A survey of potential XHTML diff/merge algorithms

Daniel Ehrenberg

Introduction

XHTML documents cannot be accurately compared and merged by simple line-by-line tools like diff3 because the tree structure may be compromised in both identifying the differences and performing a semi-automatic merge, and programs like Tidy are insufficient to clean this up. So we need a different method to compare these documents.

Several algorithms exist which may be suitable to diff HTML documents. Some algorithms, such as the Zhang-Shasha algorithm, analyzes two trees to find their minimal edit distance. Other algorithms, such as XyDiff, find a non-optimal edit distance between trees using some kind of heuristic, but incorporate a broader definition of edit operations, including moves. This tree view may, however, be inappropriate for HTML, as many elements do not induce a tree-like structure semantically, such as inline formatting elements (like). DaisyDiff presents a solution to this problem.

Several types of merging are possible. One strategy is to simply run the edit script of both versions, but this can cause inconsistencies in the output. Instead, either operational transformation or a diff3-like algorithm should be used to merge edit scripts. Alternatively, Lindholm's merge algorithm can be used directly, without using an explicit edit script.

A warning: All of the algorithms are fairly difficult to understand. I don't understand all of them; it took me months to figure out the Zhang-Shasha algorithm. You don't need to understand the details of each algorithm to roughly evaluate its advantages and disadvantages.

The Problem

The obvious solution to the problem is to just use diff3 to merge the differences. To allow diff3 to see the differences better, the tags are all split onto their own lines in a normalization pass. But diff3 this ignores the (superficial) tree structure of XML. For example, look at the following example:

 This is some text 	 This is some text 	 This is some text 	 This is some text
Original	**Part bolded**	**Part italicized**	**Line-by-line merge**

In this case, the Tidy program could fix the generated HTML. But in other cases, things get more complicated and difficult to fix up. Additionally, in a system like a WYSIWYG editor, where the user is not exposed to the HTML itself, the diff would indicate that, in the "part bolded" variant, the lines "" and "" are added. But it should give the information that "This is some" was previously not bold and it became bold.

Additionally, it'd be nice if we could track moves. If someone, say, swaps two paragraphs, it'd be nice if that could be tracked and reported rather than explained as a deletion and insertion. But this isn't absolutely necessary.

Some definitions

An XML document can be viewed as an ordered tree, where each node has an unbounded number of children and each internal node has a label. So we can solve these problems of diffing and merging XML by solving a more general problem on ordered trees, as most authors have. There are some specific aspects of XML which deserve mention (attributes, which are guaranteed to have unique names within a node; IDs, which are guaranteed to be unique within a document), but these are minor aspects which we can ignore for most of the time.

To avoid confusion, I'll define some terms I've been using or will soon start using. When I talk about "diffing", what I mean, formally, is generating an "edit script", or list of changes between two documents that can be used to get the modified document from the original. Sometimes, these edit scripts are invertible, but not always. When I talk about a "merge", I mean a way to reconcile the changes between documents to incorporate both of these changes. A merge can be an operation on edit scripts or it can be done directly on a tree matching. A "matching" is a set of correspondences between nodes in different trees; it is the basis for doing either a diff or a merge, and it's difficult to do efficiently.

The idea of diffing trees is closely related to that of comparing strings. It could be thought of as a generalization. For this, there is a $O(n^2)$ solution using dynamic programming [1] and there is also a O(nd) algorithm, where d is the edit distance, discovered by Eugene Meyers, using more advanced techniques [18]. The O(nd) algorithm is sometimes referred to as the GNU diff algorithm, for some reason. For merging, the GNU diff3 program presents a useful model which has been formally analyzed [19].

Selkow's simple algorithm

If we use a simple definition of edit distance, then the problem is relatively simple. If all you can do is insert and delete leaves of the tree, then the regular old algorithm for the longest common subsequence problem [1] can be used recursively: calculate the distance between two subtrees M and N, make a |M| x |N| array and initialize it in the same way is if you were solving Levenshtein distance. Then, fill in the array using the same technique as with Levenshtein distance, but for replacement cost, use the edit distance between the children rather than 1 if they are not equal and 0 if they are equal. Once this distance is calculated, an edit script can be traced through the intermediate tables generated. See [3] for details.

The weakness here is in the definition of edit distance. If we have a bunch of formatted text and put a big tag around it, then this algorithm will register that it was all deleted, a div node was inserted, and the text was inserted into the div node. The complexity is O(nmd), where n and m are the numbers of leaves in the two trees, and d is the maximum depth. This isn't too good, if the algorithm is supposed to be usable on large documents.

The Zhang-Shasha algorithm and extensions

The Zhang-Shasha algorithm [2] is the basic starting point when thinking about tree matching, diffing and merging. Except it isn't that basic. Dennis Shasha and Kaizhong Zhang created an algorithm to solve the approximate tree matching problem, which they described in the book Pattern Matching Algorithms. Here's the basic idea: we want to see how similar two trees are, by a weighted edit distance metric. The edit script has three operations, similar to Levenshtein distance: add a node (optionally including a contiguous subsequence of the parent node), delete a node (putting children in the parent node), and relabel a node.

With this, they were able to come up with an algorithm of complexity (basically) O((n log n)^2), where n is the number of nodes in the tree. So this can get you a matching and an edit script, or diff between two trees. This isn't great, but it's much better than previous algorithms. For large documents, it really isn't very good.

But this doesn't describe all of the changes that might take place in a tree structure that we might want to record. It constrained the edit operations so the search space would be smaller, but it comes at the cost of being less accurate in practice. For example, in Zhang-Shasha proper, moving a node from one place to another is recorded as deleting the node from one place and inserting it into another place. Another issue is that inserting or deleting a subtree is recorded as inserting the node, then inserting each of its children, or deleting the leaf nodes recursively up until you delete their parent. This all leads to counterintuitive diffs, as far as human readability goes, unless there's a postprocessing stage for that purpose.

So David Barnard, Gwen Clarke and Nicholas Duncan got together to create a modified algorithm that accommodated this modified definition of edit distance [3]. It adds three additional operations: insertTree, deleteTree, and swap. Unfortunately, this doesn't account for copying nodes, or for moves that aren't within the same parent.

Some tree matching heuristics

So, it's not very good that the Zhang-Shasha algorithm is quadratic. In fact, in many cases, it's unacceptable. For example, in my case, where I might sometimes have to compare XHTML documents which are very long, it's unacceptable. But there are some algorithms which run in a time which is dominated by the number of nodes multiplied by the edit distance, or O(ne) in effect.

There's one algorithm called FastMatch, which has insert leaf, delete leaf, update and general move operations [4]. They work on getting an edit script and matching at the same time, but the algorithm starts by matching as much as possible, top-down, before proceeding to calculate the differences. This yields a complexity of O(ne+e^2). A related algorithm, described in Chapter 7 of Tancred Lindholm's master's thesis [5] incorporates tree insertion and deletion operations for a complexity of O(ne log n).

It's important to note that both of these will still be O(n^2) in the worst case. A different XML matching algorithm was described by Grégory Cobéna in his master's thesis [6]. Cobéna calls his algorithm BULD, which stands for bottom-up lazy-down. The key to the algorithm is that, for each node, there is a hash value and a weight, both calculated bottom-up. Exact equivalence between nodes can be approximated by equal hash values, and you search for the equal hash values of nodes that have been inserted on a maxheap by weight. In his thesis, Cobéna also goes into depth about his invertible edit

script format. This algorithm doesn't necessarily generate the optimal diff, but in experiments it generates a very good one, and with a worst-case time of O(n log n). An empirical study on XML diff/merge tools [7] recommended the use of the software based off this, called XyDiff, because it was significantly faster in practice.

Three-document merge

Creating an edit script is all well and good, but it's only half of the problem: the merge. Remember that a three-document merge is one where we have the original document and two modified versions, and we want to create a fourth version with both modifications together. Here was my idea: create an edit script for both modified versions with respect to the original, then do one followed by another, with repeated modifications done only once. We know there's a conflict if the order matters, in terms of which comes first in applying to the original document.

But this will come up with more conflicts than actually exist. For example, say some node A has four children, B C D and E. In one change, we insert a new node X after B as a child of A, and in another change, we insert a node Y after D as a child of A. So a sensible merge would have A's new children be B X C D Y E, in that order. But with the model described above, there would be an edit conflict!

One solution to this is the more general strategy of operational transformation [8]. The basic idea for this technique as applied here is that, if we insert Y after inserting X, we have to add 1 to the index that Y is being inserted. If, on the other hand, Y is inserted first, we don't have to add one to the index that X is inserted on. This technique leads to fewer conflicting merges, or in OT lingo, it converges in more cases. There are a few formal properties of an operational transformation that have only recently been proven correct in the best-known algorithms. Pascal Molli used operational transformation, together with Cobéna's diff algorithm and format, in his So6 synchronization framework [9].

Tancred Lindholm went a different route altogether in creating a three-way merge, throwing out the edit script and basing it on a tree matching [10]. He based the merge definition on several large, by-hand merges of realistic XHTML and other XML documents, figuring out how to automate the results. But the algorithm isn't perfect; it, like all of the other ones mentioned so far, cannot properly handle the motivating example at the beginning. Unfortunately, I don't understand the algorithm.

DaisyDiff and my ad-hoc idea

In simple tests, none of the algorithms handled the merge at the beginning of this article properly. Proper XHTML output would be " This is some   text ", but to do this would require two things: knowledge of the XHTML schema and treating text nodes non-atomically. Instead, most algorithms treat a text node as one solid thing, which either equals another text node or doesn't, and has no subdivisions. This leads to output for the merge like " This is some text This is some text ". For a practical implementation, output like this may be reasonable if the conflict is presented well, though it's far from optimal, and it's far from obvious how to present the conflict.

A solution to this problem is to radically changes the approach to the problem is DaisyDiff [11]. Unfortunately, the only thing written about it is in Dutch. The idea is that

the linear order of the words is the most important thing in an XML document, and that surrounding elements should be treated as auxiliary. So, basically, an LCS is calculated between the two strings of words in the two documents, and then an LCS is calculated, for each word, between the stacked up parent tags. From this, we can create a sort of edit script that's very readable, if the differences in parent tags are coalesced. To present this visually, DaisyDiff uses strikethroughs for deleted words, underlines for inserted words, and special roll-over text boxes to explain changes in formatting. No merge algorithm has been constructed, though it should be possible to do so using something based on the diff3 algorithm.

I have a somewhat different, potentially easier-to-implement idea based on diff3. Diff3 is given the original document and two modified documents. It calculates the line-by-line diff between the original document and each of the modified documents. Then, it merges the changes together. See [19] for a more complete description.

So, my idea is to do diff3, but at each step make sure that we're dealing with some well-formed unit in XML. So, after the two two-way line-by-line diffs are made, they each (separately) need to have each insertion or deletion to have its scope expanded out to the smallest surrounding sequence of XML elements (in tree form). Insertions become deletions followed by insertions, and deletions become insertions followed by deletions, in general, though often the opposite preceding operations will be unnecessary. And these modified operations take place on elements of the DOM. The resulting edit scripts will do the same thing as the old edit scripts, though they will be longer. Now, the two edit scripts can be merged as they are in diff3 (see [19]), with the continued restriction that XML tags not be separated, opening from closing tag. I am still unsure whether this will work fully.

Implementations and further reading

Many, but not all, of these algorithms have usable open-source implementations available. Selkow's algorithm has been implemented in Lisp to get the diff between two s-expressions [12], and this has reportedly been used for an HTML diff (though the author in fact reinvented this algorithm without reference to Selkow). The xmldiff project implements the Zhang-Shasha algorithm and FastMatch in Python [13]. I cannot find a usable implementation of the extended Zhang-Shasha algorithm. For the XyDiff algorithm, there are two implementations: XyDiff in C++ [14] and jXyDiff in Java [15]. For Lindholm's three-way merge algorithm, there is the 3DM project in Java [16]. Unfortunately, all of these appear to be unmaintained since 2006. DaisyDiff's implementation in Java [17] looks to be still maintained, though. There are actively maintained commercial implementations of these, for use with XML, but most of them make it hard to tell what algorithm is used from the documentation, and the fact that they're closed-source makes it difficult to extend them to fit the specific domain of HTML.

For a survey on tree edit distance in an abstract, theoretical way, see [20]. For a survey of existing XML diff tools, see [21]. A good book about this topic, in the more general sense, is [22] and very technical, hard-to-read one is [23]. Additionally, getting the XML into canonical form [24] is generally useful for diffing.

By the way, if you didn't notice, the original problem is still not perfectly solved by any one algorithm.

Citations

[1] Wikipedia. “Longest common subsequence problem.” http://en.wikipedia.org/wiki/Longest_common_subsequence_problem

[2] Dennis Shasha, Kaizhong Zhang. “Approximate Tree Pattern Matching” http://citeseer.ist.psu.edu/shasha95approximate.html (This is actually really hard to read, and it might be better to look at their original paper in SIAM, K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problems. SIAM J. Computing, 18(6):1245-1252, December 1989. Unfortunately, that paper is not available for free online, or through ACM.)

[3] David Barnard, et al. “Tree-to-tree Correction for Document Trees.” http://citeseer.ist.psu.edu/47676.html (This has a good survey of older algorithms, but it’s little light on explanations.)

[4] Sudarshan Chawathe, et al. “Change Detection in Hierarchically Structured Information.” http://citeseer.ist.psu.edu/chawathe96change.html

[5] Tancred Lindholm. “A 3-way Merging Algorithm for Synchronizing Ordered Trees.” http://www.cs.hut.fi/~ctl/3dm/thesis.pdf

[6] Gregory Cobéna. “Change management of semi-structured data on the web.” http://gregory.cobena.free.fr/www/Publications/thesis_draft.pdf

[7] Sebastian Rönnau, et al. “Towards XML version control of office documents.” http://portal.acm.org/citation.cfm?doid=1096601.1096606

[8] Wikipedia. “Operational transformation.” http://en.wikipedia.org/wiki/Operational_transformation

[9] Pascal Molli, et al. “Supporting Collaborative Writing of XML Documents.” http://www.loria.fr/~molli/pmwiki/uploads/Main/OsterICEIS07.pdf

[10] Tancred Lindholm. “A Three-way Merge for XML Documents.” http://citeseer.ist.psu.edu/741860.html

[11] Guy Van den Broeck. “Stageverslag: Daisy Diff.” http://daisydiff.googlecode.com/files/stageverslag.pdf

[12] Michael Weber. Diff-Sexp source code. http://www.foldr.org/~michaelw/lisp/diff-sexp.lisp

[13] Logilab. xmldiff homepage. http://www.logilab.org/859

[14] XyDiff homepage. http://gemo.futurs.inria.fr/software/XyDiff/cdrom/www/xydiff/index-eng.htm

[15] jXyDiff homepage. http://potiron.loria.fr/projects/jxydiff

[16] 3DM homepage. http://tdm.berlios.de/3dm/doc/index.html

[17] DaisyDiff homepage. http://code.google.com/p/daisydiff/

[18] Eugene W. Myers. “An O(ND) Difference Algorithm and Its Variations.” http://www.xmailserver.org/diff2.pdf

[19] Sanjeev Khanna, et al. “A Formal Investigation of Diff3.” http://www.cis.upenn.edu/~bcpierce/papers/diff3-short.pdf

[20] Philip Bille. “A Survey on Tree Edit Distance and Related Problems.” http://grfia.dlsi.ua.es/ml/algorithms/references/editsurvey_bille.pdf

[21] Grégory Cobéna, et al. “A comparative study for XML change detection.” http://citeseer.ist.psu.edu/696350.html

[22] Dan Gusfield. Algorithms on Strings, Trees and Sequences: Computer Science and

Computational Biology.
[23] Alberto, Apostolico et al. Pattern Matching Algorithms.
[24] W3C Recommendation: Canonical XML, version 1.0. http://www.w3.org/TR/xml-c14n