Automatic Web Page Segmentation and Noise Removal for Structured Extraction Using Tag Path Sequences

Roberto Panerai Velloso, Carina F. Dorneles

Universidade Federal de Santa Catarina, Brazil
{rvelloso, dorneles}@gmail.com

Abstract. Web page segmentation and data cleaning are essential steps in structured web data extraction. Identifying a web page main content region, removing what is not important (menus, ads, etc.), can greatly improve the performance of the extraction process. We propose, for this task, a novel and fully automatic algorithm that uses a tag path sequence (TPS) representation of the web page. The TPS consists of a sequence of symbols (string), each one representing a different tag path. The proposed technique searches for positions in the TPS where it is possible to split it in two regions where each region's alphabet do not intersect, which means that they have completely different sets of tag paths and, thus, are different regions. The results show that the algorithm is very effective in identifying the main content block of several major websites, and improves the precision of the extraction step by removing irrelevant results.

Categories and Subject Descriptors: H.2.8 [**Database Management**]: Database Applications—*Data mining*; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*Information filtering*; I.1.2 [**Symbolic and Algebraic Manipulation**]: Algorithms—*Analysis of algorithms*

Keywords: noise removal, page segmentation, structured extraction, web mining

1. INTRODUCTION

One crucial step in web data mining, including structured extraction, is the cleaning phase that takes place before extracting the information. One cannot expect to get good results in the extraction phase without cleaning and removing the undesired noise first. Yi et al. [2003] mention that despite the importance of this task, relatively little work has been done in this area and, while reviewing up to date related work, we still have the impression that this is an underdeveloped field. Moreover, according to Liu and Chen-Chuan-Chang [2004], noise can seriously harm web data mining.

In structured extraction, most of the existing approaches use some sort of pattern recognition to identify the records (as defined by Liu et al. [2003]) present in the page. The problem is that, usually, we are interested only in the main content region, as depicted in Figure 4, but other regions of the page (menus, ads, etc.) often contain repeating patterns that are outputted as noise results. So, it is useful to cleanup a web page before extracting the records from it.

Currently, some of the work on noise removal and page segmentation are aimed at page indexing and clustering (i.e. they assume the main region is textual) such as the ones by Fernandes et al. [2011] and Yi et al. [2003] and, due to intrinsic differences between unstructured data and structured data, these cannot be used for structured extraction. Existing techniques that can be used for structured content either require a priori definitions [Cai et al. 2003], prior training [Chakrabarti et al. 2008], or they rely on specific HTML tags or aspects of the HTML language to work [Cho et al. 2009].

In this article, we propose and lay down a simple, computationally efficient, yet very powerful algorithm aimed at web page segmentation, noise removal and main content identification, based on

the tag path sequence of the web page. It is a general segmentation technique, presented here in the context of structured extraction, that takes into account the page's style and structure. Our main contributions are:

—Fully automatic: no training or human intervention needed;
—Domain independent: it is only required that a page contains structured content, no matter what domain it is about;
—HTML syntax independent: there are no rules defined for specific HTML tags;
—Works on single page: it requires only one page as input, which is a main advantage as discussed by Liu and Chen-Chuan-Chang [2004];
—Can be combined with extraction techniques: due to the way pruning is carried out (preserving tree structure), this algorithm can be combined with any structured extraction algorithm;
—Extraction optimization: the proposed algorithm prunes an average of 46.22% of the DOM tree, in linear time, avoiding the processing of this noise by the subsequent extraction algorithm.

To evaluate how effective our approach is, we have compared the output of MDR [Liu et al. 2003], a well-known structured extraction technique, against the output of MDR combined with our technique, as illustrated in Figure 1, yielding an average of 77.03% of noise removed from the test web pages.

This article is organized as follows. In Section 2, we give a brief survey of related work in segmentation, pointing out the differences between each one and our proposal. In Section 3, some basic definitions are given, which are needed for the problem definition and the understanding of the algorithm. In Section 4, we state the problem and explain the two hypothesis that are the basis on which we develop the proposed solution for the problem of segmentation and noise removal, targeted at structured extraction. In Section 5, a detailed description of the full algorithm and its complexity are given. In Section 6, we present the results of the tests done so far. Finally, in Section 7, a conclusion is given and possible future developments are outlined.

2. RELATED WORK

There are several work proposing ways to segment web pages and identify what is noise and what is informative content in them. We grouped them in three different categories: those based on text content, those based on the DOM tree and those that make use of visual information.

Text content based approaches. In the work of Fernandes et al. [2007], Kohlschütter and Nejdl [2008], Kohlschütter et al. [2010], Weninger et al. [2010] and Hu et al. [2013] the segmentation is done using the text content of the web page. The focus of these work, however, is not on structured extraction, but instead, on indexing and clustering of web sites. The majority of the work about page cleaning and noise removal are aimed at this kind of applications.

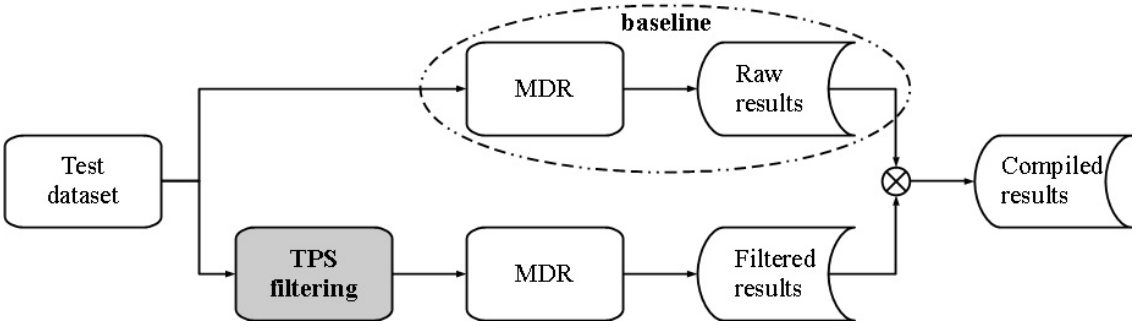

Fig. 1. Evaluation method adopted.

DOM tree based approaches. In the work of Yi et al. [2003], Chakrabarti et al. [2008], Cho et al. [2009], Fernandes et al. [2011], Zheng et al. [2012], Zheng et al. [2007] the segmentation is done using the DOM tree and, thus, they do take into account the web page's structure. However Fernandes et al. [2011] and Yi et al. [2003] require several pages from the same web site, as they are site-driven techniques. Chakrabarti et al. [2008] and Zheng et al. [2007] propose a training framework that requires a manually labeled data set to work. Cho et al. [2009] is dependent of a tag dictionary, defined a priori, to build a visual representation of the page. Finally, Zheng et al. [2012] requires a database of terms associated to "semantic roles" in order to detect data-rich regions.

Visual information based approaches. Besides text and DOM tree based techniques, there are the ones based on visual information [Cai et al. 2003; Simon and Lausen 2005; Liu et al. 2010]. They all rely on a web browser's renderer to obtain the visual information used for segmentation, what can be computationally expensive, and beyond that, the approach of Cai et al. [2003] is based on quite a large set of strong heuristic rules, each one applied to specific HTML tags. Approaches based on specific HTML tags have a serious disadvantage of being affected by changes in web page design practices and HTML syntax changes.

Structured extraction techniques. There are a number of techniques proposed to address the problem of structured extraction [Liu and Zhai 2005; Crescenzi et al. 2001; Liu et al. 2003; Miao et al. 2009; Xie et al. 2012]. The reason we chose MDR for the evaluation of our proposal is due to the level of detail provided by the publications (which allows for implementation) and availability of independent implementations. Since we are measuring only the noise suppressed in the output, and not the quality of the extraction itself, any pattern detection algorithm that complies with our constraints (fully automatic, works on single page, no training, no labeling, etc.), would suffice.

The representation of the web page used in our work (tag path sequence) was also employed by Miao et al. [2009] and Xie et al. [2012], although in both cases for structured extraction, not for segmentation. We cite them here to show that, according to their results, this representation, just like the DOM tree, is also able to expose the web page's structure and, thus, is suitable for the purpose of our work.

3. BASIC DEFINITIONS

Now we present the concepts and definitions used to state the problem in Section 4 and outline the proposed algorithm in Section 5, as well as an example to illustrate each definition.

Definition 3.1 ***DOM tree***. The DOM tree is a hierarchical structure, derived from the parsing of HTML code that represents a web page.

In Figure 2 we use a small piece of HTML code to illustrate the DOM tree and the next definitions.

Definition 3.2 ***Tag path***. A tag path (TP) is a string describing the absolute path from the root of the DOM tree to a given node. Let i be the depth-first position, in the DOM tree, of a $node_i$, then we say that the tag path TP_i is a string describing the path from the root of the DOM tree to the $node_i$.

In Figure 2, the absolute tag path TP_4 from the node *body* to the table cell node td_4 is TP_4 = "*body/table/tr/td*".

Definition 3.3 ***Tag path sequence***. We define the tag path sequence (TPS) of a DOM tree with n nodes to be the ordered sequence $TPS[1..n] = (TP_1, TP_2, TP_3, ..., TP_{n-1}, TP_n)$ where two tag paths TP_i and TP_j, with $i \neq j$, are considered equal only if their paths and style definitions are equal, otherwise they are different.

This is the same definition of Xie et al. [2012], where each different tag path is represented in the sequence by a symbol, except that here we incorporate style definitions when comparing tag paths. In Figure 2 we show the TPS for the given HTML code, where each TP is assigned a code, yielding $TPS = (1, 2, 3, 4, 4, 3, 4, 4)$.

Definition 3.4 ***Alphabet of the TPS***. Let Σ_a be a set containing all the symbols in a given sequence TPS_a of size n, we say that Σ_a is the alphabet of TPS_a defined as $\Sigma_a = \{\alpha | \exists TPS_a[i] = \alpha \wedge 1 \leq i \leq n\}$, where α is a symbol in the alphabet.

Informally speaking, the alphabet indicates all distinct symbols in a TPS. In Figure 2, the TPS is formed only by the symbols "1","2", "3" and "4", so its alphabet is $\Sigma = \{1, 2, 3, 4\}$.

Definition 3.5 ***Tag path frequency set***. Let (s, f) be a pair where s is a symbol from an alphabet of a given TPS and f is the number of times that s appears in the TPS, so we define the tag path frequency set as the set containing all possible (s, f) pairs of a TPS. Let $FS = \{(s_1, f_{s1}), (s_2, f_{s2}), (s_3, f_{s3}), \ldots, (s_{n-1}, f_{sn-1}), (s_n, f_{sn})\}$, where n is the size of the TPS.

In Figure 2, symbol "1" shows up once in the sequence, symbol "2" once too, symbol "3" twice and symbol "4" four times, so for this sequence the tag path frequency set is equal to $FS = \{(1, 1), (2, 1), (3, 2), (4, 4)\}$. The set FS is a mapping between every symbol of an alphabet and its corresponding frequency.

Definition 3.6 ***Frequency thresholds***. Given a TPS_a with alphabet Σ_a, tag path frequency set FS_a, we define the frequency thresholds FT_a to be the ordered set containing only the frequencies of FS_a. Let $FT_a = \{f | \exists (s, f) \wedge (s, f) \in FS_a \wedge s \in \Sigma_a\}$, where f is a frequency, s is the corresponding symbol of the alphabet Σ_a.

In the TPS from Figure 2, the tag path frequency set is $FS = \{(1, 1), (2, 1), (3, 2), (4, 4)\}$, in this case the frequency thresholds is equal to $FT = \{1, 2, 4\}$ because symbols "1" and "2" both have frequency equal to **1**, symbol "3" has frequency equal to **2** and symbol "4" has frequency equal to **4**. The FT set is need to filter out symbols from the TPS. If we have a set $FT = \{1, 2, 4\}$, there is no point in filtering symbols with $f = 3$, because there is none in the sequence.

Definition 3.7 ***Region***. Let a tag path sequence TPS be a concatenation of two other sequences $TPS = TPS_a.TPS_b$, we say that TPS_a and TPS_b are regions of TPS, iff $\Sigma_a \cap \Sigma_b = \emptyset$.

In Figure 2 if we divide the TPS in two subsequences $TPS_a = TPS[1..2] = (1, 2)$ and $TPS_b = TPS[3..8] = (3, 4, 4, 3, 4, 4)$, with alphabets $\Sigma_a = \{1, 2\}$ and $\Sigma_b = \{3, 4\}$, we say that TPS_a and TPS_b are distinct regions of TPS, because $\Sigma_a \cap \Sigma_b = \emptyset$.

4. PROBLEM FORMULATION

Given the definitions presented in the previous section, we formulate next the problem of page segmentation and noise removal, based on the following assumptions:

(1) different regions of a web page are described using different tag paths, so these regions will have different alphabets; and
(2) in web sites with semi-structured content (i.e. records, as defined by Liu et al. [2003]), the main region is structurally denser than the others (menus, ads, text, etc.).

The basis for assumption (1) comes from the observation that the regions of a web page are different ramifications in the DOM tree and these regions are described either using different tags for each one

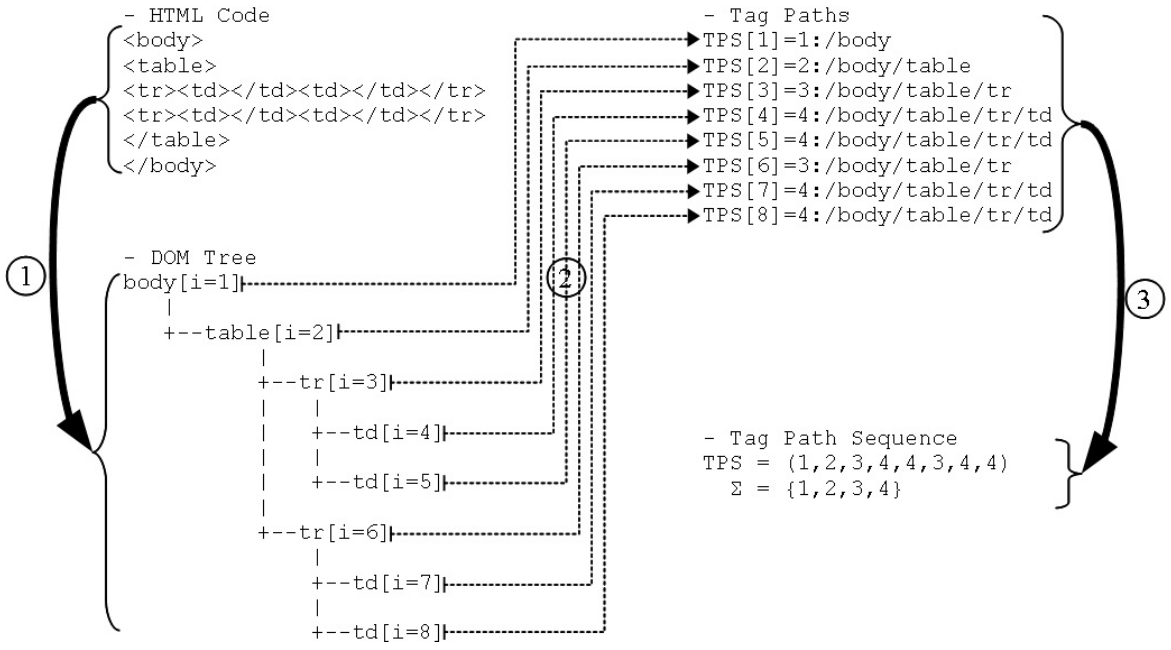

Fig. 2. An example of a TPS being built from an HTML code.

or, if the tags are the same, with different styles, so that they can easily be distinguished by the user. If all regions of a page look alike, it gets more difficult, for the user, to tell them apart. Then, from Definition 3.3 we can see that the set of symbols used in each region of a web page should be different, and so it should be possible to segment a page using Definition 3.7.

The assumption (2) comes from the context in which we apply the page segmentation proposed in this work (i.e. structured extraction). Since we are segmenting only pages containing records, and we know that in order to describe the structure of these records, in HTML, we need more nodes of the DOM tree than for unstructured data (i.e. text), it is reasonable to assume that, for a page containing records, the main region is the largest one (i.e. the one with more nodes).

Now, using the definitions in Section 3 and the above assumptions, we can state the problem of web page segmentation and main content identification to be the following: **"find the largest** *region* **in the** TPS **of a web page that has an** *alphabet* **that does not intersect with the** *alphabet* **of other smaller regions"**.

One **crucial** detail that has to be taken into account, is that there may be tag paths in a page that represent structural divisions of it (i.e. web site's visual formatting). These tag paths, if they are divisions, will show up a few times throughout the entire sequence, preventing us from finding a split, in the TPS, where the alphabets of the two parts of the sequence do not intersect. To remove this *noise* from the TPS, we filter out, iteratively, all symbols with lower frequencies. This way we can avoid this problem without harming the segmentation process, because the tag paths with higher frequencies are still being considered.

For illustration purposes, we give now an example of a web page starting and ending with the same tag path ("/body/br") and with three regions delimited by the same tag path ("/body/div"). Assuming that different tag paths are used to describe each region, without filtering out low frequency tag paths from the TPS it would not be possible to split the sequence into regions.

—HTML code

```
<body>
<br>
<div><span class='region1'></span>...<span class='region1'></span></div>
<div><span class='region2'></span>...<span class='region2'></span></div>
<div><span class='region3'></span>...<span class='region3'></span></div>
<br>
</body>
```

—TPS

```
TPS = (1,2,3,4,...,4,3,5,...,5,3,6,...,6,2)
```

The symbols 2 and 3 appear along the entire TPS.

—Filtered TPS

```
TPS = ( , , ,4,...,4, ,5,...,5, ,6,...,6, )
```

Only symbols with frequency higher than 3 are considered in the segmentation process. Now it is possible to split the TPS into regions.

5. ALGORITHMS' DESCRIPTION

In this section we present the algorithms we have developed to address the problem stated in Section 4. They are the following:

—***tagPathSequenceFilter()***. It is the main algorithm, which receives a HTML file as input and returns a pruned DOM tree with the main content region;

—***convertTreeToSequence()***. It converts the web page DOM tree into a tag path sequence;

—***searchRegion()***. It is the actual search for the main region of the TPS;

—***filterAlphabet()***. It filters an alphabet, removing lower frequency symbols, making the overall algorithm more robust and resistant to noise;

—***pruneDOMTree()***. It prunes the original DOM tree, leaving only the main content region reported by *searchRegion*, keeping the original structure of the document.

5.1 $tagPathSequenceFilter()$ Algorithm

Algorithm 1 Filters out noise from a web page

Input: $inputFile$ - an HTML file
Output: pruned $inputFile$'s DOM tree

```
1: procedure tagPathSequenceFilter(inputFile)
2:     DOMTree ← parseHTML(inputFile)
3:     convertTreeToSequence(DOMTree.body," ", tagPathSequence)
4:     searchRegion(tagPathSequence)
5:     pruneDOMTree(DOMTree.body, tagPathSequence)
6:     return DOMTree
7: end procedure
```

The procedure $tagPathSequenceFilter()$ in Algorithm 1 returns the main content region of $inputFile$. The procedure $parseHTML()$, in Line 2, converts the HTML code into a DOM tree representation; $convertTreeToSequence()$, in Line 3, converts the DOM tree into a TPS; the procedure $searchRegion()$, in Line 4, recursively searches for the largest part of the TPS that has a unique

alphabet and, finally; *pruneDOMTree*(), in Line 5, prunes out of the DOM tree every node that is not in the resulting TPS, preserving the structure of the returned document in Line 6.

Bellow we detail the algorithms *convertTreeToSequence*(), *searchRegion*(), $filterAlphabet()$ and *pruneDOMTree*(). The algorithm $parseHTML()$ is not in the scope of our work and so, will not be discussed here.

5.2 *convertTreeToSequence*() Algorithm

Algorithm 2 Converts a DOM tree to a tag path sequence representation

Input: *node* - a node from the DOM tree, initially the root of the tree
Input: *tagPath* - the previous tag path, initially empty
Input: *tagPathSequence* - the TPS built from the DOM tree, initially empty
Output: the TPS for the given DOM tree stored in *tagPathSequence*

```
1: procedure convertTreeToSequence(node, tagPath, tagPathSequence by reference)
2:     tagPath ← concatenate(tagPath,"/", node.tag, node.style)
3:     if tagPath ∋ tagPathMap then
4:         tagPathMap ← tagPathMap + {tagPath}
5:         tagPathMap[tagPath].tagPathCode ← tagPathMap.size
6:     end if
7:     tagPathSequence ← concatenate(tagPathSequence, tagPathMap[tagPath].tagPathCode);
8:     for each child of node do
9:         convertTreeToSequence(child, tagPath, tagPathSequence)
10:    end for
11: end procedure
```

The procedure *convertTreeToSequence*() in Algorithm 2 converts a web page from its DOM tree representation to a TPS representation, traversing the DOM tree in depth-first order. It is initially called in Algorithm 1 with an empty *tagPath* parameter, which represents the previous tag path string (from the previous recursive call). In Line 2, the previous tag path is concatenated with the current tag, as well as with its style definition, in order to distinguish repeated paths with different styles; in Line 3, it is checked whether or not the current tag path has been seen before ($tagPathMap$ is used for this purpose) and, if not, in Line 4, it is inserted into the set $tagPathMap$ and a new code assigned to it in Line 5, as stated in Definition 3.3; in Line 7, the tag path code is appended to the end of the sequence and, finally, the procedure is called recursively in Line 9 for each child of *node*.

5.3 *searchRegion*() Algorithm

This is the core algorithm, since it is responsible for finding the main content region, so we have provided na illustration, in Figure 3, to help understand its workings. In Figure 3, for clarity purposes, we have omitted alphabet filtering in order to keep it simple and easy to understand the main idea behind the *searchRegion*() algorithm.

Algorithm 3 Search for regions in the TPS with different alphabets

Input: *tagPathSequence* - the TPS of a given page
Output: the main region of the TPS, stored in *tagPathSequence*

```
1: procedure searchRegion(tagPathSequence[1..n] by reference)
2:     alphabet ← ∅
3:     t ← 0
```

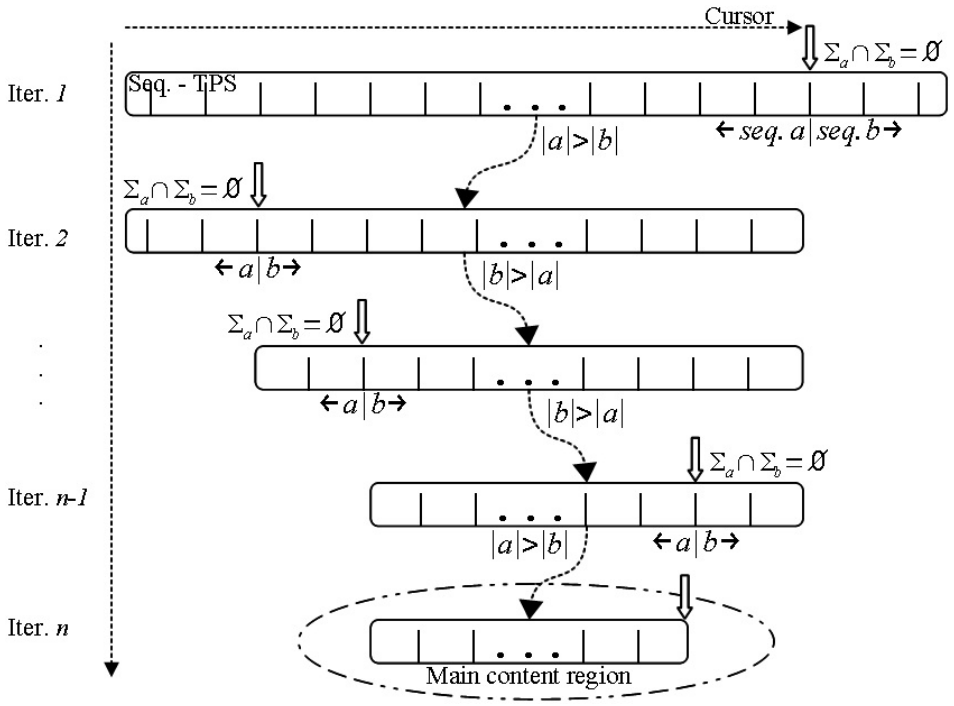

Fig. 3. Illustration of procedure *searchRegion*().

```
4:   for i ← 1..n do
5:       symbol ← tagPathSequence[i]
6:       if symbol ∋ alphabet then
7:           alphabet ← alphabet ∪ {symbol}
8:           symbolCount[symbol] ← 0
9:       end if
10:      symbolCount[symbol] ← symbolCount[symbol] + 1
11:  end for
12:  thresholds ← OrderedSetOfFrequencies(symbolCount)
13:  regionFound ← false
14:  while not regionFound do
15:      t ← t + 1
16:      currentAlphabet ← filterAlphabet(alphabet, symbolCount, thresholds[t])
17:      if currentAlphabet.size < 2 then
18:          break
19:      end if
20:      currentSymbolCount ← symbolCount
21:      regionAlphabet ← ∅
22:      for i ← 1..n do
23:          symbol ← tagPathSequence[i]
24:          if symbol ∈ currentAlphabet then
25:              regionAlphabet ← regionAlphabet ∪ {symbol}
26:              currentSymbolCount[symbol] ← currentSymbolCount[symbol] − 1
27:              if currentSymbolCount[symbol] = 0 then
28:                  currentAlphabet ← currentAlphabet − {symbol}
29:                  if currentAlphabet ∩ regionAlphabet = ∅ then
30:                      if currentAlphabet ≠ ∅ and (n − 2 ∗ i)/n > 0.20 then
```

```
31:                         regionFound ← true
32:                     end if
33:                     break
34:                 end if
35:             end if
36:         end if
37:     end for
38: end while
39: if regionFound then
40:     if i < n/2 then
41:         tagPathSequence ← tagPathSequence[i + 1..n]
42:     else
43:         tagPathSequence ← tagPathSequence[1..i]
44:     end if
45:     searchRegion(tagPathSequence)
46: end if
47: end procedure
```

The procedure *searchRegion*() in Algorithm 3 computes the TPS alphabet and corresponding symbol frequency from Lines 4 to 11; in Line 12, the frequency thresholds, from Definition 3.6, are computed; from Lines 14 to 38 the actual search is performed for a position in the TPS where a split is possible (i.e. where a region exists); in Line 15 the frequency thresholds are iterated; in Line 16 the TPS alphabet, from Definition 3.4, is filtered, as described in Section 4; in Line 22 the TPS is iterated; in Line 25 the region alphabet is computed and; from Lines 27 to 35 it is checked if there is no intersection between the alphabets of the two portions of the TPS (an empty intersection indicates that a possible region was found, as in Definition 3.7). The found region is only reported if it is at least 20% larger than the rest of the sequence, otherwise we continue iterating the frequency thresholds. This percentage is actually a parameter and its purpose is to avoid reporting a region under ambiguous conditions (in the experiments we used the value of 20%); finally from Lines 39 to 46 the TPS is split if a region was found, calling *searchRegion*() recursively in line 45, if so.

5.4 *filterAlphabet*() Algorithm

Algorithm 4 Filters out symbols with lower frequencies from the alphabet

Input: *alphabet* - the alphabet ot be filtered
Input: *symbolCount* - the tag path frequency set (FS) of the alphabet
Input: *threshold* - a frequency threshold
Output: a filtered alphabet

```
1: procedure FILTERALPHABET(alphabet, symbolCount, threshold)
2:     filteredAlphabet ← ∅
3:     for i ← 1..n do
4:         if symbolCount[alphabet[i]] ≥ threshold then
5:             filteredAlphabet ← filteredAlphabet ∪ {alphabet[i]}
6:         end if
7:     end for
8:     return filteredAlphabet
9: end procedure
```

The procedure *filterAlphabet*() in Algorithm 4 removes from *alphabet*, every symbol with frequency lower than *threshold*. in Lines 3 to 7 only the symbols with frequency greater or equal to *threshold*

are inserted in the resulting set. The result of $filterAlphabet()$ is used in Algorithm 3, Line 24, where only the symbols in $filteredAlphabet$ are considered while searching for a region.

5.5 *pruneDOMTree*() Algorithm

Algorithm 5 Prune from the DOM tree the nodes that are not in sequence

Input: *node* - a node from the DOM tree to be pruned, initially the root of the tree
Input: *sequence* - the TPS that has to remain in the DOM tree
Output: the DOM tree pointed by *node* pruned

```
1: procedure PRUNEDOMTREE(node by reference,sequence)
2:     for each child of node do
3:         if pruneDOMTree(child, sequence) = true then
4:             remove child from node
5:         end if
6:     end for
7:     if node ∋ sequence and node.childCount = 0 then
8:         return true
9:     end if
10:    return false
11: end procedure
```

The procedure *pruneDOMTree*() in Algorithm 5, traverses the DOM tree, depth first, removing the nodes that do not belong to *sequence*. In Line 3 the DOM is traversed; in Lines 7 to 9 it is decided whether or not *node* should be removed.

A node is removed from the tree, only if it is not in *sequence* **and** has no children. This way we keep the structure of the remaining tree intact, in order not to affect the subsequent structured extraction phase.

5.6 Algorithm's Complexity

As for the algorithm's complexity, if we observe Lines 14 and 22 of the procedure *searchRegion*(), we can see that the loop in Line 14 iterates the frequency thresholds until a region is found and Line 22 iterates the TPS (filtered at given frequency threshold) also until a region is found and, if so, the reported region is recursively processed.

In the worst case, when the alphabet intersection is empty only in the last index of the TPS, the complexity would be at most $O(n^2 f)$, where n is the length of the TPS and f is the size of the set *thresholds*. In practice, the size of the set *thresholds* is much smaller than the length of the TPS, so we can say the complexity approximates $O(n^2)$ as shown in Equation 1.

$$T(n) = T(n-1) + \Theta(n) \Longrightarrow \sum_{i=1}^{n} i = \frac{n(n+1)}{2} = O(n^2) \tag{1}$$

In average, if the TPS gets split in half, the complexity would be $O(n)$ as in Equation 2.

$$T(n) = T(n/2) + \Theta(n) \Longrightarrow \sum_{i=1}^{log_2^n} \frac{n}{2^i} = n - 1 = O(n) \tag{2}$$

In the best case, TPS is split in the first index, yielding $O(n)$ as in Equation 3.

$$T(n) = T(n-1) + \Theta(1) \Longrightarrow \sum_{i=1}^{n} 1 = n = O(n) \tag{3}$$

In real world scenarios, as we have seen while doing the evaluation of the algorithm, the sequences get split approximately four of five times until they cannot be split no more. So we can say that in real cases, the algorithm executes in $O(in)$ time, where n is the size of the TPS and i is the number of times the sequence gets split, which we can consider as a small constant, in this case, and say that it runs in $O(n)$.

6. EXPERIMENTAL RESULTS

In this section we describe and discuss the results of our experiments and how they are presented. To obtain the results presented in Subsection 6.2, we have implemented the algorithm and tested it against some commercial and institutional web sites. In Subsection 6.1 we detail one of the results presented, as an example, to clarify how they are compiled in Table I.

6.1 Experimental Setup

We considered the extraction results of MDR alone as our baseline to be compared with the results obtained by the combined use of TPS filtering and MDR, as illustrated in Figure 1.

When applying both approaches (MDR and TPS filtering+MDR) to a result page of YouTube web site, the following results are obtained:

—raw web page (i.e. the original page, without TPS filtering)
  —DOM tree processed: 1424 nodes;
  —MDR results: 82 records total (62 noise / 20 targets);

—pruned web page (i.e. the web page after TPS filtering)
  —DOM tree processed: 674 nodes, size $47,33\%$ of the original page, reduction of $(-52,67\%)$
  —MDR results: 20 records total (0 noise / 20 targets), noise removed 100%

In this result, we can see an improvement in the extraction of records as well as a considerable reduction in the size of the DOM tree to be processed. A percentage of 52.67% of the DOM tree was pruned without losing the target records in the process. Everything pruned out of the DOM tree was noise. Figure 4 illustrates the web page and the main content region.

Without applying TPS filtering, we get 82 records in total and, since we know there are 20 target records in this page, we can consider the value of 62 records to be 100% of noise to be removed. When we use TPS filtering, this time we get only the 20 target records in the extraction phase, yielding a precision of 100%, which means all noise was removed in this case. We calculate the percentage of noise removed to be

$$NoiseRemoved = 1 - \frac{NumRec_{totalTPS} - NumRec_{targetTPS}}{NumRec_{total} - NumRec_{target}} \tag{4}$$

Where $NumRec_{total}$ and $NumRec_{target}$ are the total number of records and the number of target records, respectively, from the original web page, and $NumRec_{totalTPS}$ and $NumRec_{targetTPS}$ are the total number of records and the number of target records, respectively, from the filtered web page.

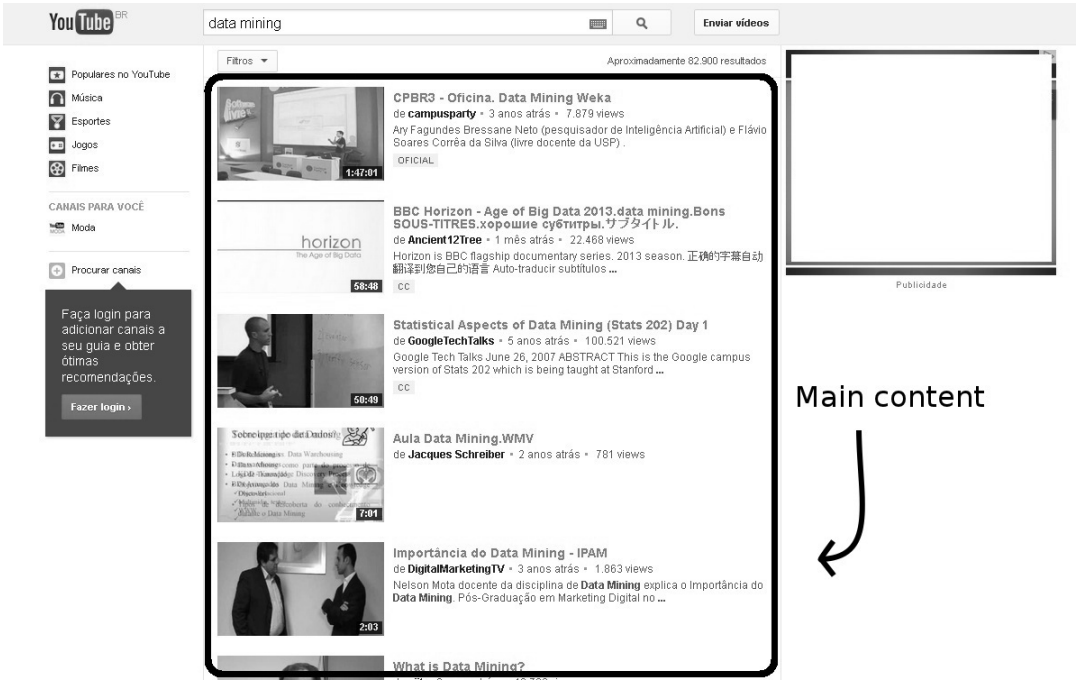

Fig. 4. A page from the YouTube web site and the main content region delimited.

6.2 Results

In Table I we present, in the first three columns, the size of the DOM tree processed by MDR and the reduction obtained after filtering. The column "Content Present" indicates whether or not the filtering process preserved the main content region. The next four columns are the results of MDR alone and combined with TPS filtering, showing the total records and target records extracted for both approaches. The last column shows the percentage of noise removed, calculated using Equation 4.

As we can see in Table I, the total of column "Content Present" indicates that the algorithm has worked in 86.96% of the sites and it has removed, for this test set, an average of 77.03% of all noise present in the data, as shown by the average of column "Noise rem.". We consider these to be good results.

The average DOM tree reduction of 46.22% is an interesting result. First, because that means almost half the DOM tree is noise in average. Second, because this number matches the value reported, independently, by Gibson et al. [2005] as page template size (between 40% and 50%), corroborating with literature work.

An interesting situation we can see in Table I is the result for the site "g1.com.br". Without filtering, MDR has reported a total of 225 records, included 10 target records. After filtering is applied, a total of 202 records are reported, none of them targets, all noise. So, after filtering, if we had reported the complementary DOM tree instead, we would get a result of 23 records in total ($202 - 225 = 23$), included here the 10 target records, which is an excellent result since it gives us a 93.99% of noise removal. We can deduce from this, that the segmentation has worked just fine for this site, only the main content was not correctly identified, since it's relatively small.

Table I. Compiled results

	DOM size (# nodes)			Content	MDR (# records) Raw		Pruned		(eq. 4)
Site	Raw	Pruned	Reduction	present	Tot	Tgt	Tot	Tgt	Noise rem.
acm.org	601	340	-43.43%	Yes	61	10	16	10	88.24%
amazon.com	3309	1054	-68.15%	Yes	368	15	27	15	96.60%
americanas.com.br	2660	710	-73.31%	Yes	211	20	20	20	100.00%
bestbuy.com	3632	1425	-60.77%	Yes	299	15	15	15	100.00%
bondfaro.com.br	3897	3069	-21.25%	Yes	231	28	178	28	26.11%
bradesco.com.br	1913	1113	-41.82%	Yes	164	10	93	10	46.10%
buscape.com.br	3608	3514	-2.61%	Yes	279	24	266	24	5.10%
ebay.com	2623	1801	-31.34%	Yes	162	50	50	50	100.00%
elsevier.com	906	160	-82.34%	Yes	120	10	32	10	80.00%
g1.com.br	900	619	-31.22%	No	225	10	202	0	N/A
globo.com	400	193	-51.75%	Yes	80	10	20	10	85.71%
google.com	1421	981	-30.96%	Yes	118	11	61	11	53.27%
itau.com.br	1111	410	-63.10%	No	77	10	11	0	N/A
magazineluiza.com.br	3167	1115	-64.79%	Yes	314	40	44	40	98.54%
mercadolivre.com.br	2401	1771	-26.24%	Yes	136	50	52	50	97.67%
reuters.com	1202	480	-60.07%	Yes	136	10	54	10	65.08%
scopus.com	4929	4688	-4.89%	Yes	114	20	75	20	41.49%
submarino.com.br	2389	1268	-46.92%	Yes	116	20	22	20	97.92%
terra.com.br	869	588	-32.34%	Yes	122	50	76	50	63.89%
valor.com.br	514	126	-75.49%	No	55	10	2	0	N/A
webmotors.com.br	2119	1361	-35.77%	Yes	113	14	19	14	94.95%
yahoo.com	760	290	-61.84%	Yes	67	10	10	10	100.00%
youtube.com	1424	674	-52.67%	Yes	82	20	20	20	100.00%
Average/Total			-46.22%	86.96%/13.04%					77.03%

6.3 Results Discussion

There are three main situations where the algorithm needs to be improved but, fortunately, only two of these can lead to loss of main content (content removal). In Table I, column "Content Present", these two situations account for 13.04% of the cases, where the content region was removed in the filtering process.

(1) **templates too homogeneous**. These are pages with little difference between the regions. In this case, using this technique, there is not much to do. We simply do not have enough information to work with, since the entire page looks alike. We do not lose the target records, but the amount of noise removed is very low;

(2) **templates too heterogeneous**. These are pages where the main content is subdivided in more than one region. In this case, the main region gets split over and over, and only the largest part passes through the filter (and it might be noise). We propose a way to work around this problem later in this Section;

(3) **pages where the main content is smaller than the rest**. That is a consequence of the second assumption we made in Section 4: "the main region is denser/bigger than the rest". In this case, noise will always be reported as content. The same proposal made for the former situation can be used to deal with this one as well.

In the case of heterogeneous templates, TPS filtering can still be used if we make some slight modifications in the algorithm. One such case of heterogeneous template are "news sites", where every record has a different structure, but they are all records from the same domain (i.e. they belong to the same entity). In this specific situation, TPS segmentation could be used to split the page in several parts, and a semantic approach used to combine the regions, reporting the main content as a set of regions instead of only one.

For situation described for the site "g1.com.br" (that happened for two other sites we tested), when the content region is smaller than the rest, we could apply a semantic technique to check whether or not the desired content is present in the reported region, if not, report the complementary DOM tree (i.e. inverse the pruning) instead. The main algorithm would look like this:

Algorithm 6 Filters out noise from a web page

Input: $inputFile$ - an HTML file
Output: pruned $inputFile$'s DOM tree

```
1: procedure tagPathSequenceFilter(inputFile)
2:     DOMTree ← parseHTML(inputFile)
3:     convertTreeToSequence(DOMTree.body," ", tagPathSequence)
4:     backupTPS ← tagPathSequence
5:     searchRegion(tagPathSequence)
6:     if tagPathSequence not content then
7:         tagPathSequence = backupTPS − tagPathSequence
8:     end if
9:     pruneDOMTree(DOMTree.body, tagPathSequence)
10:    return DOMTree
11: end procedure
```

Algorithm 6 is the same as Algorithm 1, except for Line 6 where it checks if the main content is present in the reported region and, if not, we report the complementary sequence instead (Line 7), ensuring the presence of the main content.

7. CONCLUSION

As shown in the results, the method we have proposed for page segmentation and noise removal is very effective for some commercial/institutional web sites. In most cases, a very large amount of noise is removed without compromising the main content region. Also, when applied in conjunction with MDR, we can see that the extraction precision is greatly improved.

In the situations where our algorithm fails, other techniques have to/should/could be combined depending on the targeted application. In extreme cases, where a page has either too homogeneous structure (so we cannot find a split anywhere along the TPS) or too heterogeneous structure (then the main content itself gets split in several parts), the main content block could be detected using, perhaps, semantic approaches.

The algorithm shows outstanding performance, as it works very well for the majority of large commercial web sites we have tested. It also outcomes the limitations (training requirements, HTML tag dependency, manual labeling, among others) of previous work in the area of data cleaning, page segmentation and noise removal as mentioned in Section 2.

REFERENCES

Cai, D., Yu, S., Wen, J.-R., and Ma, W.-Y. Extracting Content Structure for Web Pages Based on Visual Representation. In *Web Technologies and Applications*. Springer, pp. 406–417, 2003.

Chakrabarti, D., Kumar, R., and Punera, K. A Graph-theoretic Approach to Webpage Segmentation. In *Proceedings of the International Conference on World Wide Web*. Beijing, China, pp. 377–386, 2008.

Cho, W.-T., Lin, Y.-M., and Kao, H.-Y. Entropy-based Visual Tree Evaluation on Block Extraction. In *Proceedings of the IEEE/WIC/ACM International Joint Conference on Web Intelligence and Intelligent Agent Technology*. Milan, Italy, pp. 580–583, 2009.

Crescenzi, V., Mecca, G., Merialdo, P., et al. Roadrunner: towards automatic data extraction from large web sites. In *Proceedings of the International Conference on Very Large Data Bases*. Rome, Italy, pp. 109–118, 2001.

Fernandes, D., de Moura, E. S., da Silva, A. S., Ribeiro-Neto, B., and Braga, E. A site Oriented Method for Segmenting Web Pages. In *Proceedings of the International ACM SIGIR Conference on Research and development in Information Retrieval*. Beijing, China, pp. 215–224, 2011.

Fernandes, D., de Moura, E. S., Ribeiro-Neto, B., da Silva, A. S., and Gonçalves, M. A. Computing Block Importance for Searching on Web Sites. In *Proceedings of the ACM Conference on Information and Knowledge Management*. Lisbon, Portugal, pp. 165–174, 2007.

Gibson, D., Punera, K., and Tomkins, A. The Volume and Evolution of Web Page Templates. In *Proceedings of the International Conference on World Wide Web*. Chiba, Japan, pp. 830–839, 2005.

Hu, F., Li, M., Zhang, Y. N., Peng, T., and Lei, Y. A Non-Template Approach to Purify Web Pages Based on Word Density. In *Proceedings of the International Conference on Information Engineering and Applications*. pp. 221–228, 2013.

Kohlschütter, C., Fankhauser, P., and Nejdl, W. Boilerplate Detection using Shallow Text Features. In *Proceedings of the ACM International Conference on Web Search and Data Mining*. New York, NY, pp. 441–450, 2010.

Kohlschütter, C. and Nejdl, W. A Densitometric Approach to Web Page Segmentation. In *Proceedings of the ACM Conference on Information and Knowledge Management*. Napa Valley, California, pp. 1173–1182, 2008.

Liu, B. and Chen-Chuan-Chang, K. Editorial: special issue on web content mining. *ACM SIGKDD Explorations Newsletter* 6 (2): 1–4, 2004.

Liu, B., Grossman, R., and Zhai, Y. Mining Data Records in Web Pages. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. Washington, DC, pp. 601–606, 2003.

Liu, B. and Zhai, Y. Net – a system for extracting web data from flat and nested data records. In *Web Information Systems Engineering*. Springer, pp. 487–495, 2005.

Liu, W., Meng, X., and Meng, W. Vide: a vision-based approach for deep web data extraction. *IEEE Transactions on Knowledge and Data Engineering* 22 (3): 447–460, 2010.

Miao, G., Tatemura, J., Hsiung, W.-P., Sawires, A., and Moser, L. E. Extracting Data Records from the Web using Tag Path Clustering. In *Proceedings of the International Conference on World Wide Web*. Madrid, Spain, pp. 981–990, 2009.

Simon, K. and Lausen, G. Viper: augmenting automatic information extraction with visual perceptions. In *Proceedings of the ACM International Conference on Information and Knowledge Management*. Bremen, Germany, pp. 381–388, 2005.

Weninger, T., Hsu, W. H., and Han, J. Cetr: content extraction via tag ratios. In *Proceedings of the International Conference on World Wide Web*. Raleigh, NC, USA, pp. 971–980, 2010.

Xie, X., Fang, Y., Zhang, Z., and Li, L. Extracting Data Records from Web using Suffix Tree. In *Proceedings of the ACM SIGKDD Workshop on Mining Data Semantics*. Beijing, China, pp. 12, 2012.

Yi, L., Liu, B., and Li, X. Eliminating Noisy Information in Web Pages for Data Mining. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. Washington, DC, pp. 296–305, 2003.

Zheng, S., Song, R., Wen, J.-R., and Wu, D. Joint Optimization of Wrapper Generation and Template Detection. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. San Jose, CA, USA, pp. 894–902, 2007.

Zheng, X., Gu, Y., and Li, Y. Data Extraction from Web Pages Based on Structural-Semantic Entropy. In *Proceedings of the International Conference on World Wide Web*. Lyon, France, pp. 93–102, 2012.