Automatic Wrapper Adaptation by Tree Edit Distance Matching

Emilio Ferrara and Robert Baumgartner

Abstract Information distributed through the Web keeps growing faster day by day, and for this reason, several techniques for extracting Web data have been suggested during last years. Often, extraction tasks are performed through so called wrappers, procedures extracting information from Web pages, e.g. implementing logic-based techniques. Many fields of application today require a strong degree of robustness of wrappers, in order not to compromise assets of information or reliability of data extracted.
Unfortunately, wrappers may fail in the task of extracting data from a Web page, if its structure changes, sometimes even slightly, thus requiring the exploiting of new techniques to be automatically held so as to adapt the wrapper to the new structure of the page, in case of failure. In this work we present a novel approach of *automatic wrapper adaptation* based on the measurement of similarity of trees through improved tree edit distance matching techniques.

1 Introduction

Web data extraction, during last years, captured attention both of academic research and enterprise world because of the huge, and still growing, amount of information distributed through the Web. Online documents are published in several formats but previous work primarily focused on the extraction of information from HTML Web pages.

Emilio Ferrara
University of Messina, Dept. of Mathematics, Via Ferdinando Stagno D'Alcontres, Salita Sperone, n. 31, Italy, e-mail: emilio.ferrara@unime.it

Robert Baumgartner
Lixto Software GmbH, Favoritenstrasse 16/DG, 1040 Vienna, Austria, e-mail: robert.baumgartner@lixto.com

Most of the wrapper generation tools developed during last years provide to full support for users in building data extraction programs (a.k.a. wrappers) automatically and in a visual way. They can reproduce the navigation flow simulating the human behavior, providing support for technologies adopted to develop Web pages, and so on. Unfortunately, a problem still holds: wrappers, because of their intrinsic nature and the complexity of extraction tasks they perform, usually are strictly connected to the structure of Web pages (i.e. DOM tree) they handle. Sometimes, also slight changes to that structure can cause the failure of extraction tasks. A couple of wrapper generation systems try to natively avoid problems caused by minor changes, usually building more elastic wrappers (e.g. working with relative, instead of absolute, XPath queries to identify elements).

Regardless of the degree of flexibility of the wrapper generator, wrapper maintenance is still a required step of a wrapper life-cycle. Once the wrapper has been correctly developed, it could work for a long time without any malfunction. The main problem in the wrapper maintenance is that no one can predict when or what kind of changes could occur in Web pages.

Fortunately, local and minor changes in Web pages are much more frequent case than deep modifications (e.g. layout rebuilding, interfaces re-engineering, etc.). However, it could also be possible, after a minor modification on a page, that the wrapper keeps working but data extracted are incorrect; this is usually even worse, because it causes a lack of consistency of the whole data extracted. For this reason, state-of-the-art tools started to perform validation and cleansing on data extracted; they also provide caching services to keep copy of the last working version of Web pages involved in extraction tasks, so as to detect changes; finally, they notify to maintainers any change, letting possible to repair or rewrite the wrapper itself. Depending on the complexity of the wrapper, it could be more convenient to rewrite it from scratch instead of trying to find causes of errors and fix them.

Ideally, a robust and reliable wrapper should include directives to auto-repair itself in case of malfunction or failure in performing its task. Our solution of automatic wrapper adaptation relies on exploiting the possibility of comparing some structural information acquired from the old version of the Web page, with the new one, thus making it possible to re-induct automatically the wrapper, with a custom degree of accuracy.

The rest of the paper is organized as follows: in Section 2 we consider the related work on theoretical background and Web data extraction, in particular regarding algorithms, techniques and problems of wrapper maintenance and adaptation. Sections 3 covers the automatic wrapper adaptation idea we developed, detailing some interesting aspects of algorithms and providing some examples. Experimentation and results are discussed in Section 4. Section 5, finally, presents some conclusive considerations.

2 Related Work

Theoretical background on techniques and algorithms widely adopted in this work relies on several Computer Science and Applied Mathematics fields such as Algorithms and Data Structures and Artificial Intelligence. In the setting of Web data extraction, especially algorithms on (DOM) trees play a predominant role. Approaches to analyze similarities between trees were developed starting from the well-known problem of finding the longest common subsequence(s) between two strings. Several algorithms were suggested, for example Hirshberg [5] provided the proof of correctness of three of them.

Soon, a strong interconnection between this problem and the similarity between trees has been pointed out: Tai [14] introduced the notion of *distance* as measure of the (dis)similarity between two trees and extended the notion of longest common subsequence(s) between strings to trees. Several *tree edit distance* algorithms were suggested, providing a way to transform a labeled tree in another one through local operations, like inserting, deleting and relabeling nodes. Bille [1] reported, in a comprehensive survey on the tree edit distance and related problems, summarizing approaches and analyzing algorithms.

Algorithms based on the tree edit distance usually are complex to be implemented and imply a high computational cost. They also provide more information than needed, if one just wants to get an estimate on the similarity. Considering these reasons, Selkow [13] developed a top-down trees isomorphism algorithm called *simple tree matching*, that establishes the degree of similarity between two trees, analyzing subtrees recursively. Yang [17] suggested an improvement of the simple tree matching algorithm, introducing weights.

During years, some improvements to tree edit distance techniques have been introduced: Shasha and Zhang [19] provided proof of correctness and implementation of some new parallelizable algorithms for computing edit distances between trees, lowering complexity of $O(|T_1| \cdot |T_2| \cdot \min(depth(T_1), leaves(T_1)) \cdot \min(depth(T_2), leaves(T_2)))$, for the non parallel implementation, to $O(|T_1| + |T_2|)$, for the parallel one; Klein [7], finally, suggested a fast method for computing the edit distance between unrooted ordered trees in $O(n^3 \log n)$. An overview of interesting applications of these algorithms in Computer Science can be found in Tekli et al. [15].

Literature on Web data extraction is manifold: Ferrara et al. [4] provided a comprehensive survey on application areas and used techniques, and Laender et al. [9] give a very good overview on wrapper generation techniques. Focusing on *wrapper adaptation*, Chidlovskii [2] presented some experimental results of combining and applying some grammatical and logic-based rules. Lerman et al. [10] developed a machine-learning based system for wrapper verification and reinduction in case of failure in extracting data from Web pages.

Meng et al. [11] suggested a new approach, called SG-WRAM (Schema-Guided WRApper Maintenance), for wrapper maintenance, considering that changes in Web pages always preserve syntactic features (i.e. data patterns, string lengths, etc.),

hyperlinks and annotations (e.g. descriptive information representing the semantic meaning of a piece of information in its context).

Wong [16] developed a probabilistic framework to adapt a previously learned wrapper to unseen Web pages, including the possibility of discovering new attributes, not included in the first one, relying on the extraction knowledge related to the first wrapping task and on the collection of items gathered from the first Web page. Raposo et al. [12] already suggested the possibility of exploiting previously acquired information, e.g. queries results, to re-induct a new wrapper from an old one not working anymore, because of structural changes in Web pages.

Kim et al. [6] compared results of simple tree matching and a modified weighed version of the same algorithm, in extracting information from HTML Web pages; this approach shares similarities to the one followed here to perform adaptation of wrappers. Kowalkiewicz et al. [8] focused on robustness of wrappers exploiting absolute and relative XPath queries.

3 Wrapper Adaptation

As previously mentioned, our idea is to compare some helpful structural information stored by applying the wrapper on the original version of the Web page, searching for similarities in the new one.

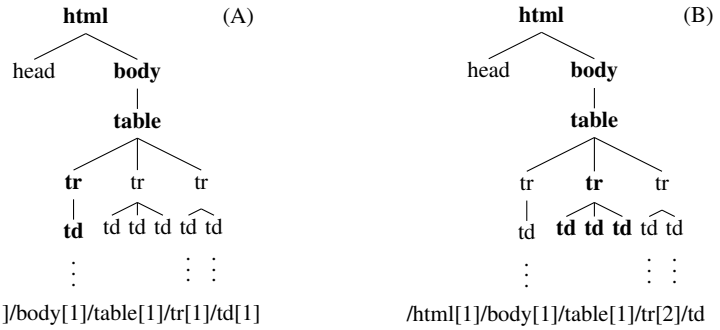
3.1 Primary Goals

Regardless of the method of extraction implemented by the wrapping system (e.g. we can consider a simple XPath), elements identified and represented as subtrees of the DOM tree of the Web page, can be exploited to find similarities between two different versions.

In the simplest case, the XPath identifies just a single element on the Web page (Figure 1.A); our idea is to look for some elements, in the new Web page, sharing similarities with the original one, evaluating comparable features (e.g. subtrees, attributes, etc.); we call these elements *candidates*; among candidates, the one showing the higher degree of similarity, probably, represents the new version of the original element.

It is possible to extend the same approach in the common case in which the XPath identifies multiple similar elements on the original page (e.g. a XPath selecting results of a search in a retail online shop, represented as table rows, divs or list items) (Figure 1.B); it is possible to identify multiple elements sharing a similar structure in the new page, within a custom level of accuracy (e.g. establishing a threshold value of similarity). Section 4 discusses also these cases.

Fig. 1 Examples of XPaths over trees, selecting one (A) or multiple (B) items.



Once identified, elements in the new version of the Web page can be extracted as usual, for example just re-inducting the XPath. Our purpose is to define some rules to enable the wrapper to face the problem of automatically adapting itself to extract information from the new Web page.

We implemented this approach in a commercial tool [1]; the most efficient way to acquire some structural information about elements the original wrapper extracts, is to store them inside the definition of the wrapper itself. For example, generating *signatures* representing the DOM subtree of extracted elements from the original Web page, stored as a tree diagram, or a simple XML document (or, even, the HTML itself). This shrewdness avoids that we need to store the whole original page, ensuring better performances and efficiency.

This technique requires just a few settings during the definition of the wrapper step: the user enables the automatic wrapper adaptation feature and set an accuracy threshold. During the execution of the wrapper, if some XPath definition does not match a node, the wrapper adaptation algorithm automatically starts and tries to find the new version of the missing node.

3.2 Details

First of all, to establish a measure of similarity between two trees we need to find some comparable properties between them. In HTML Web pages, each node of the DOM tree represents an HTML element defined by a tag (or, otherwise, free text). The simplest way to evaluate similarity between two elements is to compare their *tag name*. Elements own some particular common attributes (e.g. *id*, *class*, etc.) and some type-related attributes (e.g. *href* for anchors, *src* for images, etc.); it is possible to exploit this information for additional checks and comparisons.

[1] Lixto Suite, www.lixto.com

The algorithm selects candidates between subtrees sharing the same root element, or, in some cases, *comparable* -but not identical- elements, analyzing tags. This is very effective in those cases of deep modification of the structure of an object (e.g. conversion of tables in divs).

As discussed in Section 2, several approaches have been developed to analyze similarities between HTML trees; for our purpose we improved a version of the *simple tree matching* algorithm, originally led by Selkow [13]; we call it *clustered tree matching*. There are two important novel aspects we are introducing in facing the problem of the automatic wrapper adaptation: first of all, exploiting previously acquired information through a smart and focused usage of the tree similarity comparison; thus adopting a consolidated approach in a new field of application. Moreover, we contributed applying some particular and useful changes to the algorithm itself, improving its behavior in the HTML trees similarity measurement.

3.3 Simple Tree Matching

Let *d(n)* to be the degree of a node *n* (i.e. the number of first-level children); let T(i) to be the i-*th* subtree of the tree rooted at node T; this is a possible implementation of the *simple tree matching* algorithm:

Algorithm 1 SimpleTreeMatching(T', T'')

1: **if** T' has the same label of T'' **then**
2: $m \leftarrow d(T')$
3: $n \leftarrow d(T'')$
4: **for** $i = 0$ to m **do**
5: $M[i][0] \leftarrow 0$;
6: **for** $j = 0$ to n **do**
7: $M[0][j] \leftarrow 0$;
8: **for all** i such that $1 \leq i \leq m$ **do**
9: **for all** j such that $1 \leq j \leq n$ **do**
10: $M[i][j] \leftarrow \text{Max}(M[i][j-1], M[i-1][j], M[i-1][j-1] + W[i][j])$ where $W[i][j]$ = SimpleTreeMatching($T'(i-1), T''(j-1)$)
11: return M[m][n]+1
12: **else**
13: return 0

Advantages of adopting this algorithm, which has been shown quite effective for Web data extraction [6, 18], are multiple; for example, the *simple tree matching* algorithm evaluates similarity between two trees by producing the maximum matching through dynamic programming, without computing inserting, relabeling and deleting operations; moreover, tree edit distance algorithms relies on complex implementations to achieve good performance, instead *simple tree matching*, or similar algorithms are very simple to implement.

The computational cost is $O(n^2 \cdot \max(leaves(T'), leaves(T'')) \cdot \max(depth(T'), depth(T'')))$, thus ensuring good performances, applied to HTML trees. There are some limitations; most of them are irrelevant but there is an important one: this approach can not match permutations of nodes. Despite this intrinsic limit, this technique appears to fit very well to our purpose of measuring HTML trees similarity.

3.4 Clustered Tree Matching

Let $t(n)$ to be the number of total siblings of a node n including itself:

Algorithm 2 ClusteredTreeMatching(T', T'')

```
1: {Change line 11 with the following code}
2: if m > 0 AND n > 0 then
3:     return M[m][n] * 1 / Max(t(T'), t(T''))
4: else
5:     return M[m][n] + 1 / Max(t(T'), t(T''))
```

In order to better reflect a good measure of similarity between HTML trees, we applied some focused changes to the way of assignment of a value for each matching node.

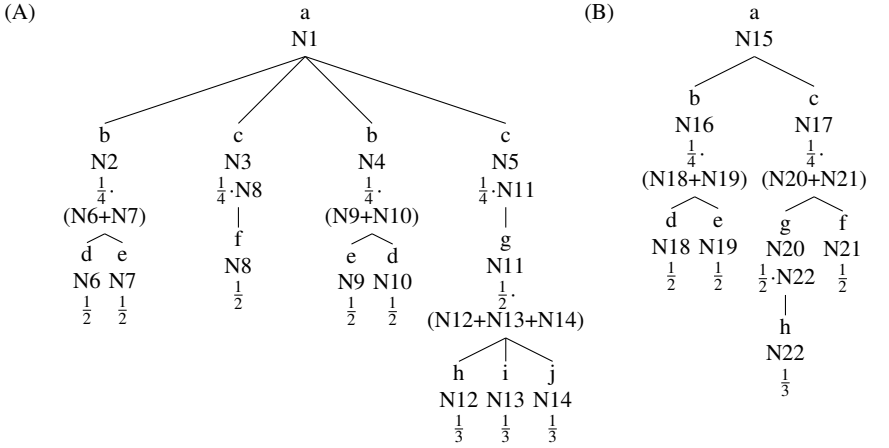
In the *simple tree matching* algorithm the assigned matching value is always 1. After leading some analysis and considerations on structure of HTML pages, our intuition was to assign a weighed value, with the purpose of attributing less importance to slight changes, in the structure of the tree, when they occur in deep sublevels (e.g. missing/added leaves, small truncated/added branches, etc.) and also when they occur in sublevels with many nodes, because these mainly represent HTML list of items, table rows, etc., usually more likely to modifications.

In the *clustered tree matching*, the weighed value assigned to a match between two nodes is 1, divided by the greater number of siblings with respect to the two compared nodes, considering nodes themselves (e.g. Figure 2.A, 2.B); thus reducing the impact of missing/added nodes.

Before assigning a weight, the algorithm checks if it is comparing two leaves or a leaf with a node which has children (or two nodes which both have children). The final contribution of a sublevel of leaves is the sum of assigned weighted values to each leaf (cfr. Code Line (4,5)); thus, the contribution of the parent node of those leaves is equal to its weighed value multiplied by the sum of contributions of its children (cfr. Code Line (2,3)). This choice produces an effect of *clustering* the process of matching, subtree by subtree; this implies that, for each sublevel of leaves the maximum sum of assigned values is 1; thus, for each parent node of that sublevel the maximum value of the multiplication of its contribution with the sum of contributions of its children, is 1; each cluster, singly considered, contributes with a maximum value of 1. In the last recursion of this top-down algorithm, the two roots

will be evaluated. The resulting value at the end of the process is the measure of similarity between the two trees, expressed in the interval [0,1]. The closer the final value is to 1, the more the two trees are similar.

Fig. 2 *A* and *B* are two similar labeled rooted trees.



Let us analyze the behavior of the algorithm with an example, already used by [17] and [18] to explain the simple tree matching (Figure 2): 2.A and 2.B are two very simple generic rooted labeled trees (i.e. the same structure of HTML trees). They show several similarities except for some missing nodes/branches.

Applying the *clustered tree matching* algorithm, in the first step (Figure 2.A, 2.B) contributions assigned to leaves, with respect to matches between the two trees, reflect the past considerations (e.g. a value of $\frac{1}{3}$ is established for nodes (h), (i) and (j), although two of them are missing in 2.B). Going up to parents, the summation of contributions of matching leaves is multiplied by the relative value of each node (e.g. in the first sublevel, the contribution of each node is $\frac{1}{4}$ because of the four first-sublevel nodes in 2.A).

Once completed these operations for all nodes of the sublevel, values are added and the final measure of similarity for the two trees is obtained. Intuitively, in more complex and deeper trees, this process is iteratively executed for all the sublevels. The deeper a mismatch is found, the less its missing contribution will affect the final measure of similarity. Analogous considerations hold for missing/added nodes and branches, sublevels with many nodes, etc. Table 1 shows M and W matrices containing contributions and weights.

In this example, ClusteredTreeMatching(2.A, 2.B) returns a measure of similarity of $\frac{3}{8}$ (0.375) whereas SimpleTreeMatching(2.A, 2.B) would return a mapping value of 7; the main difference on results provided by these two algorithms is the

Table 1 *W* and *M* matrices for each matching subtree.

W	N18	N19
N6	$\frac{1}{2}$	0
N7	0	$\frac{1}{2}$

M	0	N18	N18-19
0	0	0	0
N6	0	$\frac{1}{2}$	$\frac{1}{2}$
N6-7	0	$\frac{1}{2}$	1

W	N18	N19
N9	0	$\frac{1}{2}$
N10	$\frac{1}{2}$	0

M	0	N18	N18-19
0	0	0	0
N9	0	0	$\frac{1}{2}$
N9-10	0	$\frac{1}{2}$	$\frac{1}{2}$

W	N8
N20	0
N21	$\frac{1}{2}$

M	0	N8
0	0	0
N20	0	0
N20-21	0	$\frac{1}{2}$

W	N12	N13	N14
N22	$\frac{1}{3}$	0	0

M	0	N12	N12-13	N12-14
0	0	0	0	0
N22	0	$\frac{1}{3}$	$\frac{1}{3}$	$\frac{1}{3}$

W	N11
N20	$\frac{1}{6}$
N21	0

M	0	N11
0	0	0
N20	0	$\frac{1}{6}$
N20-21	0	$\frac{1}{6}$

W	N2	N3	N4	N5
N16	$\frac{1}{4}$	0	$\frac{1}{8}$	0
N17	0	$\frac{1}{8}$	0	$\frac{1}{24}$

M	0	N2	N2-3	N2-4	N2-5
0	0	0	0	0	0
N16	0	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$
N16-17	0	$\frac{1}{4}$	$\frac{3}{8}$	$\frac{3}{8}$	$\frac{3}{8}$

following: our *clustered tree matching* intrinsically produces an absolute measure of similarity between the two compared trees; the *simple tree matching*, instead, returns the mapping value and then it needs subsequent operations to establish the measure of similarity.

Hypothetically, in the *simple tree matching* case, we could suppose to establish a good estimation of similarity dividing the mapping value by the total number of nodes of the tree with more nodes; indeed, a value calculated in this way would be linear with respect to the number of nodes, thus ignoring important information as the position of mismatches, the number of mismatches with respect to the total number of subnodes/leaves in a particular sublevel, etc.

In this case, for example, the measure of similarity between 2.A and 2.B, applying this approach, would be $\frac{7}{14}$ (0.5). A greater value of similarity could suggest, wrongly, that this approach is more accurate. Experimentation showed us that, the closer the measure of similarity is to reflect changes in complex structures, the higher the accuracy of the matching process is. This fits particularly well for HTML trees, which often show very rich and articulated structures.

The main advantage of using the *clustered tree matching* algorithm is that, the more the structure of considered trees is complex and similar, the more the measure of similarity will be accurate. On the other hand, for simple and quite different trees the accuracy of this approach is lower than the one ensured by the *simple tree matching*. But, as already underlined, the most of changes in Web pages are usually minor changes, thus *clustered tree matching* appears to be a valid technique to achieve a reliable process of automatic wrapper adaptation.

4 Experimentation

In this section we discuss some experimentation performed on common fields of application [4] and following results. We tried to automatically adapt wrappers, previously built to extract information from particular Web pages, after some -often minor- structural changes.

All the followings are real use cases: we did not modify any Web page, original owners did; thus re-publishing pages with changes and altering the behavior of old wrappers. Our will to handle real use cases limits the number of examples of this study. These real use cases confirmed our expectations and simulations on ad hoc examples we prepared to test the algorithms.

We obtained an acceptable degree of precision using the *simple tree matching* and a great rate of precision/recall using the *clustered tree matching*. Precision, Recall and F-Measure will summarize these results showed in Table 2. We focused on following areas, widely interested by Web data extraction:

- News and Information: Google News [2] is a valid use case for wrapper adaptation; templates change frequently and sometimes is not possible to identify elements with old wrappers.
- Web Search: Google Search [3] completely rebuilt the results page layout in the same period we started our experimentation [4]; we exploited the possibility of automatically adapting wrappers built on the old version of the *result page*.
- Social Networks: another great example of continuous restyling is represented by the most common social network, Facebook [5]; we successfully adapted wrappers extracting friend lists also exploiting additional checks performed on attributes.
- Social Bookmarking: building *folksonomies* and *tagging* contents is a common behavior of Web 2.0 users. Several Websites provide platforms to aggregate and classify sources of information and these could be extracted, so, as usual, wrapper adaptation is needed to face chages. We choose Delicious [6] for our experimentation obtaining stunning results.
- Retail: these Websites are common fields of application of data extraction and Ebay [7] is a nice real use case for wrapper adaptation, continuously showing, often almost invisible, structural changes which require wrappers to be adapted to continue working correctly.
- Comparison Shopping: related to the previous category, many Websites provide tools to compare prices and features of products. Often, it is interesting to extract this information and sometimes this task requires adaptation of wrappers to

[2] http://news.google.com

[3] http://www.google.com

[4] http://googleblog.blogspot.com/2010/05/spring-metamorphosis-googles-new-look.html

[5] http://www.facebook.com

[6] http://www.delicious.com

[7] http://www.ebay.com

structural changes of Web pages. Kelkoo [8] provided us a good use case to test our approach.

- Journals and Communities: Web data extraction tasks can also be performed on the millions of online Web journals, blogs and forums, based on open source blog publishing applications (e.g. Wordpress [9], Serendipity [10], etc.), CMS (e.g. Joomla [11], Drupal [12], etc.) and community management systems (e.g. phpBB [13], SMF [14], etc.). These platforms allow changing templates and often this implies wrappers must be adapted. We lead the automatic adaptation process on Techcrunch [15], a tech journal built on Wordpress.

We adapted wrappers for these 7 use cases considering 70 Web pages; Table 2 summarizes results obtained comparing the two algorithms applied on the same page, with the same configuration (threshold, additional checks, etc.). *Threshold* represents the value of similarity required to match two trees. The columns *true pos.*, *false pos.* and *false neg.* represent true and false positive and false negative items extracted from Web pages through adapted wrappers.

Table 2 Experimental results.

		Simple Tree Matching			**Clustered Tree Matching**		
		Precision/Recall			Precision/Recall		
URL	threshold	true pos.	false pos.	false neg.	true pos.	false pos.	false neg.
news.google.com	90%	604	-	52	644	-	12
google.com	80%	100	-	60	136	-	24
facebook.com	65%	240	72	-	240	12	-
delicious.com	40%	100	4	-	100	-	-
ebay.com	85%	200	12	-	196	-	4
kelkoo.co.uk	40%	60	4	-	58	-	2
techcrunch.com	85%	52	-	28	80	-	-
Total	-	1356	92	140	1454	12	42
Recall	-		90.64%			97.19%	
Precision	-		93.65%			99.18%	
F-Measure	-		92.13%			98.18%	

In some situations of deep changes (Facebook, Kelkoo, Delicious) we had to lower the threshold in order to correctly match the most of the results. Both the algorithms show a great elasticity and it is possible to adapt wrappers with a high degree of reliability; the *simple tree matching* approach shows a weaker recall

[8] http://shopping.kelkoo.co.uk

[9] http://wordpress.org

[10] http://www.s9y.org

[11] http://www.joomla.org

[12] http://drupal.org

[13] http://www.phpbb.com

[14] http://www.simplemachines.org

[15] http://www.techcrunch.com

value, whereas performances of the *clustered tree matching* are stunning (F-Measure greater than 98% is an impressive result). Sometimes, additional checks on nodes attributes are performed to refine results of both the two algorithms. For example, we can additionally include attributes as part of the node label (e.g. *id*, *name* and *class*) to refine results. Also without including these additional checks, the most of the time the false positive results are very limited in number (cfr. the Facebook use case).

Figure 3 shows a screenshot of the developed tool, performing an automatic wrapper adaptation task: in this example we adapted the wrapper defined for extracting Google news, whereas the original XPath was not working because of some structural changes in the layout of news. Elements identified by the original XPath are highlighted in red in the upper browser, elements highlighted in the bottom browser represent the recognized ones through the wrapper adaptation process.

Fig. 3 An example of Wrapper Adaptation.

5 Conclusion

This work presents new scenarios, analyzing Wrapper Adaptation related problems from a novel point of view, introducing improvements to algorithms and new fields of application.

There are several possible improvements to our approach we can already imagine. First of all, it could be very interesting to extend the matching criteria we used, making the tree matching algorithm smarter. Actually, we already included features like analyzing attributes (e.g. *id*, *name* and *class*) instead of just comparing labels/tags or node types. The accuracy of the matching process benefits of these

additional checks and it is possible, for example, to improve this aspect with a more complex matching technique, containing full path information, all attributes, etc.

It could be interesting to compare these algorithms, with other tree edit distance approaches working with permutations; although, intuitively, *simple tree matching* based algorithms can not handle permutations on nodes, maybe it is possible to develop some enhanced version which solves this limitation. Furthermore, just considering the tree structure can be limiting in some particular situations: if a new node has only empty textual fields (or, equally, if a deleted node had only empty fields) we could suppose its weight should be null. In some particular situation this inference works well, in some others, instead, it could provoke mismatches. It could also be interesting to exploit textual properties, nevertheless, not necessarily adopting Natural Language Processing techniques (e.g. using logic-based approaches, like regular expressions, or string edit distance algorithms, or just the length of strings – treating two nodes as equal only if the textual content is similar or of similar length).

The tree grammar could also be used in a machine learning approach, for example creating some tree templates to match similar structures or tree/cluster diagrams to classify and identify several different topologies of common substructures in Web pages. This process of simplification is already used to store a light-weight snapshot of elements identified by a wrapper applied on a Web page, at the time of extraction; actually, this feature allows the algorithm to work also without the original version of the page, but just exploiting some information about extracted items. This possibility opens new scenarios for future work on Wrapper Adaptation.

Concluding, the *clustered tree matching* algorithm we described is very extensible and resilient, so as ensuring its use in several different fields, for example it perfectly fits in identifying similar elements belonging to a same structure but showing some small differences among them. Experimentation on wrapper adaptation has already been performed inside a productive tool, the Lixto Suite [3], this because our approach has been shown to be solid enough to be implemented in real systems, ensuring great reliability and, generically, stunning results.

References

1. Bille, P.: A survey on tree edit distance and related problems. Theoretical Computer Science **337**(1-3), 217–239 (2005). DOI 10.1016/j.tcs.2004.12.030
2. Chidlovskii, B.: Automatic repairing of web wrappers. In: Proceedings of the 3rd international workshop on Web information and data management, p. 30. ACM (2001)
3. Ferrara, E., Baumgartner, R.: Design of automatically adaptable web wrappers. In: ICAART '11: Proceedings of the 3rd International Conference on Agents and Artificial Intelligence (2011)
4. Ferrara, E., Fiumara, G., Baumgartner, R.: Web Data Extraction, Applications and Techniques: A Survey. Technical Report (2010)
5. Hirschberg, D.: A linear space algorithm for computing maximal common subsequences. Communications of the ACM **18**(6), 343 (1975)
6. Kim, Y., Park, J., Kim, T., Choi, J.: Web Information Extraction by HTML Tree Edit Distance Matching. In: Proceedings of the 2007 International Conference on Convergence Information Technology, vol. 1, pp. 2455–2460. Ieee (2007). DOI 10.1109/ICCIT.2007.19

7. Klein, P.: Computing the edit-distance between unrooted ordered trees. In: Algorithms — ESA' 98, *Lecture Notes in Computer Science*, vol. 1461, pp. 1–1. Springer Berlin / Heidelberg (1998)
8. Kowalkiewicz, M., Kaczmarek, T., Abramowicz, W.: MyPortal: robust extraction and aggregation of web content. In: Proceedings of the 32nd International Conference on Very Large Data Bases, pp. 1219–1222 (2006)
9. Laender, A., Ribeiro-Neto, B., Silva, A.D., JS: A brief survey of web data extraction tools. ACM Sigmod **31**(2), 84–93 (2002). DOI 10.1145/565117.565137
10. Lerman, K., Minton, S., Knoblock, C.: Wrapper maintenance: A machine learning approach. Journal of Artificial Intelligence Research **18**(2003), 149–181 (2003)
11. Meng, X., Hu, D., Li, C.: Schema-guided wrapper maintenance for web-data extraction. In: Proceedings of the 5th ACM international workshop on Web information and data management, pp. 1–8. ACM, New York, USA (2003). DOI 10.1145/956699.956701
12. Raposo, J., Pan, A., Álvarez, M., Viña, A.: Automatic wrapper maintenance for semi-structured web sources using results from previous queries. In: Proceedings of the 2005 ACM symposium on Applied computing - SAC '05, pp. 654–659. ACM Press, New York, New York, USA (2005). DOI 10.1145/1066677.1066826
13. Selkow, S.: The tree-to-tree editing problem. Information Processing Letters **6**(6), 184 – 186 (1977). DOI 10.1016/0020-0190(77)90064-3
14. Tai, K.: The tree-to-tree correction problem. Journal of the ACM (JACM) **26**(3), 433 (1979)
15. Tekli, J., Chbeir, R., Yetongnon, K.: An overview on XML similarity: Background, current trends and future directions. Computer Science Review **3**(3), 151–173 (2009). DOI 10.1016/j.cosrev.2009.03.001
16. Wong, T.l.: A Probabilistic Approach for Adapting Information Extraction Wrappers and Discovering New Attributes. In: Proceedings of the Fourth IEEE International Conference on Data Mining, pp. 257–264. Ieee (2004). DOI 10.1109/ICDM.2004.10111
17. Yang, W.: Identifying syntactic differences between two programs. Software - Practice and Experience **21**(7), 739–755 (1991)
18. Zhai, Y., Liu, B.: Web data extraction based on partial tree alignment. In: Proceedings of the 14th international conference on World Wide Web, pp. 76–85. ACM, New York, NY, USA (2005). DOI 10.1145/1060745.1060761
19. Zhang, K., Shasha, D.: Simple fast algorithms for the editing distance between trees and related problems. SIAM J. Comput. **18**(6), 1245–1262 (1989)