HTML-AWARE TECHNIQUE FOR DATA CLEANING

Mirel Cosulschi

University of Craiova
13 A. I. Cuza Street
20585 Craiova, Romania
E-mail: mirelc@central.ucv.ro

Abstract: Web pages created by human authors or dynamically generated using different scripts and data stored in a back-end database frequently contain many common mistakes. In order to obtain an acceptable result, web rendering engine component of a browser and automatic tools for extracting information from HTML pages need a preprocessing step to clean those web pages. We devised a simple algorithm whose main task is to properly close the tags and transform a page from HTML format into a XHTML format.

Keywords: bad data identification, data processing, transformations, html data, cleaning algorithm.

1. INTRODUCTION

The problem of data cleaning has been explored by many researchers as a problem that presents interesting challenges such as: dealing with missing data, handling erroneous data, and so on. Domain-independent techniques consist one direction of exploration.

The vast amount of information on World Wide Web cannot be fully exploited due to its main characteristics: web pages are designed with respect to the human readers, who interact with the systems by browsing HTML pages, rather than to be used by a computer program. The semantic content structure of web pages is the principal element exploited by many web applications: one of the latest directions is the construction of wrappers in order to structure web data using regular languages and database techniques (Laender et all, 2002), (Chang, 2001). A subsequent problem arisen here is the necessity to fix up a wide range of problems from poor generated web pages.

A HTML page may contain many types of information presented in different forms, such as text, image or applets (programs written in Java and executed, better said interpreted, inside a *virtual machine* - Java Virtual Machine, browser integrated). Hyper Text Markup Language (HTML 4.01) is a language designed for data presentation, and was not intended as a mean of structuring information and easing the process of structured data extraction. Another problem of HTML pages is related to their bad construction, language standards frequently being broken (i.e. improper closed tags, wrong nested tags, bad parameters and incorrect parameter value).

Web pages from commercial web sites are usually generated dynamically using different scripts and data stored in a back-end DBMS. The visitor can easily notice the usage of a few templates in the same site, with slight differences between them. If such a script has a small bug, then the problem will spread among all generated web pages.

2. HTML-XHTML TRANSFORMATION

World Wide Web consortium has warmly recommended usage of stricter standard Markup

Languages, such as XHTML and XML, in order to reduce errors resulted in the process of parsing various web pages created by disobeying the basic rules. Despite this recommendation, there still remains a huge quantity of web pages that do not respect the new standards, and with whom, the parser of search engine must cope.

We transform each HTML page into a *well-formed* XHTML page. Of the applications that can be involved in this process, we enumerate three of them:

- JTidy, a Java tool based on HTML Tidy (Tidy 2001) that is a W3C open source software
- CyberNeko HTML Parser (Neko HTML Parser)
- javax.swing.text.html.parser.Parser (comes embedded with the JDK)

These are complex programs, with a great degree of generality, trying to satisfy overall demands. During their usage we encounter situations when the result was not totally satisfactory.

We devised a simpler algorithm called *CleanDom* which performed well with respect to our necessities. We implemented it in Java as part of a whole system developed.

From a constructed XHTML file, the DOM tree representation used in the next step of our process, can be created with no effort.

2.1 XHTML

XHTML (Extensible HyperText Markup Language) (XHML 1.0) represents a family of document types which extends HTML4 language. In other words, XHTML is a redefinition of HTML 4.01 standards with the help of XML. Its goal is to replace the HTML language in the future and obtain cleaner documents.

- documents must be well-formed: all elements must be ested inside on unique root element <*html*>, any eement can have children elements; children elements must e correctly *closed* and *properly nested*:

```
<html>
  <head>...</head>
  <body>...</body>
</html>
```

- tag names must be in *lowercase*:

```
<body>
  <p> Sample </p>
</body>
```

- empty elements must be *closed*:

```
The text will be divided by a
horizontal line. <hr />
The image must be closed <img
src="poza.gif" />
```

- all XHTML elements must be *closed*:

```
Wrong: <p> Un text
Correct: <p> Un text </p>
```

- XHTML elements must be *properly nested*:

```
Wrong: <b><i> Bold, italic followed
by bold, italic</b></i>
Correct:<b><i> Bold, italic followed
by italic, bold</i></b>
```

- attribute names must be in *lower case*:

```
Wrong: <table WIDTH="100%">
Correct: <table width="100%">
```

- attribute values must be *quoted*:

```
Wrong: <table width=100%>
Correct: <table width="100%">
Wrong: <table width=100%>
Correct: <table width="100%">
```

- attribute minimization is *forbidden*:

```
Wrong: <frame noresize>
Correct: <frame noresize="noresize"
/>
```

- the *id* attribute replaces the *name* attribute
- the *lang* attribute applies to almost every XHTML element. It specifies the language of the content within an element
- the XHTML DTD defines *mandatory* elements: the 'html', 'head' and 'body' elements must be present, and the 'title' must be present inside the head element; all XHTML documents must have a DOCTYPE declaration

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML
1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-
strict.dtd">
<html
xmlns="http://www.w3.org/1999/xhtml">
 <head>
  <title>Titlu </title>
 </head>
 <body> Continut </body>
</html>
```

3. CleanDOM ALGORITHM DESCRIPTION

CleanDOM algorithm consists of the following main steps (see Algorithm 1):

- parse HTML page obtaining base tokens
- call Forward-Pass algorithm
- call Backward-Pass algorithm

The description of *Forward-Pass* step can be found in Algorithm 2.

After the input page was parsed and decomposed in atomic units, called tokens, the algorithm starts processing those elements using a stack for help. The stack will keep all *open tags*, tags for which their closing pair was not encountered yet. According to the rules of XHTML, a *close tag* must correspond to

each element. If such a tag cannot be found on input, then it will be created (line 14).

In lines 2-3, elements which cannot have a closing tag (e.g. <*style*>, <*script*>) or can be empty (e.g. <*hr*>, <*br*>, <*img*>) are directly added to vector B. In line 6 the *special* cases are treated first: we are talking about particular chaining of tags which will be handled differently for each situation. Due to the HTML loose syntax, there is a very high number of improper tag combinations, an exhaustively handling of all these possibilities being difficult either to foresee or to implement. In our implementation we handled the majority of situations that can be encountered in common applications with a high probability and which proved to be adequate for our proposed goal.

If the element is an *open tag* then it will be pushed onto the stack (line 7) and added to the output vector B (line 20). If the current element is a *close tag*, and if his related item (*open tag*) is presented into the stack, then all elements between this item and the stack's top will be extracted (line 12). For each element extracted from the stack, it will be created a corresponding *close tag* subsequently added to the vector B (line 14).

The *Backward-Pass* step is described in Algorithm 3.

Looking at the subroutine 3, it can easily be seen that *Backward-Pass* is very similar to *Forward-Pass*: in this situation the stack will be used to keep track of *close tags*. The element list will be run through backwards, from the last element to the first element; to keep things in an unitary way the first operation is to reverse the input vector (line 1).

The vector A is traversed element by element. Each element of vector A which does not have a closing tag or can be empty is added to the result vector B (line 3-4). If item is a *close tag* then it will be pushed onto the stack (line 8) and added to vector B. Lines 10-19 handle the case when an open-tag is encountered to whom must be associated a *close tag*. If there are other elements between stack's top and the position of *close tag*, then for each such element a correspondent *open tag* is created (line 14).

Example. Let us consider the following page:

```
<head>
 <title> First Test Page </title>
</head>
<body>
  <tag1>
    <tag2>
      <tag3>
       Inside Tag3
      </tag3>
      Outside Tag3, inside Tag2
      <tag2>
        Open second Tag2, Close Tag1
      </tag1>
      Close second Tag2
    </tag2>
    Close first Tag1
  </tag1>
</body>
</html>
```

The page tokenisation result is: <head>, <title>, 'First Test Page', </title>, </head>, <body>, <tag1>, <tag2>, <tag3>, 'Inside Tag3', </tag3>, 'Outside Tag3, inside Tag2', <tag2>, 'Open second Tag2, Close Tag1', </tag1>, 'Close second Tag2', </tag2>, 'Close first Tag1', </tag1>, </body>, </html>.

After the call of Forward-Pass, the output vector will contain the following elements: <html>, <head>, <title>, 'First Test Page', </title>, </head>, <body>, <tag1>, <tag2>, <tag3>, 'Inside Tag3', </tag3>, 'Outside Tag3, inside Tag2', <tag2>, 'Open second Tag2, Close Tag1', </tag2>}, </tag2>, </tag1>, 'Close second Tag2', </tag2>, 'Close first Tag1', </tag1>, </body>, </html> (new tags are marked with bold).

After the call of the last processing step, Backward-Pass, the page will look like:

```
<html>
 <head>
  <title>
   First Test Page
  </title>
 </head>
 <body>
  <tag1> - new
   <tag2> - new
    <tag1>
     <tag2>
      <tag3>
       Inside Tag3
      </tag3>
      Outside Tag3, inside Tag2
      <tag2>
       Open second Tag2, Close Tag1
      </tag2> - new
     </tag2> - new
    </tag1>
    Close second Tag2
   </tag2>
   Close first Tag1
  </tag1>
 </body>
</html>
```

4. CONCLUSIONS

The number of situations when we encounter wrong defined HTML pages is very large, and we didn't

proposed to treat all such cases. The aim of the algorithm is to treat the most common mistakes, to obtain fast and accurate results. The complexity of the algorithm is linearly depending of the number of tags from the parsed web page.

The algorithm was implemented in Java and the tests we have made helped us to handle many exceptional situations. We are currently using this in our other web related projects.

REFERENCES

Document Object Model (DOM) Level 3 Core specification (2004). W3C Recommendation. http://www.w3.org/TR/DOM-Level-3-Core/

HTML 4.01 Specification (1998), http://www.w3.org/TR/1999/REC-html401-19991224.

Neko HTML Parser, http://www.apache.org/~andyc/neko/doc/html/index.html.

W3C, HTML Tidy, http://www.w3.org/People/Raggett/tidy

XHTML 1.0 The Extensible HyperText Markup Language (Second Edition)}, A reformulation of HTML 4 in XML 1.0. http://www.w3.org/TR/2002/REC-xhtml1-20020801/

Chang, Chia-Hui (2001), IEPAD: Information extraction based on pattern discovery, in *Proceedings of the tenth international conference on World Wide Web*.

Laender, A., Ribeiro-Neto, B. , Da Silva, A., and Texeira , J(2002). A Brief Survey of Web Data Extraction Tools, *ACM SIGMOD Record, 31(2)*.

Algorithm 1: Algorithm CleanDOM

1: Parse HTML page obtaining base elements (tags and text strings)
2: **Call** Forward-Pass
3: **Call** Backward-Pass

Algorithm 2: Algorithm Forward-Pass

```
Input: A – the vector contains token list from original HTML page
Output: B – the vector contains the modified token list
1:      for each item from A do
2:              if  (item doesn't have a closing tag OR can be empty) then
3:                      add item to B
4:              else
5:                      if (item is a tag) then
6:                              first test some special cases
7:                              if (item is open-tag) then
8:                                      push item on Stack
9:                              else
10:                                     if (exists on the stack the pair of item element) then
11:                                             while (Stack is not empty) do
12:                                                     pop from the Stack and save into tmpTab
13:                                                     if (tmpTag ≠ tag) then
14:                                                             add to B a closed-tag in
correspondence to tmpTag
15:                                                     else
16:                                                             break
17:                                                     endif
18:                                             endwhile
19:                                     endif
20:                                     add item to B
21:                             endif
22:                     endif
23:             endif
24:     endfor
```

Algorithm 3: Algorithm Backward-Pass

```
Input: A – the vector contains token list resulted from Forward-Pass call
Output: B – the vector contains the modified token list
1:              State the order of elements from the vector A is reversed
2:      for each item from A do
3:              if (item doesn't have a closing tag ∨ can be empty) then
4:                      add item to B
5:              else
6:                      if (item is a tag) then
7:                              if (item is close-tag) then
8:                                      push item on Stack
9:                              else
10:                                     if (exists on the stack the pair of item element) then
11:                                             while (Stack is not empty) do
12:                                                     pop from the Stack and save into tmpTab
13:                                                     if (tmpTag ≠ tag) then
14:                                                             add to B a open-tag in
correspondence to tmpTag
15:                                                     else
16:                                                             break
17:                                                     endif
18:                                             endwhile
19:                                     endif
20:                                     add item to B
21:                             endif
22:                     endif
23:             endif
24:     endfor
25:     the order of elements from vector B is reversed
```