Project Thesis 2 — Module 7302 — BUAS-EIT

Difference Algorithms and Merging Strategies for Structured Documents

Lorenz Schori ⟨schol2@bfh.ch⟩

June 17, 2011

Supervisor: Prof. Dr. Olivier Bieberstein

Abstract

When dealing with long texts such as source code, articles or even books one often wants to track changes introduced by different authors. That's why we find efficient difference algorithms and merging strategies in many source code management and content management systems which produce good results when applied to flat text files.

This project aims at studying "diffing" and "merging" techniques for structured documents. The term "diffing" refers to the process of comparing two documents, structured or not, by differential analysis. The focus of this work lies on algorithms applicable to XML documents.

Contents

1. Introduction 1
1.1. Motivation and Goals 1

2. Comparing and merging flat text files 1
2.1. Fundamentals 1
2.1.1. Simple model for string sequences 1
2.1.2. Conforming edit script 2
2.1.3. Longest common subsequence vs. shortest edit script 2
2.2. GNU Diffutils 2
2.2.1. Edit scripts 3
2.2.2. Patch format 4
2.2.3. Invertibility 4
2.2.4. Commutativity 5
2.2.5. Recap 5

3. The tree to tree correction problem 5
3.1. Towards diffing structured data 5
3.1.1. Tree model 6
3.1.2. Requirements for patch format 7
3.1.3. Tree operations 7
3.2. The generic tree to tree correction problem 7
3.2.1. Tai (1979) 7
3.2.2. Zhang and Shasha (1989) 9
3.2.3. Klein (1998) 10
3.2.4. DMRW (2009) 10
3.2.5. Recap 10
3.3. Unit cost algorithms 11
3.3.1. mmdiff and xmdiff (1999) 11
3.4. Diff algorithms for structured hierarchical data 11
3.4.1. Extended Zhang Sasha (1995) 11
3.4.2. FastMatch EditScript - FMES (1996) 11
3.4.3. BULD (2001) 12
3.4.4. faxma (2006) 13
3.4.5. XCC (2010) 13
3.4.6. A note on the *move* operation 14
3.4.7. Recap 14

4. Representing changes in tree structures 14
4.1. Visualization of changes 14
4.1.1. Delta tree 15
4.1.2. DeltaXML v2 and XMLR 15

4.2. Edit script and patch formats . . . 17
4.2.1. RFC 5261 - XML Patch Operations Framework . . . 17
4.2.2. XCC patch format . . . 19

5. Conclusion and Future Work 21
5.1. Results . . . 21
5.1.1. Research on diff algorithms . . . 21
5.1.2. Research on patch formats and merging strategies . . . 21
5.2. Future Work . . . 22
5.2.1. Dynamic diff granularity . . . 22
5.2.2. Merge capable delta tree style patch format . . . 22
5.2.3. Diffing graphs along a spanning tree . . . 22

A. Reflection 24

B. Project Management 24
B.1. Methodology . . . 24
B.2. Project Livecycle . . . 24
B.3. Tools . . . 24
B.4. Chart . . . 25

C. List of Figures 26

D. Listings 26

E. References 26

1. Introduction

1.1. Motivation and Goals

The goal of the present project thesis is to identify algorithms and implementations capable of calculating and serializing differences between structured documents as well as algorithms and implementations capable of applying these to the original document or slightly modified versions thereof.

In order to find the best representation of changes between structured documents, we first need to understand the desired characteristics in terms of space efficiency and application robustness of the serialized forms, named "patches" in this context. We explain this aspects by looking at some patch formats produced by the GNU diffutils. Then we apply the concepts revealed by this analysis to existing patch formats for XML documents, especially DeltaV2 [9], RFC 5261 [23], DUL [17] and XCC [21].

Further we discuss important diff algorithms and compare them in terms of time and space complexity.

2. Comparing and merging flat text files

2.1. Fundamentals

In order to describe and compare the different approaches taken in the analyzed patch formats we need to define some terms and expressions. We present a very simple model here and introduce more aspects as we get deeper into the matter.

2.1.1. Simple model for string sequences

The difference between two documents is commonly represented as a list of operations that will convert the first document into the second [16]. A good diff-algorithm will try to minimize the number of operations required in order to produce a space efficient representation of the changes between two documents. The number of edit commands is commonly referred to as the *edit distance*.

Most of the time we're interested in the actual changes instead of just a bare number showing how much a document differs from another. The *edit script* represents this list of operations introduced above.

Definition 1 Edit Script

edit script: sequence of operations required to convert one document into another.

anchor: data unit used by the patching algorithm to identify a location where an operation must be applied.

Sometimes we need even more information in order to allow automatic verification of the applicability of a given edit script to a target document. We refer to those more sophisticated type of change representation as a *patch*.

Definition 2 Patch

patch: an edit script with context.

context: nodes and literals that all compared documents have in common in the neighborhood of where edit script operations will be applied.

full context patch: a patch containing the intersection of all compared documents in addition to the edit script.

At the very minimum two operations are required to express the conversion from one document into another: *insert* and *delete*.

Definition 3 Operation

insert $(anchor, value[, context])$

delete $(anchor[, oldvalue, context])$

Other than in *patches* operations in *edit scripts* do not include context information.

2.1.2. Conforming edit script

Usually it is not enough to just compute any edit script, instead one wants to find a particular solution conforming to a given criterion. This requirement is easy to fulfill by introducing a cost-function for operations such that we can find a conforming edit script by calculating the total cost of each candidate and selecting the one with minimal cost.

2.1.3. Longest common subsequence vs. shortest edit script

Given a constant cost function ($cost : op \rightarrow 1$) a *minimal conforming edit script* between two sequences A and B is identical to the *shortest edit script*. Myers [18] has shown that the computation of the shortest edit script is dual to the problem of finding the *longest common subsequence* (LCS) of the two sequences A and B.

2.2. GNU Diffutils

The GNU diff command compares text files on a line-by-line basis. The underlying algorithm [18] performs well for large text files with few differences. Therefore it is widely used to track changes to files by source code management tools.

Because the GNU diff command provides a variety of output formats it is perfectly suited to show some aspects of *edit scripts* and *patches*.

2.2.1. Edit scripts

Listing 1: Original Document

```
1  You must have a beer and an
2  airline. It helps if you have
3  some kind of a football team, or
4  hm, i can't remember exactly,
5  some atomic weapons, but at the
6  very least you need a beer.
7
8  Fredy Something
```

Listing 2: Modified Document

```
1  You can't be a real country
2  unless you have a beer and an
3  airline. It helps if you have
4  some kind of a football team, or
5  some nuclear weapons, but at the
6  very least you need a beer.
7
8  Frank Zappa
```

Listing 3: Resulting Edit Script

```
 1  8c
 2  Frank Zappa
 3  .
 4  4,5c
 5  some nuclear weapons, but at the
 6  .
 7  1c
 8  You can't be a real country
 9  unless you have a beer and an
10  .
```

Line 1: Delete last line from original file

Lines 2-3: Insert line "Frank Zappa"

Line 4: Delete line 4 and 5

Line 5-6: Insert line "some nuclear weapons, but at the"

Line 7: Delete line 1

Line 8-10 Insert beginning of quote

Given the input documents in Listing 1 and Listing 2 invoking `diff` with the `-e` option will produce the *edit script* shown in Listing 3. We can easily identify the operations executed by the *patch* utility when invoked with this script on the original file (explained in Listing 3 in the right column).

This example also shows another important aspect of *edit scripts*: We need some means to address the lines which should be deleted and also the location where new ones must be inserted. In this simple edit script format line numbers are used for that purpose.

It is easy to imagine what would happen if the edit script is applied to a slightly modified version of the original document, e.g. if someone saved that file with an additional newline character at the beginning of the file (Listing 4). Nothing hinders us to apply the edit script to the modified file but the result is disastrous (Listing 5):

Listing 4: Slightly modified original

```
1
2  You must have a beer and an
3  airline. It helps if you have
4  some kind of a football team, or
5  hm, i can't remember exactly,
6  some atomic weapons, but at the
7  very least you need a beer.
8
9  Fredy Something
```

Listing 5: Corrupt document

```
1  You can't be a real country
2  unless you have a beer and an
3  You must have a beer and an
4  airline. It helps if you have
5  some nuclear weapons, but at the
6  some atomic weapons, but at the
7  very least you need a beer.
8  Frank Zappa
9  Fredy Something
```

2.2.2. Patch format

Context sensitive patch formats are used to prevent data corruption when edit scripts are applied to files, which are not guaranteed to contain the exact original content. In addition to the raw changes, context sensitive patches also provide information about the surrounding lines where the operations should take place. This allows the patching program to verify if a given change can be carried out on the specified address and to look for alternative locations if this is not the case.

Using GNU diff we can produce context sensitive patches in the widely used unified diff format using the `-u` switch (Listing 6). Groups of local changes are packed into so called *hunks*.

Listing 6: Resulting Edit Script

```
1  --- /tmp/src.txt        2011-03-23 08:42:15.000000000 +0100
2  +++ /tmp/tgt.txt        2011-03-23 08:42:08.000000000 +0100
3  @@ -1,8 +1,8 @@
4  -You must have a beer and an
5  +You can't be a real country
6  +unless you have a beer and an
7   airline. It helps if you have
8   some kind of a football team, or
9  -hm, i can't remember exactly,
10 -some atomic weapons, but at the
11 +some nuclear weapons, but at the
12  very least you need a beer.
13
14 -Fredy Something
15 +Frank Zappa
```

Line 1-2: Meta information on files

Lines 3: Introduce new hunk from line 1 to 8 in source file to line 1 to 8 in target file.

Line 4: Delete line 1. Note: The whole content of line 1 is given here as context information.

Line 5-6: Insert new lines

Line 7-8: Context lines

Line 9-10 Remove lines 4-5

Line 11: Insert line

Line 12-13 Context lines

Line 14-15 Replace last line

Note: the context information not only makes patching more robust, it also enhances readability of the edit script considerably. However it also produces bigger patches.

2.2.3. Invertibility

A patch ($P : A_1 \rightarrow A_2$) with context information produced by GNU diff containing the instructions to convert a given file A_1 into a later version A_2 can be reversed ($P^{-1} : A_2 \rightarrow A_1$) when applied with the *patch* utility using its *-R* option. However

this is not possible with simple *edit scripts* because there is not enough information to turn a delete- into an insert-operation.

Invertibility of patches is especially important in the domain of version control.

2.2.4. Commutativity

When applying two patches ($P_1 : A_1 \rightarrow A_2$, $P_2 : A_2 \rightarrow A_3$) to the file $A1$ resulting in the file revision $A3$, $P2$ may be applied before $P1$ as long as context and operations of the two patches do not overlap.

2.2.5. Recap

Summing up we recognize the following characteristics of GNU diff and its file formats for representing changes:

- In the case of the *edit script* format, the *anchor* corresponds directly to the line number.
- In the unified *patch* format context information is used as an auxiliary mechanism to verify and resolve locations. Therefore in this case context is also part of the *anchor*. This process is normally referred to as *pattern matching*.
- Patches with context produced by GNU diff are *invertible* and *commutative*.

3. The tree to tree correction problem

3.1. Towards diffing structured data

In the previous section we introduced the elements of edit scripts, notable anchors and operations and the notion of context and patches. Also we showed how edit scripts and patches work using the example of GNU diff. Because XML is a widespread and accepted file format for all sorts of structured documents we concentrate on algorithms and tools designed specifically for XML.

While it is possible to use GNU diff and patch on structured text documents like XML files, the resulting patch files usually are far from ideal because of one of several reasons:

- Changes in whitespace like different indention with unchanged content may result in big patches which only express modified formatting hiding the fact that the actual content did not change at all.
- In database-style XML documents (think of a phone book) with many entries in a row with the same structure, a diff-algorithm operating on a line-by-line basis may pick up changes across record borders (see example below).

Listing 7: Original XML Document

```
1  <phonebook>
2    <person>
3      <name>Jon Zorn</name>
4    </person>
5    <person>
6      <name>Freddie Mercury</name>
7    </person>
8    <person>
9      <name>Frank Zappa</name>
10   </person>
11 </phonebook>
```

Listing 8: Modified Version

```
1 <phonebook>
2   <person>
3     <name>Jon Zorn</name>
4   </person>
5   <person>
6     <name>Frank Zappa</name>
7   </person>
8 </phonebook>
```

Listing 9: Patch produced by GNU diff

```
1        <name>Jon Zorn</name>
2      </person>
3      <person>
4  -     <name>Freddie Mercury</name>
5  -   </person>
6  -   <person>
7        <name>Frank Zappa</name>
8      </person>
9    </phonebook>
```

Note that although the end result is correct when this patch is applied, the presented changes do not reflect the meaning of the modification, namely "delete Mercurys record". Instead the patch communicates that Zappas record should be merged into its predecessor while Freddies name should be removed.

In order to understand the requirements for edit scripts and patch formats for structured documents, we need to identify their building blocks now and also extend our definitions we gave above.

3.1.1. Tree model

An XML document can be regarded as a rooted, ordered, labeled tree.

Definition 4 XML Tree: A rooted, ordered, labeled tree T consists of a set of nodes N, a finite alphabet Σ and a labelling function $L : N \to \Sigma$, which assigns a label to each node. R uniquely identifies the single root node and P denotes a function returning exactly one node representing the parent of a given node. Node order is determined by the ranking function r which returns the position of a node within its siblings. The value function V may return user data for leave nodes. This definition closely follows the one given by Bille [2].

$T = (N, R, P : N \to N, L : N \to \Sigma, \Sigma, V : N \to value, r : N \to int)$

Note that especially in graph theory node-labels normally are considered unique. In contrast in the literature on tree comparison, *node labels are not unique* and therefore may not be used for object identification.

Also note that some applications like databases may ignore node order completely while for other applications like word processors the order is an important aspect of the document and must be taken into account when comparing two files. When not specified explicitly node order is important in the following discussion of algorithms.

3.1.2. Requirements for patch format

Let's revisit our definition of edit scripts and patches and extend them now in order to match the new requirements of hierarchical structured documents.

- Edit script: *Anchors* must be extended such that we can specify paths relative to a reference node. For practical reasons most of the time the reference will be the root node.
- Patch: *Context* must be expressible in terms of nearby nodes, namely ancestors, siblings, descendants.

3.1.3. Tree operations

When operating on sequences of characters or lines it is sufficient to define two operations in order to construct an edit script, namely insert and delete. In order to express edit scripts in tree structures with the properties we defined above, we introduce one more operation: *relabel*. Several algorithms extend this basic set with methods operating on whole subtrees. Figure 1 depicts some operations on trees.

Definition 5 Basic tree operation

insert $(anchor, node[, context])$
The *anchor* points at a consecutive sequence of siblings S with a common parent P. Remove node S from P and insert the *new node* N at the place where S was before. Append S as the list of child nodes to N. After the operation P is the new parent of N and N is the new parent of S.

delete $(anchor[, oldnode, context])$
Let P be the parent of the *anchor* A pointing to the node N being deleted. Remove N from P and insert all of N child nodes where N was before.

relabel $(anchor, newlabel[, oldlabel, context])$
Replace the label of the node the *anchor* is pointing at with a new label.

We already mentioned before (3) that context information is a property of our definition of a *patch*. For invertible patch formats additional efforts are required in order to allow conversion from *insert* to *delete* operations and vice versa. Especially the exact scope of the anchor is challenging.

Before giving examples of edit scripts and patch formats, we take a look at the problem of finding the differences between two trees in the next section.

3.2. The generic tree to tree correction problem

3.2.1. Tai (1979)

In 1979 Tai expressed the *tree-to-tree correction problem* [22] as a generalisation of the edit distance on sequences. His algorithm has no practical relevance anymore today but it still serves as the basis of adapted and improved algorithms.

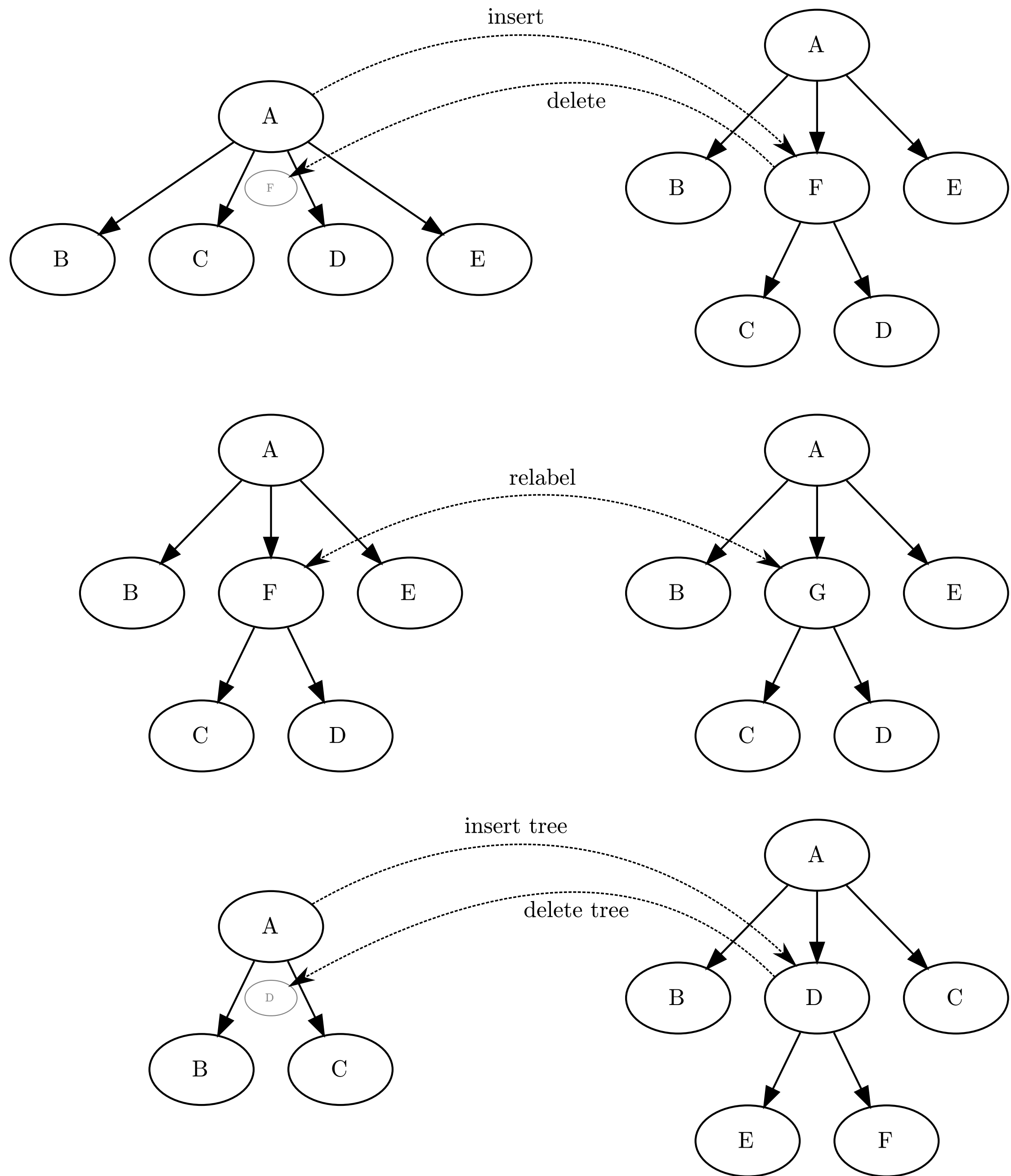


Figure 1: Tree edit operations. “Insert”, “delete” and “relabel” operate on node level while “insert tree” and “delete tree” extend to whole subtrees

Maximum complexity of Tais algorithm in time as well as in space is $O(|T_1||T_2|D_1^2D_2^2)$ where $|T_n|$ denotes the number of nodes and D_n the maximal depth of a tree. The worst case where no two nodes from T_1 and T_2 have the same label and both trees have maximal depth therefore results in an upper bound of $O(|T_1|^2|T_2|^2)$ in time and space. If both trees have a similar number of nodes n then we finally get worst case upper bound in time and space complexity of $O(n^4)$.

3.2.2. Zhang and Shasha (1989)

Zhang and Sasha published a new algorithm in 1989 improving on runtime and space requirements [24]. Key difference to Tais algorithm is the postorder traversal of the trees. In Tais algorithm the comparison of two trees starts on the root node then each of its child nodes is visited recursively until the right most leave node of both trees is reached. In contrast Zhang and Shasha algorithm starts with the leftmost leave of both trees working its way through its siblings then going through parents siblings until finally reaching the root node (see Figure 2).

Interestingly enough this algorithm actually operates on ordered forests[1] and therefore solves an even more generic problem than Tais algorithm.

In order to separate tree distance and forest distance calculations, Zhang and Sasha introduced so called keyroots, i.e. nodes having a sibling on their left and the root node. During edit distance calculations, an array of the forest distance between two keyroots is maintained(see Figure 2).

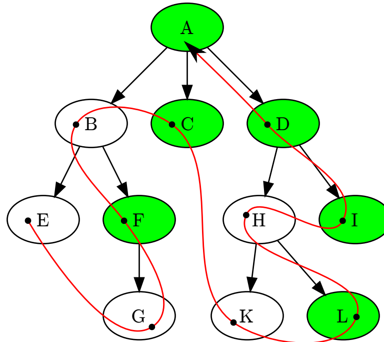


Figure 2: Zhang Sasha: postorder tree traversal (red line) and keyroots (green shaded nodes).

Upper bound in complexity of this algorithm is $O(|T_1||T_2|min(L_1, D_1)min(L_2, D_2))$ in time and $(|T_1||T_2|)$ in space. Again $|T_n|$ denotes number of nodes D_n tree depth and L_n number of leaves. Worst case input for this algorithm is a completely one-sided tree with n nodes where $\frac{n}{2}$ are leaves and with a depth of $\frac{n}{2}+1$ as shown in Figure 3. Considering two trees having the same size n, worst case time complexity becomes $O(n^4)$ while worst case space complexity remains $O(n^2)$.

[1]Ordered forest: sequence of ordered trees

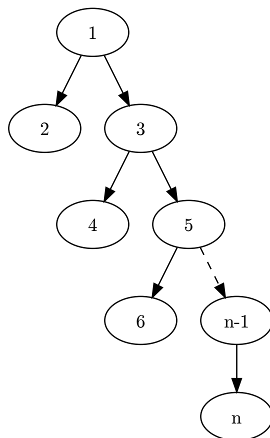


Figure 3: Worst case input tree for Zhang Sasha algorithm

3.2.3. Klein (1998)

A refinement on Zhang and Sashas algorithm was proposed by Klein [13] in 1998. The decomposition of trees into “heavy paths” yields another improvement on maximum time complexity. However Zhang and Shashas algorithm still outperforms Kleins on many input sets. On the other hand Kleins algorithm also is suited for edit distance computation between unrooted trees, i.e. trees without a designated root node.

Kleins algorithm runs with $O(|T_1|^2|T_2|log|T_2|)$ in time and with $(|T_1||T_2|)$ in space. For two similar sized trees with n nodes, the complexity therefore becomes $O(n^3\ log\ n)$ in time and $O(n^2)$ in space.

3.2.4. DMRW (2009)

In 2003 Dulucq and Touzet showed that Zhang Shasha as well as Klein algorithm can be described within a more general framework [11]. They introduced the notion of *decomposition strategy* as the discriminator between those algorithms. Based on that work Demaine, Mozes, Rossman and Weimann improved the Klein-algorithm in 2009 and achieved the new worst case time complexity of $O(n^3)$ [10].

3.2.5. Recap

In this section we pictured the milestones in the development of tree edit distance algorithms aimed at solving the generic tree to tree correction problem by finding the minimal conforming tree edit distance. In scientific domains, e.g. when working out the differences between RNA secondary structures, cubic worst case runtime may be acceptable. When comparing structured documents however it might be interesting to trade minimality of the edit script for better runtime performance.

Though the complexity bounds are important factors when comparing algorithms many algorithms perform better in most situations. For example the Zhang Shasha

algorithm is superior to Klein for certain inputs despite a worse upper bound in runtime for worst case input. Several sources mention that Zhang Sasha runs in $O(n^2\ log^2\ n)$ on balanced trees.

3.3. Unit cost algorithms

3.3.1. mmdiff and xmdiff (1999)

Recalling the duality of *shortest edit script* and *longest common subsequence* (See 2.1.3) it is possible to design more efficient edit distance algorithms by constraining the cost-function of operations to a constant (unit) value. Chawathe [3] presented two algorithms using the same technique for computing the tree edit distance like the one introduced by Myers [18] for sequences.

While mmdiff is designed to run in main memory, xmdiff is able to handle arbitrary big documents. For mmdiff upper limits in time and space is $O(|T_1||T_2|)$, boiling down to $O(n^2)$ for similar sized trees. While having constant upper limits in memory usage, the xmdiff algorithm introduces quadratic IO costs.

3.4. Diff algorithms for structured hierarchical data

3.4.1. Extended Zhang Sasha (1995)

The changes introduced by people when editing documents are normally more complex than the three basic operations *insert, delete, relabel* defined previously (see Definition 5). Therefore an edit script comprising only of basic operations may mask the actual meaning of the changes found between two document versions. In 1995 Barnard, Clarke and Duncan proposed an extended version of Zhang Sasha algorithm [1] in order enhance the expressivity of an edit script in the domain of document comparison.

Extended Zhang Shasha introduces three additional operations performed on whole subtrees: *insertTree*, *deleteTree*, *swap*. Swapping is only allowed on adjacent siblings.

While this algorithm improves on edit script semantics and generally produces smaller deltas with less operations compared to the original Zhang Sasha algorithm, it does not help narrowing complexity bounds and memory requirements. However like the original this algorithm produces minimal conforming edit scripts.

3.4.2. FastMatch EditScript - FMES (1996)

FMES was presented by Chawathe, Rajaraman, Garcia-Molina and Widom in 1996 [5] as a complementary algorithm to Zhang Sasha tailored for generating edit scripts in structured documents. The set of operations is however rather different to the one of Zhang Shasha. In FMES *insert* and *delete* operations are restricted to leaf nodes, *relabel* is substituted by an *update* operation targeting node values instead of node labels. Additionally a *move* operation is introduced capable of changing the parent of a given subtree and also its position within its siblings.

A key property of this algorithm is the separation of change detection into two sub-problems:

1. Find a good matching between two trees
2. Compute the edit script

If object identifiers are present in the data the solution to the first problem is trivial and leads to a speedup of the whole process. The algorithm is capable of assigning object identifiers if necessary based on node labels and values. For interior nodes the matching criterion is based on their child nodes.

The matching algorithm is based on a set of criteria and assumptions appropriate for structured data in the domain of document processing resulting in faster runtime at the expense of potentially non-minimal edit scripts:

Criterion 1: Leaf nodes can be matched only if their labels are equal and their values are similar enough.

Criterion 2: Internal nodes can be matched only if a certain percentage of their leaves match.

Assumption 1: Labels follow a structuring schema where certain labels only are allowed as child nodes of others.

Assumption 2: Every node in one tree only has at most one node in the other tree resembling it closely.

Upper bound in time complexity for this algorithm is $O((L(T_1) + L(T_2))e + e^2)$ where $L(T_n)$ represents the number of leave nodes of a given tree and e is the weighted edit distance (typically, $e \ll n$) [5] representing the sum of the weights of all operations where an *insert* and *delete* each count 1, an *update* counts 0 and the weight of a *move* is equal to the number of leave nodes which are descendants of the node in question. Given two similar sized trees of the size n which do not have any nodes in common, worst case time complexity becomes therefore $O(n^2)$.

LaDiff, the authors implementation of FMES, took two versions of a LaTeX document and generated a third one with annotations on additions and deletions as well as indications where parts of text were moved to another location. An example is given in [4].

3.4.3. BULD (2001)

Unlike FMES, the BULD algorithm by Cobéna, Abiteboul and Marian is designed exclusively for XML documents [8]. Operations are similar to FMES however *insert* and *delete* target subtrees and not leaf nodes.

In some XML structures it is common that certain tags occur more frequent and in consecutive sequences — for example think of paragraphs (P-Tag) in an XHTML document. Therefore XML tag names are not the best choice as a mapping criterion. Instead a hash on node values and subtrees is calculated which is used in order to find corresponding nodes and subtrees in the other document. When a DTD is available, BULD will also consider ID attributes.

The BULD algorithm runs in $O(n\ log\ n)$ time and $O(n)$ space where n is the number of nodes of both documents.

3.4.4. faxma (2006)

A rather unique approach on finding differences in XML documents was presented by Lindholm, Kangasharju and Tarkoma in 2006 [14]. Instead of matching the tree structure of documents, the sequence of tokens emitted by the XML parser is compared using a "rolling hash". A similar method is used in the `rsync` tool. After working out the common parts in the token streams, the results are mapped back to the XML tree structure (the *metadiff*).

Runtime is expected to be linear for two document versions with small and local changes. However for two completely different documents of the same size n, the algorithm runs in $O(n^2)$.

3.4.5. XCC (2010)

Recently Rönnau and Berghoff released a framework consisting of libraries and tools for diffing, patching and merging XML office documents [20]. The diff algorithm shares some aspects of BULD. The operations *insert* and *delete* are extended to address tree-sequences. The *move* operation is realized by simply referencing equivalent *insert* and *delete* operations. Like in BULD hash-values are calculated for all nodes in a bottom up manner.

The algorithm starts by calculating the longest common subsequence (LCS) over the leaf nodes of both documents. Interestingly an implementation based on Myers [18] is used in this step. After that the ancestor chain of each node in the LCS is examined for structure preserving updates, i.e. changes in attributes of internal nodes. Visited nodes are marked in order to prevent that a certain path is traced more than once. After that for each unmatched leaf in the first and the second tree where the parent is matched, the neighborhood is examined in order to detect updates on leaf nodes. Starting on the remaining leaf nodes inserted and deleted trees are detected and added to the delta now. Finally corresponding insert and delete operations are identified and referenced.

Worst case complexity is $O((L(T_1)+L(T_2))D+I(T_1)+I(T_2)+D)$ in time and $O(|T_1|+|T_2|)$ in space where $|T_1|+|T_2|$ is the sum of the number of nodes in both documents, $L(T_n)$ the number of leaves, $I(T_n)$ the number of internal nodes and D the minimal edit distance. Note that the first expression is due to the use of Myers $O(ND)$ difference algorithm for finding the longest common sequence among the leaves of both trees. Considering two completely flat trees with only one internal node each (the root node), where the value of no two leaves are equal, worst case time complexity becomes $O(n^2)$ for two trees of the size n because of parameter $D > n$. Complexity in space remains linear though.

The authors claim that the final move-detection step does not influence complexity because a hash-map with linear lookup time is used to match equivalent insert and delete operations.

3.4.6. A note on the move operation

Recall that no single algorithm for the generic tree to tree correction problem presented in the former section employed a *move* operation. In order to understand the problem let's imagine an edit script representing the changes between two ordered trees where all move operations have been deleted. The result is exactly an edit script representing the changes between two *unordered* trees. However this problem has been shown to be NP-complete for the general case [2].

So what made it possible to devise algorithms with less than quadratic complexity and support for the *move* operation at the same time? By matching corresponding *insert* and *delete* operations during a post processing phase, minimality of the resulting edit script is not guaranteed anymore which is unacceptable for the generic case but reasonable in the domain of document comparison [7].

3.4.7. Recap

We can identify several key ideas in the work done by a number of research groups in order to adapt the generic tree edit distance problem to the domain of document comparison:

1. Constrain the cost model of operations (See 3.4.1). Instead of allowing to assign a cost to each and every single operation within an edit script, the cost model is constrained to constant units in order to reduce runtime complexity.
2. Heuristically reduce possible candidates in the matching phase.
3. Accept non-minimal edit scripts in order to further reduce complexity.
4. Adapt scope of operations to enhance readability (i.e. use one insertTree instead of many insert node operations). Also introduce new operations like *move* to reduce size of edit script and to better reflect the meaning of changes. Note that this modification does not have any effect on runtime complexity.

4. Representing changes in tree structures

4.1. Visualization of changes

In the case when one is more interested into a visualization of changes between two versions of a document, the information contained in a simple edit script or a patch might not be expressive enough. GNU diff provides the option to generate a *full context patch*. Modified regions, so called hunks, are interleaved in the appropriate locations and denoted in the usual manner (see Listing 6 on page 4 for an example of the syntax). More elaborate representations adapted to a given domain are possible. Often a side by side view of two document versions with changes highlighted helps us to better understand

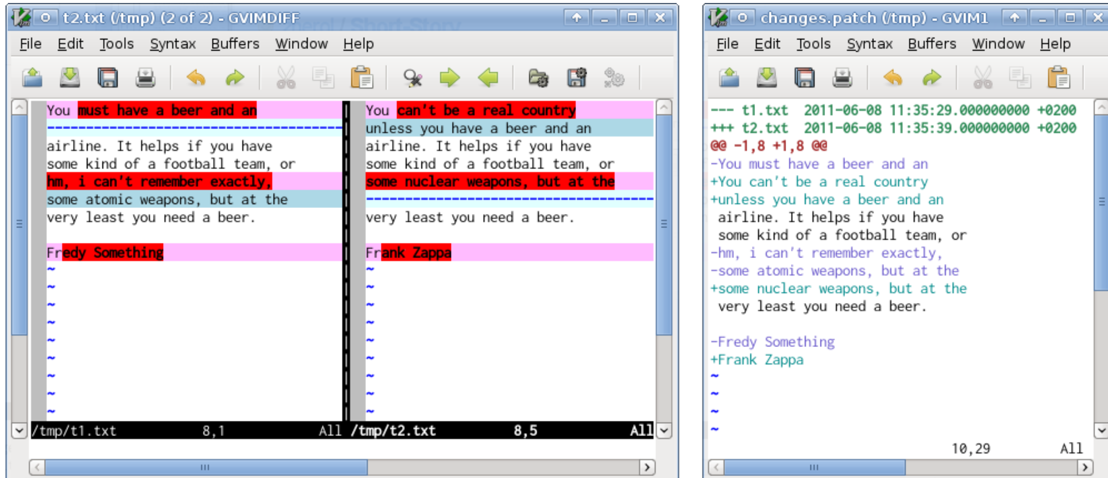


Figure 4: Left screenshot: Visualization of the changes by displaying two versions of a document side by side. Modified lines are highlighted with red color, inserted lines with blue color.
Right screenshot: The same changes in a raw patch format displayed in an editor with syntax highlighting.

which regions were modified (see Figure 4 for an example). Alternatively some programs are able to visualize changes between two versions of a document by annotating modifications, insertions and deletions directly in one of the document versions.

In the domain of visualizing changes in hierarchical structured information we find several different approaches.

4.1.1. Delta tree

Recalling the LaDiff software mentioned during the discussion of the FMES algorithm (3.4.2). This program takes two versions of a LaTeX-document and outputs a third one with annotations where paragraphs were inserted, deleted and moved around. Clearly this output format does not qualify as a *patch* as we defined it before, because it does not express the changes in a machine readable way. However a human reader easily can understand the output. Chawathe introduced the notion of a *delta tree* [4] where edit script operations are injected at the appropriate locations into the source document tree.

4.1.2. DeltaXML v2 and XMLR

The DeltaXML v2 format [9] used by the proprietary DeltaXML tools as well as the XMLR ("XML-with-references") format developed by the faxma team [14] both implement a variant of *delta tree* tailored to XML. Unchanged subtrees can be marked with a special attribute in order to shorten the delta file.

Listing 10: DeltaXML: Example of DeltaV2 full context patch [9]

```
1  <root xmlns:dxx="http://www.deltaxml.com/ns/xml-namespaced-attribute"
       xmlns:dxa="http://www.deltaxml.com/ns/non-namespaced-attribute"
       xmlns:deltaxml="http://www.deltaxml.com/ns/well-formed-delta-v1"
       deltaxml:content-type="changes-only" deltaxml:deltaV2="A!=B"
       deltaxml:version="2.0">
2    <a deltaxml:deltaV2="A=B"> </a>
3    <b deltaxml:deltaV2="A!=B">
4      <b1 deltaxml:deltaV2="B"/>
5    </b>
6    <c deltaxml:deltaV2="A!=B">
7      <deltaxml:attributes deltaxml:deltaV2="B">
8        <dxa:attr deltaxml:deltaV2="B">
9          <deltaxml:attributeValue deltaxml:deltaV2="B">hello world</
               deltaxml:attributeValue>
10       </dxa:attr>
11     </deltaxml:attributes>
12     <deltaxml:textGroup deltaxml:deltaV2="A">
13       <deltaxml:text deltaxml:deltaV2="A">hello world</deltaxml:text>
14     </deltaxml:textGroup>
15   </c>
16   <d deltaxml:deltaV2="A"/>
17   <f deltaxml:deltaV2="A"/>
18   <e deltaxml:deltaV2="A!=B">Word by Word
19     <deltaxml:textGroup deltaxml:deltaV2="A!=B">
20       <deltaxml:text deltaxml:deltaV2="A">changes</deltaxml:text>
21       <deltaxml:text deltaxml:deltaV2="B">modifications</deltaxml:text>
22     </deltaxml:textGroup>
23   </e>
24 </root>
```

Line 2: Subtree below the `<a>`-Element is identical in both documents.

Line 3-5: Subtree below the element `<b>` has changes: `<b1>` is only present in second document.

Line 6-15: In the second document the text node “hello world” is removed instead an attribute `attr=‘‘hello world’’` is added to the Element `<c>`.

Line 16-17: Both `<d/>` and `<f/>` are only present in the first document and are removed in the second.

Line 18: The `<e>`-Element contains a text where a word was substituted.

DeltaXML provides a set of XSLT filters transforming full context patches into HTML documents marking up the changes visually.

Note that the enhanced readability and the possibility to easily post-process and transform delta tree based formats stems from the fact that the entry point of operations can be derived from its location in the document tree. This property sets delta tree based formats apart from edit script and patch formats discussed in the following section. Also note that changes within a text node can be easily denoted using the `<deltaxml:text>` element. Expressing changes within a text node is more difficult in edit script / patch formats.

4.2. Edit script and patch formats

Recalling our definition of edit scripts and patches (1, 2 on page 2) and also our first attempt to formulate the requirements needed in order to expand them to the domain of rooted, ordered, labeled trees(3.1.2 on page 7). In the XML domain the obvious way to address arbitrary nodes in an XML tree is by XPath expressions [6]. Therefore in most if not all patch formats XPath or a subset thereof is used as anchor for operations. Expressing context is however not that easy like it is in the case of flat files. In the following section we discuss two different XML based file formats for representing changes between XML trees.

4.2.1. RFC 5261 - XML Patch Operations Framework

RFC 5261 describes an “XML Patch Operations Framework Utilizing XPath Selectors” [23] which was developed by Urpalainen for the IETF SIMPLE working group[2]. A reference implementation of a command line tool capable of applying xml-patch-ops files is available from the xmlpatch project website[3]. Note that RFC 5261 does not define a complete XML patch file format but rather a set of well defined operations which may be combined with additional XML elements in order to create specialized implementations. A few examples of existing patch formats are given in RFC 5261. Listing 11 depicts an example in the PIDF diff format defined in RFC 5262 [15].

The xml-patch-ops framework consists of three operations: *add*, *replace*, *remove*. All of them take an XPath expression as an anchor in their `sel`-Attribute. The operations may target any type of XML node, namely elements, attributes, namespace prefix declarations, comments, processing instructions and text nodes. Some restrictions apply, especially the root node and comments as well as processing instructions of the root node cannot be patched. Also entities are out of the scope of the xml-patch-ops framework.

In order to support insertion of attributes and namespace prefix declarations, the `<add>` element provides a special `type`-attribute. Using the `pos`-attribute one can specify whether a new element should be inserted as a sibling `before` or `after` the element specified using `sel`. If either `prepend` or `append` is specified in `pos`, the new element is inserted as the first and the last child element respectively of the target element.

There is also an XML specific extension for the `<remove>` element, namely the optional `ws` attribute. Its value specifies whether a whitespace-only text node `before` or `after` the target element also should be deleted during the removal. Note that in XML no two text-nodes can be siblings. Therefore the patching algorithm must account for that case and merge text nodes which became siblings due to a remove-operation in between them.

Listing 11: Example of RFC 5261 based delta format: PIDF diff [15]

```
1  <?xml version="1.0" encoding="UTF-8"?>
```

[2]SIP for Instant Messaging and Presence Leveraging Extensions (simple): http://datatracker.ietf.org/wg/simple/

[3]An XML Patch library: http://xmlpatch.sourceforge.net/

```
2  <p:pidf-diff
3    xmlns="urn:ietf:params:xml:ns:pidf"
4    xmlns:p="urn:ietf:params:xml:ns:pidf-diff"
5    xmlns:r="urn:ietf:params:xml:ns:pidf:rpid"
6    xmlns:d="urn:ietf:params:xml:ns:pidf:data-model"
7    entity="pres:someone@example.com"
8    version="568">
9
10   <p:add sel="presence/note" pos="before">
11     <tuple id="ert4773">
12       <status>
13         <basic>open</basic>
14       </status>
15       <contact priority="0.4">mailto:pep@example.com</contact>
16       <note xml:lang="en">This is a new tuple inserted
17         between the last tuple and person element</note>
18     </tuple>
19   </p:add>
20
21   <p:replace sel="*/tuple[@id='r1230d']/status/basic/text()">open</p:
       replace>
22
23   <p:remove sel="*/d:person/r:activities/r:busy" ws="after"/>
24
25   <p:replace sel="*/tuple[@id='cg231jcr']/contact/@priority">0.7</p:
       replace>
26
27 </p:pidf-diff>
```

Line 1-9: XML header and root element. Especially line 7 and 8 are noteworthy: The `entity` attribute holds the primary key of the object to patch while the `version` attribute specifies the revision of the object this patch is applicable to.

Line 10-19: Insert a new XML subtree before the first `<note>` element in the `<presence>` element.

Line 21: Replace the text node in the basic status of the tuple whose id is "`r1230d`".

Line 23: Remove the `<busy>` element including following whitespace only text node.

Line 25: Replace the value of the `priority` attribute of the `<contact>` element of the tuple whose id is "`cg231jcr`".

RFC 5261 describes some very important aspects of XML regarding the design of patch formats, especially the problem of whitespace-only text nodes as well as how to deal with differing namespace prefixes in target document and patch file.

The use of XPath for target node selection however poses some problems. First, XPath expressions always evaluate to a node set and never to a single node. Second, while it would be possible to express some sort of context using XPath axis (ancestor, descendant, following, preceding), the expression would quickly get complicated if not unreadable.

Because the add and remove operations are not invertible and also because all instructions of a patch must be executed in the given order, xml-patch-ops does not qualify as a *patch* but rather as an *edit script* format. Also the lack of a support for specifying the context of an operation points into that direction.

A very similar approach was proposed by Moaut as an IETF internet draft [17], which however expired without having been discussed thoroughly. His DUL (Delta Update Language) additionally allows operations in substrings of text nodes, which allows finer grained change representation in documents containing big chunks of continuous text. Additionally a *move* operation is supported. The authors implementation of a XML diff tool is based on Chawathes xmdiff and FMES algorithms (see 3.4.2, 3.3.1). An XML patch tool is also part of the open source software package[4].

Relying on XPath expressions for node identification, Mouats DUL shares the same characteristics and constraints as the xml-patch-ops framework. The main difference between the two formats is that the DUL targets document oriented XML files while xml-patch-ops is better suited for data oriented XML.

4.2.2. XCC patch format

The delta model presented in [21] suits our definition of a *patch* including context based identification and verification of anchors when applying operations, invertibility of patches as well as commutativity of patch operations. In order to keep space requirements for context information low, fingerprints are calculated and stored for the nodes within a given radius in the neighborhood of the patch operations.

In order to support patching of documents which slightly differ from the original source, candidate anchors are identified based on a weighted match quality function. Weights of node-fingerprints in the neighborhood decrease with increasing distance to the anchor. This technique is similar to the simple fuzzy matching mechanism in GNU diffutils, where the context radius is reduced until a match can be found in the target document.

Listing 12: Rönnau and Berghoff: Delta format example [19]

```
1  <?xml version="1.0" encoding="utf-8"?> <delta>
2    <update digester="fnv" path="/3/0/1" radius="4" shortened="true">
3      <fingerprint>
4        41AADF68;9A960EB2;EAA88AD9;09E5C47C;A5B43DB0;4EAACE7B;FC96EE47;
           FC96EE47;FC96EE47
5      </fingerprint>
6      <oldvalue>
7        <text:p text:style-name="Standard" />
8      </oldvalue>
9      <newvalue>
10       <text:p text:style-name="P1" />
11     </newvalue>
12   </update>
```

[4]diffxml & patchxml: Tools for comparing and patching XML files: http://diffxml.sourceforge.net/

```
13     <update digester="fnv" path="/3/0/1/0" radius="4" shortened="true">
14       <fingerprint>
15         9A960EB2;EAA88AD9;09E5C47C;A5B43DB0;4EAACE7B;FC96EE47;FC96EE47;
             FC96EE47;FC96EE47
16       </fingerprint>
17       <oldvalue>Hello, world!</oldvalue>
18       <newvalue>Hello, World!</newvalue>
19     </update>
20     <insert digester="fnv" path="/2/0" radius="4" shortened="true">
21       <fingerprint>
22         8F792681;5DBFCE63;B63D542B;F7128BEA;4B12A6AF;B4563E0E;8B673BCB;
             A5FB1C64;41AADF68
23       </fingerprint>
24       <oldvalue />
25       <newvalue>
26         <style:style style:family="paragraph" style:name="P1">
27           <style:text-properties fo:font-weight="bold"/>
28         </style:style>
29       </newvalue>
30     </insert>
31   </delta>
```

Line 2: Start of an update-Operation. The digester-Attribute specifies that the FNV-hash method should be used to calculate the fingerprints. The path-Attribute indicates the path to the XML element which should be updated second child of first child of fourth child of the root node).

Line 3-5: Fingerprints from neighbor nodes within the radius of four (indicated by the corresponding Attribute in Line 2). Note that the path-Attribute and the fingerprint together form the *anchor* of the operation.

Line 6-11: Old and new value of the element. Note that in this case only the text:style Attribute is modified and that the element name as well as its children is not touched. Also note that this syntax suggests that the text:p element is empty (does not have any child nodes) which is not true, as we see in the following operation.

Line 13-19: Update the text node of the element whose attribute was altered in the last operation.

Line 20-30: Insert a new XML Element tree as the first child of the 2nd child of the root node.

In terms of expressivity as well as functionality the XCC patch format comes very close to the unified patch format of GNU diff. In fact it is the only somewhat universal format for a wide range of XML files today providing invertibility and commutativity as well as the possibility to merge changes into a document which differs from the original document a patch was generated from in the first place. However there might be still room for improvements. We'll discuss that in the next section.

5. Conclusion and Future Work

5.1. Results

5.1.1. Research on diff algorithms

Computation of tree edit distance and related problems were explored extensively by a number of different researchers. However the generic tree diff algorithm first developed by Tai and subsequently enhanced by Zhang and Shasha, Klein and finally Demaine et al. still is not suitable for the domain of document comparison because best worst case time and space complexity remained cubic and quadratic respectively.

A number of domain specific optimizations for hierarchically structured data were first proposed by Chawathe et al. Later Cobéna et al., Lindholm et al. and recently Rönnau and Berghoff applied similar techniques to devise linear time and space algorithms for the difference analysis of XML documents. Better speed and less memory requirements come at the expense of the loss of minimality of the generated edit script which, however, is negligible in the domain of document comparison.

The BULD and XCC algorithms are suited best for DOM based comparison while faxma looks promising regarding stream based implementations.

5.1.2. Research on patch formats and merging strategies

Finding the differences between two document versions is only part of the story. After building up an appropriate edit script or delta tree the result must be presented to the user or serialized into a patch file. Other than in the case of flat files where the unified diff format can be regarded as a quasi standard, there does not seem to be any broad agreement on patch file format in the domain of XML documents.

Having reached the state of a proposed standard, the xml-diff-ops framework described in RFC 5261 may gain more acceptance in the future. Though it cannot stand on its own because it only defines the operations along with their parameters and some implementation details. The framework is intended as a base for application specific formats.

Existing patch formats can be roughly divided into two groups. In delta tree oriented flavours, annotations or edit script operations are embedded into the source document at the appropriate locations. Such formats typically are easy to transform in a way suitable for presentation with highlighted changes. This patch format can be regarded as full context patch, however normally there is a way to reduce patch sizes by leaving out unchanged subtrees. The other group consists of patch formats which are derived more or less directly from the edit script produced by the diff algorithm. Main problem here is how to specify the target node of operations in a robust way. Mostly XPath expressions or a subset thereof is used but without additional measures a patch created in such a way cannot safely be applied in the case when it is not clear if the target document corresponds exactly to the source document the patch was computed for.

Merging methods are closely coupled to the capabilities of a patch format. Only the XCC patch tool is currently capable of safely merging changes to documents which were

altered between diffing and patching. There is even a GUI tool which allows interactive merging of XML documents.

5.2. Future Work

5.2.1. Dynamic diff granularity

When working with text documents it might be interesting if the diff algorithm would be capable of picking up finer grained changes than insertion, deletion and modification of whole paragraphs. Generally comparing every text node on a word by word basis however does not make sense either, especially when comparing database style XML documents containing record-like element structures. It might be interesting to extend one of the diff algorithms with the ability to adapt diff granularity according to domain specific rules. Other things like whitespace normalization could be controlled with the same mechanics probably resulting in more expressive results than it would be possible without domain knowledge. It might be possible that some or all of the necessary information is expressible in one of XML Schema languages or the DTD.

5.2.2. Merge capable delta tree style patch format

We have given some reason why delta tree based formats are easier to postprocess in order to display the changes to a user in a visual and informative manner. We now might try to devise a patch format which combines the benefits of the delta tree model with the robustness of the XCC patch format. The DeltaV2 format might serve as a starting point.

Alternatively we might devise some methods capable of generating a delta tree style annotated document from a patch and the corresponding source document. While the patch remains in a universal format (e.g. XCC), the annotation-engine might apply document-type specific rules in order to highlight changes in a meaningful way. A visualization of differences found in two versions of an XHTML document might be realized with the application of custom CSS styles to the changed elements, while in an SVG image some colored overlays might be better suited to indicate modifications.

5.2.3. Diffing graphs along a spanning tree

It might be interesting to expand one of the analyzed algorithms to the domain of graphs in general, and the RDF (Resource Description Framework) in particular. Instead of trying to diff two graphs directly possibly having to worry about cycles, we could try to first compute a spanning tree for each graph and then apply the algorithm on the spanning trees. Analogous to the mapping constraints applied in Chawathes FMES algorithm, constraining the arcs which may participate in a spanning tree might speedup its computation.

A document by Tim Berners-Lee and Dan Conolly already points out some problems

and possible solutions as well as a patch format[5]. Swish[6], a set of RDF tools written in Haskell by Graham Klyne, provides means to show differences between RDF graphs.

[5]Delta: an ontology for the distribution of differences between RDF graphs
http://www.w3.org/DesignIssues/Diff

[6]Swish Semantic Web Inference Scripting in Haskell
http://www.ninebynine.org/RDFNotes/Swish/Intro.html

A. Reflection

We successfully identified some important algorithms in the chosen domain. Also we characterized each and depicted the relationships and differences among them. We also showed that there is no universal solution regarding patch formats for XML documents at the moment. We identified two flavours of patch formats and analyzed their benefits and drawbacks and also gave some ideas on how to develop them further.

Despite the effort to explain the most important points as precise as possible, some concepts are not defined well enough in this document. Especially the individual operation sets which differ quite substantially between the analyzed algorithms, deserve some more attention. Overall this document might profit from some more figures and examples which might help clear up some details.

The amount of research about previous work required to gain enough knowledge on the subject was unexpected high. In order to understand the performance characteristics of modern XML diff implementations it was especially important to identify the differences between them and the generic exact diff algorithms. This somewhat extensive research was necessary in order to build up the foundation required to base or future work on.

B. Project Management

B.1. Methodology

While not sticking strictly to an agile methodology/framework like Scrum or XP the core principles of agile project management are taken into account while developing this project. Risk and exploration factor are high and therefore change of coverage and even objectives should not be precluded until late in the project.

Meetings with the supervisor are held frequently on a mostly weekly or basis where current and future work is discussed.

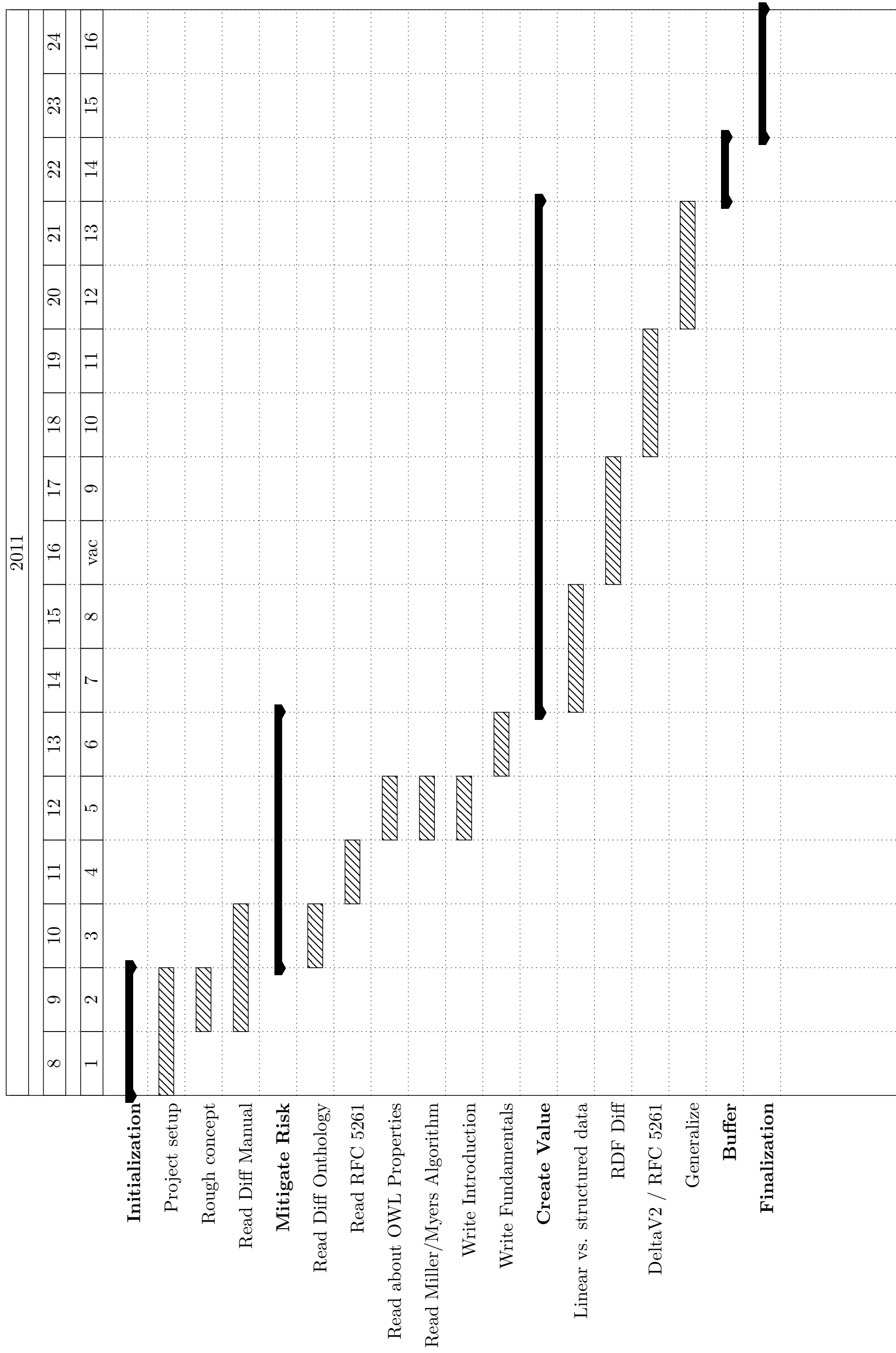
B.2. Project Livecycle

Agile methodologies are frequently divided into four to five phases ("Envision", "Speculate", "Explore", "Adapt", "Close") [12]. Applied to the livecycle of this project thesis the first phase comprises the project proposal (before the start of the semester), the second the first couple of weeks where identifying and resolving the difficult problems is important, the third phase mostly consists of writing activity while the last one is used to close up the project.

B.3. Tools

No specialized tools are used in order to schedule and prioritize tasks. Reading notes, progress reports, meeting minutes and also tasks which either are marked with [open] and [done] are recorded in a simple textfile (journal.txt). Using a text editor which is able to highlight lines matching a given pattern (like vim), it is very easy to keep the overview of open tasks.

B.4. Chart



C. List of Figures

1. Tree edit operations. “Insert”, “delete” and “relabel” operate on node level while “insert tree” and “delete tree” extend to whole subtrees 8
2. Zhang Sasha: postorder tree traversal (red line) and keyroots (green shaded nodes). 9
3. Worst case input tree for Zhang Sasha algorithm 10
4. Left screenshot: Visualization of the changes by displaying two versions of a document side by side. Modified lines are highlighted with red color, inserted lines with blue color.
Right screenshot: The same changes in a raw patch format displayed in an editor with syntax highlighting. 15

D. Listings

1. Original Document 3
2. Modified Document 3
3. Resulting Edit Script 3
4. Slightly modified original 3
5. Corrupt document 3
6. Resulting Edit Script 4
7. Original XML Document 6
8. Modified Version 6
9. Patch produced by GNU diff 6
10. DeltaXML: Example of DeltaV2 full context patch [9] 15
11. Example of RFC 5261 based delta format: PIDF diff [15] 17
12. Rönnau and Berghoff: Delta format example [19] 19

E. References

[1] David T. Barnard, Gwen Clarke, and Nicholas Duncan. Tree-to-tree correction for document trees technical report 95-372. Technical report, January 1995. Available from: http://ftp.qucis.queensu.ca/TechReports/Reports/1995-372.pdf. 11

[2] Philip Bille. A survey on tree edit distance and related problems. *Theoretical computer science*, 337(1-3):217–239, 2005. doi:10.1016/j.tcs.2004.12.030. 6, 14

[3] Sudarshan S. Chawathe. Comparing hierarchical data in external memory. In *Proceedings of 25th International Conference on Very Large Data Bases*, page 90–101, September 1999. Available from: http://www.cs.umaine.edu/~chaw/pubs/xdiff.pdf. 11

[4] Sudarshan S. Chawathe. *Managing change in heterogeneous autonomous databases*. PhD thesis, Stanford University, 1999. Available from: http://www.cs.umaine.edu/~chaw/pubs/cm.pdf. 12, 15

[5] Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. Change detection in hierarchically structured information. In *Proceedings of the 1996 ACM SIGMOD international conference on Management of data - SIGMOD '96*, pages 493–504, Montreal, Quebec, Canada, 1996. doi:10.1145/233269.233366. 11, 12

[6] James Clark and Steve DeRose. XML path language (XPath), November 1999. Available from: http://www.w3.org/TR/1999/REC-xpath-19991116/. 17

[7] Grégory Cobéna, Talel Abdessalem, and Yassine Hinnach. A comparative study for XML change detection. *Research Report, INRIA Rocquencourt, France*, 2002. Available from: ftp://tfalati.inria.fr/INRIA/Projects/gemo/gemo/GemoReport-221.pdf. 14

[8] Grégory Cobéna, Serge Abiteboul, and Amélie Marian. Detecting changes in XML documents. In *Proceedings of the International Conference on Data Engineering - ICDE 2002*, pages 41–52, February 2002. doi:10.1109/ICDE.2002.994696. 12

[9] DeltaXML. Two and three document DeltaV2 format, March 2011. Available from: http://deltaxml.com/library/deltav2-format-specification.html. 1, 15, 16, 26

[10] Erik D. Demaine, Shay Mozes, Benjamin Rossman, and Oren Weimann. An optimal decomposition algorithm for tree edit distance. *ACM Transactions on Algorithms*, 6(1):1–19, December 2009. doi:10.1145/1644015.1644017. 10

[11] Serge Dulucq and Hélène Touzet. Analysis of tree edit distance algorithms. In *Combinatorial Pattern Matching*, volume 2676, page 83–95, 2003. doi:10.1007/3-540-44888-8_7. 10

[12] James Highsmith. *Agile project management : creating innovative products*. Addison-Wesley, Upper Saddle River NJ, 2nd ed. edition, 2010. 24

[13] Philip N. Klein. Computing the edit-distance between unrooted ordered trees. *Algorithms - ESA'98*, page 1–1, 1998. doi:10.1007/3-540-68530-8_8. 10

[14] Tancred Lindholm, Jaakko Kangasharju, and Sasu Tarkoma. Fast and simple XML tree differencing by sequence alignment. In *Proceedings of the 2006 ACM symposium on Document engineering*, page 75–84, 2006. doi:10.1145/1166160.1166183. 13, 15

[15] Mikka Lonnfors, Eva Leppanen, Hirsham Khartabil, and Jari Urpalainen. RFC 5262: Presence information data format (PIDF) extension for partial presence, September 2008. Available from: http://tools.ietf.org/html/rfc5262. 17, 26

[16] Webb Miller and Eugene W. Myers. A file comparison program. *Software: Practice and Experience*, 15(11):1025–1040, November 1985. doi:10.1002/spe.4380151102. 1

[17] Adrian Mouat. draft-mouat-xml-patch-00 - a delta format for XML documents, 2005. Available from: http://tools.ietf.org/html/draft-mouat-xml-patch-00. 1, 19

[18] Eugene W. Myers. An O(ND) difference algorithm and its variations. *Algorithmica*, 1(1-4):251–266, November 1986. doi:10.1007/BF01840446. 2, 11, 13

[19] Sebastian Rönnau and Uwe M. Borghoff. Versioning XML-based office documents. *Multimedia Tools and Applications*, 43(3):253–274, March 2009. doi:10.1007/s11042-009-0271-2. 19, 26

[20] Sebastian Rönnau and Uwe M. Borghoff. XCC: change control of XML documents. *Computer Science - Research and Development*, November 2010. doi:10.1007/s00450-010-0140-2. 13

[21] Sebastian Rönnau, Christian Pauli, and Uwe M. Borghoff. Merging changes in XML documents using reliable context fingerprints. In *Proceeding of the eighth ACM symposium on Document engineering*, page 52–61, 2008. doi:10.1145/1410140.1410151. 1, 19

[22] Kuo-Chung Tai. The Tree-to-Tree correction problem. *Journal of the ACM (JACM)*, 26:422–433, July 1979. ACM ID: 322143. doi:10.1145/322139.322143. 7

[23] Jari Urpalainen. RFC 5261: An extensible markup language (XML) patch operations framework utilizing XML path language (XPath) selectors, September 2008. Available from: http://tools.ietf.org/html/rfc5261. 1, 17

[24] Kaizhong Zhang and Dennis Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM Journal on Computing*, 18(6):1245–1262, December 1989. doi:10.1137/0218082. 9