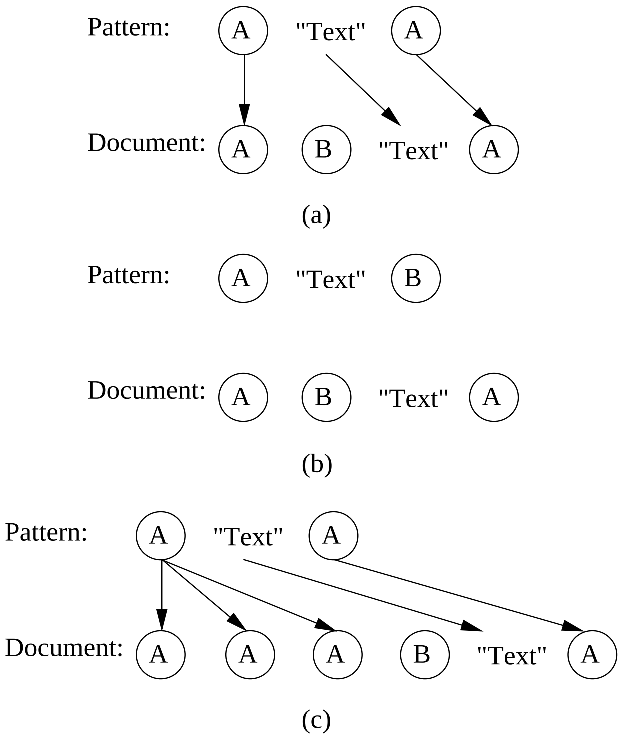


Figure 4-3: Examples of aligning two lists of child nodes. (a) shows a valid alignment. (b) shows a pair with no valid alignment. (c) shows a valid alignment using the repetitive matching scheme.

because sibling order must be preserved.

Wildcard nodes are treated specially for the alignment process. Because they may represent information that existed in one example, but not in another, we allow them to align with either zero or one document nodes during matching. This allows our patterns to be flexible enough to match the full set of given examples.

We continue to recurse down the trees P and v as long as a alignment is found for the children at each depth of the trees. For each pair of aligned nodes $P[j]$ and $T[k]$, we attempt to align $children(P[j])$ against $children(T[k])$. We recurse until $P[j]$ has no children. If we successfully find a mapping from every node in P to a node in v under these constraints, we consider the pattern matched.

The speed of this greedy algorithm for finding alignments is dependent on two factors: the size of the document and the size of the pattern. For each node in the document, we potentially attempt to match every node of the pattern against one of its descendants, resulting in a running time of $O(|T||P|)$, where T is the document being matched.

We can improve the performance of this simple matching algorithm in practice by maintaining the *height* of the pattern, or the length of the longest path from the root of the pattern to a leaf. If we also maintain the height of each node in the document tree, we can skip matching on any tree node v whose height is less than the pattern's height, as it would be impossible to find an alignment with these subtrees.

4.4 Other Improvements

The patterns formed in the previous section are surprisingly effective at matching certain types of structure on the Web. For instance, an effective pattern for the talk announcements shown in Figure 4-2 can be generated from just two examples.

There are many types of structure, however, which do not respond well to this type of straight best mapping approach. For instance, these skeleton patterns do not handle pages with variable-length lists of semantic items well because of their rigid matching structure. On other pages, it requires several examples from the user to generate useful patterns.

To deal more effectively with these issues, we introduce several other heuristics in this section which allow for more general patterns, or for generating useful patterns with fewer examples from the user.

4.4.1 Repetitive Matching

As defined above, skeleton patterns do well at matching single instances of a given structure, but often fail when the structure is a variable-length list of elements. This results from the inflexible nature of the best mapping procedure. For example, in Figure 4-4(a), we see a skeleton pattern which was created from two example lists of links, each with three items. While this pattern is correct, in that it generalizes both examples, it will only match other lists with exactly three elements. In fact, even if we provided an example with four elements, the extra list item would be removed by the best mapping procedure, because no nodes in the length-three list would map to

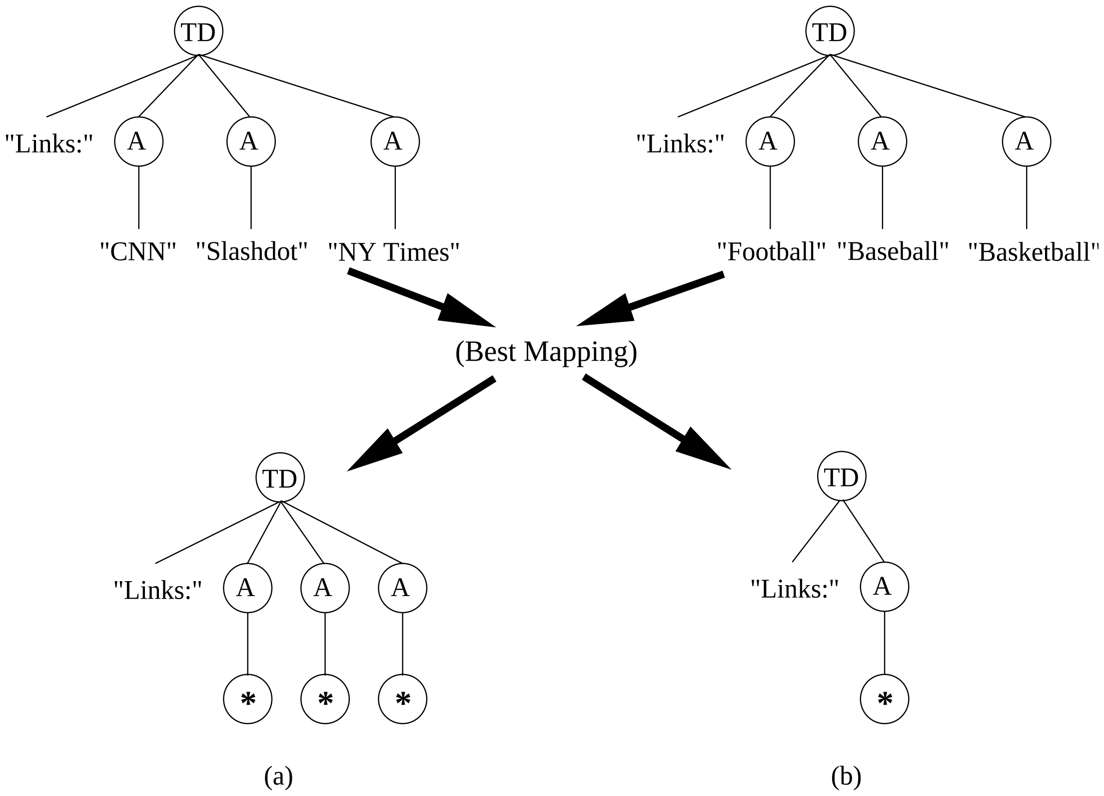


Figure 4-4: Generating a pattern with repetitive matching. The pattern on the left will only match lists with exactly three `A` tags, while the repetitive pattern on the right will match any number of `A` tags.

it.

To resolve this issue, we will adopt a scheme which allows individual nodes to match more than one element. We augment our matching algorithm from the previous section to allow a single pattern node to align with multiple document nodes at a time, as long as sibling order is still preserved. Figure 4-3(c) shows a valid alignment under the repetitive matching scheme.

A pattern built in this method is shown in Figure 4-4(b). The `A` nodes in this pattern have been combined to form a single pattern node. When we match against a new document, this single node will be allowed to match against more than one `A` node which is the child of a `TD` node, with a preceding sibling labeled `''Links:''`. In this way, we can form more general patterns which can match a variable number of listed items.

With this modification, we introduce the possibility that our patterns will over-generalize and match items which the user did not intend. While in theory this is true, we have found, in practice, that this scheme is extremely effective and rarely overgeneralizes. We discuss this in more detail in our experimental results in Chapter 7.

4.4.2 List Collapse

As mentioned above, this repetitive matching heuristic turns out to be an effective representation for many types of semantic data. We now formalize this idea of combining neighboring nodes using a *list collapse* heuristic to *automatically* collapse nodes throughout the tree.

To do this, we first introduce the notion of the *normalized cost*, $\hat{\gamma}$, of a mapping M between two trees T_1 and T_2:

$$\hat{\gamma}(T_1 \to T_2) = \frac{\gamma(T_1 \to T_2)}{|T_1| + |T_2|}.$$

Because the most expensive cost for any mapping is to delete $T_1[0]$ and insert $T_2[0]$ (that is, swap the roots), for a cost of $|T_1| + |T_2|$, and there is no negative cost, we know:

$$0 \leq \hat{\gamma} \leq 1$$

We can now utilize $\hat{\gamma}$ to *automatically* collapse nodes. We first choose a threshold cost, $\hat{\gamma}_T$. We then *collapse* any pair of neighboring nodes in the pattern which (1) have the same tag name and (2) where the mapping between them has a normalized cost less than $\hat{\gamma}_T$. Note that the our requirement that two siblings have the same tag is equivalent to ensuring that they have the same τ, because they share the same parent node. As mentioned in Section 4.2, this is a good heuristic for finding nodes with similar semantic content.

To collapse two subtrees rooted at nodes v and w with a mapping M_{vw}, we first remove any nodes from the subtree v which are *deleted* in M_{vw}, replacing them with wildcard nodes. We then delete the *entire* subtree rooted at w, including w itself. By doing this, along with our repetitive matching algorithm of the previous section, we are allowing the newly collapsed subtree at v to perform the matching function for both v and w.

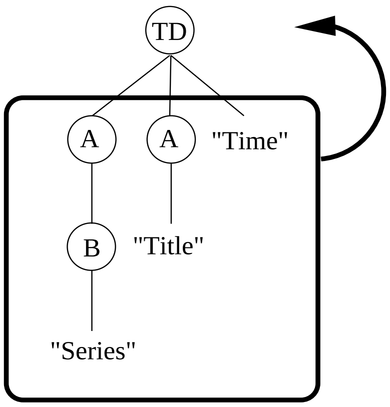
This heuristic is beneficial for several reasons. First, as mentioned above, it allows us to match variable-length lists of semantic content. In addition, it results in more compact patterns. Because we have only collapsed neighboring subtrees with a low normalized cost, we have effectively represented the same information with fewer nodes. The result is smaller, more flexible patterns.

4.4.3 Sibling Matching

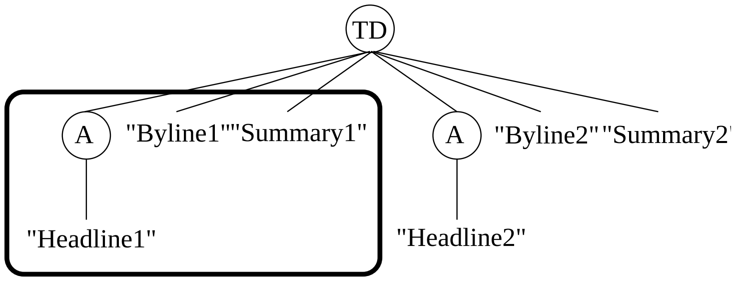
The algorithms and heuristics in the preceding sections deal entirely with examples that span a single subtree in the document. A contiguous selection in a web browser, however, may span several neighboring subtrees.

If the selection spans all children of a single parent node, we may represent it by moving the selection up to that parent without loss of generality. On the other hand, if the selection only spans a *subset* of the children of the parent node, we cannot move the root of the selection without effectively changing the nodes selected. Figure 4-5 shows an example of these two types of selections. In particular, Figure 4-5(b) shows a simplified tree from a news site, where the components of the main headlines are all lined up as siblings of the same node. If the user wishes to create a wrapper for a single story, they select only a subset of those siblings.

Unfortunately, our edit distance heuristic does not apply well to the types of unrooted or "forest" selections shown in Figure 4-5(b). Edit distance can only be computed for pairs of *rooted* trees. One approach is to add the parent of the selection as the root, ignoring the other sibling nodes which are not part of the selection. This makes an effective pattern which matches each instance of the semantic content. Unfortunately, because of our greedy algorithm for finding alignments, it also aligns in several other ways with the document, each of which is incorrect. Notably, the number of incorrect alignments grows exponentially with the number of valid matches, swamping our patterns with bad matches. Figure 4-6 shows an example of these



(a)



(b)

Figure 4-5: The two cases of multiple, sibling subtrees comprising a selection. In (a), the user has selected all children of an element, so the selection is moved up to that element without changing the meaning. In (b), the user has only selected a subset of children, so we cannot move the selected node.

d

incorrect alignments. The first document tree shows the desired alignments, while the bottom two show incorrect alignments resulting from the repetitive matching scheme.

Currently, we have not been able to extend our algorithm to deal with these cases, although we discuss several potential approaches in Section 8.2.1. Instead, to deal with partial selections, we have tied the LAPIS [26] pattern API into our system. Given a document and one or more selections, LAPIS generates a set of hypotheses, each representing a possible pattern for the selection. These hypotheses, written in a language called *text-constraints*, are linear in nature, parsing the document into tokens and learning patterns based on these tokens.

We integrate LAPIS into our wrapper induction system at the lowest level, and

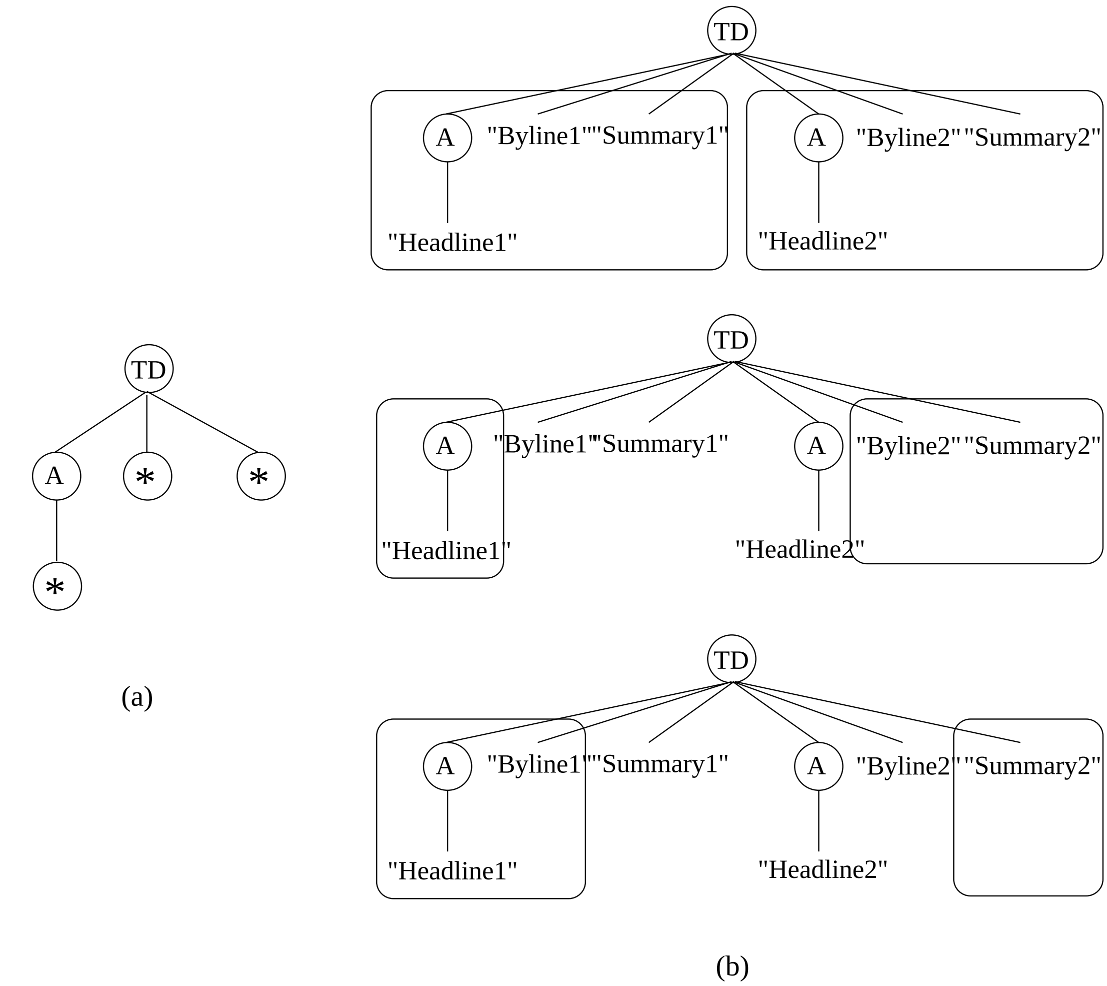


Figure 4-6: The incorrect alignment of a pattern generated from a partial selection.

only where the user's selection covers a subset of sibling elements. In this case, we pass LAPIS *only* the HTML source of the parent element of the selection, as if it were the entire document. LAPIS then generates a *text-constraints* pattern based on this restricted portion of the document and the user's selection.

Later, when we match the pattern against a new document and reach the part of the pattern that is represented by *text-constraints*, we simply pass this part of the new document back into LAPIS and have it retrieve the matches for us.

By restricting the execution of LAPIS to only the portion of the document where there is a partial selection, we can retain the power of our skeleton patterns and list collapse, combined with the flexibility of LAPIS's *text-constraints* system in dealing with partial selections.

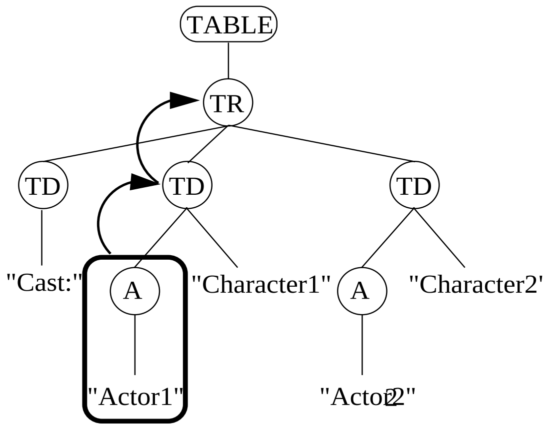


Figure 4-7: Capturing the context of a selection. In this case, the user has selected a single "Actor" node. We move the root of the pattern up the tree to capture context such as "Cast:" and other actor nodes.

4.4.4 Context

Beyond inducing a pattern for the text the user has actually selected, it is often important to capture the *context* of the example. This is especially important to avoid overgeneralization in cases where the wrapped text is only a few elements in size (for instance, a single hyperlink or image). If we did not capture the example's context, we would end up matching every hyperlink or image on the page indiscriminantly.

To capture this context, we simply move the root of the example up the tree. For each ancestor v included, we gain the context of the other children of v. An example of this process is shown in Figure 4-7.

We can place a limit on this context by restricting the size of the tree we are willing to accept to a maximum size s_{max}. Each time we move up to another ancestor, we add to the size of the example $|E|$. We continue as long as $|E| \leq s_{max}$.

Capturing context is important for several reasons. First, as mentioned, it enables us to match very simple structures such as single images without overgeneralizing. In addition, once we have gathered context, we can apply the list collapse heuristics described in Section 4.4.2 to the entire context. For example, in the case of Figure 4-7, we would not have been able to collapse any nodes in the user's original selection. However, once we have moved the root of the pattern up to get the context, we see that we can collapse the neighboring `TD` nodes. Later, when we match the pattern, we will match all actors in the list, rather than just the first.

4.4.5 URL Prefixes

Once wrappers have been formed, it is important to match them against the correct subset of pages. On many sites, semantic information of the same type is spread across multiple pages, all formatted in the same way, and thus all amenable to the same wrapper developed with our system. For instance, two different searches on Google have similar syntactic formatting, but have different URLs:

- `http://www.google.com/search?q=haystack`
- `http://www.google.com/search?q=wrapper%20induction`

Similarly, pages for two different movies on the Internet Movie Database also have different URLs:

- `http://www.imdb.com/title/tt0033467/`
- `http://www.imdb.com/title/tt0034583/`

If we attempted to tie wrappers solely to the exact URL on which they were defined, we would lose much of the benefit gained from the patterns we have generated. The user would be forced to create separate (but effectively equivalent) patterns for *each* page on a site with a different URL.

Instead, we have developed heuristics to make the reuse of wrappers possible. First, we note that wrappers are often applicable across all pages on a single host (such as `www.google.com` or `www.imdb.com`). Our first heuristic, then, is to associate wrappers with the hosts on which they were created. We then simply run all wrappers for the host on every page that the user visits on that host.

However, this process is inefficient and often incorrect. For instance, a wrapper created for Google's web search is not necessarily applicable for their newsgroup search. A wrapper developed for IMDB's movie pages is not necessarily applicable for their actor pages. Evaluating all of a host's wrappers on every page wastes time and may provide incorrect or mislabeled results to the user.

To adjust for this, we take note of the fact that most pages on a site with similar semantic content have a similar URL prefix. For instance, all Google result page URLs begin with `http://www.google.com/search?`. All IMDB movie pages have the URL prefix `http://www.imdb.com/title/`.

Thus, to determine which wrappers to execute on a given page, we simply compare the prefix of the current URL against the prefixes of the URLs of existing wrappers. When we find wrappers with the same prefix, we execute them on the current page.

We currently generate these prefixes manually, by removing either the query (the text after the "`?`" character) or the final directory (the text after the second-to-last "`/`" character). We then query the wrapper database for existing wrappers with the same URL prefix, and run only those wrappers on the current page.

In the future, we plan to maintain a suffix tree data structure [29] built from the URLs of all wrappers currently in the system. A suffix tree is useful for determining similar prefixes of a set of strings, and may be built in time linear with the size of the data set, and queried in time linear with the size of the query string. We plan to augment the suffix tree to maintain the wrapper (or wrappers) associated with a given node. We will also need to restrict our queries so they do not return results which are too general (for instance, every URL begins with the prefix "`http://`". This system will allow us to quickly and easily retrieve existing wrappers which have a common URL prefix with the current page.

4.5 Variables

The heuristics described above to augment our basic wrapper induction algorithm introduce several variables. The effectiveness of our system depends on our choice of these variables, and we describe various factors which led to our choices of two of these variables below.

4.5.1 Maximum Example Size

To improve the effectiveness of our wrappers, as well as allow them to often be formed from fewer examples, we described a method for gathering the context of the user's selection in Section 4.4.4. There are several trade-offs in the process of obtaining this context which center around the *size* of the example being considered.

The size of an example, $|E|$, is defined as the number of nodes in the example subtree. It is important to consider $|E|$ in forming wrappers because the running time of the tree edit distance algorithm, as described in Section 4.3.1, is directly dependent on the size of the two trees being considered. If we attempt to generalize examples which are too large, the user will notice an unacceptable lag in the system as it attempts to calculate the edit distance[6]. The system resources (memory and processor) required for this calculation are also dependent on the size of the trees being considered.

If we form a wrapper without gathering context for the examples that the user has selected, the size of the pattern being generated is uniquely determined by the nodes in the selection. In this case, without context, we can only maintain a hard limit on the maximum size of the example, and reject any examples that exceed this size.

On the other hand, if we decide to gather context, our system has more control over the size to which the wrapper grows. This process can be extremely important, as it often prevents overgeneralization, especially for small patterns. If we move far enough up the document tree, the context we find will allow us to more effectively "zero in" on appropriate content during matching. However, we must also place a limit on this process, or we will incur the time and system resource penalties mentioned above.

We have settled on a two-part system for limiting the size of examples and wrappers. First, we have placed a hard limit, s_{max}, on the absolute maximum size of examples allowed. This limit helps prevent us from accepting examples which en-

[6]There may be cases where this lag is acceptable - for instance, if we are attempting to create a pattern encompassing an entire page. We will not provide for this here, but do discuss it further in Section 8.2.1.

compass the entire page, which would require a large amount of time to process. If the user attempts to specify an example with size greater than s_{max}, we simply alert them that it is too large and ask that they make another selection. We have found that the value $s_{max} = 250$ is large enough to form good wrappers from few examples, but small enough to prevent unnecessary delays caused by performing the tree edit distance calculation on excessively large trees.

In addition to s_{max}, we have also provided the user the ability to set their own maximum size value, s_{user}. s_{user} is utilized in a similar way to s_{max}, to limit the size of the example to a reasonable size. It is useful on certain sites where the context of the wrapper is of a specific size, but growing the context too large would result in an overgeneralized pattern. This feature is mostly useful for expert users, and as such is kept hidden unless a user specifically requests to specify it. The user may create a pattern with a certain s_{user} by selecting the menu option "Create Wrapper (Specify Size)." Once selected, wrapper induction proceeds in the normal fashion, with the exception that the user is prompted for a value for s_{user} before the wrapper is formed. If the user selects the standard "Create Wrapper" option, we take the default value $s_{user} = 40$, which was also determined to be effective during our survey of relevant web sites.

4.5.2 List Collapse Cost Threshold

Another important factor in our ability to effectively create wrappers is the threshold at which we collapse neighboring nodes throughout the pattern tree. As described in Section 4.4.2, we utilize a *list collapse* heuristic, by which we combine similar subtrees in the pattern into a single subtree. This procedure provides us with several benefits. First, it gives us more compact patterns. In addition, these patterns are more flexible, allowing us to match variable-length lists of semantic content, rather than having to form a separate wrapper for each possible size of list.

To decide which nodes to collapse, we first calculate the normalized edit distance cost, $\hat{\gamma}$, between the two subtrees. We then employ a threshold cost, $\hat{\gamma}_T$, and collapse the two nodes if their normalized cost is less than this threshold.

The choice of $\hat{\gamma}_T$ also involves a set of trade-offs. If we set the value too low, we run the risk of not collapsing subtrees which actually contain similar content. If we set the value too high, we will collapse subtrees which are not very similar, and risk overgeneralizing our patterns.

Noting that the normalized cost is restricted to $0 \leq \hat{\gamma} \leq 1$, we have found that the value $\hat{\gamma}_T = 0.25$ is a good compromise between under- and overgeneralization of patterns. In effect, this choice requires the two trees under consideration to share at least 75% of their nodes to be collapsed. In practice, we have found that this is a good cutoff point for predicting when syntactic similarity implies semantic similarity.

4.6 Algorithm Summary

We now pause to summarize the algorithms and heuristics presented above and piece them together into a full pattern induction and matching routine.

4.6.1 Induction

To create a pattern, we first gather examples from the user in the form of subtrees from the document. For each of these examples T_i, we gather context by moving the root of the example up the tree as long as $|T_i| \leq s_{max}$, where s_{max} is the maximum example size. If specified by the user, we utilize s_{user} for gathering context instead of s_{max}.

Beginning with the first example T_1 as our pattern template P, we map P to each other example T_i using the tree edit distance algorithm described in Section 4.3.1. Each edit distance operation produces a mapping M_i containing inserts and deletes. For each deleted node in M_i, we delete the corresponding node in P, replacing it with a wildcard node. This series of mappings and deletions creates our skeleton pattern.

Once all examples have been merged into P, we next collapse neighboring nodes within the tree using the heuristics of Section 4.4.2. We walk through the nodes of the tree in preorder, skipping the root. At each node $P[i]$, we consider its right sibling, $P[i+1]$, if it exists. If $P[i]$ and $P[i+1]$ have the same tag name, we map them using the tree edit distance algorithm. If the normalized cost $\hat{\gamma}$ of this mapping is less than a predetermined threshold $\hat{\gamma}_T$, we collapse the nodes as follows. We first delete any nodes from the subtree $P[i]$ which are deleted in M, replacing them with wildcards. We then delete the entire subtree rooted at $P[i+1]$ (without replacing it with a wildcard). This allows the $P[i]$ subtree to match multiple times in our repetitive matching scheme.

Finally, to deal with partial selections, we create LAPIS *text-constraints* pattern for the affected subtree. We pass LAPIS the HTML source of the parent of the selection, along with indices indicating the extent of the user's selection. The *text-constraints* pattern generated by LAPIS is then stored at the parent node for later use in pattern matching.

4.6.2 Matching

Given a pattern P constructed as described in the previous section, we now provide a simple algorithm for finding matches in a new document tree T.

We begin by taking the nodes of T in preorder. For each node v, we check whether it is equal to the root of P, $P[0]$ (see Section 4.1 for our definition of node equality).

If $P[0]$ is equal to a node v, we recurse. We take the list of children of $P[0]$ and of v, and attempt to find an alignment as described in Section 4.3.2. To allow for the repetitive matching strategy of Section 4.4.1, we also allow individual nodes of P to align with more than one node of T, as long as the sibling order constraint is met.

We continue to recurse down the trees P and v as long as a alignment is found for the children at each level. If we reach a node where a LAPIS *text-constraints* pattern is stored, we pass the node's HTML and the pattern into LAPIS, which returns a set of matches for that node.

If we successfully find a mapping from every node in P to a node in the subtree rooted at v under these constraints, we consider the pattern matched. We record the set of alignments that make up the match for later use. We then continue through the preorder listing of T to find other matches.

When we have attempted to match against each node in T, we return the array of matches, or an empty array if no alignments were found. These matches will be used along with the semantic labels described in Chapter 5 to map semantic meaning onto pages for the user.

Chapter 5

Semantic Wrappers

In the previous chapter we outlined a powerful wrapper induction system which allows us to create compact, reusable patterns for information stored on the Web. To integrate these wrappers into the framework of the Semantic Web, we now develop a means for asserting semantically meaningful statements about the features they represent.

5.1 Ontologies

In the context of the Semantic Web, an *ontology* is a description or framework used to describe the relationships between semantically meaningful concepts. An ontology is comprised of descriptors for a single category of semantic information, such as `News`, `Search`, or `Cinema`. Ontologies on the Semantic Web are described and applied using the RDF standard [3].

For our purposes, we will be concerned with two main ontological types: *classes* and *properties*. Classes are used to categorize or classify instantiable "objects." Examples of classes on the Web include a `Book`, `NewsStory`, `SearchResult`, `Movie`, and `Actor`.

Properties are used to describe and differentiate the values of class members. Properties can, themselves, be instances of a class, or may be "primitive" types such as integers or strings. For example, a `Book` has an `author`, `title`, `publisher`, and `numberOfPages`. Some classes may have more than one instance of a single property. For example, a `JournalArticle` may have more than one `author`.

We will use the convention of representing semantic classes and properties with the syntax `Ontology:Class` and `Class:property`. Classes will be capitalized and properties lowercase. For example, the `News:NewsStory` class has the property `NewsStory:author`. When the meaning is unambiguous, we will abbreviate by leaving out the name of the ontology, for example, the `NewsStory` class and the `author` property.

Where classes and properties abstractly describe objects, an *instance* of a semantic class is a specific occurrence of that class. An instance of a class has its property slots "filled in," differentiating itself from other instances. For example, one instance of the `Book` class has the `title` "Cat's Cradle" and the `author` "Kurt Vonnegut," as well as many other properties, filled in.

In RDF, these instance bindings are stored as *statements*, described by {*subject, predicate, object*} tuples. The subject of a statement is a reference to the resource that the statement is about. The predicate is a descriptor which declares a certain property of the subject. The object is another reference which is the value of the property. For example, the following is Adenine [25] code for the properties of the book from the last paragraph:

```
add { :myBook
        Book:title        ``Cat's Cradle'' ;
        Book:author       ``Kurt Vonnegut'' ;
        Book:price        ``$12.95'' ;
        Book:image        <http://images.barnesandnoble.com/images
                            /1220000/1220677.gif> ;
        Book:publisher    ``Dell Publishing Company, Incorporated'' ;
        Book:ISBN         ``038533348X''
}
```

Interestingly, RDF statements may be chained by making the object of one statement the same resource as the subject of another. For instance, the `author` "Kurt

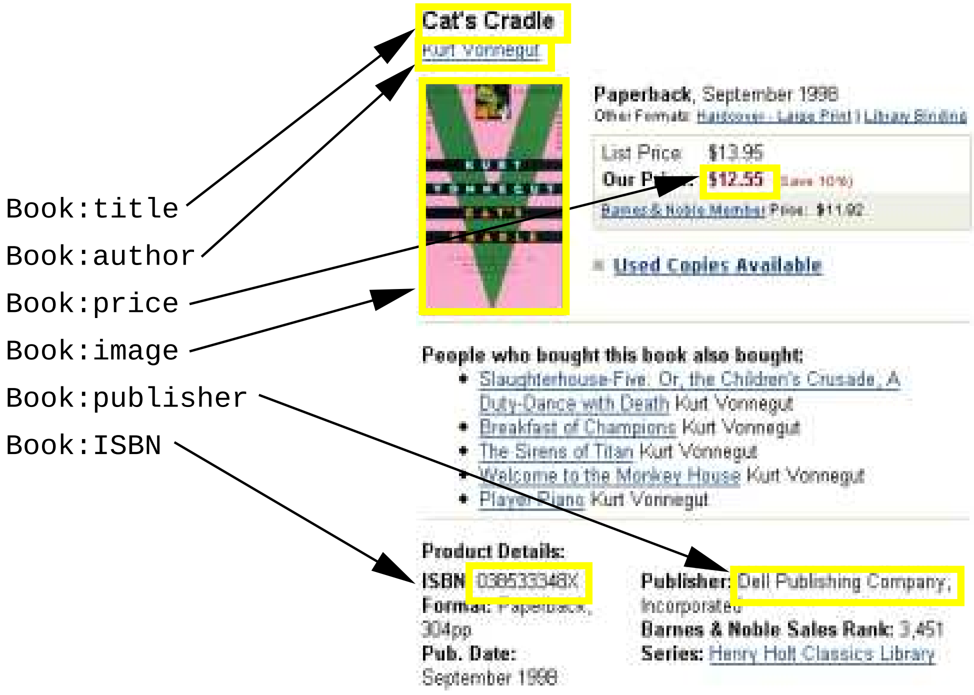


Figure 5-1: An example of mapping the properties of the class `Book` to a page from `http://bn.com/`

Vonnegut" is actually a full-fledged instance of the class `Person`. This class be the subject of its own statements, asserting its `name`, `address`, and `email`, for instance. While our semantic labeling scheme will not handle this type of "recursive" class structure, it is interesting to think of extending our wrappers to handle these cases, as discussed in Section 8.2.1.

Our goal will be to map the properties of ontological classes to the features of World Wide Web pages describing them. An example of a page describing an instance of a `Book` is shown in Figure 5-1. Here, we see abstract properties such as `author` and `title` mapped to the concrete features shown on the Web.

5.2 Labeling Wrappers

The patterns described in Chapter 4 provide an excellent opportunity to formalize the types of mapping shown in Figure 5-1. By comparing and combining several similarly-structured instances of the same semantic content, our wrappers capture

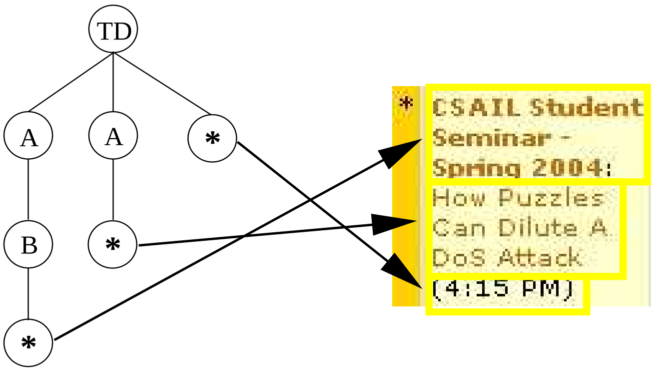


Figure 5-2: Mapping a pattern with wildcards to content.

their common syntactic structure. At the same time, the mapping process leaves empty "slots" precisely where the variable semantic properties are in the structure.

For the purposes of describing semantic content, we can think of each example as an instance of a semantic class which the user is attempting to generalize. When we created the patterns, we began by taking a single, specific instance, example T_1. We then mapped this instance to other instances, *removing* nodes that the instances did not have in common. In semantic terms, what we did by removing these specific instance nodes was turn the instances into a generic description of the structure of the semantic class they represent.

When it finds nodes which differ between examples, our pattern induction algorithm changes these nodes into *wildcards*. These wildcards provide a natural binding location for our semantic properties, as they directly map the wrapper's structure to the variable features contained in the web page's structure. In this way, we can think of the wildcards as mapping *back* onto the page when the pattern is matched. For example, Figure 5-2 shows this mapping for the pattern we originally derived in Figure 4-2.

5.2.1 Classes

Based on these observations, we describe our method for binding RDF statements to our wrappers. First, we note that by selecting an "object" on a page to wrap, the user has, in effect, stated that the selected region represents an instance of some

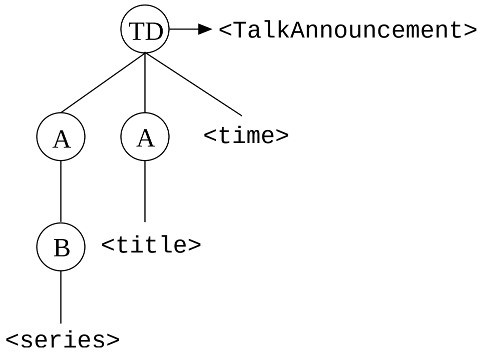


Figure 5-3: A semantically labeled pattern.

semantic class. For example, in creating a wrapper for the CSAIL events calendar page, the user selects the text of a single `Talk` instance from the page.

We can thus form a link between the syntactic page structure and the semantic ontology structure by binding the class resource to the wrapper itself. For instance, we bind the `Talk` class to the entire wrapper shown in Figure 5-2. When the wrapper is later matched against a page, each match represents an instance of this class.

5.2.2 Properties

Once the wrapper has been classified, we also need to label its wildcard nodes with the properties they represent (if any). These bindings are effectively asserting semantic statements of the form {`Class property ?x`}. Here, `Class` is the semantic class of the wrapper, described above, and `?x` is an unbound variable containing the value of `property`. `?x` is represented in the wrapper by a particular wildcard node. This statement is effectively saying that the wrapper's class has some unknown property located at the wildcard node.

For example, one of the wildcards shown in Figure 5-2 is bound to the `title` property of the `Talk` class represented by the wrapper. Abstractly, we have the statement {`Talk title ?x`}. When the match is found in Figure 5-2, the wildcard is bound to the actual text containing the title. We now have the concrete statement {`Talk title ‘‘How Puzzles Can Dilute A DoS Attack’’`}. For each labeled property, another statement is created when the pattern is matched.

Thus, by binding abstract statements of the form {`Class property ?x`} to the wildcard nodes of our wrappers, we can build up a full semantic description of the classes they represent. These abstract statements can be stored by binding the semantic class to the root of the pattern, and binding the property descriptors to the wildcard nodes. For instance, an example of the fully labeled `TalkAnnouncement` semantic pattern is shown in Figure 5-3.

When the pattern is matched, we dynamically instantiate a new instance of the type represented by the wrapper's class. We then add RDF statements about the instance's properties based on the text of the document nodes matched by the wrapper's wildcards.[1]

5.2.3 Matching Considerations

There are a few minor modifications to this description necessary to accommodate the pattern enhancements described in Section 4.4. First, adding context to an example, as described in Section 4.4.4, changes the location of its root node. If we moved the semantic class label along with the root, we would be changing the syntactic-to-semantic mapping that the user had defined, thus changing the meaning of the wrapper. Instead, we choose to bind the class to a specific node in the example, and thus to a specific node in the wrapper. Initially, this is the root of the user's selection. When we add context, we keep the class bound to that node, even when the root changes.

This method of binding the class to a specific node is important when we consider the repetitive matching scheme of Section 4.4.1. For example, if we gather context for the pattern and end up collapsing the node containing the semantic class, we may end up creating a whole list of semantic objects during matching, rather than just one. Keeping the class bound to the true semantic root of the pattern allows us to instantiate one object for each document node to which the class is bound.

[1]As defined in Section 4.3.2, a wildcard may match zero or one nodes in an alignment. If a wildcard does not match any document nodes, we simply do not add the corresponding RDF statement.

An example of this type of class binding occurs for the pattern generated for the list items in Figure 4-4. This pattern might contain objects with the class `Link`. Each time the branch of the pattern containing the `A` node and the wildcard is evaluated, we create a new instance of type `Link`, separate from the other matches.

A similar repetitive matching strategy holds for property nodes. If the semantic root matches multiple times, the wildcard nodes within that subtree will also match multiple times. We keep these matches separate, allowing us to bind the correct properties to each instance of the class. For example, the `Link` objects described above would each be bound to their own `title` property based on the node to which the wildcard was bound.

Chapter 6

User Interface

The algorithms for wrapper induction, matching, and semantic labeling outlined in the previous chapters provide a powerful tool for describing semantic information on the World Wide Web. To be useful, however, this tool must be outfitted with an easy-to-use, intuitive user interface.

To provide this interface, our algorithms have been implemented within the Haystack information management client [24]. Haystack provides users with a rich set of tools for creating, modifying, categorizing, and sharing semantic information. It is tightly integrated with the RDF standard [3], a powerful language for describing and structuring the semantic links between objects and information. This integration allows us to provide simple interfaces for labeling patterns as described in Chapter 5.

Our interface has five main parts, described below. The first allows for the creation of wrappers from an initial example on a web page, while the second allows the user to add additional examples. Once wrappers have been created, a similar interface allows users to label them with semantic properties. We also provide visual feedback and context-sensitive menus through Haystack on pages that have existing wrappers. Finally, we describe our interface which allows more advanced users to directly manipulate the structure of a wrapper.

Figure 6-1: Creating a wrapper on the CSAIL Faculty page.

6.1 Wrapper Creation

Inside Haystack, the wrapper induction functionality is always available to the web browser through the context menu - the user does not need to do anything to enable it. A user begins the wrapper induction process by simply navigating to a page containing semantic information.

Once there, the user initiates wrapper induction by highlighting the relevant content using the browser's standard mouse selection interface. They then right-click, and choose "Create a Wrapper" from the context menu that appears.[1] This menu selection is shown in Figure 6-1 on the CSAIL faculty directory page, where the user is attempting to create a wrapper which will match each `Person` in the directory.

Using Haystack's *UI continuation* framework, the user is then prompted to provide several arguments necessary for wrapper creation, as shown in Figure 6-2. This framework treats missing function arguments (such as our semantic class) as tasks which the user may complete whenever they wish. These continuations appear as modeless dialogs on the right pane of the Haystack interface.

Using the continuation, the user is asked to specify a semantic class for the wrapper

[1] If the user has not selected any text in the browser, they are prompted to do so at this point.

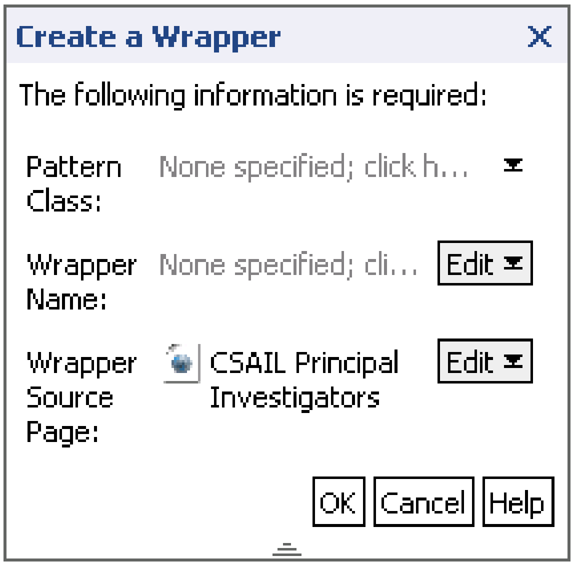


Figure 6-2: The UI continuation for creating a wrapper.

which describes its type. They may specify this class using Haystack's built-in search functionality, or type the name of the class manually. The user may also provide a name for the wrapper to help them to identify it later.

Once the user clicks "OK" from the UI continuation, the wrapper induction algorithm of Chapter 4 is run in the background, and an initial pattern is generated from this single example. This pattern is then matched against the current page, and visual feedback is provided for the user by highlighting the matched items in yellow, as demonstrated in Figure 6-3. We note here that the user has created a fairly effective pattern for the faculty members from only a single example.

At this point, the user is prompted to confirm the pattern, or may choose "Cancel" to remove it from the system, as shown in Figure 6-4. This final dialog allows the user to "back out" of pattern creation if the wrapper has been overgeneralized or has failed in some other way.

6.2 Additional Examples

While semantic objects may often be wrapped using a single example, there are many cases where more than one example is necessary. For instance, in our faculty directory

Figure 6-3: Feedback during wrapper creation by highlighting matched elements.

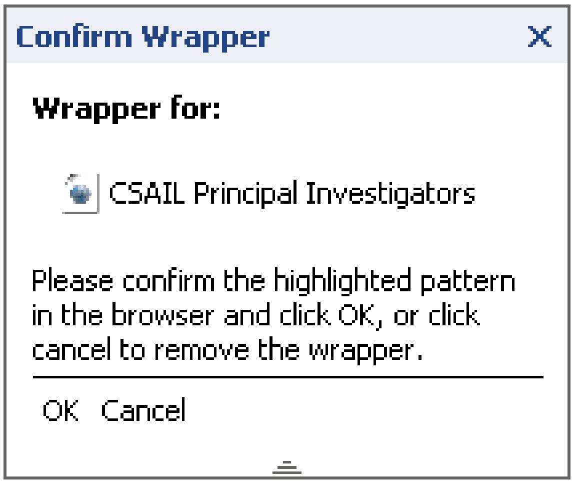


Figure 6-4: The confirmation dialog for wrapper creation.

example several faculty members do not have a "Web Site" link in their listing. An example of one such entry is shown in Figure 6-5.

To generalize the existing wrapper to include this type of entry, the user first highlights the new example, right-clicks, and chooses "Add an Example to a Wrapper" from the context menu. The user is then prompted to choose to which wrapper they wish to add the example, as shown on the right side of Figure 6-5. Once they click

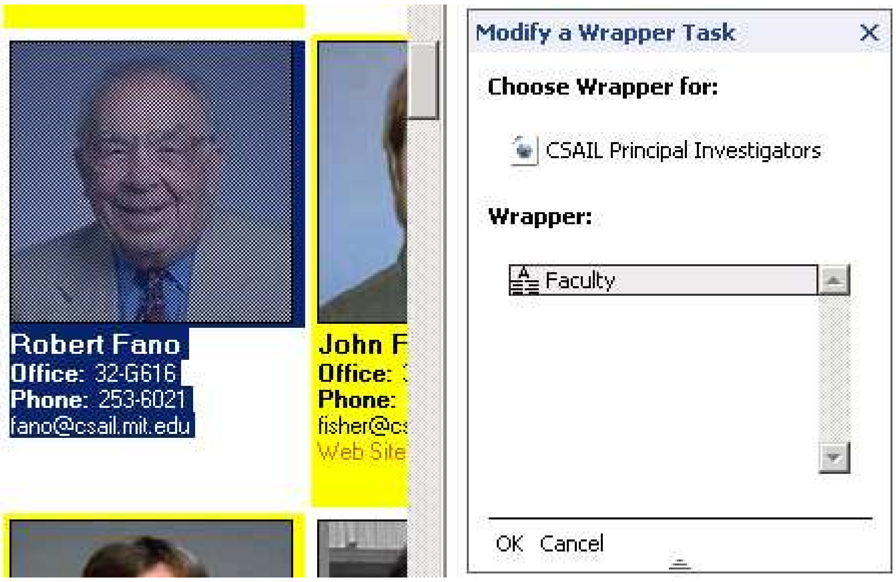


Figure 6-5: Adding an example to a wrapper.

"OK," the pattern is generalized, re-matched against the page, and the new matches are highlighted. In the case of the CSAIL Faculty page, this second example is sufficient to generalize the wrapper to match every faculty member on the page.

Of particular interest for additional examples is semantic content which spans more than one page. Many semantic items only appear once on a page, so they often cannot be generalized from a single example. Because our wrappers are stored in RDF, and accessible to the browser for every similar page via the URL prefix heuristics of Section 4.4.5, the user can provide their multiple examples from a series of pages. To do this, they simply provide the first example on one page, navigate to another similar page, and provide a second example for the same wrapper using the interface outlined above.

6.3 Assigning Properties

Once the user is satisfied with the matches the wrapper provides, they may assert statements about the wrapper's semantic properties. To do this, the user selects a portion of one of the matches on a page which represents a property. They then right-click to bring up the context menu and select "Add a Property to a Wrapper."

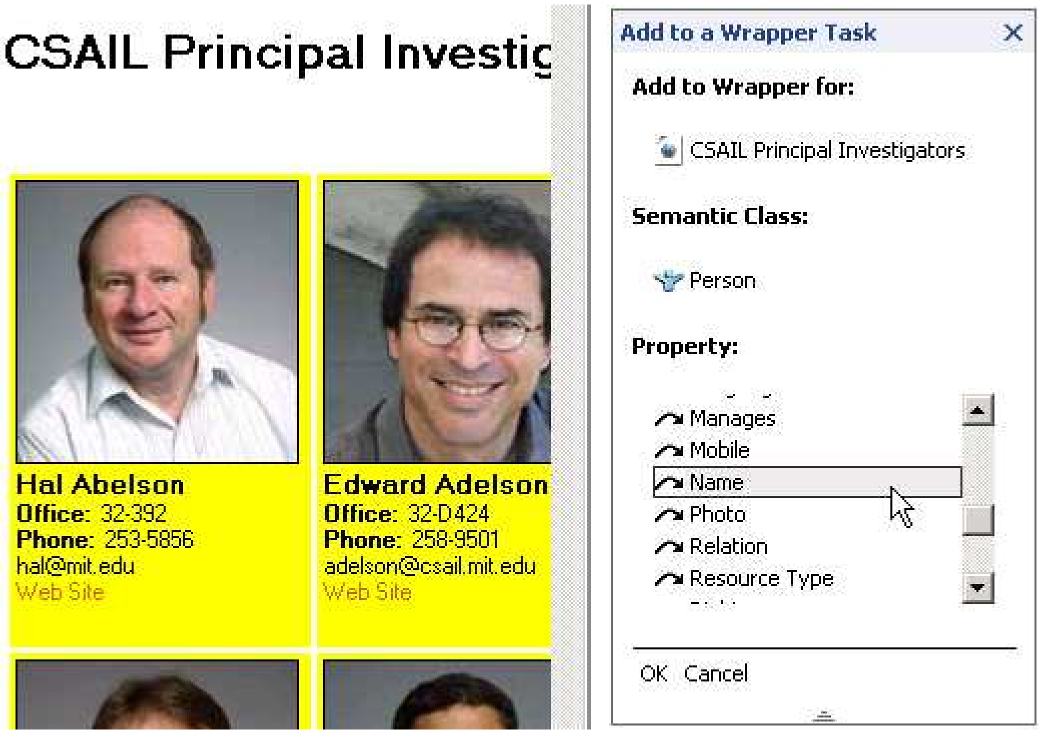


Figure 6-6: Adding a property to a wrapper.

The user is then presented with a UI continuation where they are asked to select from a list of properties appropriate for the given wrapper. This list is generated by querying Haystack's database for properties which have the current wrapper's semantic class as their `rdfs:domain`. General properties that apply to all classes, such as `dc:title`, are also listed. Once the user has selected a property, it is bound to a wildcard node in the pattern as described in Section 5.2. Figure 6-6 shows the interface being used to add the `name` property to the wrapper created for the CSAIL faculty page.

Once the property has been added, the visual feedback is augmented with additional highlighting to indicate that properties have been matched. Figure 6-7 shows an example of this highlighting once several properties have been specified for our faculty wrapper.

Figure 6-7: Visual feedback after a user has added several properties to a wrapper.

6.4 Using Wrappers

Wrappers become most useful once they are fully induced, labeled, and matched against a document. Every time a user browses to a page with a wrapper, we execute the matching algorithm for the wrapper. Any elements in the document which match the pattern are "overlaid" with dynamically generated semantic objects. These objects are fully-functional semantic instances, with properties supplied by the statements assigned in Section 6.3. Because Haystack provides content-specific context menus for semantic data, the user may now interact with semantic content on the web as if it were a first-class RDF object.

For example, in Figure 6-8, the user has right-clicked on one of the faculty members on the CSAIL directory page. Because a `Person` semantic wrapper has been defined for this page, the user is presented with a context menu relevant to that class. This includes such items as "Remind me to contact this party" and "Compose Email Message." Because the properties of these objects are drawn from the page, commands like "Compose Email Message" will have the appropriate information (in this case, an email address) to execute their actions.

Figure 6-8: Interacting with an existing wrapper on a faculty directory page.

To provide consistency for the user between visits, we utilize a heuristic when creating objects to overlay on a page. Given the properties that are bound to the wildcard nodes for a specific match of the wrapper, we query the Haystack database for objects with those properties. If we find an object with those properties, we use that object as our contextual overlay, rather than instantiating a new one. Note that the object is only required to have the properties specified by the wrapper - it may also have additional properties. This heuristic solves several interface issues. First, it prevents us from creating extra instances of objects that represent the same underlying item. In addition, utilizing an existing object means that a user can add additional markup to an object created by a wrapper match, and that markup will still exist the next time the user visits a page. For instance, a user could add one of the faculty members to their address book straight from the web page. When they revisit the page, that same wrapper match will still point to the object currently in

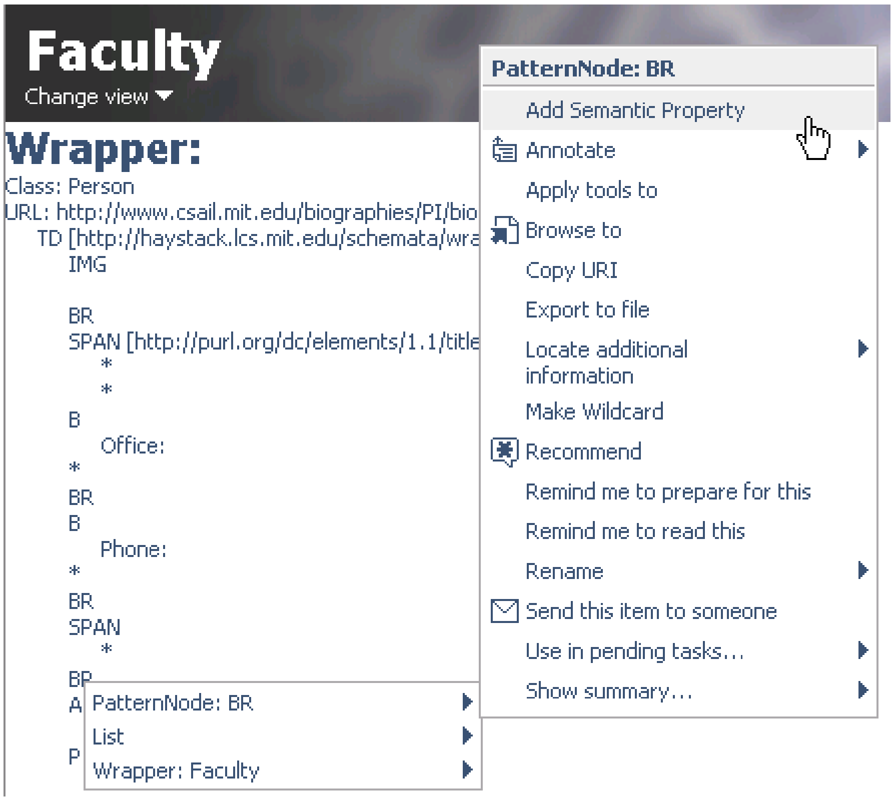


Figure 6-9: The interface for directly manipulating a pattern.

their address book, rather than being an entirely new object.

6.5 Direct Manipulation

The interfaces described in the previous sections are designed to allow users to create useful wrappers without having to know the underlying mechanics or data structures used for wrapper induction, as described in Chapters 4 and 5. While these methods are easy to use, and in most cases are powerful enough to create appropriate wrappers, advanced users may wish to manipulate the underlying patterns directly.

For this purpose, we provide an auxiliary interface for the display and modification of wrappers. This advanced view gives the user a much greater amount of control over the low-level structure of the wrapper. We currently provide operations for turning an existing node into a wildcard as well as adding semantic properties directly to a node.

This pattern manipulation interface is shown in Figure 6-9. In this case, the user is modifying the pattern for the CSAIL faculty directory, as created earlier in this chapter. Note that by right-clicking on an existing pattern node, a context menu is displayed which allows the user to perform operations directly on the given node. These operations include making the node into a wildcard ("Make Wildcard") and adding a property to it ("Add Semantic Property").

Chapter 7

Experimental Results

The wrapper induction system described in the preceding chapters has been demonstrated with examples from several web sites, such as the CSAIL faculty page. In this chapter, we will describe several other experimental results for the system. We begin by reviewing several successful wrappers created on a variety of sites, highlighting important features relevant to our algorithms. We then describe several of the failure modes for our system, and suggest possible improvements to correct them.

7.1 Web Site Survey

The development of our wrapper induction system was based on a survey of popular web sites, enumerated in Appendix A, containing a variety of semantic information. We have attempted to create a set of algorithms and heuristics which will enable the user to create reusable patterns for this content. In this section, we try to evaluate the effectiveness of our system on the same sites for which it was developed.

Appendix B gives several tables containing the results of applying our wrappers to some of the surveyed sites. The tables list the number of examples necessary to wrap a given semantic class or property. They also specify if the wrapper utilized the LAPIS system for partial selections, as noted in the "Wrapper Type" column, as well as brief comments on the wrapper.

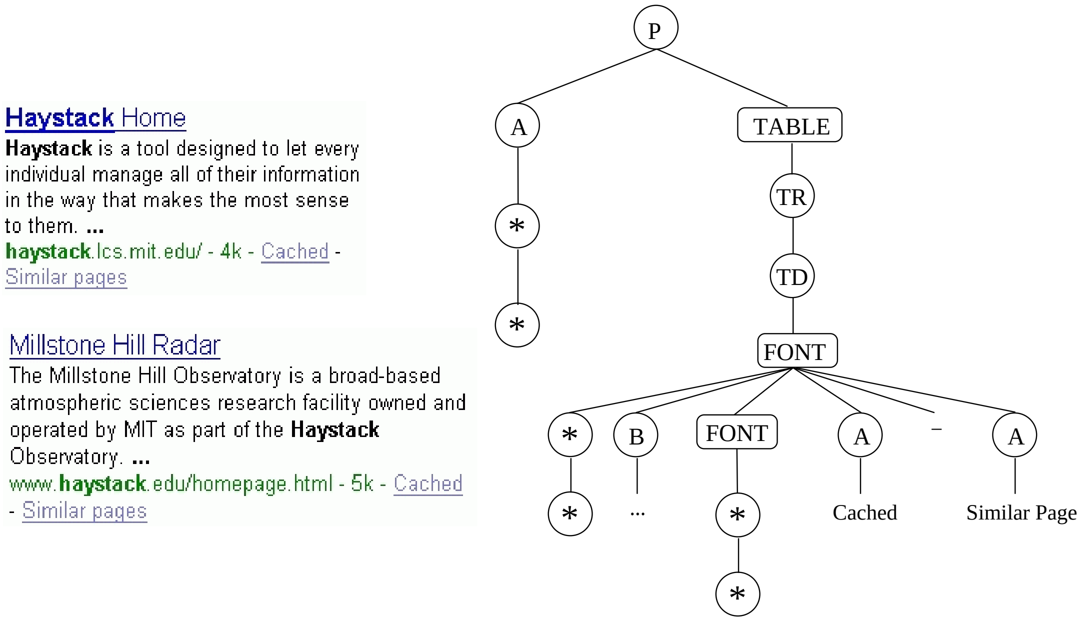


Figure 7-1: The `SearchResult` wrapper on `http://google.com`.

7.2 Successful Wrappers

Overall, our experiments validate our hypothesis that edit distance can create flexible patterns from few examples. On numerous sites, as few as one or two examples are enough to create a useful pattern. In this section we will review in more detail a selection of patterns from several of these sites.

Figure 7-1 shows the wrapper that was induced for the `SearchResult` class on `http://google.com`. We were able to create this wrapper from a single example on the search results page through the addition of *context* as described in Section 4.4.4.

One interesting feature of this wrapper is the appearance of "double" wildcards in several locations (that is, a wildcard node with another wildcard node as its child). These double wildcards are created because of search results where the query term does not appear in the title of the page. For instance, the result for the "Haystack Home" page has the word "Haystack" in bold in its title. Structurally, there is an additional `B` element in the pattern. On the other hand, the result for the "Millstone Hill Radar" does not have this bold element. Thus, when the two results are collapsed, an additional wildcard node is created.

A wrapper for the `LongRangeForecast` class on `http://weather.com` is given in

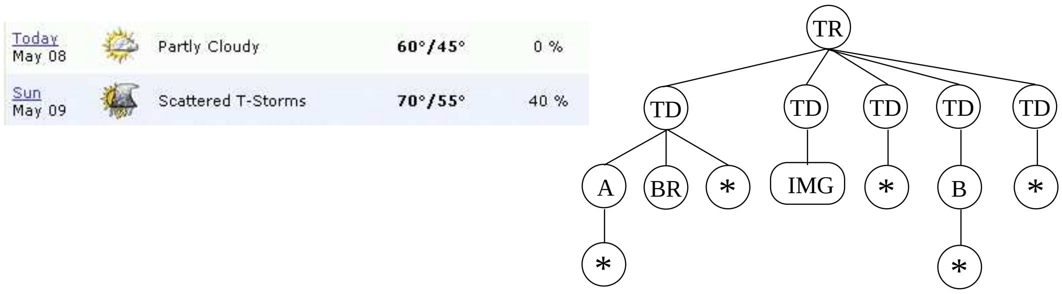


Figure 7-2: The `LongRangeForecast` wrapper on `http://weather.com`.

Figure 7-2. While structurally similar to the Google `SearchResult` wrapper discussed above with respect to context, this wrapper required two examples to create. This is because the large number of differences between the context examples means that the normalized cost, $\hat{\gamma}$, between them exceeded our threshold cost, $\hat{\gamma}_T$. Thus, our system did not automatically collapse these nodes. Rather, we were required to provide a second example to successfully generalize the pattern. In one respect, this is unfortunate, as the examples are clearly similar in structure and perhaps our system should have automatically collapsed them. On the other hand, we have decided to err on the side of not overgeneralizing our patterns by choosing a low $\hat{\gamma}_T$. Because our system only handles positive examples, this limit is important, as it ensures that the user has more control over how general the wrappers become.

The wrapper for the `Movie:actor` property on `http://imdb.com` is interesting as an example of the effectiveness of our *list collapse* heuristic, described in Section 4.4.2. This pattern was created with a single example, by highlighting one of the cast members. Our system successfully wrapped the full list by moving the context up from the originally selected `TR` node to the `TABLE` node. The list collapse heuristic then collapsed the list of actors into a single pattern subtree, shown inside the dashed line in the figure. This occurred because the roots of these subtrees all had the same tag name (`TR`), and the normalized edit distance between them was below the threshold $\hat{\gamma}_T$. We also note that the subtree containing the words "Cast overview, first billed only:" was *not* collapsed, despite having the same tag name at its parent node. This subtree had a higher edit distance cost, and because of this our algorithm correctly inferred that it did not contain the same type of semantic content. Instead,

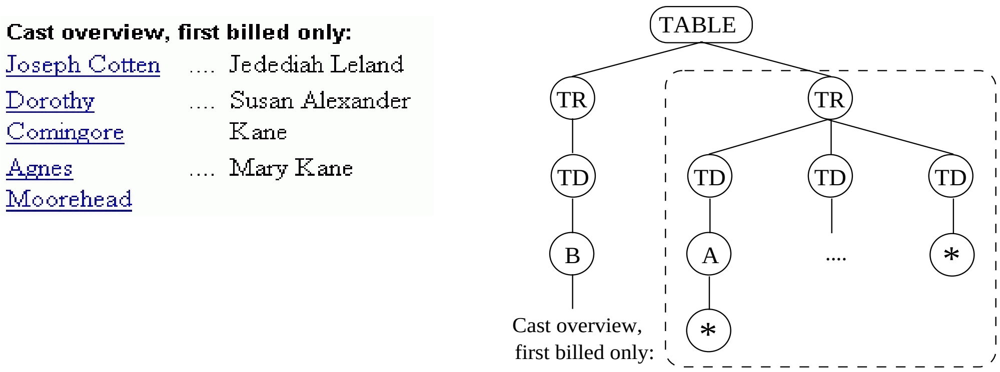


Figure 7-3: The `Movie:actor` wrapper on `http://imdb.com`.

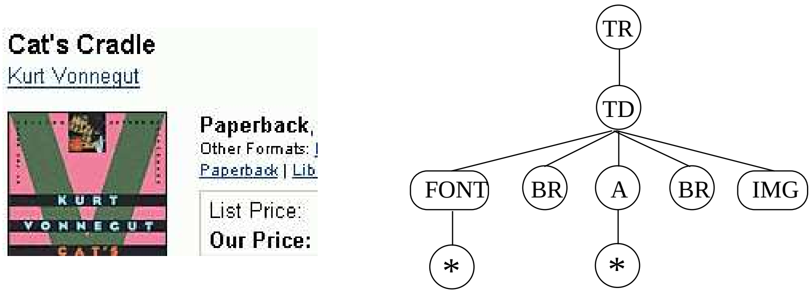


Figure 7-4: The `Book:author` wrapper on `http://bn.com`.

this subtree serves as a “flag” which allows our pattern to match only the list of actors and exclude other elements which do not begin with the text “Cast overview...”

Finally, we consider the wrapper generated for the `Book:author` on Barnes and Noble, as shown in Figure 7-4. This wrapper is interesting in that there is not enough information on a single page to effectively generalize the pattern. There is only one `author` listed on a page, so there is no way for our algorithm to know which nodes should be made wildcards. Instead, we provide another example from a different page containing a `Book` on the same site. These two examples are then merged and the appropriate wildcards are filled in, giving the pattern shown here.

7.3 Failure Modes

Despite the successes outlined above, there were several sites where we either failed to induce a wrapper, the wrapper was incorrect, or generating a valid wrapper took numerous examples. We reason about several of these failures below:

Full-page Classes Many of the semantic classes we wrapped were "full-page." For example, on `http://imdb.com`, each page represents a single instance of the `Movie` type. Because we restrict the allowable size of examples, it becomes impossible to wrap the entire class. Instead, we were able to create wrappers for the *properties* of the full-page class. These properties are often classes themselves, as in the case of an `actor` on the IMDB `Movie` page, which is an instance of type `Person`. These wrappers are still useful in and of themselves, but our system cannot tie them together at the level of the top level, full-page class. The issue of extending our system to accommodate full-page wrappers is addressed in Section 8.2.1.

Selection Inconsistencies Our system depends on reliably extracting the user's selection and finding the related subtree in the page's DOM. In several cases, the inability to do this resulted in failed wrappers. We address this issue more closely in Appendix C.2.1.

Wrapper Size In most cases, our default choices of s_{user} and s_{max} were appropriate. However, on a handful of sites, it was necessary to adjust these parameters using the "Create Wrapper (Specify Size)" option. Although this option is meant for advanced users, it seems to be necessary for some content.

Frames One site (the Java API Reference) utilized HTML frames, which divide the page into several smaller sections. Unfortunately, our system does not deal well with these types of pages because the DOM model does not provide easy access to them. This caused our wrappers to fail unless the user displays only the frame containing the relevant content.

Large Numbers of Semantic "Slots" On one site (ESPN's scoreboard page) our system required 5 examples to successfully generalize, and when it did, the wrappers took a long time to match. This resulted from the nature of the content being wrapped. The `BaseballGame` instances contained two variable semantic "slots" for each inning of play, in addition to several other slots for statistics and team information. To successfully generalize, the examples had to differ in every slot, which meant finding examples with different scores in each inning for each team. Later, when matching, the large number of wildcards meant that matching took longer than normal. This occurs because wildcards can bind to zero, one, or more document nodes, which can create an exponential blow-up in the number of possible alignments during matching.

Chapter 8

Conclusion

8.1 Contributions

In this thesis we have described a system which gives non-technical end users the ability to describe, label, and use semantic content on the World Wide Web. Previous work on labeling content on the Semantic Web, as outlined in Section 2.2, has always focused on either content providers (in the form of page authoring tools) or on technically proficient end users who know HTML and RDF. The tools described here rely on interfaces and user actions already present in existing web browsers, such as highlighting and right clicking on content.

In addition, we have provided a powerful algorithm for creating patterns from tree-structured data using the edit distance between examples. Along with several heuristics to improve its efficiency and accuracy, this method allows us to create reliable patterns with as little as a single example of the relevant content.

Finally, we have drawn out an important connection between the syntactic and semantic structures that appear on the Web. Page authors create an implicit connection between the semantics of the content they are displaying and the syntax with which they display it. Related classes and properties are grouped together on a page, certain properties are offset or highlighted, and lists of similar classes are formatted in the same way. By basing our patterns on this repeating syntactic structure, and then labeling them with the corresponding semantic information, we are making explicit these underlying connections.

8.2 Future Work

The algorithms and interfaces described here are only a first step towards building an easy-to-use, intuitive system for end users to create and manage content on the Semantic Web. Several avenues for future research have suggested themselves, which we outline below.

8.2.1 Wrapper Improvements

This section describes future research and general improvements relating to the underlying wrappers in our system:

Recursive Wrappers Much semantic information on the web is presented in a syntactically nested manner. For example, in a `TalkAnnouncement`, the speaker is not just a text property but a `Person` class. We would like give users the ability to define and associate these items recursively, instead of just as a single level of RDF class associated with a set of properties.

Document-level Classes Many times an entire web page represents a single semantic class, with items on the page detailing the properties of that class. For example, each instance the IMDB `Movie` class is displayed on its own page, with its properties (such as `director`, `actor`, and `character`) laid out throughout the page. Our current wrappers are only able to wrap *pieces* of pages, making it difficult to interact with semantic information that spans a full page. It would be useful to extend our system with the ability to label predicates throughout a page, which are mapped to the page-level class when matched.

Labeling Across Pages Some semantic labels transcend page boundaries. For instance, on the CSAIL events calendar, only the talk series, title and time are listed on the calendar page, while more information is available by clicking on the title link. We would like to develop a system which allows semantic classes and properties to span multiple pages.

Negative Examples There are often cases where our system creates wrappers that are too general in nature based on the positive examples provided by the user. We would like to allow the user the ability to make wrappers more restrictive by giving *negative* examples.

Wrapper Verification Web pages are constantly in flux, making methods for validating wrappers important [19]. We would like to develop an efficient way to verify that the semantic content being returned by the wrappers is still accurate.

Full DOM Use There is much interesting semantic content associated with the attributes of HTML tags, but our current implementation only utilizes the tag names of structural elements and the text of the page. By utilizing the full DOM tree, including attributes, we can form wrappers that capture items like the url to which an anchor tag points, or the source of an image.

Partial Selection Heuristics The wrapper algorithm described here currently makes use of the LAPIS pattern matching system when the user only selects a subset of the children of a node. Instead, it should be possible to extend our system to utilize the *linear* edit distance between two arrays of child nodes. This operation could be based on the edit distance algorithm between two strings, but in our case would use DOM nodes rather than characters as its base unit. Using this edit distance, we could generate "template/slot" patterns similar to the tree patterns described here, but useful for partial selections.

Automation and "Curious" Wrappers With some modifications and enhancements to our edit distance schemes used for pattern induction, it should be possible to automate the location of structures on the web which have the potential for semantic meaning. These structures could be recognized by their repetitiveness and location on the page, as suggested in Chapter 3. We can imagine performing edit-distance calculations on entire pages or on subtrees of pages and locating sets of subtrees with a low-cost edit distance automatically, without user interaction. When repetitive structures are found, the system could present them to the user already generalized, and ask the user to fill in the wildcards with semantic meaning. In this way, the system could become "curious" about the semantic meaning of repetitive structures on the web.

8.2.2 User-Side Improvements

In addition to improving the underlying wrappers, there are several improvements that might be made on the user-facing side of our system:

Ontology Creation and Browsing To create interesting and useful patterns, it is essential that the user has clean, intuitive interfaces for creating and browsing ontologies. The Haystack client has some current interfaces for this task, but they should be more tightly integrated with the wrapper induction tool. This applies not only to the ontology level, but also to the methods for selecting semantic classes and properties to be assigned to wrappers and their wildcards.

Wrapper Management Tools Section 6.5 describes our initial interface for the direct manipulation of wrappers by advanced users. This interface is very basic, and it would be beneficial not only to provide more powerful tools for advanced users, but to provide simple tools for non-technical users to modify and manipulate existing patterns more effectively.

"Push" Wrappers The wrappers defined here are laid out in a context of *pulling* information off of the web. However, many sites work both ways, also allowing the user to fill out forms or submit other types of information. These form entries also have semantic types associated with them, such as a `Person` class, with properties such as `name`, `address`, and `email`. One can imagine wrappers that grab information from the user's personal RDF store and automatically fill out web forms that have been labeled with semantic type information.

8.2.3 Applications

Several of the most exciting avenues of future work involve applications of the working wrapper induction system presented here:

Page Reformatting When two pages are wrapped using the same ontology, the user has implicitly specified that they contain objects of the same semantic type. The "slots" in the page where these types reside then become somewhat "interchangeable," in that other objects of the same semantic type may also be inserted or appended into the slots. This gives us the opportunity to reformat and consolidate semantic information which has been similarly labeled.

For instance, the user may have created wrappers for several sites, such as CNN, the New York Times, and the BBC, using the `News` ontology. Each of these wrappers specifies the class `Story`, with the properties `title`, `author`, and `body`. We can execute these wrappers, gathering objects of the type `Story`. Interestingly, we can also *fill in*, or *exchange* the semantic content of a page with other content of the same type. For instance, if the user likes the formatting of the New York Times' web site, we could insert stories from the BBC and CNN into the semantic slots created in the New York Times page. The user could then browse stories from all their news sites at once, using the same formatting.

Custom RSS Feeds A related idea involves the RSS standard [5], which is designed for easy syndication of news feeds. RSS contains information of a certain semantic type, specifically, the `Story` class of the `News` ontology. We could imagine combining wrappers of this type with an autonomous agent designed to execute these wrappers on a regular basis. The results from these wrappers could be serialized into the RSS format, to be incorporated into news readers compatible with the standard.

Autonomous Agents The concept of autonomous agents interacting with wrappers has broad applications in other areas, as well. One could imagine agents that monitor wrappers of the type `BankAccount` to alert the user of negative balances, or to automatically update tax or accounting software from web-based financial sites. Wrappers defined on sites with events calendars or seminar announcements could be automatically aggregated into a central calendar. On a travel information site, agents could monitor flights for delays or even book vacations for the user.

Wrapper Sharing Because they are defined and backed by the RDF language, the nature of the wrappers defined here allows them to be easily serialized. Our implementation uses this feature to store and retrieve wrappers to disk in between sessions, and to interface wrappers with the Haystack framework. However, this serialization also suggests other interesting applications. Once semantic patterns have been created for a page, they may be shared between users. One can imagine downloading a full set of wrappers for a given site and instantly enabling a full Semantic Web experience for users without the need for each user to author their own wrappers. A distributed, peer-to-peer-type network for sharing wrappers could aid not only in the adoption of the Semantic Web, but in validating wrappers for our system and in creating more useful shared ontologies.

Appendix A

Surveyed Sites

During the design and development of the wrapper induction algorithms presented here, the sites below on the World Wide Web were surveyed and used for testing purposes. They represent 29 semantic classes with 127 properties across 19 sites.

Site	URL Prefix	Class	Properties
Google	http://google.com/query	SearchResult	title, summary, category
		SponsoredLink	title, summary, url
Yahoo!	http://search.yahoo.com/search	SearchResult	title, summary, catetgory, url
		SponsoredLink	title, summary, url
Internet Movie Database (IMDB)	http://imdb.com	Movie	title, director, writer, character, actor
		Actor	name, birthday, role
New York Times	http://nytimes.com	NewsStory	headline, byline, date, summary
	http://nytimes.com/2004/...	NewsStory	headline, byline, date, storyText, relatedArticles
CNN	http://cnn.com	NewsStory	headline
	http://cnn.com/2004/...	NewsStory	headline, date, storyText
CNet News	http://news.com.com	NewsStory	headline, summary, date
Slashdot	http://slashdot.org	NewsStory	headline, poster, date, icon, summary, link
	http://slashdot.org/article.pl	Comment	comment, commenter, commentNo, commentScore
Weather.com	http://www.weather.com /weather/local/	Forecast	conditions, temperature, dewpoint, humidity, visibility, pressure, wind, radarImage
		LongRangeForecast	date, conditions, temperature, precipitation

Table A.1: Surveyed web pages.

Site	URL Prefix	Class	Properties
CSAIL Directory	`http://www.csail.mit.edu /biographies/PI/biolist.php`	`Person`	`name`, `office`, `phone`, `email`, `webSite`
CSAIL Event Calendar	`http://www.csail.mit.edu /events/eventcalendar/calendar.php`	`TalkAnnouncement`	`series`, `title`, `time`
		`TalkAnnouncement`	`title`, `speaker`, `date`, `time`, `location`, `host`, `abstract`
MIT Course Catalog	`http://student.mit.edu/catalog/`	`Course`	`title`, `prerequisite`, `time` `summary`, `professor`
Mozilla Bugzilla	`http://bugzilla.mozilla.org /show_bug.cgi`	`Bug`	`bugNumber`, `status`, `assignedTo`, `reporter`, `dependsOn`
Citeseer	`http://citeseer.ist.psu.edu/`	`Paper`	`abstract`, `citedBy`, `bibtexEntry`, `relatedDocument`, `citation`, `coCitation`, `similarDocument`
Java API Reference	`http://java.sun.com /j2se/1.4.2/docs/api`	`JavaClass`	`package`, `classname`, `superclass`, `implementedInterface`, `method`, `description`, `field`, `constructor`,
		`JavaMethod`	`name`, `declaration`, `returnType`, `description`, `returns`, `throws`, `seeAlso`

Table A.2: Surveyed web pages (continued).

Site	**URL Prefix**	**Class**	**Properties**
ESPN	`http://sports.espn.go.com/**/scoreboard`	`Game`	`homeTeam`, `visitingTeam`, `homeTeamScore`, `visitingTeamScore`, `gameTime`
	`http://sports.espn.go.com/**/clubhouse`	`Team`	`name`, `scheduledGame`, `headline`, `teamLeader`, `teamNote`
50 States	`http://50states.com/`	`State`	`statehoodDate`, `bird`, `borderStates`, `flagImage`, `flower`, `governor`, `motto`, `nickname`, `population`,
			`capital`, `tree`
EBay	`http://cgi.ebay.com/ws/eBayISAPI.dll`	`Auction`	`title`, `itemNo`, `startingBid`, `timeLeft`, `startTime`, `history`, `seller`, `description`
Barnes & Noble	`http://search.barnesandnoble.com /booksearch/isbninquiry.asp`	`Book`	`title`, `author`, `listPrice`, `salePrice`, `image`, `isbn`, `format`, `publicationDate`, `publisher`, `salesRank`, `summary`, `review`
Amazon.com	`http://www.amazon.com /exec/obidos/ASIN/`	`Book`	`title`, `author`, `listPrice`, `salePrice`, `availabilty`, `image`, `alsoBought`, `publisher`, `isbn`, `salesRank`, `editorialReview`

Table A.3: Surveyed web pages (continued).

Appendix B

Wrapper Results

The following tables give our experimental results for a subset of the web sites listed in Appendix A. For more detail on the reasons for the successful wrappers and failure modes of our system, please see Chapter 7.

Site	**Class or Property**	**Wrapper Type**	**No. of Examples**	**Comments**
Google	`SearchResult`	Standard	1	Context allowed for a single-example pattern.
Yahoo!	`SearchResult`	Standard	1	Context allowed for a single-example pattern.
IMDB	`Movie:character`	Standard	1	Wrapped full list from single example.
IMDB	`Movie:director`	Standard	2	Examples on multiple pages required.
IMDB	`Movie:writer`	Standard	2	Examples on multiple pages required.
New York Times	`NewsStory`	LAPIS	2	Failed to match last headline.
Slashdot	`NewsStory`	LAPIS	fail	Inconsistencies because of selection.
Slashdot	`NewsStory:icon`	Standard	1	Individual property of `NewsStory` wrapped easily.
Slashdot	`NewsStory:poster`	Standard	1	Individual property of `NewsStory` wrapped easily.
Weather.com	`Forecast:conditions`	Standard	2	Examples on multiple pages required.
Weather.com	`LongRangeForecast`	Standard	2	List collapse cost threshold too high for single example.

Table B.1: Results of wrapping surveyed sites.

Site	**Class or Property**	**Wrapper Type**	**No. of Examples**	**Comments**
CSAIL Directory	`Person`	Standard	2	Second example necessary for people without a “Web Page” listing.
CSAIL Event Calendar	`TalkAnnouncement`	Standard	2	Second example necessary for talks without a “series”
MIT Course Catalog	`Course`	Standard	1	
Mozilla Bugzilla	`Bug:bugNumber`	Standard	2	Examples on multiple pages required.
Mozilla Bugzilla	`Bug:status`	Standard	2	Examples on multiple pages required. Specified $s_{user} = 50$ to capture appropriate context.
Mozilla Bugzilla	`Bug:assignedTo`	Standard	2	Examples on multiple pages required. Specified $s_{user} = 50$ to capture appropriate context.
Citeseer	`Paper:abstract`	LAPIS	1	Slight overgeneralization dependent on exact selection.
Citeseer	`Paper:bibtexEntry`	Standard	2	Examples on multiple pages required. Specified $s_{user} = 50$ to capture appropriate context.
Java API Reference	`JavaClass`	Standard	fail	Page uses frames, and wrapper induction could not find selection
Java API Reference (one frame)	`JavaClass:method`	Standard	2	Second example necessary for variable number of arguments

Table B.2: Results of wrapping surveyed sites (continued).

Site	**Class or Property**	**Wrapper Type**	**No. of Examples**	**Comments**
ESPN	`BaseballGame`	Standard	5	Several examples necessary because of box score entries. Resulting wrapper has many wildcards, causing matching to be slow.
ESPN	`Team:headline`	Standard	2	
50 States	`State:capital`	Standard	2	Examples on multiple pages required.
EBay	`Auction:title`	Standard	2	Examples on multiple pages required.
EBay	`Auction:startingBid`	Standard	2	Examples on multiple pages required.
EBay	`Auction:timeLeft`	Standard	2	Examples on multiple pages required.
Barnes & Noble	`Book`	Standard	fail	Issues with getting correct user selection.
Barnes & Noble	`Book:title`	Standard	2	Examples on multiple pages required.
Barnes & Noble	`Book:salePrice`	Standard	2	Examples on multiple pages required.
Amazon.com	`Book:title`	Standard	2	Examples on multiple pages required.
Amazon.com	`Book:author`	Standard	2	Examples on multiple pages required.

Table B.3: Results of wrapping surveyed sites (continued).

Appendix C

Implementation Details

In developing the user interfaces and algorithms described in this thesis, we have built several useful programming interfaces as well as encountered several issues which are important for future developers. This appendix details these items.

C.1 Java Interfaces

The system described here has been implemented using the Java [14] programming language. While our initial implementation has used Microsoft Internet Explorer as the browser component, we have designed our wrapper induction system to be easily ported to other browsers or HTML parsers.

The implementations of our algorithms interact with interfaces which extend the DOM standard [2] for parsing HTML documents. Table C.1 lists the Java interfaces from package `org.w3c.dom` which our system requires.

In addition to these methods, we have implemented additional interfaces of our own which extend the base DOM interfaces. These provide additional functionality not specified in the DOM standard. Tables C.2, C.3, and C.4 briefly describe these interfaces, the superinterfaces they extend, and their key methods.

We have designed these additional interfaces with two important facets in mind. First, they must provide the required functionality to our algorithms and user interfaces, including the ability to retrieve highlighted text, add and remove elements

Interface	Description
`org.w3c.dom.Node`	Top-level interface for all DOM nodes.
`org.w3c.dom.Document`	Root node of an HTML document.
`org.w3c.dom.Element`	A structural node within a document
`org.w3c.dom.Text`	A text node
`org.w3c.dom.NodeList`	A list of `Node` objects

Table C.1: Required DOM interfaces.

from the document, and highlight portions of the page. Second, we attempted to keep the required methods within the facilities provided by most modern browsers. We hope that our algorithms may be easily ported to other potential browsers, such as Mozilla[1] or Konqueror[2].

C.2 Issues

In this section we will describe several issues which arose during the implementation of the algorithms and interfaces described here. This is meant as a reference for future developers who may utilize our tools.

C.2.1 Internet Explorer

Many of the issues that arose during implementation were related to our choice of Microsoft Internet Explorer as the browser for our system. This choice was made for several reasons:

- The Haystack system is built on the SWT toolkit[3], which utilizes Internet Explorer for its web browser component on Windows operating systems..
- A large percentage of the Haystack user base uses Windows.
- Internet Explorer's DOM interfaces are mature, and fairly well documented.

[1] `http://mozilla.org`
[2] `http://konqueror.org`
[3] `http://eclipse.org`

Interface or Method	Description
`ITree extends org.w3c.dom.Document`	Methods for dealing with tree structures
`int getSize()`	Returns the size (number of nodes) of the tree.
`INode[] getNodes()`	Returns the nodes of this tree, in postorder.
`IDOMDocument extends ITree`	Extended methods for querying or modifying DOM documents.
`String getURL()`	Returns the URL of this document.
`String getTitle()`	Returns the title of this document.
`String getDomain()`	Returns the domain, or server, of this document.
`String getPathname()`	Returns the path of this document on its respective server.
`IDOMElement getActiveElement()`	Returns the currently active element (the last element clicked on by the user).
`IDOMElement getElementAtPoint(int x, int y)`	Returns the element at the given point.
`DOMSelection getSelection()`	Returns an object representing the current selection.
`void write(String input)`	Appends the given string to the document.
`DOMSelection`	Represents the currently selected portion of the document
`boolean isEmpty()`	Returns true if this selection is empty (contains no elements).
`boolean isActiveElementSelection()`	Returns true if this selection was generated from the active element, rather than from an actual selection.
`boolean isPartialSelection()`	Returns true if this selection covers a subset (of size greater than one) of the parent element's child nodes.
`String getHtmlText()`	Retrieves the underlying HTML of the selection.
`IDOMElement getParentElement()`	Retrieves the parent element of the selection
`IDOMElement[] getSelectedElements()`	Retrieves the subset of children of the parent element contained by this selection.
`NodeID getNodeID() throws NodeIDException`	Generates a `NodeID`, or root-to-node path, for this selection.

Table C.2: Extended interfaces.

Interface or Method	Description
`INode extends org.w3c.dom.Element`	Methods for dealing with individual nodes in a tree.
`NodeID getNodeID()`	Returns the `NodeID`, or root-to-node path, representing this node's position in the tree.
`int getSize()`	Returns the size (number of nodes) of the subtree rooted at this node.
`int getHeight()`	Returns the height of this node, or the length of the longest path from this node to a leaf. A leaf is defined to have height 1.
`INode[] getPostorderNodes()`	Retreives the nodes in the subtree rooted at this node using a post-order traversal.
`INode[] getPreorderNodes()`	Retreives the nodes in the subtree rooted at this node using a pre-order traversal.
`int getSiblingNo()`	Returns the sibling number of this child (i.e. which index child it is of its parent).
`void setSiblingNo(int siblingNo)`	Sets the sibling number of this child (i.e. which index child it is of its parent).
`boolean isOnlyChild()`	Returns true if this INode is the only child of its parent.
`INode getChild(int index)`	Returns the child of this node at the given index.
`INode[] getChildren(String tagName)`	Returns all immediate children of this node with the given tag name.
`INode getAncestor(int generation)`	Retrieves the Nth ancestor of this node. N=0 returns this node, N=1 returns its parent, N=2 its grandparent, etc. Returns null if that ancestor does not exist.
`void setParent(INode parent)`	Sets the parent of this node to the given node.
`NodeList getSiblings()`	Retreives an array containing the siblings of this node, including this node.
`INode removeNode()`	Removes this node and all its decendents from the tree.
`List removeChildNodes()`	Removes the children of this node, returning them as an ordered list.

Table C.3: Extended interfaces (continued).

Interface or Method	Description
`IDOMElement extends INode` (continued)	
`boolean equals(INode other)`	Returns true if the node given is the same as this node for the purposes of editing distance between trees.
`String toString(int depth, String indent)`	Returns this node recursively as a string, with the specified indentation
`int getDeleteCost()`	Returns the cost to delete this node.
`int getInsertCost()`	Returns the cost to insert this node.
`int getChangeCost(INode other)`	Returns the cost to change this node to the given other node.
`IDOMElement extends INode`	Extended methods for querying or modifying DOM elements.
`String getTagName()`	Returns the tag name of this node.
`String getNodeText()`	Retrieves all text (i.e. non-markup) from within this element and its descendents.
`String getOuterHTML()`	Returns all HTML for this node and its children.
`String getInnerHTML()`	Returns all HTML for this node's children (not including this node).
`String startTagHTML()`	Retrieves the text of html representing the start tag of this object. If this is a TEXT_NODE, returns the text of this node.
`String endTagHTML()`	Retrieves the text of html representing the end tag of this object, or ”” if this object does not have a closing tag (e.g. ¡BR¿). If this is a TEXT_NODE, returns the empty string (””).
`IDOMElement highlight(String highlightColor, String textColor)`	Highlights this element.
`void unhighlight()`	Returns this element to its original, un-`highlight()`-ed style.
`String getAttribute(String attributeName)`	Returns the value of the given attribute.

Table C.4: Extended interfaces (continued).

Unfortunately, this choice required us to deal with several bugs, some of which often seemed nondeterministic. We outline these below, for the benefit of future developers.

Inconsistent Sibling Numbers We heavily utilize the notion of a node's *sibling number*, as defined in Section 4.1. We often found it very difficult to determine a node's sibling number in practice, and sibling numbers occasionally differed on subsequent visits to the same page.

Selection-to-DOM Translation The DOM standard does not deal with the notion of a user's "selection" in the document at all. Microsoft has extended the DOM interfaces, though, to allow for the retrieval of the current selection. However, the selection interface only returns the HTML source of the selection, in flat text format. It does not provide any interfaces for translating between this text and the underlying DOM model. We went to great lengths to work around this missing functionality in implementing the `DOMSelection` interface.

Inconsistent Selections The ability to consistently gather the user's intended selection and turn it into a wrapper is central to our work. However, we found that selecting certain elements in the browser was difficult, or often impossible. At other times, identical-looking selections returned different underlying HTML. Also, the HTML returned by the selection is often inconsistent with the HTML returned by Internet Explorer's `getOuterHTML` method. This made it very difficult to consistently determine the user's intent when gathering the selection.

As mentioned earlier, our hope is that our abstract wrapper induction model, because it makes generous use of Java interfaces, will be easy to port to other web browsers which may have a more consistent user experience.

C.2.2 Other Issues

Several other issues occurred which were unrelated to our choice of browser. These are outlined below, along with our solutions or intended future work:

LAPIS We chose to interface with the LAPIS system [26] to handle cases where the user has only partially selected the children of a node (see Section 4.4.3). In many cases, LAPIS generalized effectively given only one or two examples. However, there are several cases where the pattern was overgeneralized, or where no LAPIS pattern could be found. We are working with the authors of the LAPIS system to improve its effectiveness in our use case.

Semantic Roots When gathering context for a wrapper, as described in Section 4.4.4, we move the root of the pattern up the DOM. During this process, it is necessary to keep track of the initial root, as this is the node with semantic meaning. To do this, we tag this node with the semantic predicate `<wrapperinduction:semanticRoot>`. Later, when we match, we look for this predicate and use it to generate a unique resource representing the semantic class of the object.

Caching Results Our initial implementation re-matched all wrappers on the page every time the user right-clicked or in some other way interacted with the wrapped content. This proved to be slow, and unnecessary, as the page had not changed since the initial matches were made. We have implemented a caching system within Haystack's browser part which stores these results and allows them to be accessed as long as the page is displayed.

Instantiating Objects Each time the user clicks on a matched portion of the page, an instance of the wrapped class is created and populated with the semantic information from the match. Currently, a new object is instantiated every time the user clicks on an object. This creates issues not only with redundant data, but also with the user's expectation that every click is interacting with the same object. To fix this, we plan to augment our result caching system to include the instantiated objects as well, and to retain pointers to these objects for future visits using a hash of the matched document subtree.

Bibliography

[1] The CGI specification, Version 1.1. http://hoohoo.ncsa.uiuc.edu/cgi/interface.html.

[2] Document Object Model (DOM). http://www.w3.org/DOM/.

[3] Resource Description Framework (RDF) specification. http://www.w3.org/RDF, 1999.

[4] XML Path language (XPath) specification. http://www.w3.org/TR/xpath, 1999.

[5] RDF Site Summary (RSS) 1.0 specification. http://web.resource.org/rss/1.0/spec, 2001.

[6] xpath2rss HTML to RSS scraper. http://www.mnot.net/xpath2rss/, 2003.

[7] T. Berners-Lee, J. Hendler, and O. Lassila. The semantic web. *Scientific American*, 284(5):35, May 2001.

[8] R. Cole, R. Hariharan, and P. Indyk. Tree pattern matching and subset matching in deterministic $O(n \log^3 n)$-time. In *SODA: ACM-SIAM Symposium on Discrete Algorithms (A Conference on Theoretical and Experimental Analysis of Discrete Algorithms)*, 1999.

[9] M. Collins and S. Miller. Semantic tagging using a probabilistic context free grammar. In *Proceedings of 6th Workshop on Very Large Corpora*, Montreal, Canada, 1997.

[10] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, second edition, 2001.

[11] M. Dubiner, Z. Galil, and E. Magen. Faster tree pattern matching. *J. Association of Computing Machinery*, 41(2):205–213, March 1994.

[12] D. Freitag and A. McCallum. Information extraction with HMM structures learned by stochastic optimization. In *AAAI/IAAI*, pages 584–589, 2000.

[13] J. Golbeck, M. Grove, B. Parsia, A. Kalyanpur, and J. Hendler. New tools for the semantic web. In *Proceedings of 13th International Conference on Knowledge Engineering and Knowledge Management*, Oct 2002.

[14] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification Second Edition.* Addison-Wesley, Boston, Mass., 2000.

[15] C. M. Hoffmann and M. J. O'Donnell. Pattern matching in trees. *J. Association of Computing Machinery*, 29(1):68–95, January 1982.

[16] J. HopCroft and R. Tarjan. *Isomorphism of Planar Graphs*, pages 131–152. Plenum Press, 1972.

[17] J. Kahan and M. Koivunen. Annotea: an open RDF infrastructure for shared web annotations. In *World Wide Web*, pages 623–632, 2001.

[18] D. Karger, B. Katz, J. Lin, and D. Quan. Sticky notes for the semantic web. In *Proceedings of the 8th International Conference on Intelligent User Interfaces*, pages 254–256, 2003.

[19] N. Kushmerick. Wrapper verification. *World Wide Web*, 3(2):79–94, 2000.

[20] N. Kushmerick, D. Weld, and R. Doorenbos. Wrapper induction for information extraction. In *Intl. Joint Conference on Artificial Intelligence (IJCAI)*, pages 729–737, 1997.

[21] D. Matula. An algorithm for subtree identification. *SIAM Rev.*, 10:273–274, 1968.

[22] B. McKay. Practical graph isomorphism. *Congressus Numerantium*, 30:45–87, 1981.

[23] I. Muslea, S. Minton, and C. Knoblock. A hierarchical approach to wrapper induction. In Oren Etzioni, Jörg P. Müller, and Jeffrey M. Bradshaw, editors, *Proc. of the Third International Conference on Autonomous Agents*, pages 190–197, Seattle, WA, USA, 1999. ACM Press.

[24] D. Quan, D. Huynh, and D. Karger. Haystack: A platform for authoring end user semantic web applications. In *Proc. 2nd International Semantic Web Conference*, 2003.

[25] D. Quan, D. Huynh, V. Sinha, and D. Karger. Adenine: A metadata programming language. In *Student Oxygen Workshop*, 2002.

[26] R.Miller and B. Meyers. Lightweight structured text processing. In *Proc. of USENIX 1999 Annual Technical Conference*, pages 131–144, Monterey, CA, USA, June 1999.

[27] K. Seymore, A. McCallum, and R. Rosenfeld. Learning hidden Markov model structure for information extraction. In *AAAI 99 Workshop on Machine Learning for Information Extraction*, 1999.

[28] L. Shih and D. Karger. Learning classes correlated to a hierarchy. Technical report, MIT AI Lab, 2001.

[29] P. Weiner. Linear pattern matching algorithms. In *Proc. of the 14th IEEE Symposium on Switching and Automata Theory*, pages 1–11, 1973.

[30] E. Weisstein. Graph isomorphism complete. From MathWorld - A Wolfram Web Resource, http://mathworld.wolfram.com/GraphIsomorphismComplete.html.

[31] S. Wu and U. Manber. Agrep – a fast approximate pattern-matching tool. In *Proceedings USENIX Winter 1992 Technical Conference*, pages 153–162, San Francisco, CA, 1992.

[32] K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM J. Computing*, 18(6):1245–1262, December 1989.