Lowdown Diffing Engine

Kristaps Dzonsons

BSD.lv

Lowdown Diffing Engine

In this paper, I briefly describe the “diff” engine used in lowdown-diff(1) tool in lowdown. The work is motivated by the need to provide formatted output describing the difference between two documents—specifically, formatted PDF via the **-Tms** output, although **-Thtml** and the other output modes are of course supported.

This documents an early work in progress—both source code and documentation. The source is documented fully in diff.c. This paper itself is available as diff.md, or downloadable as diff.pdf. Please direct comments to me by e-mail or just use the GitHub interface.

For a quick example of this functionality, see diff.diff.html (or diff.diff.pdf), which shows the difference between this document and a [fabricated] earlier version. .

Introduction

Let two source files, *foo.md old.md* and *bar.md new.md* , refer to the old and new versions of a file respectively. The goal is to establish the changes between these snippets in formatted output. Let ’s begin with the old version, *old.md* .

```
*Lorem* ipsum dolor sit amet, consectetur adipiscing elit, sed do
eiusmod tempor incididunt ut [labore](index.html) et dolore magna
aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco
laboris nisi ut _aliquip_ ex ea commodo consequat. Duis aute irure dolor
in reprehenderit...
```

In the new version, *new.md*, I add some more links and styles.

```
*Lorem* ipsum dolor sit amet, consectetur adipiscing elit, sed do
eiusmod tempor incididunt ut [labore](index.html) et dolore [magna
aliqua](index.html). Ut enim ad minim veniam, quis nostrud exercitation
ullamco laboris nisi ut _aliquip_ ex ea commodo consequat. Duis *aute
irure* dolor in reprehenderit...
```

The most simple way of viewing changes is with the venerable diff(1) utility. However, this will only reflect changes in the input document—not the formatted output.

```
--- old.md      Tue Oct 17 11:25:01 2017
+++ new.md      Tue Oct 17 11:25:01 2017
@@ -1,5 +1,5 @@
 *Lorem* ipsum dolor sit amet, consectetur adipiscing elit, sed do
-eiusmod tempor incididunt ut [labore](index.html) et dolore magna
-aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco
-laboris nisi ut _aliquip_ ex ea commodo consequat. Duis aute irure dolor
-in reprehenderit...
+eiusmod tempor incididunt ut [labore](index.html) et dolore [magna
+aliqua](index.html). Ut enim ad minim veniam, quis nostrud exercitation
+ullamco laboris nisi ut _aliquip_ ex ea commodo consequat. Duis *aute
+irure* dolor in reprehenderit...
```

Not very helpful for any but source-level change investigation. And given that Markdown doesn't encourage the usual "new sentence, new line" of some languages (like mdoc(7)), even this level of change analysis is difficult: a single change might affect multiple re-wrapped lines.

A similar possibility is to use wdiff(1), which produces a set of word-by-word differences.

```
*Lorem* ipsum dolor sit amet, consectetur adipiscing elit, sed do
eiusmod tempor incididunt ut [labore](index.html) et dolore [-magna
aliqua.-] {+[magna aliqua](index.html).+} Ut enim ad minim veniam, quis
nostrud exercitation ullamco laboris nisi ut _aliquip_ ex ea commodo
consequat. Duis [-aute irure-] {+*aute irure*+} dolor in
reprehenderit...
```

One could then extend the Markdown language to accept the insertion and deletion operations and let the output flow from there. (In fact, that was my first approach to solving the problem.)

Unfortunately, doing so entails extending a language already prone to extension and non-standardisation. Here is some added text. More unfortunately, a word-based diff will not be sensitive to the shmarkdown Markdown language itself, and in establishing context-free foo bar baz sequences of similar words, will overrun block and span element boundaries.

On the other end of the spectrum are difference tools specific to the output media.

One can directly analyse PDF output using (for example) the poppler tools, which would extract text, then examine the output with a linear difference engine such as Diff, Match, Patch. This is not an optimal solution because, as with the word diff above, it only compares words and cannot distinguish semantic artefacts such as in italics, links, code blocks, and so on.

There are even more HTML diff tools available, so it's tempting to use one of these tools to produce an HTML file consisting of differences, then further use a converter like wkhtmltopdf to generate PDFs.

Since the HTML difference engines often respect the structure of HTML, this is much more optimal in handling semantic difference. However, re-structuring the difference does not easily produce a document of the same style or readability as the PDFs themselves.

The most elegant (and reliable) solution is to attack the problem from the language-level itself. Since the lowdown(3) library is able to produce a full parse tree for analysis, it stands to reason, given the wealth of literature on tree differences (instead of the usual linear difference, as in the case of diff(1) and friends), one can work within the language to produce differences. [1]

Algorithm

The algorithm is in effect an ordered tree diff. I began with well-studied algorithms for a well-studied problem: XML tree differences. (The HTML difference tools described above inherit from those algorithms.) For an overview of algorithms, see Change Detection in XML Trees: a Survey[2] . I base the lowdown-diff(1) algorithm off of Detecting Changes in XML Documents[3] .

The reason for this choice instead of another is primarily the ease in implementation. Moreover, since the programmatic output of the algorithm is a generic AST, it's feasible to re-implement the algorithm in different ways, or augment it at a later date.

The BULD algorithm described in this paper is straightforward. It begins with a short sanitisation pass.

1. Annotate each node in the parse tree with a hash of the subtree rooted at the node, inclusive. (diff.c, `annotate_sigs()`)
2. Annotate each node with a weight corresponding to the subtree rooted at the node. (diff.c, `annotate_sigs()`)
3. Enqueue the new document's root node in a priority queue ordered by weight. Then, while the priority queue is non-empty: (diff.c, `lowdown_diff()`)

[1] This is just to illustrate a removed footnote.

[2] Change Detection in XML Trees: a Survey (2005), Luuk Peters.

[3] Detecting Changes in XML Documents (2002), Gregory Cobena, Serge Abiteboul, Amelie Marian.

    1. Pop the first node of the priority queue.
    2. Look for candidates in the old document whose hash matches the popped node's hash. (diff.c, `candidate()`)
    3. Choose an optimal candidate and mark it as matched. (diff.c, `optimality()`)
    4. If the no candidates were found, enqueue the node's children into the priority queue.
    5. A a candidate was selected, mark all of its subtree nodes as matching the corresponding nodes in the old tree ("propagate down"), then mark ancestor nodes similarly ("propagate up"). (diff.c, `match_up()`, `match_down()`)

4. Run an optimisation phase over the new document's root node. (diff.c diff.c , `node_optimise()` `node_optimise_bottomup()` and `node_optimise_topdown()`)

5. Step through both trees and create a new tree with nodes cloned from both and marked as inserted or deleted. (diff.c, `node_merge()`)

My implementation changes or extends the BULD algorithm in several small ways, described in the per-step documentation below.

Sanitise

Before the BULD algorithm is run, the input tree is sanitised. This process merges all adjacent text nodes into a single text node. By doing so, possible differences are pushed into large blocks of contiguous text—which in this case are managed by the word-difference algorithm described later in this paper.

Annotation

Each node in the tree is annotated with a hash and a weight. The hash, MD5, is computed in all data concerning a node. For example, normal text nodes (`LOWDOWN_NORMAL_TEXT`) have the hash produced from the enclosed text. Autolinks (`LOWDOWN_LINK_AUTO`) use the link text, link, and type.

There are some nodes whose data is not hashed. For example, list numbers: since these may change when nodes are moved, the numbers are not part of the hash. In general, all volatile information that may be inferred from the document structure (column number, list item number, footnote number, etc.) is disregarded.

Non-leaf nodes compute their hashes from the node type and the hashes of all of their children. Thus, this step is a bottom-up search.

Node weight is computed exactly as noted in the paper.

Optimal candidacy

A node's candidate in the old tree is one whose hash matches. In most documents, there are many candidates for certain types of nodes. (Usually text nodes.)

Candidate optimality is computed by looking at the number of parent nodes that match on both sides. The number of parents to consider is noted in the next sub-section. The distance climbed is bounded by the weight of the sub-tree as defined in the paper.

In the event of similar optimality, the node "closest" to the current node is chosen. Proximity is defined by the node identifier, which is its prefix order in the parse tree.

"Propagate up"

When propagating a match upward, the distance upward is bound depending on the matched sub-tree as defined in the paper. This makes it so that "small" similarities (such as text) don't erroneously match two larger sub-trees that are otherwise different. Upward matches occur while the nodes' labels are the same, including attributes (e.g., link text).

I did modify the algorithm to propagate upward "for free" through similar singleton nodes, even if it means going beyond the maximum number allowed by the sub-tree weight.

Optimisation

The lowdown-diff(1) algorithm has two optimisations, both lightly derived from the paper: top-down and bottom-up propagation.

Top-down

The top-down optimisation, which is performed first, takes matched nodes and matches un-matched, non-terminal children by label. The children examined must be siblings of adjacent matching nodes.

This is useful when, say, a document consists of several paragraphs where the text has changed within paragraphs. It won't be able to match the text content, but it will match the paragraphs, which will push the difference downward in the tree.

Bottom-up

In the bottom-up propagation, the weight of any given sub-tree is used to compute how high a match will propagate. I extend the paper's version optimisation by looking at the cumulative weight of matching children.

This works well for Markdown documents, which are generally quite shallow and text-heavy.

For each unmatched non-terminal node with at least one matched child, the weights of all matched children with the same parents (where the parent node is equal in label and attributes to the examined node) are computed. If any given parent of the matched children has greater than 50% of the possible weight, it is matched.

Merging

The merging phase, which is not described in the paper, is very straightforward. It uses a recursive merge algorithm starting at the root node of the new tree and the root node of the old tree.

1. The invariant is that the current node is matched by the corresponding node in the old tree.
2. First, step through child nodes in the old tree. Mark as deleted all nodes not being matched to the new tree.
3. Next, step through child nodes in the new tree. Mark as inserted all nodes not having a match in the old tree.
4. Now, starting with the first node in the new tree having a match in the old tree, search for that match in the list of old tree children.
5. If found, mark as deleted all previous old tree nodes up until the match. Then re-run the merge algorithm starting at the matching child nodes.
6. If not found, mark the new node as inserted and return to (2).
7. Continue (after the recursive step) by moving after both matching nodes and returning to (2).

If adjacent nodes are un-matched normal text, the normal text nodes are compared word-by-word to compute a shortest-edit script. This is enabled by libdiff, which implements an algorithm for computing the longest common subsequence[4] . See [5] for background information.

Tables

Tables are currently in the "hacks" state in that they're considered as opaque bodies. Tables that have changed in any way are simply deleted and re-added: there's no attempt to discern actual differences.

There are ways to reduce this opacity, such as being able to detect and account for added or removed rows. However, ultimately there is some opacity in that changed table headers do not have a representable form in the output.

---

[4] An O(NP) sequence comparison algorithm (1990), Sun Wu, Udi Manber, Gene Myers.

[5] An Algorithm for Differential File Comparison(1976), James W. Hunt, M. Douglas McIlroy.

Metadata

Like tables, metadata key-value pairs are opaque bodies. Metadata has a complex relationship with Markdown documents, which leaves how to handle the "difference" uncertain.

The difference engine, after computing differences like any other opaque nodes, simply passes the difference to front-ends, which determine how to handle this for themselves. For front-ends that use the metadata in creating document headers (e.g., HTML, LaTeX, roff), the policy is not to process deleted metadata.

Thus, metadata won't strictly represent the document differences.

Footnotes

Footnotes required some consideration because the order in which the definitions and references are printed must be synchronised. Now a process keeps track of all references (added, deleted, unchanged) and renumbers then. When references are emitted, these are emitted in the correct order.

API

The result of the algorithm is a new tree marked with insertions and deletions. These are specifically noted with the `LOWDOWN_CHNG_INSERT` and `LOWDOWN_CHNG_DELETE` variables.

The algorithm may be run with the `lowdown_diff()` function, which produces the merged tree.

A set of convenience functions, `lowdown_buf_diff()` and `lowdown_file_diff()`, also provide this functionality.

Future work

There are many possible improvements to the algorithm.

Foremost is the issue of normal text nodes. There should be a process first that merges consecutive text nodes. This happens, for example, when the `w` character is encountered at any time and might signify a link. The parsing algorithm will start a new text node at each such character.

The merging algorithm can also take advantage of libdiff when ordering the output of inserted and deleted components. Right now, the algorithm is simple in the sense of stopping at the earliest matched node without considering subsequences.

Lastly, the **-Tms** and **-Tman** output needs work to make sure that the insert/delete macros don't disrupt the flow of text.

Document last updated: $Date$