Structure-preserving Difference Search for XML Documents

Erich Schubert[1], Sebastian Schaffert[2], and François Bry[1]

[1] Institut für Informatik, Ludwig-Maximilians-Universität München, Germany
[2] Salzburg Research Forschungsgesellschaft m.b.H., Salzburg, Austria

Extreme Markup Languages 2005
4th August 2005, Montréal, Canada

salzburg|research

REWERSE reasoning on the web

LMU

Contents

Motivation
- Why Difference Search?
- State-of-the-Art
- Structure-Based Differencing

Structure Retainment
- Document Graphs
- Node Similarity
- Nodeset Correspondences
- Retained Relations

Algorithm
- Search Tree
- Dijkstra's Algorithm
- Optimised Cost Function

Results
- Experimental Results
- Prototype
- Summary

Contents

Motivation
- Why Difference Search?
- State-of-the-Art
- Structure-Based Differencing

Structure Retainment
- Document Graphs
- Node Similarity
- Nodeset Correspondences
- Retained Relations

Algorithm
- Search Tree
- Dijkstra's Algorithm
- Optimised Cost Function

Results
- Experimental Results
- Prototype
- Summary

Motivation

Why Difference Search?

Difference Search in general:

- determine, store, and distribute the (usually smaller) *increment* between different versions of a document
- well-known applications: GNU diff, CVS, Subversion, Eclipse

Uses:

- efficient transmission of changes (“patch systems”)
- efficient storage of multiple versions (“revision control systems”)
- reviewing changes by humans
- merging concurrent changes into a single document

Motivation

Why Difference Search?

Difference Search in general:

- determine, store, and distribute the (usually smaller) *increment* between different versions of a document
- well-known applications: GNU diff, CVS, Subversion, Eclipse

Uses:

- efficient transmission of changes (“patch systems”)
- efficient storage of multiple versions (“revision control systems”)
- reviewing changes by humans
- merging concurrent changes into a single document

Motivation

Why Difference Search?

Observation:

- differences need to be processable by machines and humans
- but requirements for machines and humans are different (efficient storage vs. presentation)
- the most widely used output format of GNU diff is not the smallest, but the format easiest to read by humans

Empiric observation: human processing does not only require a different output format, it sometimes requires a different set of changes, that better reflect the way we think about the changes, instead of being as few steps as possible. Apparently, minimality is less important than readability for many uses.

Motivation

Why Difference Search?

Observation:

- differences need to be processable by machines and humans
- but requirements for machines and humans are different (efficient storage vs. presentation)
- the most widely used output format of GNU diff is not the smallest, but the format easiest to read by humans

Empiric observation: human processing does not only require a different output format, it sometimes requires a different set of changes, that better reflect the way we think about the changes, instead of being as few steps as possible. Apparently, minimality is less important than readability for many uses.

Motivation
State-of-the-Art

Text Differencing:

- usually based on efficient *longest common subsequence* (LCS) algorithms, e.g. GNU diff
- compares documents line-by-line
- changes given as line operations
- often only “add” and “delete” operations, no “move”

Not well-suited for XML:

- line unit unsuiteable (single line XML documents common)
- reordering common, and often not significant
- complex operations such as wrap-around or flatten

Motivation

State-of-the-Art

Text Differencing:

- usually based on efficient *longest common subsequence* (LCS) algorithms, e.g. GNU diff
- compares documents line-by-line
- changes given as line operations
- often only “add” and “delete” operations, no “move”

Not well-suited for XML:

- line unit unsuiteable (single line XML documents common)
- reordering common, and often not significant
- complex operations such as wrap-around or flatten

Motivation
State-of-the-Art

XML Differencing:

- often just longest-common-subsequence on “tokens” instead of lines
- existing approaches try to minimize a so-called *edit script* consisting of pre-defined “atomic” operations
- some can not handle reordering, few can handle wrap-around
- little parameterization possible

Consequences:

- usually does not reflect actual changes done by document authors (“*human aspect*”)
- often unsuited for visualising differences in an editor or for reviewing changes

Motivation
State-of-the-Art

XML Differencing:

- often just longest-common-subsequence on “tokens” instead of lines
- existing approaches try to minimize a so-called *edit script* consisting of pre-defined “atomic” operations
- some can not handle reordering, few can handle wrap-around
- little parameterization possible

Consequences:

- usually does not reflect actual changes done by document authors (“*human aspect*”)
- often unsuited for visualising differences in an editor or for reviewing changes

Motivation

State-of-the-Art – Example

```
<?xml version="1.0"?>
<root>
  <sub>
    <node>0</node>
    <node><b>1</b></node>
    <node>2</node>
    <node>3</node>
  </sub>
  <sub2>
    <node>0</node>
    <node>2</node>
    <node>3</node>
  </sub2>
  <other/>
</root>
```

```
<?xml version="1.0"?>
<root>
  <sub>
    <node>0</node>
    <node>1</node>
    <node>2</node>
    <node>3</node>
  </sub>
  <sub2>
    <node>0</node>
    <node><b>2</b></node>
    <node>3</node>
  </sub2>
  <other/>
</root>
```

Two revisions of a document, differences highlighted

Motivation

State-of-the-Art – Example cont'd

This is the output by Logilab xmldiff (an existing xmldiff application):
Logilab xmldiff uses the easiest-to-read (but non-XML) output format of the applications we tested:

```
[rename, /root[1]/sub[1]/node[2]/b[1], node]
[move-after, /root[1]/sub[1]/node[2]/node[1], /root[1]/sub[1]/node[1]]
[insert-after, /root[1]/sub2[1]/node[1], <node/> ]
[rename, /root[1]/sub2[1]/node[3], b]
[move-first, /root[1]/sub2[1]/b[1], /root[1]/sub2[1]/node[2]]
[remove, /root[1]/sub[1]/node[3]]
```

Can you comprehend and verify this edit script?

Motivation

State-of-the-Art – Example cont'd

This is the output by Logilab xmldiff (an existing xmldiff application): Logilab xmldiff uses the easiest-to-read (but non-XML) output format of the applications we tested:

```
[rename, /root[1]/sub[1]/node[2]/b[1], node]
[move-after, /root[1]/sub[1]/node[2]/node[1], /root[1]/sub[1]/node[1]]
[insert-after, /root[1]/sub2[1]/node[1], <node/> ]
[rename, /root[1]/sub2[1]/node[3], b]
[move-first, /root[1]/sub2[1]/b[1], /root[1]/sub2[1]/node[2]]
[remove, /root[1]/sub[1]/node[3]]
```

Can you comprehend and verify this edit script?

Motivation

Query by example

Xcerpt: rule-based querying and reasoning

- data, query and output (can) use the same syntax
- uses “query-by-example”
- syntax is very easy to read (“human aspect”)
- has constructs for loose matching (nesting, ordering, wildcards)
- theoretical background called *simulation unification*
- by S. Schaffert, F. Bry and others
- see xcerpt.org and the Proceedings 2004

Question: Is it possible to search for differences by using one document as loose “query-by-example” for the other?
Can we find the “maximal query” that matches both documents?

Motivation

Query by example

Xcerpt: rule-based querying and reasoning

- data, query and output (can) use the same syntax
- uses “query-by-example”
- syntax is very easy to read (“human aspect”)
- has constructs for loose matching (nesting, ordering, wildcards)
- theoretical background called *simulation unification*
- by S. Schaffert, F. Bry and others
- see xcerpt.org and the Proceedings 2004

Question: Is it possible to search for differences by using one document as loose “query-by-example” for the other?
Can we find the “maximal query” that matches both documents?

Motivation

Structure-Based Differencing

- based on maximal retainment of structure instead of minimal edit script ("finding commonalities instead of changes")
- inspired by *Simulation Unification* algorithm used in Xcerpt
- structure: not limited to "parent-child", may be *any* relation:
  - ancestor-descendant
  - sibling (unordered), next-sibling (ordered)
  - element-attribute
  - ID-IDREF
  - any combination of the above (or others) – XPath expressions work fine
- easily extensible: e.g. fuzzy matching

Motivation

Structure-Based Differencing – Example cont'd

Informally, our structure-based differencing produces the following steps:

1. Store ("cut") the text contents of the `<b>1</b>` node.
2. Cut the now empty `<b/>` node.
3. Reinsert the text ("1") cut earlier instead.
4. Cut the text "2" of the lower `<node>2</node>`.
5. Insert the `<b/>` node there.
6. Insert the text "2" into the `<b/>` node.

Note that our approach also takes 6 steps, but arguably these are more intuitive. Furthermore, they could be written as 3 overlapping move operations – if the output format allows move operations.

Contents

- Motivation
  - Why Difference Search?
  - State-of-the-Art
  - Structure-Based Differencing
- Structure Retainment
  - Document Graphs
  - Node Similarity
  - Nodeset Correspondences
  - Retained Relations
- Algorithm
  - Search Tree
  - Dijkstra's Algorithm
  - Optimised Cost Function
- Results
  - Experimental Results
  - Prototype
  - Summary

Structure Retainment

3 Relations

- Document Graphs:
  describe the “structural” relations inside the documents we want to retain as good as possible
- Node Similarity:
  defines which nodes are considered to be similar in different documents (usually label equality)
- Nodeset Correspondence:
  describe where nodes of the first document are located in the second document (possible solutions)

LMU

Document Graphs

Structure of the document

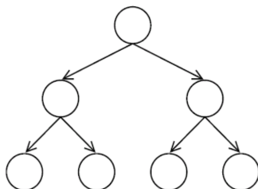
- describe the structure we want to retain
- formally: a possibly weighted graph

$$D = (V, E) \quad \text{or} \quad D = (V, E, \omega)$$

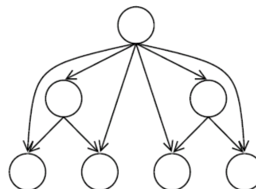
  where V is a finite set of nodes, $E \subseteq V \times V$ is a relation called "edges", and $\omega : E \longrightarrow \mathbb{R}$ is a weight function
- E is an arbitrary relation between nodes, does not necessarily correspond to DOM or Infoset tree
- attributes: treated like elements with label and content

Document Graphs

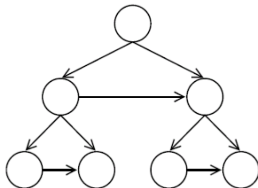
Example



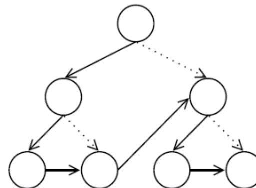
E_1: child relation



E_2: descendant relation



E_3: child or following-sibling relation



E_4: document order (immediate successors solid, child relation dotted, others not drawn)

Node Similarity

Matching restrictions

- describes which nodes are allowed to match
- relation between nodes (usually node labels) of different documents
- used to determine “acceptable” nodeset correspondences
- formally: given two document graphs $D = (V, E)$ and $D' = (V', E')$, any $\tau \subseteq V \times V'$ is a node similarity.
- typical similarity relations:
  - label equality (naïve but sufficient in most cases)
  - semantic equality (e.g. `<em>` and `<i>` in HTML)
  - substring, Levenshtein distance
  - query strings like “*.doc” or regular expressions
  - more complex, e.g. based on ontology information on the Semantic Web, dictionaries, etc.

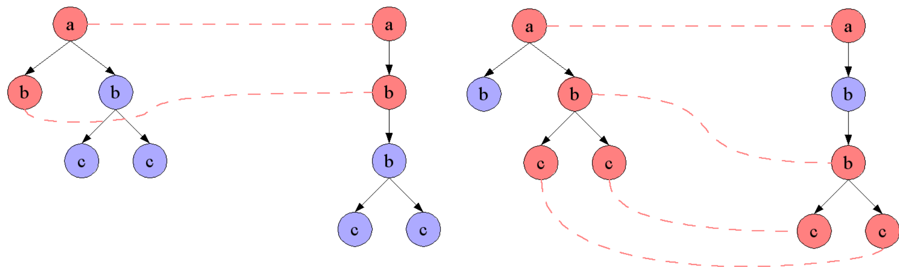
Nodeset Correspondences

Interim solutions

- describe the "location" of nodes from the first document in the second document – i.e. which nodes correspond
- there are many different nodeset correspondences
- usually we are only interested in *maximal nodeset correspondences*
- abstract representation of possible "solutions"
- the "best" (by some measure) correspondence is the solution we search
- formally: a nodeset correspondence is a partial one-to-one mapping $S : V \to V'$ from nodes V of the first document to nodes V' in the second document

Nodeset Correspondence

Example



Two different nodeset correspondences (nothing said about quality!).

Retained Relations

Measuring the quality of a correspondence

- measure indicating how many relations of the first document have been retained in the second document
- defines the quality of a nodeset correspondence
- formally, a relation $v \rightarrow w \in E$ $(v, w \in V)$ in the first document $D = (V, E)$ is considered *retained* by a nodeset correspondence S, iff $S(v) = v', S(w) = w'$, and $v' \rightarrow w' \in E'$ is a relation in the second document $D = (V', E')$
- for weighted document graphs, a measure ψ is applied (an example ψ using the document graphs' weight functions is given in the article)
- simplest case: count of retainted relations gives quality

LMU

Retained Relations

Measuring the quality of a correspondence

- measure indicating how many relations of the first document have been retained in the second document
- defines the quality of a nodeset correspondence
- formally, a relation $v \to w \in E$ ($v, w \in V$) in the first document $D = (V, E)$ is considered *retained* by a nodeset correspondence S, iff $S(v) = v', S(w) = w'$, and $v' \to w' \in E'$ is a relation in the second document $D = (V', E')$
- for weighted document graphs, a measure ψ is applied (an example ψ using the document graphs' weight functions is given in the article)
- simplest case: count of retainted relations gives quality

Retained Relations

Measuring the quality of a correspondence

- measure indicating how many relations of the first document have been retained in the second document
- defines the quality of a nodeset correspondence
- formally, a relation $v \rightarrow w \in E$ ($v, w \in V$) in the first document $D = (V, E)$ is considered *retained* by a nodeset correspondence S, iff $S(v) = v', S(w) = w'$, and $v' \rightarrow w' \in E'$ is a relation in the second document $D = (V', E')$
- for weighted document graphs, a measure ψ is applied (an example ψ using the document graphs' weight functions is given in the article)
- simplest case: count of retainted relations gives quality

Contents

- Motivation
  - Why Difference Search?
  - State-of-the-Art
  - Structure-Based Differencing
- Structure Retainment
  - Document Graphs
  - Node Similarity
  - Nodeset Correspondences
  - Retained Relations
- Algorithm
  - Search Tree
  - Dijkstra's Algorithm
  - Optimised Cost Function
- Results
  - Experimental Results
  - Prototype
  - Summary

LMU

Algorithm

Searching for the best solution

Task:

- determine the maximal nodeset correspondence S with the highest quality measure $\psi(S)$

Implementation:

- using well known search techniques: Dijkstra's Algorithm for Shortest Path Search together with optimized cost functions

Algorithm

Searching for the best solution

Task:

- determine the maximal nodeset correspondence S with the highest quality measure $\psi(S)$

Implementation:

- using well known search techniques:
  Dijkstra's Algorithm for Shortest Path Search
  together with optimized cost functions

Algorithm
Search Tree

- search through a tree of *states*,
  each representing a nodeset correspondence
- a state is a *solution*, if its associated nodeset correspondence is maximal.
- a solution is the *optimal solution*, if the quality measure of the associated correspondence is maximal. Dijkstra finds the optimal solution first.
- initial state:
  empty nodeset correspondence (no mappings, retains no structure)
- state transitions:
  add a single relation between a node in the first document and a node in the second document to the nodeset correspondence
- state transitions: nodes may be skipped (mapped to or from $\perp$)
- cost of a state:
  number/weight of *relations that cannot be retained* (informally)

Algorithm

Dijkstra's Algorithm

- graph algorithm for solving the shortest path problem for a directed graph with non-negative edge weights ("costs")

```
queue = []                         /* priority queue containing nodes */
initialNode.actual_cost = 0        /* initial node has no actual cost */
queue.insert(initialNode)

while not queue.isEmpty():
  candidate := queue.first()

  if isSolution(candidate):
     return candidate              /* best solution found; break loop */
  else:
     for nextNode in candidate.successors() do:
       queue.insert(nextNode)

     /* order queue ascending by path cost */
     queue.sortBy(lambda x,y: cmp(c(x),c(y)) )

return "no␣solution␣found"
```

Algorithm

Optimised Cost Function

- simply measuring the number of unretained relations for each state offers only a very low selectivity, in particular in the beginning of the algorithm
- improvements try to *predict* the number/weight of relations that we will have to drop later:
    - honour unavoidable drops of relations: a drop of an $a \longrightarrow b$ relation is unavoidable if the number of $a \longrightarrow b$-relations in the two documents differ
    - predict losses for cases where only one node is already mapped: If node a_1 in the first document is mapped to a_2 in the second, and the number of $a \longrightarrow b$ relations involving a_1 and a_2 differ, predict further losses
- if prediction is optimistic – i.e. it never overestimates – the search is guaranteed to still find the best solution
- careful handling of complex cases is required to avoid overestimation with above techniques
- details: see article

Contents

Motivation
- Why Difference Search?
- State-of-the-Art
- Structure-Based Differencing

Structure Retainment
- Document Graphs
- Node Similarity
- Nodeset Correspondences
- Retained Relations

Algorithm
- Search Tree
- Dijkstra's Algorithm
- Optimised Cost Function

Results
- Experimental Results
- Prototype
- Summary

Results

An example

```
<charts>
  <title>Top-Selling Harry Potter novels</title>
  <book>
    <rank>1</rank>
    <title>Harry Potter</title>
    <title2>and the Order of the Phoenix</title2>
    <edition>Hardcover</edition>
    <author>J. K. Rowling</author>
    <price>19.79</price>
    <isbn>043935806X</isbn>
  </book>
  <book>
    <rank>2</rank>
    <title>Harry Potter</title>
    <title2>and the Order of the Phoenix</title2>
    <edition>Paperback</edition>
    <author>J. K. Rowling</author>
    <price>8.99</price>
    <isbn>0439358078</isbn>
  </book>
</charts>
```

```
<charts>
  <title>Top-Selling Harry Potter novels</title>
  <book>
    <rank>1</rank>
    <title>Harry Potter</title>
    <title2>and the Order of the Phoenix</title2>
    <edition>Paperback</edition>
    <author>J. K. Rowling</author>
    <price>8.49</price>
    <isbn>0439358078</isbn>
  </book>
  <book>
    <rank>2</rank>
    <title>Harry Potter</title>
    <title2>and the Order of the Phoenix</title2>
    <edition>Hardcover</edition>
    <author>J. K. Rowling</author>
    <price>17.99</price>
    <isbn>043935806X</isbn>
  </book>
</charts>
```

Results

An example – cont'd

Results of this text case:

- our prototype recognizes the reordering of the "book" elements and the updated price and ranks as shown.
- other XMLdiff applications such as logilab xmldiff update the "price", "edition" and "isbn" tag contents.
- we didn't even need to add the "anchestor::book/isbn/text()" relation, our default worked fine.
  Using this would put extra emphasis on retaining the ISBN (in its role of a unique ID)

More examples are given in the article.

Results

Our Prototype

We have developed a prototype application:

- Written in C++ using libxml, developed and tested on Linux
- Released as Open Source on http://ssddiff.alioth.debian.org/ (the Debian GNU/Linux SourceForge equivalent)
- Allows arbitrary XPath expressions as document graphs
- Supports only label equality as node similarity right now
- No support for weights
- Approximative "fast" mode (experimental)
- Output as: XUpdate, annotated source files or merged document

LMU

Summary

A promising new approach for and old problem

- difference search for XML documents is a problem with many applications
- existing XML difference algorithms try to reduce edit scripts, disregarding human readability of the results
- our structure-based difference search is a flexible algorithm trying to retain as much structure as possible instead of minimising changes (edit scripts)
- implementation is rather straightforward using Dijkstra's shortest path algorithm and appropriate cost functions
- experiments indicate that results are indeed more suited for humans
- it is likely that there are benefits for machine-processing areas such as the merging of changesets, too

LMU

thank you for your attention!

salzburg|research
REWERSE reasoning on the web
LMU

Outlook

Other stuff to do. . .

- investigate further speed improvement ideas
- reduce memory usage
- 3-way mode (for merging)
- schema learning
- database applications
- RDF and other formats
- . . . your suggestion?