Learning (k,l)-Contextual Tree Languages for Information Extraction

Stefan Raeymaekers[1], Maurice Bruynooghe[1], and Jan Van den Bussche[2]

[1] K.U.Leuven, Dept. of Computer Science, Celestijnenlaan 200A, B-3001 Leuven
{stefanr, maurice}@cs.kuleuven.ac.be
[2] Universiteit Hasselt, Dept. Theoretical Computer Science,
Agoralaan D, B-3590 Diepenbeek
jan.vandenbussche@uhasselt.be

Abstract. This paper introduces a novel method for learning a wrapper for extraction of text nodes from web pages based upon (k, l)-contextual tree languages. It also introduces a method to learn good values of k and l based on a few positive and negative examples. Finally, it describes how the algorithm can be integrated in a tool for information extraction.

1 Introduction

The World Wide Web is an indispensable source of information. Extracting its content for further processing however, is difficult because it is formatted in HTML, which is primarily focussed on presentation. A wrapper is a general name for a procedure that extracts data (often from machine generated HTML-pages) based on the structure of the documents, commonly without the use of linguistic knowledge. Various tools are designed to facilitate wrapper building, but the process remains tedious. Hence the efforts [5, 12, 13, 14, 17, 21] to create algorithms that learn wrappers from examples.

Several approaches [6, 7, 17] process documents in a string representation. Flattening the tree structure of the document to a string representation though can project sibling nodes arbitrarily far from one another, and increases the complexity of the wrapper to express relations between these nodes. In [13], documents are represented as (ranked) binary trees; this improves locality and gives better results. An unranked tree representation is for the first time used in [12]. Combined with a number of ad-hoc design decisions, it leads to superior results.

Note that most string-based approaches can extract a substring of a text node, while most tree-based approaches aim to extract tree nodes (either a whole text node or a subtree of the document). If a task needs sub-node extraction though, it is very natural to learn first a wrapper to retrieve the containing text node, and focus then on learning a (string based) wrapper that extracts the required information from this text.

The contributions of this paper are:

- The introduction of the notion of a (k,l)-contextual tree language for unranked trees and an algorithm to infer such a language from positive examples (trees) only. A major virtue is that this algorithm needs very few examples to learn. This algorithm is then applied on marked trees to induce wrappers. We obtain better results than [12] while avoiding its ad-hoc design decisions. All this is described in Section 2.

- Whereas [12] needed cross validation to learn the parameters (i.e., a fully annotated data set), we introduce a method to learn the parameters with only a few negative examples (Section 3).
- In Section 4, we integrate our results into an interactive system that guides the user in building a wrapper by posing equivalence queries. For example, if a user wants to extract book prices from www.amazon.com, he clicks on an example page (in the browser of the GUI-front end) on one or more prices. The algorithm then learns a wrapper from these (positive only) examples and highlights all elements that are extracted by this wrapper. When the current hypothesis is erroneous, the user can either click on a highlighted item to indicate it as a false positive or click on an item that is not yet highlighted to indicate that it is a false negative. The application then adjusts the wrapper. This interaction continues (possibly with other example pages), until the user is satisfied.

In Section 5 we round up with a discussion and a summary.

2 Induction from Positive Examples Only

In the language learning approach to information extraction setting, it is important that we can start learning from positive examples only, because that is typically all we have to begin with. Only after the learner has inferred a hypothesis, false positives give us sensible negative examples, which we can then exploit to refine the hypothesis. Unfortunately, the whole class of regular languages cannot be learned from positive examples only [10] . Intuitively the reason is that there is no boundary to end the generalization, and therefore the resulting language will accept everything. A common solution for this negative result is to define a learnable subclass of the regular languages. Examples of learnable subclasses of string languages are k-reversible languages [2], k-contextual languages [16] and k-testable languages [9]. The latter two are often referred to as k-local languages as they are equivalent [1]. Similar developments occurred for tree languages. Algorithms for induction of string automata have been upgraded for tree automata. Several works exist for ranked trees, e.g., [8, 11] (k-testable tree languages) and [20] (probabilistic extensions). In ranked trees, the number of children of a node is fixed in advance (determined by its label). HTML or XML documents are clearly *not* ranked; hence an awkward encoding is needed in order to apply k-testable tree language learning to Web information extraction [13].

Therefore, in this section, we introduce (k, l)-contextual tree languages, which are unranked, and therefore directly applicable. But first some background is introduced.

2.1 Preliminary Definitions

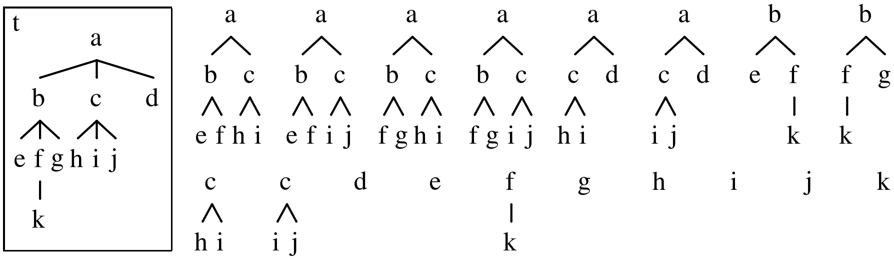
We define the alphabet Σ as a finite set of symbols. The set of all finite trees with nodes labeled by elements of Σ can be recursively defined as $T(\Sigma) = \{f(s) \mid f \in \Sigma, s \in T(\Sigma)^*\}$. We usually denote $f(\epsilon)$, where ϵ is the empty sequence, by f. A *tree language* is any subset of $T(\Sigma)$. The set of $(k$,$l)$-roots of a tree $t = f(t_1, \ldots, t_n)$ is the singleton $\{f\}$ if l=1; otherwise, it is the set of trees obtained by extending the root f with $(k, l-1)$-roots of k successive children of t (all children if $k > n$). Formally, we can define inductively[1]:

[1] f(S) denotes $\{f(s) \mid s \in S\}$.

$$R_{(k,l)}(f(t_1 \ldots t_n)) = \begin{cases} \{f\} & \text{if } l=1 \\ f(R_{(k,l-1)}(t_1) \ldots R_{(k,l-1)}(t_n)) & \text{if } l>1 \text{ and } k>n \\ \bigcup_{p=1}^{n-k+1} f(R_{(k,l-1)}(t_p) \ldots R_{(k,l-1)}(t_{p+k-1})) & \text{otherwise} \end{cases} .$$

Finally, a (k,l)-*fork* of a tree t is a (k,l)-root of any subtree of t. The set of (k,l)-forks of t is denoted by $F_{(k,l)}(t)$.

Example 1. Below we show graphically the $(2,3)$-forks of a tree t. The first 6 of these forks, are the $(2,3)$-roots of t.



2.2 (k,l)-Contextual Tree Languages

Let G be a set of trees over Σ, such that every tree in G has height at most l, and every node of each tree in G has at most k children. The (k,l)-*contextual tree language based on* G is defined as $L_{k,l}(G) = \{t \in T(\Sigma) \mid F_{(k,l)}(t) \subseteq G\}$; i.e., a tree is part of the language defined by a set of forks, if each of its (k,l)-forks is an element of that set. Obviously, $L_{k,l}(G_1) \subseteq L_{k,l}(G_2)$ iff $G_1 \subseteq G_2$. Let G be the set of forks of a given set of examples. The (k,l)-contextual languages that accept these examples are those based on a superset of G. $L_{k,l}(G)$ is the least (most specific) language accepting the examples.

Our inference algorithm avoids overgeneralisation by learning for a given k and l, the most specific (k,l)-contextual language that accepts all the examples. It does so by collecting all the (k,l)-forks of the examples. Checking for membership of a tree t in $L_{k,l}(G)$ is done by checking whether all (k,l)-forks of t are among the forks of G.

As is the case for local languages such as k-contextual languages [16, 1], k-testable languages [9], and k-testable tree languages [8, 11], (k,l)-contextual tree languages learned from a given training set are anti-monotone in the parameters; i.e., increasing either k or l decreases the set of trees accepted by the learned language.

Our definition generalizes to unranked trees the notion of k-testable string language "in the strict sense". If we had wanted to generalize the more expressive notion of k-testable, studied by McNaughton [15], we would have taken a set of sets of forks for G (one for each example), and would have then accepted a tree if its forks are a subset of those from one example. Our experiments (Section 2.5) indicate that k-testable languages in the strict sense are sufficiently expressive, hence we explore only the strict notion.

The local unranked tree automata of [12] correspond to the special case $l = 2$ in our approach. The lack of expressiveness in vertical direction was remedied with some extra preprocessing (see Section 2.4).

2.3 Wrapper Induction

We follow [12, 13] for defining wrappers. A wrapper is a language that accepts only trees that are correctly marked. Marking a node s consists of replacing it by a marked equivalent s_x. To decide whether to extract a node, the candidate node is marked. When the wrapper accepts the resulting tree, the original text of the marked node is extracted. The wrapper is learned from examples, as described above, where each example is a HTML page with one target node marked. However, simply collecting all forks from a few examples typically results in a too specific wrapper.

A first problem is that text nodes are from an (almost) infinite alphabet and cannot be learned from a small number of examples. To solve this, we follow [12, 13]: More generalization is obtained by replacing all text nodes by a special symbol (@)[2]. Sometimes this leads to overgeneralisation as a text node close to the target is needed to disambiguate between a positive and a negative example. A preprocessor finds such a distinguishing context and text nodes containing it are not replaced.

looseness 2 A second problem is that a small number of examples does not cover all the variance of possible forks in areas far away from the targets. One can argue that the forks containing the marker provide the local context needed to decide whether a node should be extracted or not, while the other forks describe the general structure of the document. The latter merely serve to decide whether the document is in the class of documents that contains relevant information. Learning that class typically requires substantially more examples than learning the local context. However, in our setting, we assume all documents are from the right class; hence there is no need to learn the document class and we can ignore all forks that do not contain the marker during learning and extraction.

Combining the preprocessing of text nodes with the filtering of forks, one obtains a lot more generalization and wrappers can be learned from a small set of examples.

2.4 Expressiveness

To compare the expressiveness of our languages with that of [12], we first explain the latter briefly. As already mentioned above, after preprocessing, each text node is either a marker (x), a distinguished context (c) or a generalized text node (@). The method basically infers a $(k, 2)$-contextual language. However, the tree representing an example is subject to two other preprocessing steps.

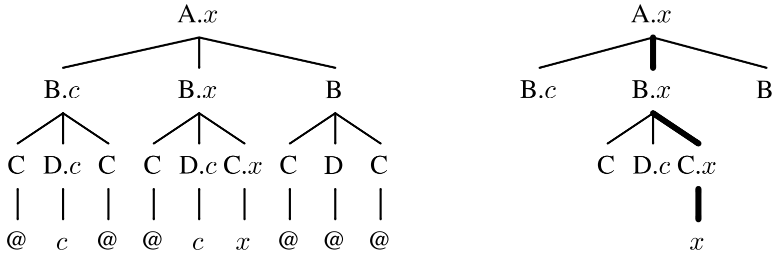
The first transformation replaces every node f into a node $f.x$, if its subtree contains the x-node. If the subtree does not contain the x-node but a c-node then it is replaced by $f.c$. Hence, limited information is passed infinitely upwards, making the method not purely local. However, the subclass remains inferable and the expressiveness is enhanced.

The second transformation in [12], although part of the inference algorithm, can also be explained as a preprocessing step. The automaton accepts everything below a node that is not of the form $f.x$, i.e., all subtrees below such nodes can be removed and only the path from the root to the x-node is left, together with the siblings of the nodes

[2] When the node is extracted, the original test is returned.

on that path; parts farther away from the marked node are ignored. This enhances the generalizing power of the resulting language (and reduces the expressiveness).

Example 2. The left tree below shows a tree after the first transformation, while the tree on the right shows the result of applying the second transformation to that same tree.



Thanks to the first transformation, the algorithm of [12] (K) can express some global vertical relations. While added to retain information in the vertical direction, it can also describe the relation between a node and an ancestor that is an arbitrary number of levels higher. Our algorithm (KL) is purely local and does not have this expressiveness. Our experiments showed that local information in the vertical direction (the l parameter) was sufficient for all data sets.

The second transformation in K makes it less expressive than KL as all information about the siblings of the target node is removed while KL retains the neighborhood. We encountered several data sets where that information was needed to disambiguate positive and negative examples[3].

2.5 Experiments

We evaluate our approach on the WIEN[4] data sets. We use the F1 score as a fitness criterium. Given E, the number of text nodes extracted from the test set, C, the number of correctly extracted text nodes, and T, the total number of text nodes to be extracted from the test set. Precision(P) is defined as P=C/E, recall(R) as R=C/T. The F1 score is defined as the harmonic mean: F1=2PR/(P+R).

Our algorithm as well as the K algorithm [12] are expressive enough to handle all tasks of the WIEN data sets, i.e., given enough examples, they reach a 100% F1 score. In those tasks where sub-node extraction is required, both algorithms return the text node containing the substring to be extracted. In comparison, in [17] it is stated that neither STALKER nor WIEN [14] are expressive enough to handle all tasks. Also STALKER with Aggressive Co-testing still fails on some tasks according to [18]. Note that on the tasks where the other algorithms did not reach the maximal score, this was not due to the fact that the sub-node extraction posed extra difficulties. [12] compares the K algorithm also with HMM [7] and BWI [6]. They report an experiment where the K algorithm reaches a 100% F1 score whereas the other ones have a significantly lower score on some (difficult) WIEN data sets (the number of examples was limited in this

[3] E.g., a table with bargains. The aim is to extract those with a picture of the item. The picture, when present, occupies the first cell of the row, (a sibling of the cell containing the target) .

[4] These are available at the RISE repository: http://www.isi.edu/info-agents/RISE/index.html.

Table 1. Results for data sets with 5 examples

Data set	ctx	K	KL	Data set	ctx	K	KL	Data set	ctx	K	KL
s1-1		89.1	**100.0**	s11-2	✔	100.0	100.0	s23-1		97.6	**100.0**
s1-3		90.4	**98.7**	s12-2		98.4	**98.5**	s23-3		94.4	**100.0**
s1-4		78.8	**100.0**	s13-2		100.0	100.0	s25-2		97.2	**100.0**
s3-2		97.6	**100.0**	s13-4		100.0	100.0	s29-1		96.6	96.6
s3-3		98.2	**100.0**	s14-3		99.5	**100.0**	s29-2		**100.0**	87.8
s4-1		91.6	**100.0**	s15-2		97.1	**100.0**	s30-2		96.0	**100.0**
s5-2		93.8	**98.9**	s19-4		100.0	100.0	bigbook-2		94.3	**100.0**
s8-2		100.0	100.0	s20-3	✔	98.5	**100.0**	bigbook-3		88.0	**100.0**
s8-3		100.0	100.0	s20-4	✔	97.5	**100.0**	okra-1	✔	100.0	100.0
s10-2		100.0	100.0	s20-5	✔	97.5	**100.0**	okra-2	✔	99.3	**100.0**
s10-4		100.0	100.0	s20-6	✔	98.5	**100.0**	okra-3	✔	99.1	**100.0**
s11-1	✔	100.0	100.0	s22-2		93.3	**100.0**	okra-4	✔	99.1	**100.0**

experiment). This is a strong indication that our method is more expressive than previous methods. Only the K algorithm has a comparable expressivity, however it contains a number of ad-hoc design decisions.

A second experiment compares these both algorithms in their ability to learn from a small set of positive examples. Each experiment randomly selects 5 examples (each one target in a document) in a data set and compares the F1 score of both algorithms (for optimal parameter setting) with the whole data set as test set. This experiment is not intended to measure the number of examples needed by each algorithm but to measure which one learns best from a given sample of (incomplete) data. We use a well-defined subset of 36 extraction tasks from the available WIEN data tasks, namely those that extract a complete text node and for which the information on the nodes to be extracted is available in the WIEN data. Tasks aiming at the extraction of a n-tuple are split in n extraction tasks. We refer to them with the name of the original data set and the index of the field in the tuple. Table 1 shows for each data set the mean over 5 experiments. The variance over the different experiments was low. In most cases when a mean does not reach 100%, all the experiments do not reach 100%. The column *ctx* indicates whether both algorithms used a (same) distinguishing context. One can observe that our KL algorithm gives a better F1 score for 24 tasks out of 36 and a worse one for only 1 data set. This is evidence that it learns better from a small set of positive examples.

3 Learning the Parameters

As shown in Section 2.5, our (k, l)-contextual tree language improves upon the local unranked tree automata of [12] by being able to learn from fewer examples. However, a problem shared with [12] is that the method needs **parameter tuning** for each task. Selecting the optimal parameters requires to run the program on a set of completely annotated documents to obtain precision and recall. Hence parameter selection is in fact based on a large set of positive and negative examples.

Here, we describe how to learn parameters based on a small set of negative examples. In addition, it is indicated when (k, l)-contextual tree languages are not expressive enough to reach a 100% F1-score for the extraction task at hand.

3.1 Algorithm

Order relations. We can distinguish two order relations on languages. Firstly, a partial order $\geq$ defined as $L_1 \geq L_2 \Leftrightarrow L_2 \subseteq L_1$ which is anti-monotonic in the parameters. Secondly, let S be a finite set of trees and $\#acc(S, L)$ the number of trees from S that is accepted by the language L (the *count*). Then we define the total order $\geq^{\#}_{S}$ as $L_1 \geq^{\#}_{S} L_2 \Leftrightarrow \#acc(S, L_1) \geq \#acc(S, L_2)$ [5]. Note that $\forall S \mid L_1 \geq L_2 \Rightarrow L_1 \geq^{\#}_{S} L_2$, hence $\geq^{\#}_{S}$ is also anti-monotonic in the parameters, i.e., the count decreases with increasing parameter values.

Solutions. A solution is a (k, l)-contextual language that is consistent with the examples. We define a solution L_1 to be better than L_2 when it extracts more solutions from the documents used to learn the wrapper; more formally, when $\#acc(S, L_1) \geq \#acc(S, L_2)$ where S has a tree for each candidate node (with the candidate marked cnfr. Section 2.3). Hence the best solution is the solution that is maximal in the order $\geq^{\#}_{S}$.

Heuristic. In what follows, we denote with $[k, l]$ the (k, l)-contextual language learned from the given examples. Due to the anti-monotonicity, we have that $\#acc(S, [k, l]) \leq \#acc(S, [k-1, l])$ and $\#acc(S, [k, l]) \leq \#acc(S, [k, l-1])$, hence $\#acc(S, [k-1, l])$ and $\#acc(S, [k-1, l])$ are upper bounds on the value of $\#acc(S, [k, l])$. The algorithm uses them to estimate the value of $\#acc(S, [k, l])$ and, at each step, computes the count of the language with the best estimate. The search stops when the best estimate cannot improve upon the best current solution.

Initialisation. All $(k, 1)$-contextual languages extract all single node forks from the examples, hence are overly general and of no interest. Therefore, the search starts from the $(1, 2)$-contextual language as it has the largest count.

Algorithm. To reduce the space requirements, our algorithm maintains for a given l-value the count of at most one (k, l)-contextual language. If the (k, l)-contextual language is a solution, then the $(k + 1, l)$-contextual language is of no interest as it has a lower count; if it is inconsistent, then its count is discarded as soon as the count of the $(k + 1, l)$-contextual language is computed. These counts are maintained in a *front* (of the search). For each l-value, the front maintains the k-value ($F.k[l]$), the count ($F.c[l]$) and whether it is a solution ($F.sol[l]$) (see the right of Figure1). In each step, the algorithm selects the minimal value l such that the language $[F.k[l], l])$ is most promising for exploration (the function BestRefinement): $[F.k[l], l]$ is not a solution and the estimation of its refinement has the highest bounds on its count. For $k > 1$, the refinement is the language $[F.k[l] + 1, l]$, however for $k = 1$, also $[1, l + 1]$ is a refinement.

Example 3. Given the data in Figure 1, the languages $[1, 5]$, $[4, 3]$ and $[2, 5]$ are candidates for refinement. Although $[4, 3]$ has the highest count, its refinement $[5, 3]$ has a count bounded by 33 while both refinements of $[1, 5]$ have a count bounded by 48, hence the latter is selected for refinement.

[5] A total order over equivalence classes with the same count.

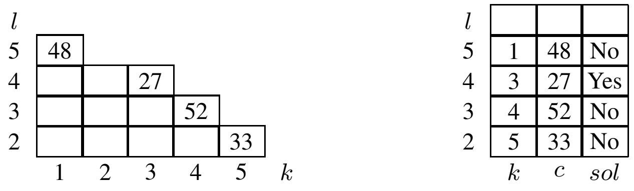


Fig. 1. Parameter Space and Data Representation

A final point to remark is that it is useless to consider a language $[k, l]$ with k larger than $MaxK(P, N, l)$, the maximum branching factor for the forks of a given depth l (it depends on l because only the forks containing the target are considered). Indeed, an increase of k will not affect the number of extractions. The algorithm below achieves this by setting the k-value at level l to ∞ and the count to 0 when refining it. When this happens for all l values, then it means that no wrapper based on (k, l)-contextual tree languages is expressive enough to reach a 100% F1-score. Note that there is always a solution when all examples come from a single document. The final set of forks then becomes ultimately the set of marked versions of the whole document.

Algorithm 1. Learning the Parameters

```
Input: $P$ and $N$, The sets of positive and negative examples.
Output: The parameters $k$ and $l$ of the wrapper.
 1: calc($P$,$N$, 1, 2) // initialisation
 2: $bestL$ = 2
 3: while not $F.sol[bestL]$ do
 4:    if $F.k[bestL]$=1 then
 5:       calc($P$,$N$, 1, $best$L+1)
 6:    end if
 7:    calc($P$,$N$, $F.k[bestL]$+1, $bestL$)
 8:    bestL = BestRefinement($F$);
 9: end while
10: return $F.k[bestL]$ and $bestL$

Function: calc($P$,$N$, $k$, $l$)
 1: if $k >$ maxK($P$,$N$, $l$) then
 2:    $F.k[l]$=$\infty$
 3:    $F.c[l]$=0
 4: else
 5:    $F.k[l]$=$k$;
 6:    $W$ = learnWrapper($P$, $k$, $l$)
 7:    $F.sol[l]$=$W$ rejects all $N$
 8:    $F.c$ = cnt(extractions($W$,$P$,$N$))
 9: end if
```

The algorithm is sketched in Algorithm 1. F is the array representing the front as shown in Fig. 1. For a given l value, the values $F.k[l]$, $F.c[l]$, and $F.sol[l]$ give respectively the k-value, the count and whether $[k, l]$ is a solution. It is initialized for

$l = 2$ with k-value 1. The function $BestRefinement(F)$ returns the l-value of the best candidate for refinement (as described above) if it exists, otherwise it either returns the l-value of the solution or reports failure. The function $calc(P, N, k, l)$ updates $F[l]$ with the appropriate values. Note that two refinements are computed when the selected best candidate has a k-value of 1.

3.2 Learning with Context

We define a new preprocessing step to identify a distinguishing context, to replace the ad hoc procedure in [12, 13] that returns zero or one context string. For each positive example we collect the set of text nodes that occur in the marked (k, l)-forks for that example. We define the *context* as the set of text nodes that is the common subset of all these sets. This way text nodes are only generalized when there is a positive example for which they do not occur in its parameterized neighborhood. This procedure guarantees that (given sufficient examples) all the strings in the resulting set are context for the target node. It is possible though that some discriminative context string is not found (for example the target is a node with as context either c1 or c2). We haven't yet encountered the need for a more elaborate procedure. Note that the count of a wrapper decreases with increasing context and that, given this procedure, the context increases with an increase in k or l, hence the anti-monotonicity property is still valid and our algorithm can easily be extended to learn a wrapper with context.

Not all data sets need a context. In principle, one could learn the wrapper with context and the wrapper without context independently of each other. However, one can easily integrate both in one algorithm that maintains two fronts and selects the most promising point of both for refinement. Note that, for a given point (k, l), the count of the wrapper with context is bounded by the count of the wrapper without context; i.e., the latter value can be used as an extra bound on the count of the former (hence selection is such that the former will only be evaluated when that bound is already known).

4 Induction with Equivalence Queries

Arbitrary sets of positive and negative examples contain often redundant information. It is more efficient to use queries. The system will ask itself the information that it needs to improve its hypothesis. In this section we present a system based on the algorithms from previous section, that uses equivalence queries[3]. The system allows the user to inspect its hypothesis by checking the extraction results (possibly for different pages). When detecting an error, the user signals it to the system as a counterexample (a false positive or a false negative), so that it can update its hypothesis.

In Section 4.1 we indicate how to adapt the algorithm of previous section for an efficient interactive use. In Section 4.2 we discuss some details of the implementation of our system and finally in Section 4.3 we give an evaluation of its usability.

4.1 Interactive Algorithm

After each interaction the system updates its hypothesis. This is done by finding the $\geq_S^{\#}$-most general language that is consistent with the current set of examples. For this update step we can use the algorithm from Section 3. However, an incremental algorithm is feasible. This would certainly improve the timings in Table 2 (see Section 4.3).

Adding a positive example (a false negative) to the set of examples increases the set of forks, hence the counts of all wrappers. However, a (k, l)-wrapper that covers negative examples still does so and cannot become a solution. It means that the search of a solution can start from the current front. The initialization of the new search for parameters consists of updating the count fields ($F.c$) in the front.

Adding a negative example (a false positive) does not affect the set of forks. However the solution is invalid as it covers the new negative example. After updating the (true) solution fields ($F.sol$)[6], the search can resume from the current front.

In short, the algorithm from Section 3 can be used. When a new example is received, the values in the front are updated and the search resumes.

4.2 Implementation

Representing the wrappers as sets of forks is straightforward, and works fine most of the time. For some tasks (requiring large k and l-values, and with pages with a large branching factor), the time for learning and extraction becomes noticeable and becomes an annoyance in an interactive application. We developed an implementation that represents the wrappers by unranked tree automata based on a technique described in [19]. This substantially reduces the memory consumption and the execution time without affecting the language accepted by the wrapper.

We added a graphical user interface to our application, which is basically a HTML-compliant browser, that allows the user to right-click on an element of the page to add an extra example. The system colors the background of all elements that are extracted by its hypothesis. A click on a colored element is interpreted as a false positive, a click on a plain element is interpreted as a false negative. This way the user is restricted to give only counterexamples to the equivalence query posed by the system.

Table 2. Number of interactions needed to learn the wrappers

Data set	P/N	ms	Data set	P/N	ms	Data set	P/N	ms	Data set	P/N	ms
s1-1	1/1	87	s10-2	1/1	33	s19-4	1/1	53	s29-1	3/2	2446
s1-3	4/1	915	s10-4	1/1	555	s20-3	1/0	35	s29-2	4/2	5628
s1-4	1/0	27	s11-1	1/2	885	s20-4	1/1	1364	s30-2	2/1	46
s3-2	1/1	56	s11-2	1/2	766	s20-5	1/1	1568	bigbook-2	1/2	2013
s3-3	1/1	127	s12-2	1/2	108	s20-6	1/1	1472	bigbook-3	1/1	723
s4-1	1/0	10	s13-2	1/2	45	s22-2	2/1	200	okra-1	1/2	123
s5-2	2/1	230	s13-4	1/1	584	s23-1	1/2	242	okra-2	1/1	684
s8-2	1/1	38	s14-3	1/0	26	s23-3	1/1	38	okra-3	1/2	235
s8-3	1/2	181	s15-2	1/0	18	s25-2	1/1	25	okra-4	1/1	536

[6] When the example is from a new document, also the counts are updated.

4.3 Evaluation

To evaluate our system, we use the same tasks as in the second experiment of Section 2.5. Each task is learned until a 100% F1-score is obtained. In Table 2 we show the number of interactions that are needed to learn the wrapper. The first column contains the data set, the second indicates the numbers of positive and negative examples[7] needed, and the last column indicates the total time needed by all the learning steps in Algorithm 1. The number of examples needed (P+N) is in the same range as those reported in [18] forAggressive Co-Testing for tasks where the latter reaches 100% F1-score. The system is highly responsive and suited for interactive use.

5 Conclusion

We have introduced a new subclass of the regular unranked tree languages, called (k, l)-contextual tree languages, that is learnable from positive examples only. We applied this class of languages to the problem of wrapper induction by representing a wrapper as a language of marked trees. Experiments on generally used data sets show the expressiveness of this wrapper representation to be superior over other approaches. We made an in-depth comparison with a wrapper inference algorithm based on Local Unranked Tree automata [12], which corresponds to $(k, 2)$-contextual tree languages; they lack expressivity and their authors tweak the representation of the documents by annotating the path from the root to the target node. An experiment learning wrappers from a small set of positive examples shows that our pure local languages usually yield a better wrapper than theirs.

Both our new algorithm as [12] need to tune parameters for each task. In [12] this is solved by evaluating wrappers on a sufficiently large set of completely annotated documents (representing positive and negative examples) to find the optimal parameter setting for a given extraction task. We developed a technique that learns a good parameter setting from a small set of positive and negative examples.

Another limitation of [12] was the need for an ad-hoc preprocessing step to identify a so called distinguishing context that in some applications is needed to disambiguate positive from negative examples. We developed a technique that preserves text nodes close to the target node when they occur in all examples.

We integrated the algorithm in an interactive system that allows a user to build a wrapper by selecting an initial positive example, and possibly a small number of false positives or false negatives, in sample documents. Experiments show that the resulting system is indeed able to learn a wrapper from a few positive and negative examples for a large number of extraction tasks. Interestingly, the system indicates failure when the extraction task is not expressible as a (k, l)-contextual tree language. In this case, one could switch to more expressive languages, e.g., the tRPNI algorithm [4] that needs a set of completely annotated documents (so far we have not met an existing data set requiring this).

[7] P/N = 1/0 means that the initial (1,2)-wrapper given one positive example is a solution.

References

1. H. Ahonen. *Generating grammars for structured documents using grammatical inference methods*. PhD thesis, University of Helsinki, Department of Computer Science, 1996.
2. D. Angluin. Inference of reversible languages. *Journal of the ACM (JACM)*, 29(3):741–765, 1982.
3. D. Angluin. Queries and concept-learning. *Machine Learning*, 2:319–342, 1988.
4. J. Carme, A. Lemay, and J. Niehren. Learning node selecting tree transducer from completely annotated examples. In *International Colloquium on Grammatical Inference*, volume 3264 of *Lecture Notes in Artificial Intelligence*, pages 91–102. Springer Verlag, Oct. 2004.
5. B. Chidlovskii, J. Ragetli, and M. de Rijke. Wrapper generation via grammar induction. In *Proc. 11th European Conference on Machine Learning (ECML)*, volume 1810, pages 96–108. Springer, Berlin, 2000.
6. D. Freitag and N. Kushmerick. Boosted wrapper induction. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Innovative Applications of AI Conference*, pages 577–583. AAAI Press, 2000.
7. D. Freitag and A. McCallum. Information extraction with HMMs and shrinkage. In *AAAI-99 Workshop on Machine Learning for Information Extraction*, 1999.
8. P. García. Learning k-testable tree sets from positive data. Technical report, Technical Report DSIC-ii-1993-46, DSIC, Universidad Politecnica de Valencia, 1993.
9. P. García and E. Vidal. Inference of k-testable languages in the strict sense and application to syntactic pattern recognition. *IEEE Trans. Pattern Anal. Mach. Intell.*, 12(9):920–925, 1990.
10. E. M. Gold. Language identification in the limit. *Information and Control*, 10(5):447–474, 1967.
11. T. Knuutila. Inference of k-testable tree languages. In H. Bunke, editor, *Advances in Structural and Syntactic Pattern Recognition: Proc. of the Intl. Workshop*, pages 109–120, Singapore, 1993. World Scientific.
12. R. Kosala, M. Bruynooghe, H. Blockeel, and J. V. den Bussche. Information extraction from web documents based on local unranked tree automaton inference. In *Intl. Joint Conference on Artificial Intelligence (IJCAI)*, pages 403–408, 2003.
13. R. Kosala, J. Van den Bussche, M. Bruynooghe, and H. Blockeel. Information extraction in structured documents using tree automata induction. In *PKDD*, volume 2431 of *Lecture Notes in Computer Science*, pages 299–310. Springer, 2002.
14. N. Kushmerick, D. S. Weld, and R. B. Doorenbos. Wrapper induction for information extraction. In *Intl. Joint Conference on Artificial Intelligence (IJCAI)*, pages 729–737, 1997.
15. R. McNaughton. Algebraic decision procedures for local testability. *Math. Systems Theory*, 8(1):60–76, 1974.
16. S. Muggleton. *Inductive Acquisition of Expert Knowledge*. Addison-Wesley, 1990.
17. I. Muslea, S. Minton, and C. Knoblock. Hierarchical wrapper induction for semistructured information sources. *Journal of Autonomous Agents and Multi-Agent Systems*, 4:93–114, 2001.
18. I. Muslea, S. Minton, and C. Knoblock. Active learning with strong and weak views: A case study on wrapper induction. In *Intl. Joint Conference on Artificial Intelligence (IJCAI)*, 2003.
19. S. Raeymaekers and M. Bruynooghe. Extracting information from structured documents with automata in a single run. In *Proc. 2nd Int. Workshop on Mining Graphs, Trees and Sequences (MGTS 2004, Pisa, Italy)*, pages 71–82, Pisa, Italy, 2004. University of Pisa.
20. J. R. Rico-Juan, J. Calera-Rubio, and R. C. Carrasco. Probabilistic k-testable tree languages. In A. Oliveira, editor, *Proceedings of 5th International Colloquium, ICGI*, pages 221–228, 2000.
21. S. Soderland. Learning information extraction rules for semi-structured and free text. *Machine Learning*, 34(1-3):233–272, 1999.