XPath-Wrapper Induction by generalizing tree traversal patterns

Tobias Anton
Technical University of Darmstadt, Germany
tobias.anton@web.de

Abstract

We introduce a wrapper induction algorithm for extracting information from tree-structured documents like HTML or XML. It derives XPath-compatible extraction rules from a set of annotated example documents. The approach builds a minimally generalized tree traversal pattern, and augments it with conditions. Another variant selects a subset of conditions so that (a) the pattern is consistent with the training data, (b) the pattern's document coverage is minimized, and (c) conditions that match structures preceding the target nodes are preferred. We discuss the robustness of rules induced by this selection strategy and we illustrate how these rules exhibit knowledge of the target concept.

1 Introduction

The internet is a huge source for almost any kind of information. Many web applications provide a view to a database by encapsulating records of a single table in formatting HTML code. Our goal is to make the data behind such pages accessible for machine processing. For a machine, however, the underlying structure is not obvious, mostly due to ambiguities in the usage of tags for different entities as well as lack of semantics in HTML tags. Even though great efforts are taken to enrich the web with semantics [Berners-Lee, et al., 2001], it is still necessary today to hand-craft wrappers, which essentially consists of a series of typical tasks: (1) Define the target schema to be extracted and annotate portions of example documents as belonging to the schema, (2) find patterns in or around the annotated examples, (3) create a parser that recognizes these patterns and (4) write a program to extract the contents identified by them.

If the schema underlying the presented data is simple enough and if the source is sufficiently regular, these tasks can be transferred in part to a computer. If only task (1) is left to the user, the automated part is referred to as "wrapper induction".

2 Related Work

Information extraction is recognized as an application of standard machine learning techniques to the problem of classifying document fragments based on features derived from their context [Finn & Kushmerick 2004]. As such, Wrapper induction exists in supervised and unsupervised flavours, even semi-supervised variants have been presented [Scheffer, et al., 2001].

Many wrapper induction systems use a high-level representation for the wrapping task, so that the code generation in tasks (3) and (4) reduces to interpret the set of patterns output by task (2) with a generic wrapper engine. Depending on the flexibility of the wrapper engine, more or less complex wrappers can be induced by the learning component of the system. The complexity of rules that can be learned by an algorithm is referred to as "expressiveness".

Wrapper induction systems can also be divided up into string-based, token-based and HTML-aware ones. String-based and token-based wrapper induction systems regard the document as a sequence of characters or tokens, respectively. The algorithms behind these systems usually search for delimiters, delimiter patterns or regular delimiter languages that suit the training data well. HTML-aware systems are either designed like their token-based counterparts, but use a tokenizer that is adapted to HTML, or they try to exploit the tree-structure of HTML explicitly, with the expectation that the tree-view on the document exhibits structure that would remain hidden otherwise.

WIEN [Kushmerick, et al., 1998] by Nicholas Kushmerick was the first wrapper induction system to our knowledge. It is a string-based supervised learning procedure that creates wrappers by finding the shortest prefix and suffix that delimit all training examples in the example documents and nothing else. Under not too pessimistic assumptions, if such a pattern exists, the HLRT variant of the algorithm will discover it in quadratic time with respect to the document size [Kushmerick, 2000].

That algorithm and the derived classes HLRT, OCLR and HOCLRT are somewhat limited because they only consider regular delimiter patterns without variables. To bypass this limitation, BWI [Freitag & Kushmerick, 2000] was developed on top of these algorithms. It learns an set of LR-wrappers using AdaBoost, combining rules by weighted voting.

The **STALKER**-algorithm [Muslea, et al., 2000] from the ARIADNE-Project [Ambite, et al., 1998] is a token-based approach that also creates extraction patterns from examples. It is more expressive than WIEN, because the patterns it learns may contain wildcards and even disjunctive rules. The learning component treats the documents as a sequence of tokens, but it allows a hierarchical structuring of the learning task: Multiple rules can be nested to allow for recognition of complex structures such as lists of tuples or lists of lists. The extraction result of one wrapper is used as the input for the wrappers on the deeper level. This nesting model also allows to represent parent-child relationships analogous to the HTML structure of the source, but the nesting structure must be defined manually.

SoftMealy [Hsu & Dung, 1999] is a token-oriented, statistical approach that learns a markov model. In the model, some states are associated with data fields, while others are only used to skip document fragments. For ex-

tracting data, the model is matched to a new document and for each token, the most likely state defines whether to skip it and otherwise into which field to extract it.

RoadRunner [Crescenzi, et al., 2001] is an unsupervised, token-based learning system that learns target schema based on the contents of two or more unannotated example documents. A wrapper is represented as a sequence of tokens with wildcards, repetitions and optional elements. Patterns can be learned incrementally. Whenever a mismatch occurs, a generalization step is done: Depending on the type of mismatch, the conflicting tokens are replaced for a wildcard, a repetition or an optional sequence of tokens.

XWrap Elite [Han, et al., 2001] discovers tree-patterns in an unsupervised manner. The user presents a page and chooses a heuristic. Based on the result, which is a list of extracted elements, she may impose further constraints on the number of results or choose another heuristics. After some refinement, a piece of java source code is generated that represents the learned concept. However, since no annotated documents are accepted as examples, no consistency constraints can be imposed to the system.

A few interactive programming approaches were found that explicitly exploit the tree structure of HTML:

W4F [Sahuguet & Azavant, 1999] requires the user to write a valid extraction rule in a language called "HEL", which can be seen as a hybrid between XPath and SQL. It assists the user only in discovering the "tree path" that leads to a target node. The tree path is the sequence of tag names of all ancestor tags from the document root to the target node, each decorated with a serial number. The user can generalize this path manually by substituting serial numbers for variables and defining relationships between these variables in an SQL-style "WHERE"-clause.

Lixto [Baumgartner, et al., 2001], in contrast, has a convenient user interface for selecting one starting node. Based on the selection, a path is generated, but in contrast to W4F, the path is a sequence of tag names or wildcards derived from the sequence of nodes preceding the selection. Further constraints can then be defined interactively by the user.

ANDES [Myllymaki & Jackson, 2002] is an information extraction system from IBM. It is completely based on open standards. It consists of a web crawler whose configuration files are stored in XML and a wrapper shell that uses XSLT scripts as extraction patterns. These scripts require XPath expressions to identify target elements that must be hand-crafted by the user. Even though [Myllymaki & Jackson, 2002] describe in detail how robust extraction rules should be written, no learning component is presented.

In fact, no html-aware tools were found that induce wrappers from examples specified by the user. The purpose of our paper is to fill this gap and to provide a proposal for a learning algorithm suited for XSLT-based information extraction. Due to the close relation to the approach presented here, we will outline the concepts from [Myllymaki & Jackson, 2002], but before, we will provide a short introduction to XPath in the next section.

3 DOM and XPath

XPath [W3C, 1999] is a language for traversing DOM trees, where a DOM tree is a representation for a well-formed XML document.

A DOM tree is an ordered tree, where every node has assigned a name and a value. We distinguish two types of nodes: Element nodes and text nodes. Text nodes may only occur as leaves. Their name is constantly "`#text`", their node value is an arbitrary string. The name of an element node E can be chosen arbitrarily, while its value is constrained to be the concatenated value of its text node descendants, if any. Additionally, every element node is also associated to a set of attribute-value pairs.

Figure 1 shows a small DOM tree that result from parsing the following XML document:

```
<BODY>
 <IMG SRC="filename.gif"/>
 <H1>Playing tonight:</H1>
 <P>
  Star Wars
  <HR/>
 </P>
</BODY>
```

For simplicity, only node names are shown, while node values as well as the attribute-value pairs are omitted.

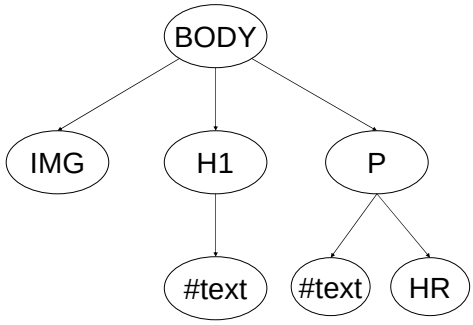

Figure 1 – a sample DOM tree

Now, an XPath statement (or path, synonymously) defines a traversal through a DOM tree. We will only provide a quick overview, while we refer to [W3C, 1999] for the details. A path is a sequence of steps, separated by slashes. Each step consists of three parts: A search direction (also called "axis"), followed by "::", a node filter and a (possibly empty) sequence of predicates.

The axis describes the direction of document traversal, e.g. to the parent, to the child, to following or preceding neighbours. Node filters restrict the set of applicable nodes along the search direction. Finally, predicates are XPath expressions in square brackets that are used to restrict the set of results further. Examples for XPath statements are:

1. /descendant-or-self::H1
2. /child::BODY/child::H1
3. /child::BODY/child::H1[following-sibling::P[child::HR]]
4. /descendant-or-self::H1[preceding-sibling::IMG [@src='filename.gif']]/child::text()

Each step is evaluated on a set of DOM nodes and yields a set of DOM nodes as its result. A path is then evaluated by evaluating the first step on the initial set of nodes and using each step's result as the input for the following step.

For example, statement 1 evaluates to a set of nodes containing all H1 elements in the document. When applied to the sample DOM tree from Figure 1, statements 1, 2 and 3 result in the set consisting of the single H1 ele-

ment. Statement 4 yields the text node inside the same H1 element.

In general, every step in an XPath statement is evaluated with respect to a set of input nodes as follows:

1. Let Output be an empty set of DOM nodes.
2. For each node N in the input node set:
   - Let $Result_N$ be the sequence of nodes that are on the current search direction relative to N
   - remove from $Result_N$ all elements that don't match the current node filter
   - remove from $Result_N$ every element for which any predicate is not satisfied.
   - add $Result_N$ to Output.
3. return Output

For the understanding of the constructs used in the following, it is necessary to mention the following details:
Node filters match elements whose tag name corresponds to the value of the node filter. Special node filters are the "`*`", which matches all element nodes, but no text nodes; "`text()`", which matches all text nodes, but no element nodes, and finally "`node()`", which matches both.
If the node filter is a tag name, the axis and the double colon may be dropped. In that case, the axis of the step is assumed to be "`child`".

Expressions inside predicates are evaluated in a boolean context. Particularly, path expressions are true for non-empty result sets, arithmetic expressions and functions are true if they have a nonzero value. A number constant C is a shortcut to select the C'th node in $Result_N$, e.g. the predicate "`[2]`" would filter out all but the second element of $Result_N$. Finally, expressions starting with "@" are interpreted as accesses to attributes.

4 Traversal graphs

In this section, we introduce traversal graphs as a formal representation of tree traversal patterns that can generally be expressed as XPath statements.

4.1 Linear traversal graphs in ANDES

[Myllymaki & Jackson, 2002] proposes traversal sequences to represent extraction rules. A traversal sequence is a linear directed graph whose nodes are called "anchors", connected by "hops". Hops that specify the descendant axis as their search direction are called "search", others are "path"s. On every anchor, one condition is evaluated and all nodes that don't comply are removed from the list of targets.

The ANDES authors propose three different types of anchors: Depending on whether the anchor imposes a constraint on text content, on attribute content or on the existence of related elements with a certain node name, it is called a "content", "attribute" or "structure" anchor.

Such traversal sequences can be expressed as XPath statements. This is done by converting each hop into one XPath step, appending the condition of the following anchor as a predicate and concatenating all steps by delimiting slashes.

4.2 Extending traversal graphs

The machine learning algorithms presented here will use a similar, but more general pattern representation. First, we allow an anchor to impose multiple constraints on the previous hop's result set, or equivalently, we allow multiple predicates to be added to one XPath search step. Next, we extended the "search"-hops by also allowing them to search the "following-sibling" or "preceding-sibling" axis. Last, we extended the traversal graph to be able to express branched extraction patterns.

The last extension probably needs some explanation. The XPath representation of a branched extraction pattern can only be expressed using a sub-path inside a predicate to filter nodes depending on the existence of the branch. A document traversal along the branch, however, would lead away from the destination node. Figure 1, for example, shows the traversal graph for the XPath statement `/BODY/P[preceding-sibling::H1]/child::text()[count(preceding-sibling::node())=0]`.
The target anchor is the `#text`-node at the bottom. The tree path to it leads from the root node over `/BODY/P` to the text node, but it has two additional structural constraints: First, the node traversed during the second step must have a preceding H1 element. Second, the `#text` node reached in the third step must *not* have any preceding siblings. Both constraints require a subordinate path to be evaluated. In such a case, the path can simply be written as a predicate behind the step, as shown in the XPath representation. When the XPath interpreter evaluates the second step, it will recursively evaluate the sub-path "`preceding-sibling::H1`" relative to each node of the intermediate result set. A sub-path in an XPath predicate thus corresponds to a branch in the traversal graph. This makes the traversal graph a tree, but still, the existence of a sequential "stem" can be supposed that spans all anchors and hops between an arbitrarily chosen root and the target node. However, throughout this paper, we will always select an ancestor of the target node as the root.

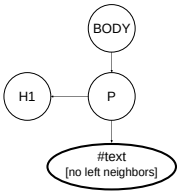

Figure 1:
An extended traversal graph with a target, one branch and one anchor condition.

5 Wrapper induction

In the next section, we will introduce the setting in which the wrapper induction algorithms work, as well as the algorithms themselves.

5.1 The wrapper induction scenario

Documents are seen as collections of DOM nodes connected through parent-child and following-preceding-sibling relations. A set of nodes, chosen by the user, is considered the positive training data. Another set of nodes, disjoint to the positive training data, is considered the negative training data. All documents that contain any positive or negative training example are called the training documents, or example documents. Another set of documents that contain neither positive nor negative training examples is referred to as the set of test documents. Positive examples are provided by the user in an interaction sequence: After annotating one or more DOM nodes as positive training examples, the user may run the algorithm on the set of training documents and/or test documents. Depending on the result, she may choose to annotate another example from the training or test documents or accept the generated pattern.

The learning task

The learning task is formulated as a search problem for a generalized tree traversal pattern that, when evaluated on all training documents, will have a result set that contains all positive examples and no negative ones.

The hypothesis language

We introduce augmented traversal graphs as a representation for the extraction rules generated by the learning algorithms. An augmented traversal graph, or ATG for short, is an acyclic, undirected graph. To avoid confusion with nodes in DOM trees, we call its nodes "anchors" and its edges "hops".

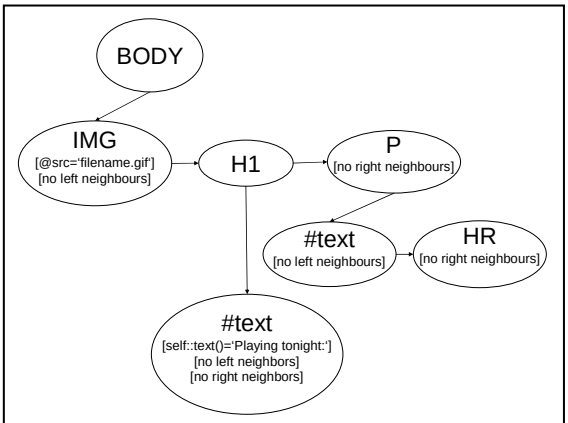

Figure 2: sequential node test for the DOM tree from Figure 1

Definition 1: augmented traversal graph

An augmented traversal graph ATG is a tuple (A, H, t, D), where A is a set of anchors, H is a set of hops, t is one element of A, D is a partial mapping from A×A → X, and N is a partial mapping from A×A → IN. An anchor is a tuple (NF, P), where NF can take any value the name of a DOM node can take and P is a vector of XPath predicates. A hop is a 2-elemented set (F, T) where F and T are elements from A. t is called the "target" of ATG. D is called the traversal direction function, where X is an XPath axis. For every $h=(A_{from},A_{to}) \in H$, both $D(A_{from},A_{to})$ and $D(A_{to},A_{from})$ exist and are opposed to each other. N is called the max-skip function and its domain equals that of the traversal direction function. Its default value is 0. A sequence $A_0, H_1, A_1, H_2, \ldots A_{n-1},H_n$ with $A_i \neq A_j$ for all $i \neq j$, $(A_i,A_{i+1}) \in H$ for all $0 \leq i < n$ and $H_i \in A$ for all $0 < i < n$ is called a path of ATG. For every pair of anchors of A, there is exactly one path to every other anchor of A. A hop in the context of a path is said to be an X-hop iff $D(A_{i-1},A_i)=X$. Additionally, it is said to be an immediate X-hop iff $N(A_{i-1},A_i)=0$.

Conversion to XPath

Every augmented traversal graph can be transformed into an XPath statement and be applied to a DOM tree. Thus, T is said to extract a node N from a DOM tree D iff an XPath generated from T and evaluated on D has a result set that contains N. The equivalent XPath predicate for an anchor A1 is "`[self::`", followed by the value of A1's NF, followed by all elements A1's P in square brackets. If N(A1,A2)=0 and D(A1,A2)="`descendant`", the XPath equivalent step for a pair of anchors A1, A2 is "`child::`" followed by A2's NF, followed by A2's predicates, otherwise it is the string D(A1,A2) "`::node()`", followed by "`[1]`" if N(A1,A2) = 0, followed by the equivalent XPath predicate for A2.

The complete procedure for converting an ATG into an XPath works as follows: Choose any anchor as the starting anchor. Let $TP=(A_0,H_1,A_1,\ldots H_N,A_N)$ be the path between the starting anchor and the target. Create an empty XPath X and append "`/descendant-or-self::node()`" as well as the equivalent XPath predicate for A0. Then, for each $0<i<N$, append to X a slash and equivalent XPath step for A_{i-1},A_i. If there any hops connect A_i to any nodes that are not part of TP, do for each maximal tree BR that does not contain A_i: Let A_{br} be a node of BR adjacent to A_i and choose a new target node t_{br}. Append to X a predicate containing the XPath equivalent step of (A_i, A_{br}) followed by all but the first step of this function evaluated recursively on $(A,H\backslash\{A_i,A_{br}\},t_{br},D)$ with A_{br} as the new starting node.

5.2 Deriving STTs

For every root of a DOM tree, one can construct an augmented traversal graph that extracts exactly the root node. We call such an extended traversal graph a "sequential tree test", or STT for short. Constructing an STT is done as follows: Start at the root of the subtree in question. In the example from Figure 1, this would be the "BODY"-element. Construct an anchor A_0. Let its N be the name of the current node, "BODY" in our example. If the node is an element node, add one predicate to **P** for each attribute. Then, if the current node is a leaf, create a predicate that defines the leaf node content: For elements, add to **P** the expression "`[count(child::node())=0]`", and for text nodes, add "`[self::text()='value']`", where *value* is the current node's node value. Otherwise, do for all children C_i of the current node: Construct a new anchor A_i, let its NF be the name of the C_i and execute this procedure recursively for C_i. Remove the target node from the set of anchors and replace all references to t in H, D, and N for A_i. If i=1, insert into H and D a "descendant"-hop from A_{i-1} to A_i, otherwise insert a "following-sibling" hop. When the first or last child nodes are processed, add to the current anchor the predicate "`[count(preceding-sibling::node()=0)]`" or "`[count(following-sibling::node())=0]`", respectively.

As an example, Figure 2 shows the resulting augmented traversal graph of applying this procedure to the DOM tree from Figure 1. The equivalent XPath would be:

```
/body[child::node()[1]
[self::IMG[@src='filename.gif']
/following-sibling::node()[1]
[self::H1[child::node()[1]
[self::text()='Playing tonight:']
[count(following-sibling::node())
=0]]/following-sibling::node()[1]
[self::P[child::node()[1]
[self::text()='Star Wars']
/following-sibling::node()[1]
[self::HR][count(following-sibling::node()=0]]]
```

5.3 Creating CCPs from the flipped tree

Even though STTs can readily be used to identify nodes within documents by means of XPath statements, their expressive power does not exceed that of a sequential token-matching algorithm, because DOM trees are still traversed strictly in document order. In order to produce traversal graphs that reflect and exploit the tree structure of a document, we aim for traversal graphs whose paths between root and target node consist exclusively of child- and descendant-hops. We call such a traversal graph a "constrained child path":

Definition 2: constrained child path

A constrained child path is a tuple (A, H, t, D, r), where A, H, t, and D are defined as in Definition 1. r is called the "root" and is an element of A. The path from r to t is called the stem. The stem consists solely of "descendant"-hops. Every maximal tree of anchors and hops that does not contain any stem anchors is called a branch and every stem node to which at least one branch is connected, is called a forking anchor. Every path that has no stem nodes except for a forking anchor starts with a series of one or more "preceding-sibling"-hops or with a series of "following-sibling"-hops. Eventually, these hops may be followed by a series of "descendant"-hops.

Constructing a CCP can be done by changing the algorithm slightly: Informally speaking, we create the stem whilst traversing the DOM tree from the target node up to the root and connect one branch for the left and right neighbours on each level. During the traversal, a cursor is used to keep track of the current node. Thus, we define a cursor to be a reference to a DOM node.
This is the exact algorithm:

1. Let cursor C be the target node.
2. Let A be an anchor.
3. If C is an element node, create a STT for C and replace its target node for A. Otherwise, insert a predicate into A that describes C's content.
4. Let A's NF be the name of C. If C is an element node, insert one predicate into A for each attribute of C.
5. Set C_L = C and A_L=A.
6. Repeatedly, let C_L be the next left neighbour of C_L and do:
    a. create an anchor A_{Lnew} and connect it to A_L by a "preceding-sibling"-hop.
    b. repeat steps 3 and 4 for C_L and A_L.
    c. set $A_L = A_{Lnew}$
7. Insert into A_L the predicate "`[count(preceding-sibling::node() = 0]`".
8. Process the left neighbours analogously.
9. If C is the document root, return P
10. Let C be the parent node of C
11. Let A_{new} be an anchor. Connect A to A_{new} with a "parent"-hop.
12. go to (4)

Applying this procedure to the text child of the H1 element in the DOM tree of Figure 1 would yield the XPath:

```
/body
/child::node()[self::H1]
[preceding-sibling::node()
[self::IMG][@src='filename.gif']
[count(preceding-sibling::node())=0]]
[following-sibling::node()[self::P]
[child::node()[1][self::text()='Star Wars']
/following-sibling::node()[1] [self::HR]]
[count(following-sibling::node())=0]]
/child::node() [self::text()='Playing tonight:']
[count(preceding-sibling::node())=0]
[count(following-sibling::node())=0]
```

Note that, although this expression is equivalent to the sequential tree pattern from the previous section when evaluated under a Boolean context, their structure is quite different. Figure 2 shows the order of document traversal of the STT compared to the CCP. The important difference between these variants is the connection from a node to its child: In the sequential tree test, a parent is always connected its first child. The constrained child path, on the other hand, defines a straight traversal from the root node to the target and imposes conditions to the left and right neighbour chains along its stem.

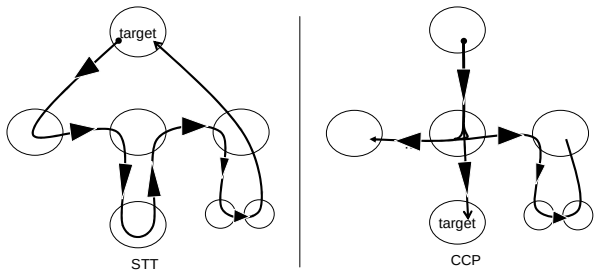

Figure 2: Structure of a sequential tree test (STT) and a constrained child-path (CCP). The CCP has a stem that leads from the root to the target node. Branches check the document structure to the left and to the right of the stem. The STT, on the other hand, just traverses the DOM tree in document order. Branches of the CCP are sequential tree tests.

5.4 Generalizing constrained child-paths on multiple target nodes

In this section, we will introduce the first of two extraction algorithms as a modification to the procedure described above that we call "F-GCP", which stands for Full Generalized Child-path Pattern. F-GCP is essentially identical to the algorithm that derives a CCP, but works on a set of target nodes instead on a single one. It creates a path that will match all at least all target nodes. Generalization is done by ignoring content and attributes that differ among the cursor nodes, as well as by skipping non-matching nodes and adapting the traversal pattern accordingly. These are the modifications compared to the algorithm for finding a CCP:

First, the cursor C is enhanced to contain a set of nodes instead of merely a single one. Thus, we define the generalized cursor C as a set of DOM nodes with equal name.

Second, a heuristic is introduced to find a generalized description of the common structures under a cursor position to replace step 3. Specifically, instead of creating an STT in step 3, all possible child-paths (i.e. paths that consist of a series of immediate descendant-hops) that match at least one DOM node from every element of C are found and appended to A.

Third, step 4 is modified to insert only predicates for attributes that are satisfied for every element of C.

Fourth, the cursor movements of step 6 and 10 are replaced for a search algorithm that yields the nearest matching cursor position in a given search direction. The operation takes a cursor C and yields another cursor C_{new} that may be empty if no matching neighbours were found. This is the search algorithm:

1. Start with a cursor C and an axis a. Let n be 1.
2. terminate if no element of C has an n'th neighbour along axis a.
3. for each element e of C, do:
4. find the n'th element along the axis a
5. If that neighbour can be matched (by name for elements and by value for text nodes) to any neighbour of all other elements in C with a distance of at most n, return that set of neighbours as the new cursor.
6. increase n by 1 and go to 2.

Fifth, during the steps 6a and 11, we additionally set the value of the max-skip function for $N(A,A_{new})$ or $N(A_L,A_{Lnew})$ to n-1, where n is the last value assigned to n during the search procedure above.

Last, step 7 is skipped if the search failed with n>1.

This algorithm yields a heuristic approximation for the least general generalization of a set of completely constrained paths. Intuitively speaking, it tries to match the trees around the positive training examples to each other and computes an intersection of all those subtrees, augmented with information about gaps between the hops that are reflected by the values of the max-skip function and by predicates indicating the absence of structure in a direction.

When the F-GCP is converted into an XPath statement, this information is reflected by three kinds of expressions:

1. The first predicate on a "preceding-sibling"- or "following-sibling"-step is "`[1]`" (i.e. "consider only the first resulting element"), if there is no gap between the two adjacent anchors.
2. Vertical path steps on the stem of the child-path pattern use the "child"-axis instead of the "descendant"-axis if the max-skip function of the corresponding hop is zero.
3. The last predicate on a "preceding-sibling" or on a "following-sibling"-step is "`[count(preceding-sibling::node() = 0]`" or "`[count(following-sibling::node() = 0]`", if the search for the corresponding cursor position failed with n=1.

Limitations

Since every additional example in the positive training set may shrink the generalized pattern, the number of available conditions decreases as the number of training examples increases. If the positive examples are too diverse, over-simplification will occur. This tendency to over-simplify is a severe limitation of the learning algorithm: If a *negative* example matches all conditions of the pattern, the concept can no longer be learned.

In practice, however, we found the extraction rules generated by this software to be highly precise. In fact, when applying the system to automatically generated pages, we encountered really large generalized child-path patterns. More than 50 conditions were not unusual, so, in the one-per-document experiments, we did not encounter any problems with over-simplification. The lack of robustness was a greater problem: Frequently, conditions in extraction rules were only found because the training examples are not drawn independently. For example, some rules contained a test for the following condition:

```
/HTML/BODY[preceding-sibling::HEAD
[child::TITLE="Meldung vom 27.08.2004"]]
```

Obviously, this condition will produce a wrapper that fails after at most one day. Many other examples like this could be found. This problem gets even worse if beside the contents, also the structure of an HTML template changes slightly over time.

Robustness

For generating robust wrappers, it is thus necessary to keep the number of features used in the rule as small as possible. However, the F-GCP learner selects all conditions it possibly can, from which only a small fraction contributes to the precision of the resulting extraction rule.

According to [Myllymaki & Jackson, 2002], robustness can be achieved by "relying less on page structure and more on content." In other words, by preferring content and attribute search patterns over structure search patterns. We think that in addition to this, the total number of hops should be minimized because every additional hop increases the risk of failure. Anyhow, minimizing the number of anchors is equivalent to the proposal of [Myllymaki & Jackson, 2002], because every additional hop essentially is a constraint on page structure.

5.5 Incorporating negative training data

In order to create more robust wrappers, we will now introduce the second algorithm "M-GCP", which is based on the F-GCP algorithm, but it additionally uses negative examples to minimize the number of conditions and hops in the generalized child-path pattern. Therefore, we considered all nodes of all training documents as negative examples unless they were labelled positive.

For explaining M-GCP, let us recall that F-GCP incorporates all conditions into the augmented traversal pattern that match all positive training examples. This can be thought of as "growing" the tree from the target anchor. M-GCP, in contrast, checks whether a particular condition has any discriminative power before adding it to the pattern. This is done by calculating the false positive rate of the traversal graph before and after adding another predicate. If the false positive rate is not influenced, the predicate is rejected.

Provided that the GCP is large enough to separate all positive from all negative examples, this feature selection strategy will reject every new candidate condition once a consistent set of conditions has been found. But still, candidate conditions added later might imply the truth value of any candidate condition added earlier. For that reason, irrelevant conditions are pruned away in a second pass.

5.6 Discussion of M-GCP's selection strategy

The order of creation of hops and anchors, as well as the matching strategy for cursor elements defines the search bias of M-GCP towards certain hypotheses. Since M-GCP traverses the documents the same way as F-GCP does, conditions whose containing anchor's path towards the target anchor contains fewer descendant-hops are always preferred over those with more. Among those conditions with an equal number of descendant-hops, conditions preceding the stem are preferred to those following it. Among each of those, anchors that are nearer, i.e. whose path to the stem has fewer hops in total, are preferred. Among those, again, tests for text content is preferred over tests for attribute content, and those are preferred over tests for attribute existence. But every condition that is not completely irrelevant is picked up in the rule. This is a major difference to most other rule learning systems that usually maximize the discriminative power of conditions. M-GCP, however, focuses on another quality of the available conditions, which is their document coverage. Since robustness is strongly correlated to small document coverage and the robustness considerations from [Myllymaki & Jackson, 2002] were also respected by this selection strategy, we think that this system is ideally biased for creating robust extraction rules in the XPath language.

6 Experiments and Results

To evaluate robustness and expressiveness, we tested both algorithms on automatically generated web pages from two content providers. To retrieve the content pages, we configured a web crawler for each web site. For the extraction task, we used only pages with exactly one record per page.

The tests were conducted incrementally, by repeatedly running the algorithm and labelling one example document on which the extraction rule failed, until a consistent extraction rule had been learnt, but we waived to run the algorithms on a single example page. The number of manually labelled example pages is shown in Table 1.

We will now examine the initial extraction rules that were chosen by M-GCP for the "author"-field on www.heise.de. The author-field regularly follows every news message on www.heise.de. Depending on the author's name, affiliation and email address, the corresponding HTML code could for example be `(<a href='mailto:anw@ct.heise.de'>anw</a>/c't)` or `(<a href='mailto:jk@ct.heise.de'>jk</a>/c't)`. The user can recognize the author-field as a pair of round braces at the end of every article that contain the author's initials inside a link to his email address, followed by a slash and the name of the magazine. We were interested in the link itself, say, in the `<A>`-tag surrounding the author's initials. The following table shows the extraction rules generated by M-GCP after the second, third and fourth example page had been labelled:

2	2%	//A[child::text() = "ad"]
3	99%	//A[following-sibling::node()[1] [self::text()][self::node()="/c't)"]]
4	100%	//A[following-sibling::node()[1][self::text()] /following-sibling::node()[1] [self::BR][@CLEAR="all"]]

This sequence shows how M-GCP generalizes: The first and the second example came from the same author, so that the author's initials were identical. For that reason, the set of conditions common to the target anchors provided enough discriminative power to generate an extraction rule from text inside the link ("find an `<A>`-tag with contents "`ad`"). The GUI we use for labelling assists the user in discovering that this expression is not general enough by presenting him the first page for which no extraction result was found. So, the next example scheduled for labelling did not come from the same author. This made the common text "ad" disappear from the generalized traversal pattern, forcing the algorithm to find another set of conditions to be consistent with the training data. It chose a test for the following text node, so the rule reads: "find an `<A>`-tag that is followed by the text "`/c't)`". Since all but two pages in the training set come from the magazine c't, the following text node is identical in nearly all example pages, so the extraction rule already achieves 99% accuracy. However, the training set also contained two news articles from another magazine, on which that extraction rule did not work. So, the GUI presented one of the remaining two messages for manual labelling. The final generalization step done by the system was to skip over the text node after the target anchor and to use the immediately following `<BR clear=all>`-tag in the final rule. Indeed, this hypothesis is consistent with heise's page layout and did work 100% correct. Whether that extraction rule is more satisfying than the previous version is questionable, though.

Next, we will have a look at the hypotheses that M-GCP found for the "parken"-field on www.burgenwelt.de. The content pages on www.burgenwelt.de show information on castles of Europe. For each castle, the authors are supposed to contribute a piece of text for each of the categories history, gastronomy, parking lots, accommodation, guided tours, and entry prices. We let F-GCP and M-GCP generate extraction rules for each of them. Certainly, F-GCP frequently required more examples for the extraction task than M-GCP. In fact, extraction rules for name, accommodation and entry prices were discovered by providing only two examples, we investigated the hardest case, which is the "parken"-field. The intermediate and the final extraction rules are shown below:

# ex's	Accuracy	hypothesized extraction rule
2	99,8%	//TD[preceding-sibling::node()[1][self::TD]/CENTER/IMG[@ALT = "Parkmöglichkeiten"]]
3	99,9%	//TD[preceding-sibling::node()[1][self::TD]/CENTER/IMG[@SRC = "../grafiken/314.gif"]]
4	79,2%	//TR[count(preceding-sibling::node()) = 0] /TD[@WIDTH = "560"][@BACKGROUND = "../grafiken/bglaufg2.jpg"]
5	79,3%	//TR[count(preceding-sibling::node()) = 0] /TD[preceding-sibling::node()[position() = 1][self::TD] [attribute::BACKGROUND = "../grafiken/bglaufg2.jpg"][attribute::WIDTH = "40"]]
6	57,8%	//TR[count(preceding-sibling::node()) = 0] /TD[preceding-sibling::node()[position() = 1][self::TD][attribute::VALIGN]]
7	94,6%	//TR[count(preceding-sibling::node()) = 0] /TD[preceding-sibling::node()[position() = 1][self::TD] [count(preceding-sibling::node()) = 0][attribute::BACKGROUND]]
8	100%	//TR[count(preceding-sibling::node()) = 0] /TD[preceding-sibling::node()[position() = 1][self::TD] [count(preceding-sibling::node()) = 0]][count(following-sibling::node()) = 0]

During this test, we discovered that the pages from www.burgenwelt.de are not automatically generated, but hand-crafted with an HTML editor using a template page. Even though their visual appearance is uniform, their mark-up differed in only two of the 976 instances that were part of the test. None of the two examples labelled initially were irregular, so consequently, the first extraction rule induced by M-GCP did cover all but the two irregular examples. The goal, however, was to achieve 100% accuracy, so we labelled the irregular pages, too. It

web site	www.heise.de			www.burgenwelt.de				
total # of pages	191			976				
field name	Datum	Headline	Autor	Name	Eintritt	Parken	Gastronomie	Geschichte
ex's required by F-GCP	2	5	4	7	9	9	11	12
ex's required by M-GCP	2	2	2	2	2	8	6	7

Table 1: overview over the experiments

worked, but it resulted in an XPath that spans a much larger portion of the document. As discussed above, this is believed to result in low robustness, and indeed, we observed the accuracy dropping to 57.8% in the following rounds. This trend could only be stopped by replacing an attribute anchor for the meta-node ("`count(preceding-sibling::node()) = 0`") in the seventh step, and in the eighth step, 100% accuracy could only be achieved by adding such a constraint on the following-sibling axis of the same anchor, too. Again, the result is a 100% score in the end, but – compared to the result achieved with two training examples - at the cost of labelling six additional pages, and of getting a much larger extraction pattern.

7 Lessons learnt

From these experiments, we conclude that M-GCP can successfully be used to create precise extraction rules that act on the level of DOM nodes. For wrapping similar pages that are automatically generated from the same source or at least generated by hand from the same template page, disjunctive expressions (as for example induced by the STALKER algorithm) are usually expendable. However, to find conjunctive expressions that can be used for continuous information extraction, the usefulness of generalization steps that are necessary to cover more examples must be assessed individually. We have encountered two non-trivial cases where robustness is being traded off for generality: When few examples are used, the resulting wrapper may not be general enough. On the other hand, when more examples are added, the expression tends to span a larger portion of the document, and thus loses robustness with respect to changes in page layout.

Common structures around the training examples are often large enough to derive a good rule from it. Sometimes, the use of multiple conditions and the introduction of structural constraints of the type "`count(SOME_AXIS::node()) = 0`" is necessary. Thus, the sequential traversal graph as pointed out by the ANDES project is not expressive enough as a hypothesis space for a machine learning algorithm. One can argue that most trees could be converted into a sequential traversal that would still be consistent with the training data, but the concept of having multiple conditions match document parts that are unrelated to each other and that possibly reside in different relative positions to the target anchor, is better represented by a tree than by a sequence.

References

[Ambite, et al., 1998] Ambite, J.-L.; Ashish, N.; Barish, G.; Knoblock, C. A.; Minton, S.; Modi, P. J.; Muslea, I.; Philpot, A.; and Tejada, S. Ariadne: A system for constructing mediators for internet sources. *Proc. ACM SIGMOD International Conference on Management of Data*, ACM Press, 1998.

[Baumgartner, et al., 2001] Robert Baumgartner, Sergio Flesca, Georg Gottlob. Visual Web Information Extraction with Lixto. *Proc. 27th International Conference on Very Large Data Bases*: 119 – 128, 2001

[Berners-Lee, et al., 2001] T. Berners-Lee, J. Hendler, O.Lassila, *The Semantic Web*, Scientific American, May 2001

[Crescenzi, et al., 2001] Valter Crescenzi, Giansalvatore Mecca, Paolo Merialdo. RoadRunner: Towards Automatic Data Extraction from Large Web Sites. *Proc. 27th Intl. Conf. on Very Large Data Bases*, 2001.

[Finn & Kushmerick, 2004] Aidan Finn and Nicholas Kushmerick. Information Extraction by Convergent Boundary Classification. *Proc. Workshop Adaptive Text Extraction and Mining*, AAAI 2004.

[Freitag & Kushmerick, 2000] Dayne Freitag and Nicholas Kushmerick. Boosted wrapper induction. *Proc. American Nat. Conf. Artificial Intelligence*, AAAI 2000.

[Han, et al., 2001] Wei Han and David Buttler and Calton Pu. Wrapping web data into XML. *Special section on advanced XML data processing*, 30(2): 33-38, ACM Press, 2001

[Hsu & Dung, 1998] Chun-Nan Hsu and Ming-Tzung Dung. Generating Finite-State Transducers for Semi-Structured Data Extraction from the Web. *Information Systems*, 23(8): 521-538, 1998.

[Kushmerick et al., 1998] Nicholas Kushmerick. Wrapper Induction for information extraction. *Proc. 15th Intl. Conference on Artifical Intelligence*, ICAI 1998.

[Kushmerick, 2000] Nicholas Kushmerick. Wrapper Induction: Efficiency and Expressiveness. *Artificial Intelligence,* 118(1-2):15-68, 2000

[Muslea, et al., 2000] Ian Muslea, Steve Minton, Craig Knoblock: Stalker: Learning extraction rules for semistructured, web-based information sources, *Proc. AAAI-98: Workshop on AI and Information Integration*. AAAI Press, 1998

[Myllymaki & Jackson, 2002] J. Myllymaki, J. Jackson. Robust Web Data Extraction with XML Path Expressions. *IBM Research Report*, May 2002.

[Sahuguet & Azavant, 1999] A. Sahuguet, F. Azavant. Building light-weight wrappers for legacy Web data-sources using W4F. *Proc. 25th Intl. Conf. on Very Large Data Bases*, ACM Press, 1999.

[Scheffer et al. 2001] Tobias Scheffer, Christian Decomain, Stefan Wrobel. Active Hidden Markov Models for Information Extraction. *Proc. Intl. Symposium on Intelligent Data Analysis*, IDA 2001.

[W3C, 1999] World Wide Web Consortium. XML Path Language (XPath) Recommendation. http://www.w3c.org/TR/xpath/, 1999.