HTML Page Analysis Based on Visual Cues

Yudong Yang and HongJiang Zhang
Microsoft Research China
No. 49, Zhichun Road, Beijing, China
yangyud@cn.ibm.com, hjzhang@microsoft.com

Abstract

In this paper, we present a novel approach to automatically analyzing semantic structure of HTML pages based on detecting visual similarities of content objects on web pages. The approach is developed based on the observation that in most web pages, layout styles of subtitles or records of the same content category are consistent and there are apparent separation boundaries between different categories. Thus these subtitles should have similar appearances if they are rendered in visual browsers and different categories can be separated clearly. In our approach, we first measure visual similarities of HTML content objects. Then we apply a pattern detection algorithm to detect frequent patterns of visual similarity and use a number of heuristics to choose the most possible patterns. By grouping items according to these patterns, we finally build a hierarchical representation (tree) of HTML document with "visual consistency" inferred semantics. Preliminary experimental results show promising performances of the method with real web pages.

1. Introduction

The World Wide Web has become one of the most important information sources today. Most of data on web are available as pages encoded in markup languages like HTML intending for visual browsers. As the amount of data on web grows, locating desired contents accurately and accessing them conveniently become pressing requirements. Technologies like web search engine and adaptive content delivery [1,2,3,4,5,6,7] are being developed to meet such requirements. However web pages are normally composed for viewing in visual web browsers and lack information on semantic structures.

To extract these structures, documents wrappers are commonly used. Building wrappers, however, is not a trivial task. Normally, wrappers are built for specific web pages by having people examine these pages and then figure out some rules that can separate the chunks of interests on these web pages. Based on these special rules, we can write the wrapper to extract information from pages that belong to exactly the same class. Many wrappers are just lexical analyzers as that discussed in [8]. Methods like [9] make some improvements by using heuristics in addition to lexical analyzers. There are also approaches trying to derive some semantic structures directly. Approach presented in [10] discusses a "concept" discovery and confirmation method based on heuristics. Another one [11] introduces a method to find the relationships between labeled semi-structured data.

As we can see that methods listed above are some limited because detection of content chunks is actually done by human. These methods are not feasible if a large amount and variations of web pages are to be processed. Automatic methods or semi-automatic methods are much more effective in this situation. Only recently, several proposals discuss ways of automatic analysis. In [14], a method to parse HTML data tables and generate a hierarchical representation is discussed. The approach assumed that authors of tables have provided enough information to interpret tables. The authors of [13] introduce a method that detects chunk boundary by combining multiple independent heuristics. With specific field of interests, wrappers can also be implemented based on semantic rules. Approach discussed in [12] is such an idea.

HTML, as it was introduced with web technology, is the most commonly used standard of current web pages. However it lacks the ability of representing semantic related contents. For some reasons, it was designed to take both structural and presentational capability in mind. And these two were not clearly separated (In the first version of HTML most of the tags were for structures. But many layout and presentation tags were stuffed into following versions and are widely used today. Some of the histories can be found in [15]). Further widely misuses of structural HTML tags for layout purpose make the situation even worse. Cascade Style Sheet (CSS) [21] was later developed as a remedy to this, but only recently several popular browsers begin to have better CSS support [21]. The recent W3C recommendation of XML provides a better way to organize data and represent semantic structures of data. However, most of web contents are still authored in HTML.

Because of the common misusages, we consider that HTML tags are not stable features for analyzing structures of HTML documents. For semantic rules based approaches, limited fields of interests and difficulties to learn new rules automatically restrict their feasibilities with general web pages.

In this paper, we propose a novel method to extract semantic structures from general HTML pages. This method doesn't

require *a priori* knowledge of web pages. It uses features derived directly from layout of HTML pages. As we observed, it's common for web pages to keep consistent visual styles with parallel subtitles or records in same content categories and different categories are separated by apparent visual boundaries. The objective of our approach is to detect these visual cues and construct the hierarchies of categories.

The paper is organized as following. Section 2 talks about measurements of visual similarities. Then, in section 3 we introduce our heuristics. After that, we talk about our method to detect visual patterns and then to construct document structures based on these heuristics. Experimental results are discussed in section 4. In section 5 we present examples of the method being used in our adaptive web content delivery test-bed. Finally, section 6 sums up our discussions.

2. Visual Similarity of HTML Objects

Good organization of contents is an essential factor of good content services. Figure 2 and Figure 3 on later pages show some examples of typical web pages. From these examples we can see that it's quite common to divide contents into categories and each category holds records of related subtitles. In addition, records in one category are normally organized in ways having a consistent visual layout style. Boundaries between different categories are marked apparently with different visual styles or separators. As we have said, the basic idea of our approach is to detect these visual cues and then records and categories.

Appearances of HTML objects are defined by factors like layout and style. With current W3C recommendations, layout and style of HTML pages should be defined by CSS. However, due to history reasons, CSS is not very popular yet and most of the web pages are still patchworks of structural and *deprecated* [20] presentation tags. Also by reasons like tricks to reduce page size, laziness, mistakes, etc or because of some authoring tools of limited functions, two HTML objects that look similarly in browsers don't denote that they use the same combinations of HTML tags. Different tags and in different orders may all have the same results. Approaches that relay on tags will not be effective in these complex situations.

From previous analysis we can see that visual similarity should be potentially a good feature for structure extraction. This assumption is justified later by our experiment results. In following discussions, we will use some terms as defined below:

- **Simple object:** None-breakable visual HTML objects that do not include other HTML tags (like paragraphs of pure texts or tags as `<IMG>`, `<HR>`) or are representations of one embedded media object (like `<OBJECT>`, `<APPLET>`).
- **Container object:** An ordered set of objects that consists of at least one simple object or other container object and these objects must be adjacent if they are rendered in browsers. Order of these elements is normally defined by reading habits ("Left to Right" and "Top to Bottom" in most languages). We represent a container object C as a string of elements $\{e_1, e_2, \ldots, e_n\}$, where e_i is simple objects or other container objects.
- **Group object:** Special container objects where all elements are simple objects and these elements are rendered on the same text line without deliberate line breaks by visual browsers.
- **List object:** Special container objects where all elements satisfy some consistency constraint (like visual similarity defined later).
- **Structured document:** HTML documents converted to hierarchical structures of container objects and simple objects.

Table 1. Fuzzy rules for comparing simple objects

Comp-are	Rules
Text & Text	Starting from x=1.0 Compare key HTML attributes (like `<H1>` … `<H6>`, `<A>`) $x = x\cdot\begin{cases} Key_Mod, & \text{Not Equal} \\ 1, & \text{Equal}\end{cases}$ Compare font size attribute $x = x\cdot\begin{cases} Size_Mod, & \text{Not Equal} \\ 1, & \text{Equal}\end{cases}$ Compare styles (bold, italic, underline, …) $x = x\cdot\begin{cases} Style_Mod, & \text{Not Equal} \\ 1, & \text{Equal}\end{cases}$ Compare font face $x = x\cdot\begin{cases} Font_Mod, & \text{Not Equal} \\ 1, & \text{Equal}\end{cases}$ Compare text length $x = x\cdot\left(\frac{\min(len_1, len_2)}{\max(len_1, len_2)}\right)^{Adj}$
Image & Image	Starting from x=1.0 Compare key HTML attributes (like `<H1>`…`<H6>`, `<A>`) $x = x\cdot\begin{cases} Key_Mod, & \text{Not Equal} \\ 1, & \text{Equal}\end{cases}$ Compare color attributes (pseudo color, true color, grayscale) $x = x\cdot\begin{cases} Col_Mod, & \text{Not Equal} \\ 1, & \text{Equal}\end{cases}$ Compare image dimension $x = x\cdot\left(\frac{\min(x_1, x_2)\cdot\min(y_1, y_2)}{\max(x_1, x_2)\cdot\max(y_1, y_2)}\right)^{Adj}$

2.1 Visual similarity of simple objects

To compare visual similarities of more complex objects, we will firstly start from simple objects. During the process to parse HTML documents and to extract *simple objects*, we extract text rendering parameters by keeping a stack of tags that affect text attributes like font face, styles, size, and color. For other embedded media objects like images, we extract information from tag attributes or by analyzing their file headers. According to these parameters, we define some fuzzy comparison rules to decide visual similarity. Table 1 lists some of the rules we used in our experiments where $X_Mod \in [0,1]$ and $Adj \in [0,1]$ are user-defined values that represent the level of impacts on similarity measurements when correspondent parameters are not equal. A modifier equals to zero means that two objects are distinct or can't be compared at all (as in the case of text and image). For simplicity, we only listed cases with image and text media types.

In our experimental systems, all common visual HTML objects (like <HR>) are also considered.

2.2 Visual similarity of container objects

We define visual similarity of *container objects* based on that of *simple objects*. To keep appropriate semantic granularities, we define *group objects* as contents that are considered tightly related from our visual cues based view (such as sentences and paragraphs). And we do not break up *group objects* during the analysis process. A *simple object* is treated as a *container object* with only one element when it is compared with other *container objects*. Beside these, *list objects* have their specialties because we use them to represent detected categories and records. And instead of using whole objects, we pick typical elements from *list objects* to compare with other objects. With two *container objects*, we define two kinds of visual similarity measurements:

- **Approximate Similarity:** Comparison of two element strings that enables weighted mismatches and omissions (skips).
- **Parallel Similarity:** Comparison of two element strings that enables only weighted mismatches.

Table 2. Approximate String Compare Algorithm

```
compare(x, NULL) = skip_weight(x);
compare(simpleX, simpleY) = def by Table 1;
compare(strI[1..lthI], strJ[1..lthJ])
{
  dim cmp[0..lthJ];
  cmp[0] = 1;
  lastv10 = 1;
  for(j=1; j<=lthJ; j++)
    cmp[j] = cmp[j-1] * compare(NULL, strJ[j]));
  for(i=1; i<=lthI; i++) {
    lastv11 = cmp[0];
    cmp[0] = lastv10 * compare(strI[i], NULL);
    lastv10 = cmp[0];
    for(j=1; j<=lthJ; j++){
      v11 = lastv11 * compare(strI[i], strJ[j]);
      v10 = cmp[j] * compare(strI[i], NULL);
      v01 = cmp[j-1] * compare(NULL, strJ[j]);
      lastv11 = cmp[j];
      cmp[j] = max(v11, v10, v01);
    }
  }
  return cmp[lthJ];
}
```

From the definitions we can see that *approximate similarity* is more robust than *parallel similarity* if there are outliers in strings. In our experiments *parallel similarity* is simply an one-by-one comparison. Algorithm of *approximate similarity* measurement using dynamic programming is shown in Table 2. Weight of skipping may differ from element to element because some of the objects (as <H1>...<H6>) could be very important and skipping of them would be costly (small weight) or not allowed (zero weight).

3. Pattern Detection and Construction of Document Structures

In this section, we present our method to detect visual similarity patterns and then records and categories of contents based on previously defined similarity measurements. As we can see, visual similarity patterns are not appearing as very stable forms even with "well composed" web pages. Their lengths can change from one to one, and outliers are common. Beside these we do not have known boundaries to separate potential patterns. Many proposals have been introduced to detect frequent sequential patterns in large databases. However, due to differences of applications, their expected constraints are different from us.

In our approach, we start from an exact pattern detection method based on suffix trees and then we expand exact patterns according to *approximate similarity*. Each time a *container object* is constructed, it is checked for potential patterns. These patterns are converted to *list objects* then. Adjacent *list objects* are checked for visual similarities and are merged if they are similar.

Here we define some terms used in following discussions. For *container object* $C=\{e_1, e_2, \dots, e_n\}$, an object o is represented by a sub-string of C as $\{e_s, \dots, e_{s+l-1}\}$. Visual pattern p is represented as a set of "equal" objects $\{o_1, \dots, o_m\}$ and some times represented by a typical element o_p of the pattern. We also follow some heuristics as listed below for locating possible patterns:

- **Equal Judgment:** Two objects are equal only if their similarity measurement is not below a threshold E_p.
- **Minimal Frequency:** A pattern must contain at least F_p objects.
- **No Overlap:** Objects in one pattern do not overlap with others.
- **Alignment:** Objects in one pattern are normally aligned tidily.
- **Paragraphs:** Contents that reside in the same unbroken text line should be tightly related and thus will be treated as one element (This is what group objects stand for).
- **Minimal Deviation:** Standard deviations of objects' distributions (positions) and lengths in potentially better patterns should be smaller.
- **Maximum Coverage:** The better patterns should have bigger coverage of elements in C.
- **Sub-pattern Replacement:** If all objects in a pattern are concatenations of "equal" sub strings (sub-pattern), then these objects are expanded to sub-strings. Assume a pattern as $\{\{e_1, \dots, e_m\}, \{e_{m+1}, \dots, e_{m+k}\}, \dots\}$ and $e_i == e_j, \forall i,j$, then the pattern is expanded to $\{e_1,\dots, e_m, e_{m+1},\dots, e_{m+k}, \dots\}$.
- **Significant Token:** Records in one category should have similar prefix elements (A pattern starts at a significant token).

3.1 Quantization

To reduce the complexity of frequency counting, we first cluster candidate elements according to similarity measurements between each element. These clusters are then labeled with unique identifiers. Elements in the same cluster are assigned with same identifier and are considered as equal to each other. As we said before we call this process as *quantization*. Currently, we use a DBSCAN [16] like clustering algorithm because we do not know the number of possible clusters (or groups of similar elements) at the beginning. Another reason is that our heuristics have specified two values (E_p and F_p) that are just the epsilon and minimal density as required by DBSCAN. Given a distance function between two elements (as similarity measurement) and a

minimal acceptable density (as defined by heuristic "minimal frequency" and "equal judgment"), we can define our *Eps-neighbourhood* and *core point condition* [16] as following. Other terms like *density-reachable* and *density-connected* can be defined respectively and thus *cluster* and *noise*.

- **Eps-neighbourhood:** $N_{Eps}(e)=\{e'\in C \mid similarity(e, e') \geq E_p\}$, where E_p is from "equal judgment".
- **Core point condition:** $|N_{Eps}(e)| \geq F_p$, where F_p is defined by "minimal frequency".

For $C=\{e_1, e_2, \ldots, e_n\}$, if the clustering result is m clusters as $T_1=\{e_a, e_b, \ldots, e_x\}, \ldots T_m=\{e_s, e_t, \ldots, e_y\}$, we construct a token string $T=\{t_1, t_2, \ldots, t_n\}$ with t_i equals to the cluster identifier that e_i belongs to. The token string is then passed to the frequency counting stage. In following discussions we use an example as $C=\{e_1, e_2, \ldots, e_{13}\}$ and clustering result as $T=$ {C, A, B, D, A, B, E, D, A, B, C, A, B} with 4 clusters labeled as ABCD and 1 outlier labeled as E.

3.2 Frequency Counting

Frequencies of quantized patterns are counted efficiently using a suffix tree representation of token string *T*. Starting from the root node, the "label of path" of a node is actually what we called as a pattern and leaves under the node are positions of the pattern in string. The number of leaves under each node is exactly how many times the pattern appears in the string. Figure 1 gives an example of pattern counting of string *T* using suffix tree.

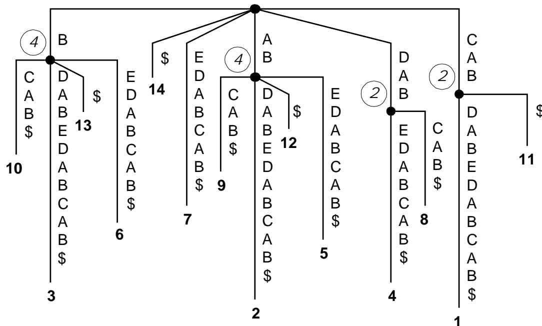


Figure 1. Pattern frequency counting

To build the tree, we borrow some code from [19] which is an implementation of Ukkonen's algorithm [18]. We modify it slightly to fit with our requirements. The algorithm is $O(n)$ complexity. We won't repeat the details here since the author gives a very good introduction over it.

3.3 Selection and Confirm

From the results of frequency counting, we choose the best patterns based on heuristics listed above. As we can see from Figure 1, frequency of {A, B} and {B} is the highest and are good candidates. And {A, B} is superior to {B} according to heuristic "maximum coverage". However {A, B} can only cover a part of the elements because of outliers as {C, D, E}. To cop with these outliers we expand these patterns based on *approximate similarity* measurements and heuristic "significant token". Currently we use a naïve method – starting from a strict pattern, we try to append succeeding elements after each object of the pattern. The consistency of the pattern is checked during the process and it stops if an appendant breaks the consistency. To illustrate the process, we list the steps of expanding pattern {A, B} below:

$\{e_1, \{\mathbf{e_2}, \mathbf{e_3}\}, e_4, \{\mathbf{e_5}, \mathbf{e_6}\}, e_7, e_8, \{\mathbf{e_9}, \mathbf{e_{10}}\}, e_{11}, \{\mathbf{e_{12}}, \mathbf{e_{13}}\}\}$
→ the original pattern {A,B}

$\{e_1, \{\mathbf{e_2}, \mathbf{e_3}, e_4\}, \{\mathbf{e_5}, \mathbf{e_6}\}, e_7, e_8, \{\mathbf{e_9}, \mathbf{e_{10}}\}, e_{11}, \{\mathbf{e_{12}}, \mathbf{e_{13}}\}\}$
→ one element appended

… … (repeat)

$\{e_1, \{\mathbf{e_2}, \mathbf{e_3}, e_4\}, \{\mathbf{e_5}, \mathbf{e_6}, e_7\}, e_8, \{\mathbf{e_9}, \mathbf{e_{10}}, e_{11}\}, \{\mathbf{e_{12}}, \mathbf{e_{13}}\}\}$
→ final result

From the example we can see that heuristic "significant token" might some time miss possible best patterns like $\{\{e_1, \boldsymbol{e_2}, \boldsymbol{e_3}\}, \{e_4, \boldsymbol{e_5}, \boldsymbol{e_6}, e_7\}, \{e_8, \boldsymbol{e_9}, \boldsymbol{e_{10}}\}, \{e_{11}, \boldsymbol{e_{12}}, \boldsymbol{e_{13}}\}\}$ which do not have a "significant token" at the beginning.

3.4 Construction of Structured Document

*Structured document*s are constructed in a recursive manner. Starting from *simple object*s and *group object*s, we divide these elements into initial *container object*s roughly based on block-level tags [20]. Then we apply the pattern detection algorithm to elements of these initial *container object*s, and detected patterns are converted to *list object*s. For example, using *container object* and patterns of section 3.3, we can create a new *container object* as $\{e_1, \underline{\{\{\boldsymbol{e_2}, \boldsymbol{e_3}, e_4\}, \{\boldsymbol{e_5}, \boldsymbol{e_6}, e_7, e_8\}, \{\boldsymbol{e_9}, \boldsymbol{e_{10}}, e_{11}\}, \{\boldsymbol{e_{12}}, \boldsymbol{e_{13}}\}\}}\}$ where the underscored element is a *list object*. Note that outliers between two list elements are appended as *do-not-cares*. We then expand *container object*s to upper levels by merging objects under the same parent if they are not enclosed in important structures (Currently we think `<H1>…<H6>`, `<FORM>`, `<ADDRESS>`, `<TABLE>`, `<OL>`, `<UL>` and `<DL>` should be kept). After expanding, we check if two adjacent *list object*s are similar and merge them into one if they are. The whole process then repeats until `<BODY>` of HTML document has been processed. The final *container object* is the hierarchical *structured document* that is actually a tree representation of original page.

3.5 Special Consideration of HTML Table

In this section, we discuss how to apply our visual cue based method to analyze structures of HTML tables. Tables are the most frequently used layout tools of HTML pages. From regular data tables to general content layouts (Here we name data tables as that represent cleanly organized data like stock, price, etc), tables provide a powerful way to control positions and alignments. For this reason, tables are considered a very important source of semi-structured data. Several approaches have been developed to extracting structured contents from data tables. Typical approaches like [8] does this by manually specifying rules and pattern strings to locate wanted data. Method introduced in [14] made further steps by automatically analyzing data tables with titles and headers. These approaches, however, did not mention

ways to decide if a table is data table automatically.

As we can see that data tables are normally organized tidily and thus should hold very strong visual similarity patterns. In addition, many general contents tables also hold the strong visual cue. It's not strange to make use of the alignment nature of tables as the starting point of structure analysis. We start the analysis from a pseudo rendering process that counts the rows and columns of a table at first (HTML table 'colspan' and 'rowspan' attributes are also taken care of in this step). Then, all empty rows and columns are stripped since these are only for spacing and other layout purposes. Because column-wise and row-wise organizations are quite common for data tables, we detect this situation firstly by checking if the table has headings and footings (such as that specified by <TH> <THEAD> <TFOOT> tags). Then, we compare elements in rows and columns to check if similarity consistency holds. If none of the above checks is successful, we will then try a more aggressive method that divides the table into smaller rectangular blocks and these blocks are checked for similarity consistency. Currently we divide these blocks according to divisors of the number of columns and rows based on an assumption that one category of contents should be held in one table only. We pass the table back to default detector if all efforts fail.

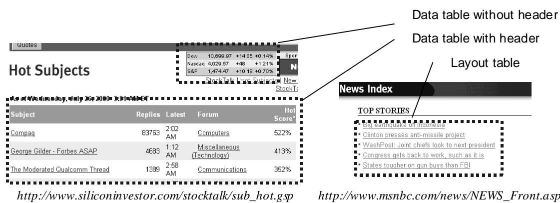


Figure 2. Examples of common HTML tables

4. Experimental Result and Analysis

We have implemented an experimental detector to test our ideas. Test data are web pages randomly collected from popular sites listed by http://www.100hot.com. Beside these we also collect the first page (normally contains a directory list) and search results of several popular search engines. Total number of pages is 50. We then run our test program and save extracted structure as text files for manual analyzing later. Experimental results are listed in following table. Because of lacking impersonal ground-truth references, we can only list some numbers for reference purposes (If you want to have a test with your own data, you may contact *hjzhang@microsoft.com* for executable of the detector).

Table 3. Experimental results with a test set of 50

Results	Number of Documents
All chunks are detected	46
Missed some apparent chunks	2
Failed to parse the document	2

Although the test is still based on a small data set, from these results we can see that most of the category chunks are detected successfully with some minor exceptions. There are 2 pages our detector misses some apparent chunks on them. In common, these pages are a bit cluttered and have very complex table based layouts that our detector fails to analysis effectively. Our HTML parser fails over the other 2 pages because of HTML syntax errors.

Table 4. Analysis of exceptions in results

Exceptions	Reasons
Wrongly confirmed chunks	•Mistakes by visual similarity measurement
One chunk broken up to two or more	•Mistakes by visual similarity measurement
Mis-aligned boundary	•Detected patterns are skewed •Heuristic significant token does not hold
Missed chunks	•Style Sheets are not supported •DHTML and scripts are ignored currently
Failed	•HTML Tidy failed to parse the page

5. Application in Adaptive Content Delivery System

We have used an early version of the structure detector in a test-bed of adaptive web content delivery [1, 17]. To give a quick summary, for users with very slow Internet connections, the basic idea of our adaptation process is to summarize web pages to some levels that will not affect human comprehension too much in favor of download speed and client (device/browser) capability. The process uses some heuristics based on basics of web UI design:

- Categories/items occupying larger display area are more important (potential of user interests) than those smaller ones.
- Users prefer to see full images or videos in their browsers and dislike scrolling a window on large images.
- Important items in are organized in front of trivial ones.
- Users can catch the ideas of a big category with only a small subset of content items in it.
- Impatient users should be in favor of deeper organization structure of information (smaller pages thus faster download but with deeper hierarchies) instead of wider ones (larger pages thus slower download but with less hierarchies).

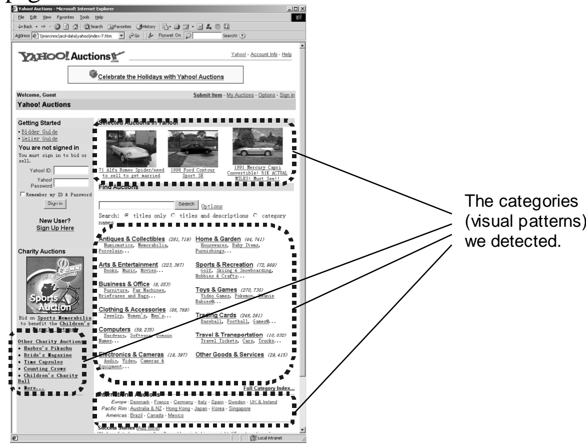
From these heuristics, we deduce some rules to reduce contents with tolerable information loses:

- Importance values of categories/items are initialized by areas they occupy.
- In case of deletion, contents are preserved according to their importance values.
- In case of summarization, larger categories are summarized prior to small ones.
- List objects are summarized by truncating tailing items.
- Large images are shrunk to fit inside the browser's screen.
- Contents deleted in a step can be made accessible further.

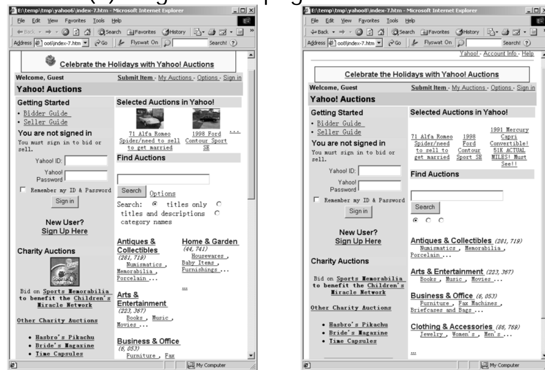
The adaptation process consists of several steps as:

1. **Content structure analysis:** The structure detector introduced here is used to locate content categories on HTML pages and to build abstract representations of contents that include these structure information and additional content attributes as areas of objects, object size, etc.
2. **Content reduction:** Content reduction rules are applied to abstract representations according to detected network bandwidth and type of client devices in favor of speed.
3. **Page reconstruction:** The reduced abstract representations are mapped back to original HTML pages to generate reduced web

pages that should be downloaded faster.



(a) Original web page from "Yahoo!"



(b) Adaptation result 1 (c) Adaptation result 2

Figure 3. Experimental results of the adaptive web content delivery system

Figure 3 shows an example of the adaptation results. As we can see that detected content structures plays a very important role. With guided reduction of contents, rich information and original organizations can be preserved.

6. Conclusions

In this paper, we have presented a visual cue based approach to extraction of semantic structure of HTML documents. It relays on the observation that for most web pages layout styles of subtitles or records in the same content category are kept the same and apparent separation boundaries exist between different categories. We have tested the method with a set of 50 web pages collected randomly from directories of http://www.100hot.com. An example of using it in our adaptive content delivery test-bed is introduced briefly later. These preliminary experimental results show that the approach works well with only a few exceptions. Also note that this approach is not exclusive of other methods. It's possible to achieve better results if we combine it with other approaches. We hope future works will prove this.

Acknowledgements

The HTML parser in our experiments is based on HTML Tidy [22] that is W3C open source software and is by Dave Raggett. The suffix tree construction algorithm is based on Mark Nelson's codes [19]. Here we thank the authors for their great works.

References

[1] Y.D. Yang, J.L. Chen, and H.J. Zhang, "Adaptive Delivery of HTML Contents", WWW9 Poster Proceedings, May, 2000, pp24-25.

[2] A. Fox, S.D. Gribble, *et al*, "Adapting to Network and Client Variation Using Infrastructural Proxies: Lessons and Perspectives", IEEE Personal Communication, V5, I4, 1998, pp10-19.

[3] T.W. BickMore, and B.N. Schilit, "Digestor: Device-independent access to the World Wide Web", Proc. of the 6th International World Wide Web Conference, 1997, pp655-663.

[4] M. Liijeberg, H. Helin, M. Kojo, and K. Raatikainen, "Enhanced services for World Wide Web in mobile WAN environment", Report C-1996-28, 1996, University of Helsinki Finland. http://www.cs.helsinki.fi/research/mowgli/

[5] W.Y. Ma, I. Bedner, G. Chang, A. Kuchinsky, and H.J. Zhang, "A Framework for Adaptive Content Delivery in Heterogeneous Network Environments", Proc. MMCN2000 (SPIE Vol. 3969), 2000.

[6] J.R. Smith, R. Mohan, and C.S. Li, "Scalable Multimedia Delivery for Pervasive Computing", Proc. of the 7th ACM International Conference on Multimedia, 1999, pp131-140.

[7] Web Clipping, 3Com, http://www.palm.com/

[8] J. Hammer, H.Garcia-Monlina, J. Cho, R. Aranha, and A. Crespo, "Extracting semistructured information from the web", Proc. PODS/SIGMOD'97, May 1997.

[9] N. Ashish, and C. Knoblock, "Wrapper generation for semi-structured Internet sources", Proc. PODS/SIGMOD'97, May 1997.

[10] D. Simth, and M. Lopez, "Information extraction for semi-structured documents", Proc. PODS/SIGMOD'97, May 1997.

[11] S. Nestorov, S. Abiteboul, R. Motwani, "Inferring Structure in Semistructured Data", Proc. PODS/SIGMOD'97, May 1997.

[12] D.W. Embley, Y.K. Ng, L, Xu, "Filtering Multiple-record Web Documents Based on Application Ontologies", http://www.deg.byu.edu/papers/vldb00.ps

[13] D.W. Embley, Y. Jiang, Y.K. Ng, "Record-Boundary Discovery in Web Documents", In Proc. SIGMOD'99, 1999, pp467-478.

[14] S.J. Lim, Y.K. Ng, "An Automated Approach for Retrieving Hierarchical Data from HTML Table", In Proc CIKM'99, pp466-474.

[15] D. Siegel, "The Web is Ruined and I Ruined It", http://webreview.com/97/04/11/feature/

[16] M. Easter, H-P. Kriegel, J. Sander, X.W. Xu, "A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise", In Proc KDD'96, 1996.

[17] J.L. Chen, Y.D. Yang, H.J. Zhang, "An Adaptive Web Content Delivery System", in Proc. International Conference on Adaptive Hypermedia and Adaptive Web-based Systems (AH-2000), Aug. 2000.

[18] E. Ukkonen, "On-line construction of suffix trees", Algorithmica, 14(3), Sept. 1995, pp249-260.

[19] M. Nelson, "Fast string searching with suffix trees", Dr. Dobb's Journal, August 1996.

[20] W3C, "HTML 4.0 specification", http://www.w3.org/TR/html4/

[21] W3C, "Cascading Style Sheets", http://www.w3.org/Style/CSS/

[22] W3C, "HTML Tidy", http://www.w3.org/People/Raggett/tidy/