

# Automatic Partitioning of Web Pages Using Clustering

Richard Romero and Adam Berger

Nokia Inc.  
501 Martindale Street  
Pittsburgh PA 15212

**Abstract.** This paper introduces a method for automatically partitioning richly-formatted electronic documents. An automatic partitioning system has many potential uses, but we focus here on one: dividing web content into fragments small enough to be delivered to and rendered on a mobile phone or PDA. The segmentation algorithm is analyzed from a theoretical and an empirical basis, with a suite of measurements.

## 1 Introduction

Segmentation is often necessary before transmitting large files to mobile devices. For one, mobile phones often suffer from limited memory and therefore cannot digest large files. Second, gateways (e.g. GGSN products) that mediate mobile data traffic may truncate or refuse to propagate large files. In general, even in the absence of strict file size limits, it is ill-advised to transmit large files over a low-throughput cellular data network, since the recipient may experience an unacceptable latency and/or airtime cost.

There has been limited activity in the field of partitioning richly-formatted documents, but considerably more in the related fields of document summarization and distillation, and in expository text segmentation. The specific contributions of this paper are twofold:

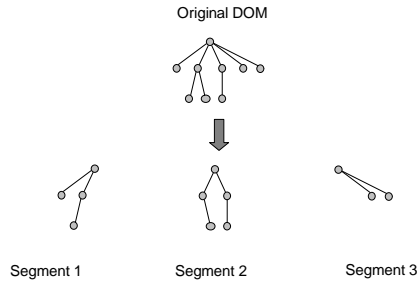
- Introduce a technique, based on clustering, for partitioning richly-formatted content.
- Describe and evaluate a real-time service, based on this technique, which is designed to increase the usability of mobile web browsing.

## 2 Segmenting via clustering

This section introduces a tree-based clustering algorithm for document partitioning.

The algorithm takes as input a DOM, the data structure analogue of XML. There exist publicly available parsers that convert XML and HTML files into DOM [2]. In addition, there also exist publicly available utilities that convert PDF and Microsoft Office™ file formats into XHTML, an XML-compliant version of HTML. What this means is that the segmentation algorithm described here can easily be adapted to process the most common file formats.

For pedagogic purposes, however, we will assume that the DOM reflects an HTML document: the nodes of the DOM tree correspond to HTML elements like `<p>`, `<b>`, `<table>`, `<td>`, `<it>`, and `<anchor>`, and the leaves correspond to interactive, viewable elements such as text, images, and form widgets.



**Figure 1:** The segmenting procedure operates on a tree-based document format called a DOM. The input DOM is divided into sub-trees, not necessarily equal in size, but each no larger than a pre-set limit. (DOMs corresponding to real-world documents are, of course, substantially larger.)

One can compute the size of a DOM node recursively; for example, the size of a “b” node is seven bytes (<b>...</b>), plus the size of its children. The size of a text node is the number of bytes in the string.

A naïve DOM-segmentation technique is to perform a left-right traversal of the tree’s leaves, adjoining leaves to the current segment until adding the next leaf to the current segment would cause the current segment to exceed the specified size threshold. The segment counter is then incremented and a new segment begins. A problem with this approach is that it is insensitive to the inherent structure of the document. For example, it makes no effort to avoid splitting sibling or related nodes: two paragraphs belonging to the same story, for instance, or a heading and the following paragraph. Conversely, it is not positively disposed towards inserting a segment boundary at a natural seam in the page, such as before a long block containing a sequence of hyperlinks.

We now describe a more sophisticated technique, designed to insert seams at “structurally appropriate” locations in the DOM.

## 2.1 A clustering approach

We formulate the DOM segmentation problem as clustering. The algorithm starts by assigning each leaf to its own segment. Adjacent leaves are then aggregated together. Pairs are chosen according to a cost function which encourages merging related nodes (e.g. two adjacent paragraphs) and discourages merging unrelated nodes (e.g. two different frames). The cost function assigns an infinite cost to illegal merges; for instance, a merge that creates a too-large segment. The algorithm terminates when no finite-cost merges exist.

**Algorithm:** *Iterative DOM Segmentation*

**Input:** DOM document  $D$   
 Cost function  $C(x,y)$ : a non-negative penalty value for merging DOM segments  $x$  and  $y$

**Output:** A set of segments, each consisting of a (disjoint) set of contiguous DOM leaf nodes

1. Assign each leaf in  $D$  to its own segment.
2. Compute the cost  $C(x,y)$  of merging each adjacent pair  $(x,y)$  of segments in  $D$ .
3. If  $C(x,y)=\infty$  for all  $x,y$  then end.
4. Locate the  $x=x^*$  and  $y=y^*$  for which  $C(x,y)$  is minimal.
5. Merge segments  $x^*$  and  $y^*$ .
6. Go to step 2.

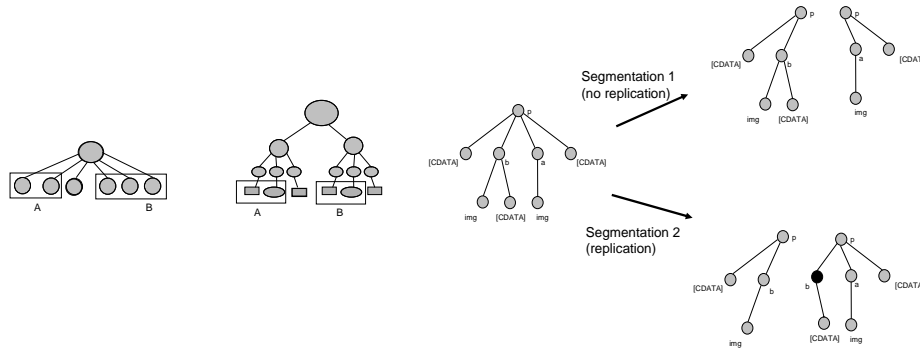
## 2.2 Constructing a cost function

The cost function  $C(x,y)$  guides the behavior of the segmentation algorithm. Different assignments for  $C(x,y)$  result in algorithms with different behaviors and different results.

We begin with a simple and intuitive cost function:  $C(x,y) = |x| + |y|$ . Here  $|s|$  is shorthand for “the size of the smallest subtree whose leaves are  $s$ ”; that is, the size of  $s$  plus all the ancestors of  $s$ . At a “micro” level, this cost function favors merging smaller segments together; at a “macro” level, the tendency is towards segments of balanced sizes.

Internal DOM nodes may have to be replicated when converting segments into well-formed documents. Often an internal node in the original DOM will appear in two or more segments. The cost function  $C(x,y) = |x| + |y|$  overcounts replicated nodes: if node  $n$  appears in segment  $x$  and  $y$ , then it should only be counted once when calculating the size of  $xy$ . We therefore need a subtractive term in the cost function, which we denote by  $r(x,y)$ : the size (in bytes) of the DOM content which appear in both  $x$  and  $y$ .

Lastly, we observe that two segments related only through a distant ancestor constitute a less compelling merge than two segments related through a parent. To account for this, we append an extra term  $d(x,y)$  to the cost function, where  $d(x,y)$  is the shortest path in the DOM from  $x$  to  $y$ .



**Figure 2:** Considerations in constructing the cost function. **Left:** The unaffiliated leaf can be merged into segment  $A$  or segment  $B$ . Assuming leaves are equal-sized, merging with  $A$  should be lower cost, since that merge balances the sizes of the resulting segments. **Middle:** The unaffiliated leaf node between  $A$  or  $B$  shares a grandparent with segment  $A$ , but only a great-grandparent with  $B$ . All else being equal, we expect a lower cost for merging with segment  $A$ . **Right:** Two candidate segmentations of a simple DOM. Notice how in the second segmentation, the black node was replicated, since it is a parent of leaves in both resulting segments. When it comes to node replication, less is better, since replicated nodes increase the total encoding size.

Many mobile browsers and/or gateways require that the incoming file not exceed a fixed size  $B$ . To accommodate this constraint, we require that the cost function assign an infinite cost to the merge of segments  $x$  and  $y$  when the size of the resulting segment,  $|xy|$ , exceeds  $B$ .

Combining these constraints together, we arrive at a parametric form for the cost function  $C$ :

$$C(x,y) = \begin{cases} \alpha(|x|+|y|) - \lambda r(x,y) + \beta d(x,y) & \text{if } |xy| < B \\ \infty & \text{otherwise} \end{cases} \quad (1)$$

For the experiments reported below, we used  $\alpha=1$ ,  $\beta=100$ , and  $\lambda=1$ . ( $\beta$  is unitless, but  $\alpha$  and  $\lambda$  are in bytes<sup>-1</sup>.) These values were determined beforehand, by manual optimization on a held-out collection of web content.

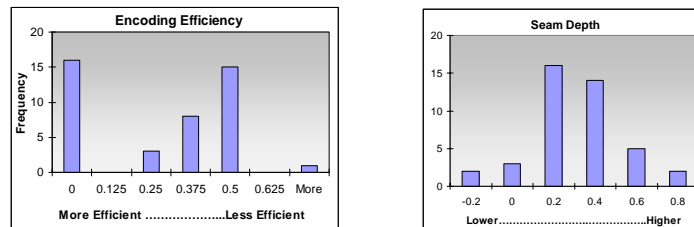
Real-world constraints may force a more complex cost function. For example, some mobile web browsers will dispatch HTTP GET requests for only the first  $n$  images in a web page, for a browser-specific value of  $n$ . After requesting  $n$  images, the browsers simply give up, issuing no further GET requests. To avoid this, one could adapt (1) to assign an infinite cost to  $C(x,y)$  when the number of `<img>` elements in  $xy$  exceeds  $n$ .

The algorithm has time requirements of amortized  $O(n \log n)$  and space requirements of  $O(n)$ , where  $n$  is the number of nodes in the tree. (Space prohibits a full analysis.)

### 3 Results

We applied the system to a subset of 41 of the largest web sites from the KeyNote consumer and business web sites<sup>1</sup>. This dataset included URLs as varied as [www.dilbert.com](http://www.dilbert.com), [www.intel.com](http://www.intel.com) and [www.fedex.com/us](http://www.fedex.com/us). For these experiments,  $B= 5000$  bytes. The algorithm was executed on a single-processor 2.4GHz x86 machine, on which the segmentation algorithm requires an average of 60ms of CPU time per web page.

The first experiment was designed to measure how close to “optimal” the algorithm is in its segmentation. We define the encoding efficiency as  $E=(N_c- N_l)/ N_l$  where  $N_c$  is the number of segments generated by the clustering segmentation algorithm for that web page, and  $N_l$  is the ideal number of segments: the total size of the web page divided by the pre-set size limit  $B$ . Note that the *ideal* segmentation doesn’t necessarily represent a *valid* segmentation; splitting a web page every  $B$  bytes is impossible because internal nodes will have to be replicated, pushing the size of some segments over the limit.



**Figure 3:** Two experiments on the KeyNote dataset. **Left:** Encoding efficiency compares the number of generated segments against the theoretical minimum number. **Right:** Relative DOM height of the seam nodes, compared with average depth of a DOM node. A value of zero means the seam appears at exactly the mean depth of an internal DOM node. Generally speaking, higher is better: a seam higher in the DOM separates higher-order structure in the DOM and causes less node replication.

Another quality measure relates to the placement of seams. Generally speaking, seams placed higher in the DOM (closer to the root) are preferable. A seam between two nodes high in the DOM tree is likely to divide two high-level structures: two tables, for instance, or perhaps a paragraph and a form. A seam between two nodes closer to the leaf level is more likely to split related content: an image and a paragraph, or two text blocks.

Define by  $\langle d \rangle$  the expected depth of a non-leaf node in the DOM, and by  $d_c$  the average depth of a computed seam in the DOM. We define the “seam height ratio” as  $S = (\langle d \rangle -$

<sup>1</sup> The full list of 50 is available at [www.keynote.com](http://www.keynote.com). Nine of these sites were small enough to fit within a single 5KB segment, and were discarded for these experiments.

$d_c/d$ . Intuitively,  $S$  measures (in relative terms) how much higher the computed seam is in the DOM, compared to the height of a randomly placed seam. The average value of  $S$  over the dataset was 0.22, with only four of the pages exhibiting a negative value of  $S$ . (In each of these cases, the page had only one seam.)

Also, for illustration, Figure 4 depicts the behavior of the system on a single web page.



**Figure 4:** Real-world example of the partitioning algorithm, this time with  $B=1400$ . In this case, the partitioning algorithm generated five segments, outlined in black.

## 4 Previous Work

Chen *et al* [1] have proposed a system that includes a page-segmentation procedure, which analyzes the document both in DOM *and* in pixel space. The domain of their proposed solution presupposes a certain macro-level structure of a web page (header, footer, left sidebar, right sidebar, body). This approach is not designed to account for network or device-imposed size limits.

Li *et al* [3] propose improving the quality of web search by dividing web pages into cohesive “micro-units,” each covering a single topic. The segmentation procedure proposed by the authors involves creating a tag tree (similar to a DOM), and then applying two heuristics to aggregate tree nodes into segments. These two heuristics—merge headings with the following content, and merge adjacent text paragraphs—may be sufficient for creating indexable fragments of the page, but they are too limited for the more general problem of segmentation: the resulting segments will be smaller than desirable, since the algorithm cannot merge segment pairs that don’t qualify under these two rules.

## References

- [1] Y. Chen, W. Ma and H. Zhang. Detecting web page structure for adaptive viewing on small form factor devices. In *Proceedings of the 12th Annual World Wide Web Conference*. Budapest, Hungary. 2003.
- [2] HTML Tidy Library Project: <http://tidy.sourceforge.net/>
- [3] X. Li, B. Liu, T. Phang and M. Hu. Using micro information units for Internet search. In *Proceedings of the 11th International Conference on Information and Knowledge Management*, McLean, VA. 2003.