# NET – A System for Extracting Web Data from Flat and Nested Data Records

Bing Liu and Yanhong Zhai

Department of Computer Science
University of Illinois at Chicago
851 S. Morgan Street, Chicago, IL 60607
{liub, yzhai}@cs.uic.edu

**Abstract.** This paper studies automatic extraction of structured data from Web pages. Each of such pages may contain several groups of structured data records. Existing automatic methods still have several limitations. In this paper, we propose a more effective method for the task. Given a page, our method first builds a tag tree based on visual information. It then performs a post-order traversal of the tree and matches subtrees in the process using a tree edit distance method and visual cues. After the process ends, data records are found and data items in them are aligned and extracted. The method can extract data from both flat and nested data records. Experimental evaluation shows that the method performs the extraction task accurately.

## 1 Introduction

Structured data objects are an important type of information on the Web. Such objects are often data records retrieved from a backend database and displayed in Web pages with some fixed templates. This paper also calls them *data records*. Extracting data from such data records enables one to integrate data from multiple sites to provide value-added services, e.g., comparative shopping, and meta-querying.

There are two main approaches to data extraction, wrapper induction and automatic extraction. In wrapper induction, a set of data extraction rules are learnt from a set of manually labeled pages [5, 8, 9, 14]. However, manual labeling is labor intensive and time consuming. For different sites or even pages in the same site, manual labeling needs to be repeated because different sites may follow different templates. For large scale web data extraction tasks, manual labeling is a serious drawback.

For automatic extraction, [1, 4, 6] find patterns or grammars from multiple pages containing similar data records. Requiring an initial set of pages containing similar data records is, however, a limitation. [6] proposes a method that tries to explore the detailed information pages behind the current page to segment data records. The need for such detailed pages behind is a drawback because many data records do not have such pages or such pages are hard to find. [3] proposes a string matching method. However, it could not find nested data records. A similar method is proposed in [11]. [7] and [15] propose some algorithms to identify data records, which do not extract data items from the data records, and do not handle nested data records. Our previous system DEPTA [13] is able to align and extract data items from data records, but does not handle nested data records.


**(a). An example page segment**

| | | | | | |
|---|---|---|---|---|---|
| image 1 | Canning Jars by Ball | 8-oz | Canning Jars, Set of 4 | ***** | $4.95 |
| image 1 | Canning Jars by Ball | 1-pt | Canning Jars, Set of 4; Blue Gingham | **** | $5.95 |
| image 2 | Canning Tools by Norpro | 12-dia | Canning Rack | **** | $4.95 |

**(b). Extraction results**

**Fig. 1: An example page and the extraction results**

This paper proposes a more effective method to extract data from Web pages that contains a set of flat or nested data records automatically. Our method is based on a tree edit distance method and visual cues. It is called NET (*N*ested data *E*xtraction using *T*ree matching and visual cues). Given a Web page, it works in two main steps:

1. Building a tag tree of the page. Due to erroneous tags and unbalanced tags in the HTML code of the page, building a correct tree is not a simple task. A visual based method is used to deal with this problem.
2. Identifying data records and extracting data from them. The algorithm performs a post-order traversal of the tag tree to identify data records at different levels. This ensures that nested data records are found. A tree edit distance algorithm and visual cues are used to perform these tasks.

Experimental evaluation shows that the technique is highly effective.

## 2 Problem Statement

Fig. 1a gives an example page segment that contains two data records. In this segment, the first data record has two nested data records, i.e., the same type of products but different sizes, looks, prices, etc.

*Our task*: Given a Web page that contains multiple data records (at least two), we discover the underlying data records, extract the data items in them and put the data in a relational table. For Fig. 1a, we aim to produce the table in Fig. 1b. Due to space limitations, we omitted the red spot in Fig. 1a. Note that "image 1" and "Canning Jars by Ball" are common for the first and the second rows due to nesting.

## 3 Building the Tag Tree

In a Web browser, each HTML element is rendered as a rectangle. Instead of using nested tags (which have many errors) in the HTML code to build a tag tree, we build a tag tree based on the nested rectangles (see [13] for more details).

1. Find the 4 boundaries of the rectangle of each HTML element by calling the em-

bedded parsing and rendering engine of a browser, e.g., Internet explorer.
2. Detect containment relationships among the rectangles, i.e., whether one rectangle is contained inside another. A tree can be built based on the containment check.

## 4 The Proposed Algorithm

Before presenting the algorithm, we discuss two observations about data records in Web pages, which simplify the extraction task. The observations were made in [7].

1. A group of data records that contains descriptions of a list of similar objects are typically presented in a contiguous region of a page and are formatted using similar HTML tags, e.g., the data records in Fig. 1a.
2. A group of similar data records being placed in a region is reflected in the tag tree by the fact that they are under one parent node, although we do not know which parent (the algorithm will find out). In other words, a set of similar data records are formed by some child sub-trees of the same parent node.

These observations make it possible to design a tree based method for data extraction.

The basic idea of our proposed algorithm is to traverse the tag tree in post-order (or bottom-up). This ensures that nested data records are found at a lower level based on repeating patterns before processing an upper level. For example, in Fig. 1a, the following two nested data records are found first at a lower level:

| | | | |
|---|---|---|---|
| 8-oz | Canning Jars, Set of 4 | ***** | $4.95 |
| 1-pt | Canning Jars, Set of 4; Blue Gingham | **** | $5.95 |

Then at an upper level, all the data records are found as shown in Fig. 1b.

The overall algorithm NET is given in Fig. 2.

```
Algorithm NET(Root, T)
1   Traverse(Root, T);
2   Output().

Traverse(Node, T)
1   if Depth(Node) => 3 then
2       for each Child ∈ Node.Children do
3           Traverse(Child, T);
4       Match(Node, T);
```

**Fig. 2: The overall NET algorithm**

Line 1 of Traverse() says that the algorithm will not search for data records if the depth of the sub-tree from *Node* is 2 or 1 as it is unlikely that a data record is formed with only a single level of tag(s). Match() is the procedure that performs tree matching on child subtrees of *Node* (see below). *T* is a threshold for a match of two trees to be considered sufficiently similar. Output() in NET() outputs the extracted data to the user in relational tables (as a page may have multiple *data areas* with different structured data, and data in each area are put in a separate table). Note that some simple optimization can be performed to the NET algorithm. For example, if *Child* does not have any data item, e.g., text, image, etc, Traverse() may not be performed on *Child*.

The Match procedure is given in Fig. 3. TreeMatch() matches two child subtrees under *Node* (line 4). In lines 2 and 3, we set TreeMatch() to be applied to every pair of child nodes, which ensures all necessary data item matches are captured. AlignAndLink() aligns and links matched data items (leaf nodes) (line 5). The details of these procedures are given below.

PutDataToTable() extracts the matched data items and puts them in tables. This will be discussed below together with GenPrototypes(). Note that PutDataToTable() does not output the final results, which is done by Ouput() (line 2 of NET() in Fig. 2).

```
Match(Node, T)
1   Children = Node.Children;
2   for each ChildF in Children do
3       for each ChildR in Children = Children – {ChildF} do
4           if TreeMatch(ChildF, ChildR) > T then
5               AlignAndLink();
6               if all items in ChildR are aligned and linked then
7                   Children = Children – {ChildR}
8   if some alignments have been made then
9     PutDataInTables(Node);
10    GenPrototypes(Node);
```

**Fig. 3: The Match procedure**

```
Simple_Tree_Matching(A, B)
1.  if the roots of the two trees A and B contain distinct symbols or there is a visual
      conflict between A and B
2.  then return (0);
3.  else  m:= the number of first-level sub-trees of A;
4.        n:= the number of first-level sub-trees of B;
5.        Initialization:    M[i, 0]:= 0 for i = 0, …, m;
                             M[0, j] := 0 for j = 0, …, n;
6.        for i = 1 to m do
7.          for j = 1 to n do
8.            M[i,j]:=max(M[i,j-1], M[i-1, j], M[i-1, j-1]+W[i, j]);
                  where W[i,j] = Simple_Tree_Matching(A_i, B_j)
9.        return (M[m, n]+1)
```

**Fig. 4: The simple tree matching algorithm**

## Tree Matching: TreeMatch()

TreeMatch() uses a tree edit distance or matching algorithm. Since a list of data records form repeated patterns, this procedure basically finds such tree patterns.

In this work, we use a restricted tree matching algorithm, called *simple tree matching* (STM) (Yang 1991). STM evaluates the similarity of two trees by producing the maximum matching through dynamic programming with complexity $O(n_1 n_2)$, where $n_1$ and $n_2$ are the sizes of trees $A$ and $B$ respectively.

Let $A$ and $B$ be two trees and $i \in A, j \in B$ are two nodes in $A$ and $B$ respectively. A *matching* between two trees in STM is defined to be a mapping $M$ such that for every pair $(i, j) \in M$ where $i$ and $j$ are non-root nodes, (parent($i$), parent($j$)) $\in M$. A *maximum matching* is a matching with the maximum number of pairs.

Let $A = <R_A, A_1, A_2, \ldots, A_m>$ and $B=<R_B, B_1, B_2, \ldots, B_n>$ be two trees, where $R_A$ and $R_B$ are the roots of $A$ and $B$, and $A_i$, $B_j$ are the $i$th and $j$th first-level sub-trees of $A$ and $B$ respectively. When $R_A$ and $R_B$ match, the maximum matching between $A$ and $B$ is $M_{A,B}$+1 ($M_{A,B}$ is the maximum match between $<A_1, A_2, \ldots, A_m>$ and $<B_1, B_2, \ldots, B_n>$).

In the Simple_Tree_Matching algorithm in Fig. 4, the roots of $A$ and $B$ are compared first (line 1). If the roots match, then the algorithm recursively finds the maximum matching between first-level sub-trees of $A$ and $B$ and save it in $W$ matrix (line 8). Based on the $W$ matrix, a dynamic programming scheme is applied to find the number of pairs in a maximum matching between two trees $A$ and $B$.

Note that we add a visual based condition in line 1. That is, we want to make sure that $A$ and $B$ has no visual conflict. For example, based on the visual information, if

the width of *A* is much larger than that of *B*, then they are unlikely to match. Due to space limitations, we are unable to present all the rules that we use here. These rules help to produce better match results and also to reduce the computation significantly.

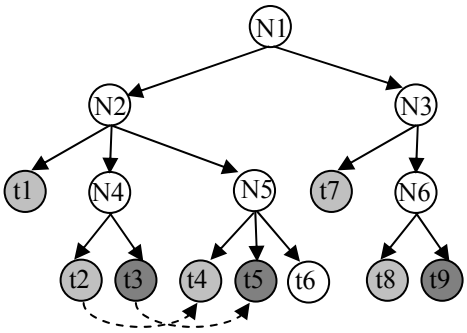


**Fig. 5: Aligned data nodes under N2 are linked**

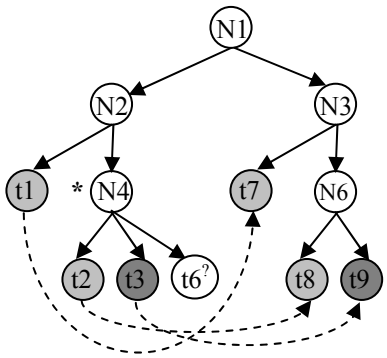


**Fig. 6: Aligned data nodes under N1 are linked**

## Align Matched Data Items: AlignAndLink()

AlignAndLink() aligns the data items after tree matching. We simply trace back in the *M* matrices to find the matched/aligned items in the two trees. When there is more than one match for a node that gives the maximum result, we choose the one appearing the earliest in the tree. This performs well (visual information may also be used).

All the aligned data items are then linked. The links are directional, i.e., an earlier data item will point to its next matching data item. Fig. 5 gives an example, where ti represents a terminal (data item) node, and Nj represents a tag node. Since NET performs post-order traversal, at the level N4-N5, t2-t4 and t3-t5 are matched (assume they satisfy the match condition, line 4 of Fig. 3). They are aligned and linked (dash lines). N4 and N5 are data records at this level (nested in N2). t6 is optional.

At the level of N2-N3, TreeMatch() will only match N4 subtree and N6 subtree. We see that t2-t8 and t3-t9 are linked. t1 and t7 are also linked as they match (Fig. 6).

The subtree at N5 is omitted in Fig.6 because N5 has the same structure as N4. N4 is marked with a "*" in Fig. 6 as it is turned into a prototype data record by GenPrototypes(). Note that t6 is inserted into N4 as an optional node, denoted by "?". A prototype represents a standard/typical data record containing the complete structure so far. We may not be able to use the first data record (e.g., N4) directly because it may not contain some optional items, which may cause loss of data items in upper level matches. Note also that a prototype may consist of multiple child nodes (not just one as N4) as a single data record may consist of multiple child nodes. To produce the prototype, in general we need to perform multiple alignments of data records [13]. However, we use a simpler method based on the extracted data (see below).

## Put Data in Tables and Generate Prototypes: PutDataInTables() and GenPrototypes()

PutDataInTables (Fig. 7) puts the linked data items in tables (as there may be more than one data areas with different structures). A table here (*DataTable* in Fig. 7) is a linked list of one dimensional arrays, which represent columns. This data structure makes it easy to insert columns in appropriate locations for optional data items.

```
PutDataInTables(Node)
1  Children = FindFirstReturnRest(Node.Children); // find the first child node of a data area
     and return all child nodes from it onward
2   Tables = {};      // multiple data areas may exist under Node
3   DataTable = create a table of suitable size; // store data items
4   Row = 0;          // DataTable row that a data item will be inserted in
5   for each Child in Children in sequence do
6       if Child has aligned data items then
7          if a link to Child exists from an data item in an earlier column then  // a variable
               can keep track of the current column
8             Row = Row + 1;
9          for each data item d in Child do
10            if d does not have in-coming link then
11               add a new column and insert d in Row;
12            else Insert d in Row and the same column as the data item with an out-going
                    link pointing to d.
13      else     Tables = Tables ∪ {DataTable};
14           if no aligned data in the rest of Children nodes then
15              exit loop
16           else DataTable = create a table of suitable size;
17                Row = 0;
```

**Fig. 7: The PutDataInTables procedure**

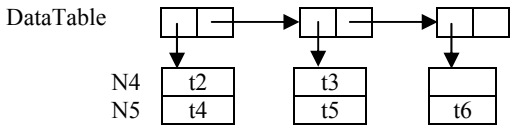


**Fig. 8: DataTable for N4 and N5 (children of N2)**

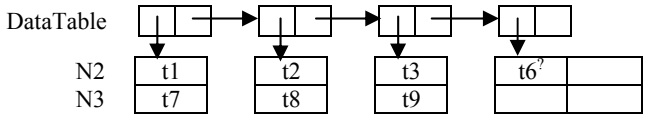


**Fig. 9: DataTable for N2 and N3 (children of N1)**

| | | | |
|---|---|---|---|
| t1 | t2 | t3 | |
| t1 | t4 | t5 | t6 |
| t7 | t8 | t9 | |

**Fig. 10: Final output (children of N1)**

The basic idea of the algorithm is as follows: All linked data items are put in the same column (line 9-12). A new row is started if an item is being pointed to by another item in an earlier column (line 7). For example, for node N2 in Fig. 5 this procedure produces the *DataTable* in Fig. 8. For node N1 in Fig. 6, it produces the *DataTable* in Fig. 9. Here we only use *N4, but not N5 (in Fig. 5). PutDataInTables() is a linear algorithm.

After putting data in tables, producing prototypes from each table is fairly simple. GenPrototypes() follows the tree structure based on the first data record (e.g., N4 in Fig. 5) and inserts those tree paths representing optional items not in the first data record, but in other data records. The optional items are clear from the table because they occupy some columns that do not have data items in the first data record. In the example of Fig. 9, $t6^{?}$ is added to N4 as an optional item which gives *N4 (the proto-

**Table 1: Experimental results**

| URL | DEPTA | | NET | |
|---|---|---|---|---|
| | Wr. | Corr. | Wr. | Corr. |
| **Without Nesting** | | | | |
| http://accessories.gateway.com/ | 0/0 | 15/15 | 0/0 | 15/15 |
| http://google1-cnet.com.com/ | 0/0 | 180/180 | 0/0 | 180/180 |
| http://google-zdnet.com.com/ | 0/0 | 80/80 | 0/0 | 80/80 |
| … | … | … | … | … |
| http://sensualexpression.com/ | 0/0 | 12/12 | 0/0 | 12/12 |
| http://www.shopping.com | 0/0 | 35/35 | 0/0 | 35/35 |
| http://www.tigerdirect.com/ | 0/0 | 70/70 | 0/0 | 70/70 |
| Recall | 97.15% | | 98.99% | |
| Precision | 99.37% | | 98.92% | |
| **With Nesting** | | | | |
| http://froogle.google.com/ | 12/0 | 124/136 | 0/0 | 136/136 |
| http://www.cooking.com/ | 0/15 | 48/63 | 1/0 | 62/63 |
| http://www.kmart.com/ | 38/0 | 0/38 | 0/0 | 38/38 |
| http://www.rei.com/ | 45/26 | 32/103 | 2/0 | 101/103 |
| http://www.sonystyle.com/ | 78/23 | 0/101 | 0/0 | 101/101 |
| http://www.target.com/ | 0/43 | 36/79 | 0/0 | 79/79 |
| http://www.walmart.com/ | 50/0 | 28/78 | 0/0 | 78/78 |
| http://www1.us.dell.com/ | 189/0 | 32/221 | 0/0 | 221/221 |
| Recall | 36.63% | | 99.63% | |
| Precision | 42.13% | | 100% | |

type). In Fig. 6, we can also see that $t6^?$ is attached to *N4.

### Output Data and Prototypes: Output()

Output() outputs the final data tables, which are tables not covered by any upper level tables. For example, Fig. 10 shows the final table for the tree in Fig. 5. Note that the nested data are expanded and included. The procedure also outputs the final prototypes that can be used to extract data from similar pages from the same site.

## 5 Empirical Evaluation

This section evaluates NET. We compare it with the most recent system DEPTA [13], which does not find nested data records. We show that for flat data records, NET performs as well as DEPTA. For nested data records, NET also performs very well. Our experimental results are given in Table 1.

Column 1 lists the site of each test page. Due to space limitations, we could not list all of them here. The number of pages that we used in our experiments is 40. The first 32 pages (without nesting) are from DEPTA. The last 8 pages all contain nested data records. We did not collect more nested pages for testing because such pages are relatively rare and quite difficult to find.

Columns 2 and 4 give the numbers of data items extracted wrongly (Wr.) by DEPTA and NET from each page respectively. In x/y, x is the number of extracted results that are incorrect, and *y* is the number of results that are not extracted. The tree similarity threshold is 0.7, which is the default of our system and is used in all our

experiments. Columns 3 and 5 give the numbers of correct (Corr.) data items extracted by DEPTA and NET from each page respectively. Here, in x/y, x is the number of correct items extracted, and y is the number of items in the page.

From the table, we can observe that without nesting the results of DEPTA and NET are both accurate. With nesting, NET is much better. DEPTA still can correctly extract some data items because not all data records in the pages have nested records. The precision and recall are computed based on the extraction of all pages.

## 6 Conclusions

In this paper, we proposed a more effective technique to perform automatic data extraction from Web pages. Given a page, our method first builds a tag tree based on visual information. It then performs a post-order traversal of the tree and matches subtrees in the process using a tree edit distance method and visual cues. Our method enables accurate alignment and extraction of both flat and nested data records. Experimental results show that the method performs data extraction accurately.

**Acknowledgements**: This work was partially supported by National Science Foundation (NSF) under the grant IIS-0307239.

## References

1. Arasu, A. and Garcia-Molina, H. Extracting Structured Data from Web Pages. *SIGMOD'03,* 2003.
2. Buttler, D., Liu, L., Pu, C. A fully automated extraction system for the World Wide Web. *IEEE ICDCS-21*, 2001.
3. Chang, C. and Lui, S-L. IEPAD: Information extraction based on pattern discovery. *WWW'10*, 2001.
4. Crescenzi, V. Mecca, G. Merialdo, P. Roadrunner: towards automatic data extraction from large web sites. *VLDB'01*, 2001.
5. Kushmerick, N. Wrapper induction: efficiency and expressiveness. *Artificial Intelligence*, 118:15-68, 2000.
6. Lerman, K., Getoor L., Minton, S. and Knoblock, C. Using the Structure of Web Sites for Automatic Segmentation of Tables. *SIGMOD'04*, 2004.
7. Liu, B., Grossman, R. and Zhai, Y. Mining data records from Web pages. *KDD'03*, 2003.
8. Muslea, I., Minton, S. and Knoblock, C. A hierarchical approach to wrapper induction. *Agents'99*, 1999.
9. Pinto, D., McCallum, A., Wei, X. and Bruce, W. Table extraction using conditional random fields. *SIGIR'03*, 2003.
10. Reis, D. Golgher, P., Silva, A., Laender, A. Automatic Web news extraction using tree edit distance, *WWW'04*, 2004.
11. Wang, J.-Y., and Lochovsky, F. Data extraction and label assignment for Web databases. *WWW'03*, 2003.
12. Yang, W. Identifying syntactic differences between two programs. *Softw. Pract. Exper.*, 21(7):739–755, 1991.
13. Zhai, Y and Liu, B. Web data extraction based on partial tree alignment. *WWW'05*, 2005.
14. Zhai, Y and Liu, B. Extracting Web data using instance-based learning. To appear in *WISE'05*, 2005.
15. Zhao, H., Meng, W., Wu, Z., Raghavan, V., Yu, C. Fully automatic wrapper generation for search engines. *WWW'05*, 2005.