INSTITUT FÜR INFORMATIK
Lehr- und Forschungseinheit für
Programmier- und Modellierungssprachen

Oettingenstraße 67 D–80538 München

Ludwig
Maximilians–
Universität
München

Structure-Preserving Difference Search in Semistructured Data

Erich Schubert

Projektarbeit im Rahmen des Fortgeschrittenenpraktikums

Beginn der Arbeit:	10.11.2003
Abgabe der Arbeit:	4.1.2004
Betreuer:	Prof. Dr. François Bry
	Dr. Sebastian Schaffert

Zusammenfassung

Semistrukturierte Daten wie XML spielen eine zunehmend wichtigere Rolle im Web und Internet. Während es zwar zahlreiche Programme gibt, Differenzen für solche Daten berechnen und ausgeben können, haben alle ihre Schwächen. Meistens wird sogar das GNU "diff"-Programm verwendet, welches eigentlich nur für normale Textdateien geeignet ist und keine semantisch sinnvollen Änderungen für XML-Dateien ausgibt.

Der in dieser Arbeit präsentierte Ansatz orientiert sich nicht wie bestehende XML-Diff-Anwendungen an der Minimierung eines so genannten "Edit-Scripts", also der Minimierung von Edit-Operationen, sondern versucht die Struktur der Dokumente möglichst gut in Übereinstimmung zu bringen. Dahinter steht die Annahme, dass die Struktur bei Semistrukturierten Daten besonders wichtig ist. Dafür kann der Nutzer einen XPath-Ausdruck verwenden, um die Struktur anzugeben, die möglichst gut beibehalten werden soll.

Kernstück der Arbeit bildet eine verbesserte Kostenfunktion für die A^*-Suche – einem aus der Künstliche-Intelligenz-Forschung bekannten Suchalgorithmus – um die "beste strukturelle Übereinstimmung" zwischen den beiden Dokumenten effizient finden zu können.

Abstract

Semistructured data like XML play an increasingly important Role in the Web and Internet. While there are several programs capable of calculating and outputting differences for such data files, all have weaknesses. Usually the GNU "diff" tool is even being used, which was actually made for plain text files and whose output does not make semantic sense.

The approach presented in this work does not follow the approach of minimizing "edit scripts" – the minimizing of operations needed to transform one document to the other – but tries to maximize the structure preserved from one document to the other. This is based on the assumption that structure is an essential part of semistructured data. For this, the user can provide an XPath expression defining the structure to be preserved as good as possible.

The core part of this work is an improved cost estimation function to be used in an A^*-Search – an algorithm well-known from artificial intelligence – to find the "best structural correspondence" between the two documents faster.

Contents

1 Introduction

1.1 Motivation

XML is a generic file format for structured data, defined by the World Wide Web Consortium [W3C98], and used in more and more applications.

This file format currently is the number one choice for exchanging structured and semistructured data between applications running on different architectures, on different platforms, or written by different programmers.

One example is the XHTML format for web pages [W3C00], the latest version of the HTML format for web pages. Other examples are the XMPP protocol [IET04] (also known as Jabber Protocol) for instant messaging applications published by the Internet Engineering Task Force (IETF) and a draft standard by OASIS and used by OpenOffice for office documents saved as XML [OAS04], trying to specify a common interchange format.

There is a wide variety of other applications, tools, and libraries available for almost any platform. Many (newer) programming languages such as Java include XML support in their core libraries [JAX97].

Tracking changes in documents is an important functionality that can be used in numerous applications and has many uses such as in revision control. The difference output between two revisions of the same document is often used to manually review changes of documents, so their format must be easily readable. For example, modifications to the Linux Kernel must be submitted as GNU "diffutils" patch files (the most commonly used format for text file differences).

Up to now, the GNU "diffutils" applications [EHH+88] and file format are usually used for this task, since it is very fast, has a human-readable output and is available on most platforms (and is for example used in the cvs revision control system [BGea89]). For binary files, there is LibXDiff [Lib03], but this is only occasionally used (although supported for example by the PHP web scripting language).

The GNU "diff" application (part of "diffutils") has major flaws for application onto XML data: First of all, it operates line-based, which is a bad smallest unit for changes, since in XML whitespace such as line wraps is often not significant. In fact, a big XML file consisting of a single line may be equivalent to a multi-line file with respect to their parsed contents. Secondly, GNU "diff" is not aware of the structured nature of the XML data, but treats it as a string of characters. As a third point, ordering of subtrees in an XML file may be insignificant (for example, entries in an address book that can easily be sorted when needed) – GNU "diffutils" don't support such re-orderings, but will consider these insertions and deletions.

The biggest problem of applying GNU "diff" (and similar token-, not line-based longest-common-subsequence implementations) to XML files is that it may mismatch closing and opening tags, as Figure 1 illustrates.

The changes as calculated by GNU "diff" violate the well-formedness of an XML document: These are not elementary tree operations (i.e. additions, deletions, moves), but remove the closing tag of one node, and the opening tag of a different node.

The hand-made "patch" (the output format of GNU "diff" is usually referred to as a "patch")

First source document:

```
<?xml version="1.0"?>
<a>
  <b>
    <c/>
  </b>
  <b>
    <c/>
  </b>
</a>
```

Second source document

```
<?xml version="1.0"?>
<a>
  <b>
    <c/>
    <c/>
  </b>
</a>
```

Result as generated by GNU “diff”:

```
 <?xml version="1.0"?>
 <a>
   <b>
     <c/>
-  </b>
-  <b>
     <c/>
   </b>
 </a>
```

Hand-made “patch”:

```
 <?xml version="1.0"?>
 <a>
   <b>
     <c/>
+    <c/>
   </b>
-  <b>
-    <c/>
-  </b>
 </a>
```

Figure 1: Applying GNU “diff” for XML data

only uses elementary tree operations such as the insertion of a node or the deletion of a subtree.

1.2 Existing XML Diff Applications

The task of generating a “patch” for XML files basically consists of two parts. The first part is identifying the changes (or measuring the similarity), the second is to generate an output for these changes.

For each of these parts, there are different approaches. For measuring the similarity, S. Flesca et al [FMM$^+$02] interpret the documents as time series and use fast fourier transformations, the traditional approaches are to do a longest-common-subsequence (LCS) search on tokens (like done by GNU “diff” for lines), or to identify identical subtrees and try to “grow” this to a complete solution.

Most published algorithms try to minimize the edit script used to transform one document to the other. Others do not actually try to find the minimal edit script, but try to speed up the process, accepting to achieve a suboptimal result faster.

There is an extensive comparison of various XML diff applications done by a French national research group at INRIA [CAH02].

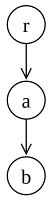
Example applications include: xydiff by Verso at INRIA [CAM02], DeltaXML [Mon00], xdiff [WDC03], mmdiff [MT01], logilab xmldiff [Log] and IBMs XML Diff and Merge Tool [IBM99]. Each of these has its good and its bad sides. For example the xdiff algorithm [WDC03] – which claims to find the minimal editing script – only considers nodes with the same labels on the path from the root to be matches and thus cannot cope with root renames at all. Apparently this approach fails to recognize nesting changes in XML documents, which are a frequent operation in XHTML files (for example adding a `<div>` element around a document part for layout reasons)

The example files shown in Figure 2 (containing a simple nesting inversion) already pose a problem for many diff applications. Logilab XMLdiff for example just replaces the whole first document by the second, not even keeping the root node (which has not changed). xydiff, probably one of the best XML diff applications available, does keep the root node, but neither the `<a/>` nor the `<b/>` node.

First document:

```
<?xml version="1.0"?>
<r>
  <a><b/></a>
</r>
```

Graph:



Second document:

```
<?xml version="1.0"?>
<r>
  <b><a/></b>
</r>
```

Graph:

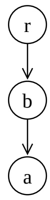


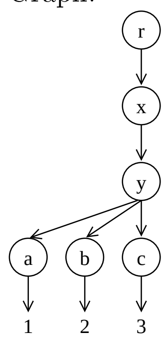
Figure 2: Simple test documents

The test documents in Figure 3 produce very complex results by Logilab xmldiff, that certainly do not reflect the intuitive changes applied to the document: moving the `<a/>` and `<b/>` subtrees into the `<c/>` node, replacing the text "`3`". Instead a new root node is added, the old root node itself is renamed to `<x/>` and moved around, the `<x/>` node is named `<y/>`, the `<y/>` node renamed to `<c/>` etc. – see Figure 4.

First document:

```
<?xml version="1.0"?>
<root>
  <x><y>
    <a>1</a>
    <b>2</b>
    <c>3</c>
  </y></x>
</root>
```

Graph:



Second document:

```
<?xml version="1.0"?>
<root>
  <x><y>
    <c>
      <a>1</a>
      <b>2</b>
    </c>
  </y></x>
</root>
```

Graph:

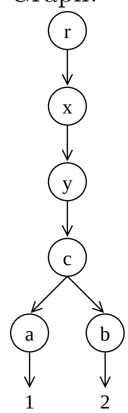


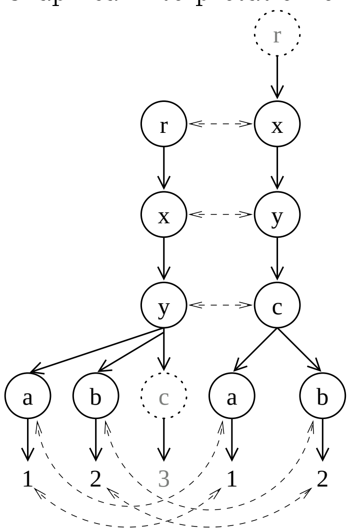
Figure 3: More test documents

Existing xmldiff applications often have an own diff format, only some support known formats such as XUpdate. The used formats have different properties: some are XML formats themselves, others are not (e.g. Logilab xmldiff output format), some allow aggregation of changes without knowledge of the original file (e.g. xydiff format) and sometimes there is no program

Output by Logilab xmldiff in its own format:

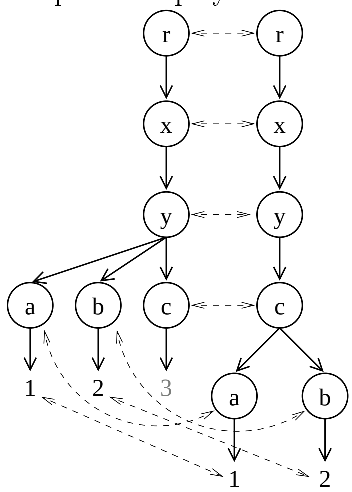
```
[append-first, /, <root/> ]
[rename, /root[2], x]
[move-first, /x[1], /root[1]]
[rename, /root[1]/x[1]/x[1], y]
[rename, /root[1]/x[1]/y[1]/y[1], c]
[remove, /root[1]/x[1]/y[1]/c[1]/c[1]]
```

Graphical interpretation of the output:



Edit operations: One deletion, one move, one addition, three renames.

Graphical display of the intended changes:



Edit operations needed: One deletion, two move operations.

Figure 4: Result produced by Logilab xmldiff

available to apply the generated output to an XML file.

This goal of minimizing the "edit script" maybe is not always the best primary goal. For example, producing an human-understandable edit script may be more valuable in some applications (e.g. for reviewing changes). Slightly non-minimal edit scripts are already often preferred for getting results faster – getting a human-readable output might be another reason for this.

The amount of edit operations needed depends on the output format (which may for example not include "move" operations), and on the update processor model: in XML with DOM (XML Dynamic Object Model), text nodes will collapse when the separating node is removed, in other models this is different. By doing all text node removals first and all text node additions last, many but not all of these cases can be covered. In some cases, additional "dummy" nodes will need to be inserted and removed afterwards, increasing the real edit operation count. Also, deleting a sequence of normal nodes differs from deleting a sequence of mixed text and normal nodes: when all normal nodes are deleted, the text nodes have collapsed and can be deleted in one operation. Again, a human-understandable edit script is preferable to a minimal one – editing the text nodes at the same time as editing the neighboring nodes is preferred.

When you just count the number of update statements in XUpdate for example, you can transform one document to any other with just two XUpdate statements: one deleting the old document, one inserting the new.

Therefore, the count of edit operations is not a well-suited measure for comparing the quality of a difference description.

1.3 A Different Approach

The first thing that comes to mind when thinking about a better quality comparison, is to look at the number of nodes kept (or changed).

Unfortunately, this allows for unnecessary (and unwanted) moves of nodes in the document – as long as the node is still anywhere in the other document, this would be considered good. Structure is not necessarily preserved, changes may be hard to review.

This leads to the concept of trying to preserve the structure: Instead of looking at the number of nodes preserved, the number of relations between preserved nodes that exist in both documents is looked upon. This is an extension of above approach: a document not keeping a node will not be able to keep any of the relations of this node, whereas a document keeping an additional node will keep at least as many relations.

The approach presented in this paper differs from the ones cited in the previous section because it tries to *maximize the structure preserved* during the transformation, where structure is given in form of a configurable relation on the nodes – for example using an XPath expression. In a second run, when the best structure preservance has been found, the output functions will then try to optimize the edit script to achieve the intended changes in an efficient and understandable way.

For understanding this approach, it is important to become familiar with this notion of "structure" and "structure similarity". Albeit the common use will define structure as being the child relation or the descendant relation (and thus "structure similarity" as having similar child relations for matched nodes), the algorithm is not limited to that. Using different relations gives additional possibilities: for example when using the sibling relation as well, it will support ordered documents. Also a weight function may be used to assign different importance to the individual relations.

Instead of trying to minimize the edit script we can try to match as much as possible of the structure of the XML documents, while allowing for custom preferences (by choosing the relations and weights). The algorithm doesn't look for changes, but for keeping similar structures intact.

It is not limited to application onto labeled trees or directed acyclic graphs (DAG), but can work with arbitrary graphs. In fact it should be well-suited for generic labeled graph similarity search: we search for the mapping between the two documents such that a maximum number of edges is kept. Algorithms working on trees are often very efficient, but unfortunately, when adding additional relations, this property of the XML document is lost (adding the descendant and following-sibling relations will make the graph become a DAG, adding relations to all siblings will make it a normal graph).

Using XML documents and similar semistructured data allows a more intuitive approach to the topic than generic graphs as well as showing the key strengths.

Using the algorithm on records with a fixed structure – such as database tables – makes no sense (since the structure is fixed and cannot be preserved better or worse), there are more efficient algorithms for such cases. This algorithm is designed for finding similar structures in semistructured documents such as XML files.

Choosing the relations and weights, the user may also force exact matches for example on id attributes (i.e. unique attributes) and even force propagation to neighboring nodes. The choice of relations, weight functions and approximations will of course influence the efficiency of the algorithms, too.

Finally, it may be preferable to use a modified algorithm that is faster – even when it does not give the "optimal" result.

2 Preliminaries

Since we'll deal with many different relations, it is important not to confuse them. Therefore, we'll give a short overview of the relations to be used and defined:

1. Each document has a relation representing the structure we want to retain (such as child relation, descendant relation etc.). This relation is called the *edges* of the graph that represents the document.
2. Between two node sets there is a *node similarity* relation, matching "similar" nodes (for example label equality). This specifies, which nodes of the first document are candidates for being mapped to which nodes of the second document.
3. A *node set correspondence* is a collection of similar nodes, forming a partial one-to-one mapping between the two documents. This is a "possible solution".
4. Each set of *retained relations* stores four nodes, two in each tree, such that in each tree the nodes are connect by an edge and both top and both bottom node pairs are "similar" (with respect to the second relation in this list). This is the set we want to measure and weight to compare the quality of a match.

Note: In the following we assume all our sets and structures are finite. Any two directed node sets will be assumed disjoint unless being explicit noted subset of a given node set.

2.1 Document Graphs

A document is represented by an arbitrary directed (possibly weighted) graph $D = (V, E)$ or $D = (V, E, \omega_E)$ consisting of a finite set of nodes V, a relation $E \subseteq V \times V$ called "edges" and eventually a weight function $\omega_E : E \to \mathbb{R}$. The notations $v_1 \to v_2$ or $v_1 \to_E v_2$ will be used as shorthand for the edge $(v_1, v_2) \in E$.

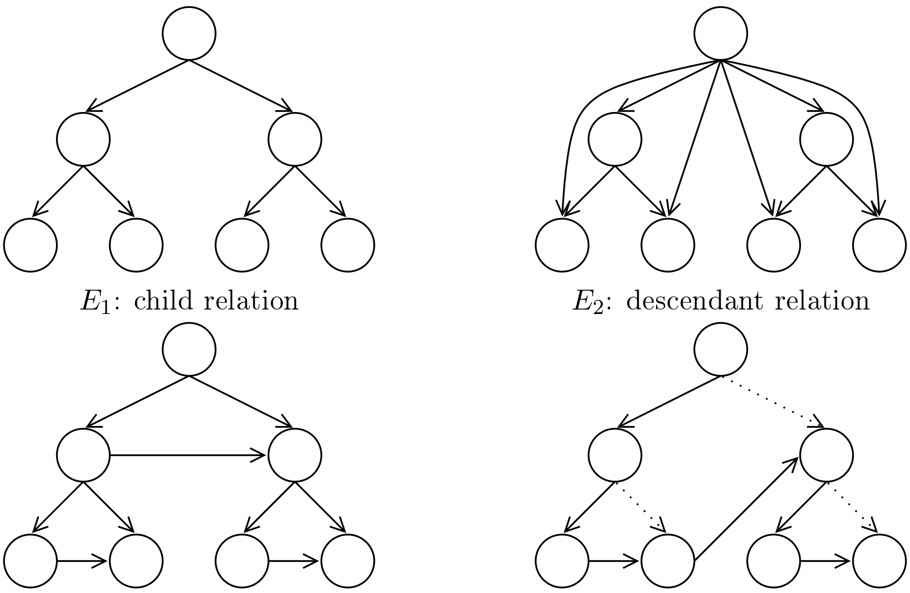
Multi-graphs (graphs with more than one edge between the same two nodes) are not allowed.

Note that this relation does not have to be the edges of the document tree (i.e. the child relations), but can be arbitrary. Usually the edges of the tree itself will be part of the relation, but they need not, or they might get assigned a low weight in specific use cases.

Examples: Let V be the set of nodes from a given (finite) tree or XML document.

1. Let E_1 be the child relation of the tree.
2. Let E_2 be the ancestor-descendant relation (E_2 is the transitive closure of E_1) and $\omega_{E_2}((v_1, v_2)) = \frac{1}{d}$ with d the distance of the nodes in E_1.
3. Let $E_3 := E_3' \cup E_3''$, E_3' the parent-child relation and E_3'' the sibling order (i.e. a true subset of the document order).
4. Let E_4 be the document order (as obtained by depth-first-traversal).

Then $D_1 = (V, E_1)$, $D_2 = (V, E_2, \omega_{E_2})$, $D_3 = (V, E_3)$ and $D_4 = (V, E_4)$ are document graphs. Figure 5 shows examples of these relations.



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)

Figure 5: Examples for relations E_1, E_2, E_3, E_4 on a small tree

We have now defined how a document looks like for this approach. Next we have to start interconnecting nodes from two documents:

2.2 Node Similarity Relations

We have to state when two nodes (without the context) are candidates for matching. Usually this will be simple things such as label equality, but we can use an arbitrary relation. A relation which fulfills a certain form of symmetry and transitivity that allows us to build "equivalence classes" of nodes (that span both documents) will be called a "node equivalence". Working on equivalence classes can significantly contribute to a faster algorithm, making the cost prediction function more precise.

Definition: Let $D = (V, E_1)$, $D_2 = (W, E_2)$ be two document graphs.

A *node similarity (relation*[1]*)* $S_{V,W}$ between V and W is an arbitrary subset $S_{V,W} \subseteq V \times W$. Two nodes $v \in V$, $w \in W$ are called *similar* $(v \sim w)$, iff $(v, w) \in S_{V,W}$.

Examples:

1. "label equality" on the nodes of two labeled trees is a node similarity.
2. "substring-of", "superstring-of" and "substring-or-superstring-of" on the text node contents of two trees are node similarities
3. "is-regular-expression-for" is a node similarity that could be used to match a query document with a database.
4. "has-the-same-labeled-path" is the node similarity relation used in xdiff [?].

Not every node similarity is well-suited for every use. There is a special class of similarity relations with nice properties, similar to equivalence relations in mathematics (but between the elements of two disjoint sets):

Definition: Two nodes $v_1, v_2 \in V$ in the same node set V are called *equivalent* $(v_1 \equiv v_2)$ *with respect to the node similarity* $S_{V,W}$ *and the node set* V, iff they are similar to the same nodes in the second node set W, that is $v_1 \equiv v_2 :\Leftrightarrow \forall_{w \in W} (v_1 \sim w \Leftrightarrow v_2 \sim w)$.

Definition: Let $S_{V,W}$ be a node similarity between node sets V and W.

Let $\overline{v_1} := \{v \in V | v \equiv v_1\}$ be the corresponding equivalence class on V.

Let $\overline{\emptyset} := \emptyset$ and let $\overline{V} := \{\overline{v_1} | v_1 \in V\} \cup \{\overline{\emptyset}\}$ be the set of equivalence classes in V.

Let $\hat{v}_1 := \{w | v_1 \sim w\} \overset{\text{by def.}}{=} \{w | \exists_{v \in \overline{v_1}} v \sim w\} =: \hat{\overline{v_1}}$ be the image of v_1 (or any of its equivalent nodes) under the similarity relation $S_{V,W}$. Let $\hat{\emptyset} := \{w | \nexists_{v \in V} v \sim w\}$ be the set of nodes in the node set W, where no node $v \in V$ is similar to.

Using the symmetry of the node similarity definition, we can also define $\overline{W}$ and $\overline{w}$ the same way.

The following are equivalent (no proof given here) definitions of a *node equivalence* $S_{V,W}$:

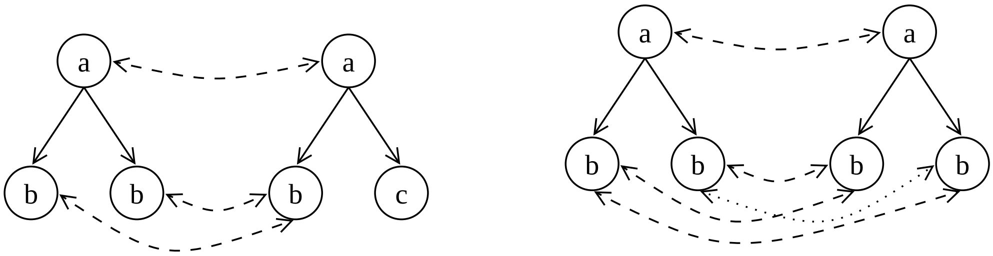
1. The union $\bigcup_{\overline{v_1} \in \overline{V}} \hat{v}_1 \cup \hat{\emptyset} = W$ is disjoint, i.e. each node in W is mapped to by only one equivalence class of V (i.e. $\forall_{w \in W} \forall_{v_1, v_2 \in V} v_1 \sim w \wedge v_2 \sim w \Rightarrow \overline{v_1} = \overline{v_2}$).

[1]The term "relation" will not be written explicitly to avoid confusion with the relation of the document graph.

2. There is a bijective correspondence (transformation) of equivalence classes between the documents $\pi : \overline{V} = \{\overline{v} \mid v \in V\} \to \overline{W}$ such that $\pi(\hat{v}) = \overline{w} \Leftrightarrow v \sim w$.

3. $\forall_{v_1,v_2 \in V} \forall_{w_1,w_2 \in W} v_1 \sim w_1 \wedge v_1 \sim w_2 \wedge v_2 \sim w_1 \Rightarrow v_2 \sim w_2$, i.e. if two nodes from the first document are both similar to one node in the second document, they are always either both or none similar to a given node.

Examples: Figure 6 shows an example for a node equivalence relation and a relation violating equivalence.



this is an equivalence relation

if the dotted relation is missing, this is no longer an equivalence relation

Figure 6: Example node similarity/equivalence relation

Examples 1 and 4 given before are node equivalence relations, 2 and 3 are not equivalences in general: for example consider the "substring-of" relation $\leq$: "a" $\leq$ "ab" and "a" $\leq$ "ac", "b" $\leq$ "ab" but not "b" $\leq$ "ac" (violating characterization 3 obviously)

Remark: Reflexivity and transitivity are only defined on one set. Since node similarity is (in general) defined between two different sets, the use of the term "node equivalence relation" needs to be well grounded. Let V and W be disjoint, then try to extend the given relation to a mathematical equivalence relation on the union set:

Let $O := V \dot{\cup} W$ ($\dot{\cup}$ denotes disjoint union). Extend the node similarity relation $S_{V,W} \subseteq V \times W$ to $O \times O$ by adding symmetry $S'_O = \{(w, v) \mid v \sim w\} \subseteq W \times V$ and reflexivity $S''_O = \{(o, o) \mid o \in V \vee o \in W\}$. Then $(S_{V,W} \dot{\cup} S'_O \dot{\cup} S''_O) \cap V \times W = S_{V,W}$ obviously holds.

Iff for the transitive closure $\overline{S_{V,W} \dot{\cup} S'_O \dot{\cup} S''_O} \cap V \times W = S_{V,W}$ holds as well (i.e. $S_{V,W}$ is "transitively closed" but restricted to $V \times W$), then S is a node equivalence relation, i.e. if there is any directed path relating two nodes, then they must be immediately related (i.e. similar).

2.3 Node Set Correspondences

In the last two sections we defined how a document looks like, and which nodes are similar to each other. Using this we can now build a connection between two documents, consisting of one-to-one mappings of "similar" nodes of each tree. Such a mapping is what a "solution" looks like. In the next section, we will define the quality of such solutions then.

Definition: Let $D_1 = (V, E_1)$, $D_2 = (W, E_2)$ be two documents with a given node similarity relation $S_{V,W}$ on V, W.

If $\sigma \subseteq S_{V,W} \subseteq V \times W$ is a bijective partial mapping ($\sigma : V \rightharpoonup W$) between these two node sets, written as $v \sim_\sigma w :\Leftrightarrow (v, w) \in \sigma$ or $\sigma(v) = w$, mapping each node of each set to at most one similar node in the other set, i.e.

$$\forall v \in V\, \forall w_1, w_2 \in W : (v \sim_\sigma w_1 \wedge v \sim_\sigma w_2 \Rightarrow w_1 = w_2)$$

$$\forall v_1, v_2 \in V\, \forall w \in W : (v_1 \sim_\sigma w \wedge v_2 \sim_\sigma w \Rightarrow v_1 = v_2)$$

then call σ a *node set correspondence.* An element $(v, w) \in \sigma$ is called a *node correspondence.*

Let $S^{\mathcal{P}}_{V,W} \subseteq \mathcal{P}(S_{V,W})$ be the the set of node set correspondences (which is a part of the power set of node correspondences).

A node set correspondence σ is called *maximal* with respect to $S_{V,W}$, if there is no node set correspondence τ such that $\sigma \subsetneq \tau \subseteq S_{V,W}$.

We call two node set correspondences disjoint and *non-conflicting*, iff they don't map the same nodes, i.e. $\nexists_v \exists_{w_1,w_2} \sigma(v) = w_1 \wedge \tau(v) = w_2$ and $\nexists_w \exists_{v_1,v_2} \sigma(v_1) = w \wedge \tau(v_2) = w$.

Remark: There might be multiple maximal node set correspondences for the same documents D_1, D_2 and the same similarity relation $S_{V,W}$.

Maximal node set correspondences are what we search for as solutions, other node set correspondences are interim steps (incomplete solutions).

Examples: Figure 7 shows two maximal node set correspondences for the same two simple trees.

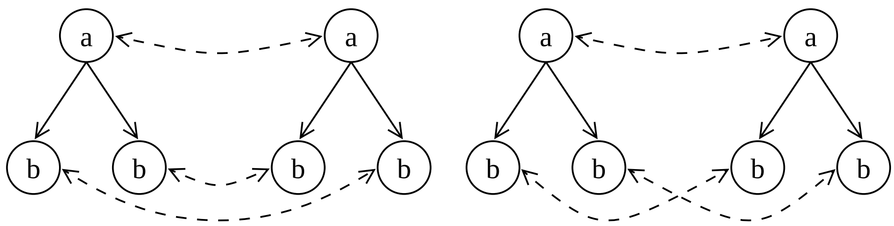


Figure 7: Two maximal node set correspondences. Leaving away any dashed arrow leads to a non-maximal node set correspondence

1. Let $D_1 = D_2 = (V, E)$ and $\sigma = \{(v, v) | v \in V\} \subseteq S_{V,V}$. Then σ is a node set correspondence with respect to $S_{V,V}$. Note that the relation $S_{V,V}$ may not allow identity matches everywhere. Then σ may be not maximal.

2. Let $D_1 = D_2 = (V, R)$ and a similarity relation $S'_{V,V} \subseteq \{(v, v) \mid v \in V\}$ subset of the equality on V. Then $S'_{V,V}$ itself is the only maximal node set correspondence with respect to itself.

2.4 Retained Relations

The core idea of this paper is to measure the quality of a mapping by measuring the relations retained.

Therefore, we need to formalize when a relation is retained by a node set correspondence.

Definition: Let $D_1 = (V, E_1)$, $D_2 = (W, E_2)$ be two document graphs and $S_{V,W} \subseteq V \times W$ be a given node similarity relation.

Define the *set of retained relations* of a given node set correspondence $\sigma \subseteq S_{V,W} \subseteq V \times W$ for two documents D_1 and D_2 as

$$\mathcal{R}(\sigma, D_1, D_2) := \mathcal{R}(\sigma, E_1, E_2) := \{((v_1, w_1), (v_2, w_2)) \in \sigma \times \sigma \mid v_1 \sim v_2 \wedge w_1 \sim w_2\} \subseteq S_{V,W} \times S_{V,W}$$

The overall situation is explained in Figure 8. We have a relation from each document, such that the "first" nodes in each documents are in the node set correspondence (therefore they are similar, too), and the bottom nodes as well.

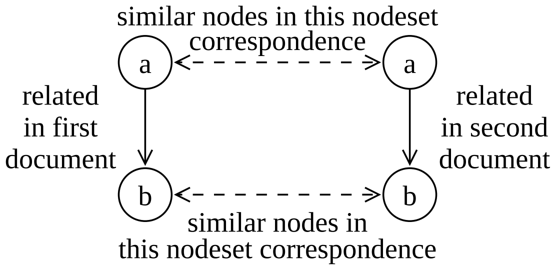


Figure 8: Overall situation for a retained relation

They need to be in the same order and may not be swapped around (as shown in figure 9). Different maximal node set correspondences can have different sets of retained relations, as shown in figure 10.

We'll often also write $\mathcal{R}(\sigma)$ where it is obvious which D_1, D_2 are meant.

The notion $\mathcal{R}(S_{V,W}) := \{((v_1, w_1), (v_2, w_2)) \in S_{V,W} \times S_{V,W} \mid v_1 \sim v_2 \wedge w_1 \sim w_2\}$ will be called the set of retainable relations. Since $S_{V,W}$ is not always a node set correspondence, this is not a set of retained relations! Still this is a useful set: It is obvious that $\mathcal{R}(\sigma) \subseteq \mathcal{R}(S_{V,W})$ holds for $\sigma \subseteq S_{V,W}$.

2.5 Retainment Quality Functions

In order to compare different node set correspondences (i.e. different solutions), we need a quality function to rank them. Depending on the function, integers may be sufficient, computer implementations will choose integer or float numbers. For the mathematical view we allow $\mathbb{R}$.

Using the mathematical structure of measures we can skip proving much, therefore we will try to step down from the more general level of all allowed functions and limit ourselves to such functions derived from certain well-formed measures.

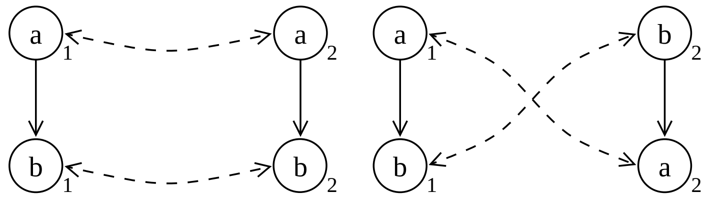


Figure 9: Retained and non-retained child relations in the previous example tree (the $b_2 \to a_1$ arrow would need to point in the other direction in order to be "retained")

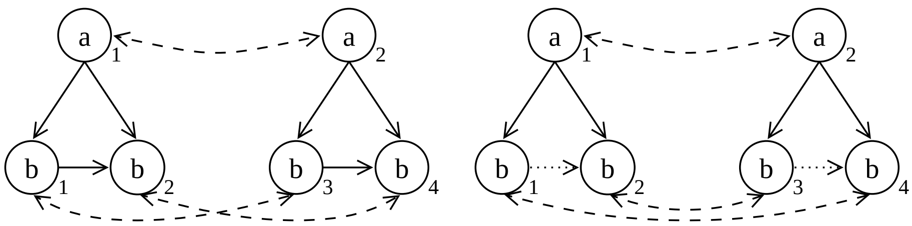


Figure 10: The left node set correspondence maintains 3 relations on each side (solid lines), the node set correspondence on the right doesn't retain the dotted relations (following-sibling relation)

Definition: Let $D_1 = (V, E_1)$, $D_2 = (W, E_2)$ be two document graphs and $S_{V,W} \subseteq V \times W$ a given node similarity relation.

Iff the map from the set of node set correspondences $\varphi : S_{V,W}^{\mathcal{P}} \to \mathbb{R}$, $\varphi \geq 0$ is additive or superadditive for $A, B \subset S_{V,W}$ disjoint, that is $\varphi(A \dot\cup B) \geq \varphi(A) + \varphi(B)$ holds for all A, B, call φ a *(structure) retainment quality function.*

Superadditivity basically means that if we match two non-conflicting subsets at the same time the quality is at least as good as (usually even better than) if we try to match them independently. While additivity is required, superadditivity will usually be the case: matching a single child node isn't worth much, but adding a matching child node to a matching context makes the context match more valuable, too. (Super-) Additivity will be needed for an optimistic cost function in the A^*-Search algorithm (explained in the next section). It prevents the case that matching additional nodes will decrease the total quality.

Proposition: Let the map from the power set of retainable relations, $\psi : \mathcal{P}(\mathcal{R}(S_{V,W})) \to \mathbb{R}$, $\psi \geq 0$ be a measure. Then $\varphi(\sigma) := \psi(\sigma)$ is a *retainment quality function.*

Proof: For σ, τ disjoint and non-conflicting, $\mathcal{R}(\sigma \dot\cup \tau) \supseteq \mathcal{R}(\sigma) \cup \mathcal{R}(\tau)$ by definition of $\mathcal{R}$ and since ψ is a measure $\psi(\mathcal{R}(\sigma \dot\cup \tau)) \geq \psi(\mathcal{R}(\sigma) \cup \mathcal{R}(\tau)) \geq \psi(\mathcal{R}(\sigma)) + \psi(\mathcal{R}(\tau))$. Thus $\varphi(\sigma \dot\cup \tau) = \psi(\mathcal{R}(\sigma \dot\cup \tau)) \geq \psi(\mathcal{R}(\sigma)) + \psi(\mathcal{R}(\tau)) = \varphi(\sigma) + \varphi(\tau)$.

For a measure it is sufficient to give the values on a σ-Algebra, for example the single retainable relations. This is an easy and intuitive way of giving a measure: assigning weights to the individual edges.

Proposition: Let $\mu : E_1 \cup E_2 \to \mathbb{R}$, $\mu \geq 0$ be a map from the edges of documents $D_1 = (V, E_1)$, $D_2 = (V, E_2)$ to nonnegative numbers. Then μ induces a measure $\mu' : \mathcal{P}(\mathcal{R}(S_{V,W})) \to \mathbb{R}$ in the following way: $\mu'\left(\{((v,w),(v',w'))\}\right) := \mu((v,w)) + \mu((v',w'))$ and $\varphi(\sigma) := \mu'(\sigma)$ is a retainment quality function.

Examples: Using a measure on the set of retained relations is the easiest approach to define a retainment quality function, allowing to use many well-known properties of measure functions. The following are retainment quality functions:

1. The count of matched nodes, $\varphi_c(\sigma) := |\sigma|$ is additive. (But it can be derived from a measure function if the relations on the node sets contain identity.)
2. $\varphi_m(\sigma) := |\mathcal{R}(\sigma)|$, the count of retained relations, since $\psi_m(R) := |R|$ is a measure.
3. Let $D_1 = (V, E_1, \omega_{E_1})$, $D_2 = (W, E_2, \omega_{E_2})$ be document graphs as given in example 3 of the document graphs, weighting longer ancestries less than short ones and choose a falloff exponent k:

   Let $d_k : ((v,w),(v',w')) \mapsto (\omega_{E_1}(v,v') \cdot \omega_{E_2}(w,w'))^k$.

   Then $\psi'_k(R) := \sum_{r \in R} d_k(r)$ is a measure and let $\varphi'_k(\sigma) := \psi'_k(\mathcal{R}(\sigma))$.

   An example for this is shown in figure 11. Using $k = 1$ the left tree has $\varphi'_1(\sigma_1) = 2$ and the right tree $\varphi'_1(\sigma_2) = 1.5$.

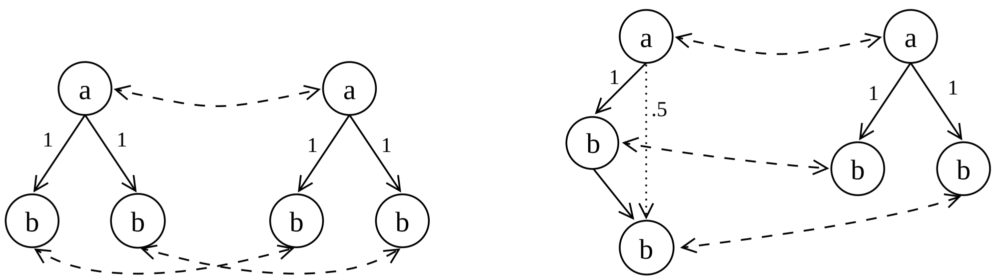


Figure 11: Two node correspondences (the retained relations are not plotted) with a distance function assigned to the relations. The dotted relation is a descendant relation of length 2, assigned $\frac{1}{2}$.

Lemma: Given a set of node set correspondences $\{\sigma_j\}$, $\varphi(\sigma_j)$ defines a total pre-order on this set.

Especially, a set of maximal node set correspondences is totally pre-ordered by such a function.

Proof: trivial: $\varphi(\sigma_j) \in \mathbb{R}$ is total and $\mathbb{R}$ is ordered.

2.6 Optimal Structure Correspondence

Since we are searching for the "optimal structure correspondence", we have to formally define what we consider to be the best solution. This of course uses the previous notion of node set correspondences and retainment quality:

Definition: An *optimal structure correspondence* with respect to a retainment quality function $\varphi : \mathcal{P}(\mathcal{R}(S_{V,W})) \to \mathbb{R}$ is a node set correspondence which is maximum with respect to the order given by the retainment quality function φ.

An *optimal structure correspondence* with respect to a measure $\psi : \mathcal{R}(S_{V,W}) \to \mathbb{R}$ on the set of retainable relations is a node set correspondence which is maximum with respect to the order given by the retainment quality function $\varphi(\sigma) := \psi(\sigma)$.

An *optimal structure correspondence* with respect to a measure $\mu : E_1 \cup E_2 \to \mathbb{R}$ on the edges of the documents is a node set correspondence which is maximum with respect to the order given by the retainment quality function $\varphi(\sigma) := \mu'(\sigma)$ and the induced measure $\mu'(\{(v, w), (v', w')\}) := \mu(v, w) + \mu(v', w')$.

Lemma: A retainment quality function $\varphi(\sigma) := \psi(\mathcal{R}(\sigma))$ induced by a measure on the set of retained relations is monotone with respect to the subset relation on $S_{V,W}$. That is $\sigma \subseteq \sigma' \Rightarrow \varphi(\sigma) \leq \varphi(\sigma')$.

If $\varphi(\sigma')$ is strictly positive, the subset partial order of node set correspondences is a subset of the partial order induced by the retainment quality function φ, i.e. any node set correspondence σ with $\varphi(\sigma)$ maximum will be a maximum node set correspondence.

Proof: Obvious, since ψ is a measure:

$\sigma \subseteq \sigma' \Rightarrow \mathcal{R}(\sigma) \subseteq \mathcal{R}(\sigma') \Rightarrow \psi(\mathcal{R}(\sigma)) \leq \psi(\mathcal{R}(\sigma)) \Rightarrow \varphi(\sigma) \leq \varphi(\sigma')$

Definition: A retainment quality function φ coming from the measure ψ is called *invariant* with respect to the node set equivalence relation $S_{V,W}$ if for $v_1 \equiv v_1' \wedge v_2 \equiv v_2' \wedge w_1 \equiv w_1' \wedge w_2 \equiv w_2' \Rightarrow \psi((v_1, v_2), (w_1, w_2)) = \psi((v_1', v_2'), (w_1', w_2'))$. Then it is sufficient to give v_1, v_2 to this function to evaluate the value.

Comment: An optimal structure correspondence together with a retainment quality function gives a distance function for graphs.

3 Using A^*-Search for finding solutions

A^*-Search is a generic algorithm for searching solutions in a graph of possibilities. Dijkstra's shortest-path algorithm is a special case of A^*-Search. It is a best-first expansion search using a cost estimation function. When given a specific cost estimation function, it can also behave like breadth-first search.

A^*-Search is shown to be optimal with respect to a given optimistic and admissible heuristic cost function, i.e. no other search will expand less nodes when using the same heuristic.

An optimistic cost estimation function is a function which never overestimates the real costs, an admissible cost function (not cost estimation function) is never decreasing during the search. In a nutshell, the optimistic cost estimation is used to decide which branch is most likely to contain the best solution, whereas the admissible cost function is used to verify that no other branch is better.

The cost estimation functions presented here are both optimistic and admissible during the search process, and at the end the result will even be the real costs. This removes the need for having a separate admissible cost function.

The cost estimation functions require a node *equivalence* relation and an the improved functions also an *invariant* retainment quality function.

3.1 A^*-Search

State information object:

1. current traversal position for first document
2. list of node correspondences (i.e. list of previously matched nodes)
3. current cost estimation and "credits" information for cost function (explained in following sections)

Algorithm: The basic A^*-Search Algorithm
queue := *list of state information objects, sorted by cost*
startState := *initial state*
solution := *no solution*
finished := false
insert startState *into* queue
while (not finished)
    bestState := *get first element from* queue
    if bestState *is complete* **then**
        solution := bestState
        finished := true
    else
        foreach newState **in** *calculate next moves from* bestState **do**
            estimate costs for newState
            insert newState *into* queue
/ solution contains the best result */*

Extensions: The algorithm can be extended using the usual A^* improvement techniques (especially for solving memory troubles, or when searching for approximate solutions). Also a B^*-Search might be possible.

By using a secondary sort on the queue for ranking states with identical costs – for example sorting by the count of already matched nodes – the search can be made slightly faster, too. Other optimizations are suggested in the "outlook" section of this document.

3.2 Calculating the Next Moves

In each step, the algorithm can either

1. match the current node from the first document with one unmatched similar node from the second document or
2. drop the current node. This will be represented by a dummy mapping $\sigma(a) = \perp$.

Each of these operations will be assigned a specific cost estimation in the following section.

In order to consider all matching possibilities, we have to test each node from the first document once against each similar, unmatched node in the second. This can be achieved by traversing the document in an arbitrary order, for example in document order. The order influences the search process, and might allow for significant optimizations, but this has not been investigated. If there are no more nodes left in the first document we have found a solution (i.e. a state is complete iff no unprocessed nodes are left in the first document); the A^*-Search together with the optimistic and admissible cost function guarantees, that the first complete solution found is optimal.

Algorithm: Get possible moves
current := *current state information object*
current.position := *current traversal position in first document*
current.matchedNodes := *list of already matched nodes*
results := *empty set of state information objects*
/ now try to find possible matches for the current node */*
foreach node2 **in** *get similar nodes for* self.position
    if node2 *not found in* current.matchedNodes
        newState := *copy of current*
        advance newState.position
        append mapping (current.position, node2) *to* newState.matchedNodes
        insert newState *into* results
/ consider dropping the current node */*
newState := *copy of* current
advance newState.position
append mapping (current.position, *unmapped marker* $\perp$) *to* newState.matchedNodes
insert newState *into* results

3.3 Simple Cost Function

Let $D_1 = (V, E_1)$ and $D_2 = (W, E_2)$ be the two documents to process.

This cost function, like all presented cost functions, is designed to work with a retainment quality function derived from a measure $\mu : E_1 \cup E_2 \to \mathbb{R}$ on the edges of the documents. While this cost function does not require invariance, the improved cost functions will.

The simplest cost function using retained relations is obtained by just checking which relations $a_1 \to b_1 \in E_1$ in the first and $a_2 \to b_2 \in E_2$ in the second document are not retained and assigning a cost increment of $\mu(a_1 \to b_1)$ (respective $\mu(a_2 \to b_2)$) to them:

For all relations $a_1 \to b_1 \in E_1$ (and similar for all $a_2 \to b_2 \in E_2$), check if $\sigma(a_1) = \perp$ – i.e. the node a_1 is to be dropped –, $\sigma(b_2) = \perp$ or $\sigma(a_1) \to \sigma(b_1) \notin E_2$ then assign costs of $\mu(a_1 \to b_1)$, since the relation $a_1 \to b_2$ cannot be retained. If either $a_1$or b_1 is not yet decided or the relation is actually retained ($\sigma(a_1) \to \sigma(b_1) \in E_2$), do not assign costs.

At the end, when (by the iteration process we use as explained in the previous section) there are no unmapped nodes left in the first document, all yet unmapped nodes in the second document are assumed to be unmapped, i.e. all relations with them are considered lost and are assigned costs.

The cost function can be written as

$$\sum_{a_1 \to b_1 \in E_1} \begin{cases} 0 & \text{if } \exists_{a_2 \to b_2 \in E_2} \sigma(a_1) = a_2 \wedge \sigma(b_1) = b_2 \\ \mu(a_1 \to b_1) & \text{if } \sigma(a_1) = \perp \vee \sigma(b_1) = \perp \\ 0 & \text{if } a_1 \text{ or } b_1 \text{ not yet processed} \\ \mu(a_1 \to b_1) & \text{otherwise} \end{cases} +$$

$$+ \sum_{a_2 \to b_2 \in E_2} \begin{cases} 0 & \text{if } \exists_{a_1 \to b_1 \in E_1} \sigma(a_1) = a_2 \wedge \sigma(b_1) = b_2 \\ \mu(a_2 \to b_2) & \text{if } \nexists_{a_1 \in V} \sigma(a_1) = a_2 \vee \nexists_{b_1 \in V} \sigma(b_1) = b_2 \\ 0 & \text{if } a_2 \text{ or } b_2 \text{ not yet processed} \\ \mu(a_2 \to b_2) & \text{otherwise} \end{cases}$$

This function is obviously optimistic – when additional nodes are processed they can only cause additional costs – and admissible: the costs equal the number of relations that were present in the documents and that could not be retained.

Unfortunately, there is a negative existential quantifier in the second sum, which can only be evaluated when all nodes in V are processed. Therefore, A^*-Search with this heuristic is not very efficient. In fact, this cost function has some very bad characteristics:

1. the costs at the beginning show little difference between different options (i.e. the first step has always costs of 0 if it does not drop the node, since no other nodes are yet mapped, and therefore no relation is lost for sure)

2. many costs generated are "obvious" costs that all solutions will have – even when one document is identical to a subtree of the second document it will have costs

3. the last step often contains a big amount of the costs, due to the negative existential quantifier

4. the behavior is asymmetric, especially for documents with different numbers of nodes

For A^*-Search to be fast it is essential that good solutions can be reached at low costs, whereas worse solutions are detected early by higher costs than the good solutions. When a "perfect" solution – e.g. where D_1 is identical to a subtree of D_2 – has already costs n, then this means that all nodes in the search tree with a cost of less than n have to be expanded. Therefore, the "perfect" match – if present – should have costs of 0; costs of inferior matches should grow early in the search process (while of course remaining optimistic).

There are some tricks to lessen some of these problems – for example by choosing the document with more relations or more nodes as first document or by switching documents each step – but they do only have limited effect.

To effectively counter these problems, the cost function needs to be improved by assigning costs earlier wherever possible. At the same time eliminating "obvious" costs to get the "perfect" match down to a cost level of 0. The cost estimation function explained in the next paragraphs significantly improves over this first cost function and achieves this last goal.

The key point are "obvious costs". For example, if there is a node in D_2 that no node in D_1 is similar to, none of its relations will ever be retained. Eliminating such costs will improve the process a lot, but this calculation can be taken a step further:

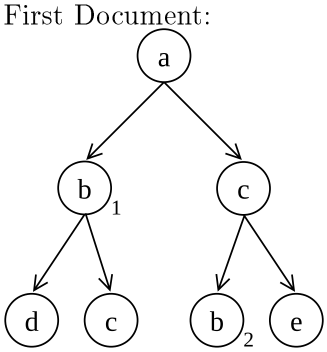
Instead of looking at which nodes are unique in each document, we again look at the relations instead. And we do not only consider the case, where a relation type does only occur in one document, but we look at the difference between the occurrence in both documents.

3.4 Improved Cost Estimation Function

Motivation: Given two nodes $a, b \in V$ and a relation $a \to b \in E$ in the first document, we can obtain the relation equivalence class $\overline{a \to b} := \overline{a} \to \overline{b}$ (this definition is well-defined, the proof is trivial). If the retainment quality function is *invariant* and derived from a measure function on the edges of the documents, all edges in this class will have the same weight. This process explained here can be extended to allow non-invariant retainment quality functions, but this would make the concept harder to understand at little benefit.

Using this we can count the number $r_{a\to b}$ of relations in the first document that relate nodes of equivalence class $\overline{a}$ with nodes of equivalence class $\overline{b}$ (see Figure 12). Similar we can count the number $r'_{a\to b}$ of relations in the second document, that relate nodes of equivalence class $\pi(\overline{a})$ (the nodes similar to nodes of class $\overline{a}$) with nodes of equivalence class $\pi(\overline{b})$ as shown in Figure 13 (for this to be well-defined, a node equivalence relation is needed). We can safely set $\overline{a \to b} = \overline{\pi(a) \to \pi(b)}$.

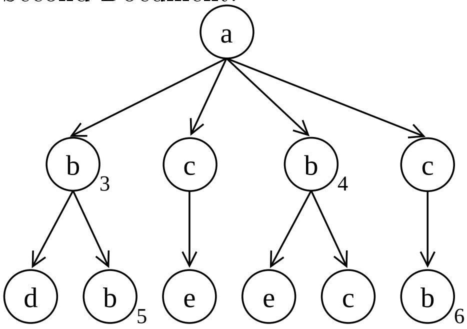
First Document:



Relation class sizes:
Using the child relation the calculated class sizes are: $r_{a\to b} = 1$, $r_{a\to c} = 1$, $r_{b\to c} = 1$, $r_{b\to d} = 1$, $r_{c\to b} = 1$, $r_{c\to e} = 1$
Using the descendant relation, they are: $r_{a\to b} = 2$, $r_{a\to c} = 2$, $r_{a\to d} = 1$, $r_{a\to e} = 1$, $r_{b\to c} = 1$, $r_{b\to d} = 1$, $r_{c\to b} = 1$, $r_{c\to e} = 1$

Figure 12: Credit Calculation: Calculating class sizes in the first document

Second Document:



Relation class sizes:
child relation: $r'_{a\to b} = 2$, $r'_{a\to c} = 2$, $r'_{b\to b} = 1$, $r'_{b\to c} = 1$, $r'_{b\to d} = 1$, $r'_{b\to e} = 1$, $r'_{c\to b} = 1$, $r'_{c\to e} = 1$
descendant relation: $r'_{a\to b} = 4$, $r'_{a\to c} = 3$, $r'_{a\to d} = 1$, $r'_{a\to e} = 2$, $r'_{b\to b} = 1$, $r'_{b\to c} = 1$, $r'_{b\to d} = 1$, $r'_{b\to e} = 1$, $r'_{c\to b} = 1$, $r'_{c\to e} = 1$

Figure 13: Credit Calculation: Calculating class sizes in the second document

The numbers $r_{a\to b}$, $r'_{a\to b}$ are constant for fixed documents (i.e. independent of the other document or the difference search process).

The maximum number of relations in a relation class $\overline{a \to b}$ that could be maintained – ignoring the actual structure of the documents – obviously is $\min(r_{a\to b}, r'_{a\to b}) =: r^{\sim}_{a\to b}$.

This implies that we will have to "give up" $r_{a\to b} - r^{\sim}_{a\to b} =: c^{+}_{a\to b}$ relations of this class in the first document, and $r'_{a\to b} - r^{\sim}_{a\to b} =: c^{-}_{a\to b}$ relations in the second document. Either $c^{+}_{a\to b}$ or $c^{-}_{a\to b}$ is 0 by this estimation.

Credits: Therefore, the difference $c_{a\to b} := r_{a\to b} - r'_{a\to b} = c^{+}_{a\to b} - c^{-}_{a\to b}$ contains all necessary information and is called the initial *credits value* for $\overline{a \to b}$. A positive credit value means we will have to give up more relations in the first document, a negative means more in the second. This value has some interesting properties:

For a given node set correspondence σ, we can calculate $r^{\sigma}_{a\to b}$, the number of relations in class $\overline{a \to b}$ actually retained when matching the documents as given by σ. This number is by definition the same for both documents – we can only retain a relation in one document if we have one in the other, too. This yields the equations $r^{\sigma}_{a\to b} = r_{a\to b} - l^{\sigma}_{a\to b}$ and $r^{\sigma}_{a\to b} = r'_{a\to b} - l^{\sigma\prime}_{a\to b}$ with $l^{\sigma}_{a\to b}$ and $l^{\sigma\prime}_{a\to b}$ being the count of "lost" (i.e. non-retained or "dropped") relations. From this we obtain the equation $r_{a\to b} - r'_{a\to b} = l^{\sigma}_{a\to b} - l^{\sigma\prime}_{a\to b} = c_{a\to b}$, showing that we'll always lose additional relations in both documents, never only in one. Therefore it is sufficient to track losses only in one document.

When taking into account the maximum number of retainable relations calculated earlier, we can also calculate $r^{\sigma}_{a\to b} = r^{\sim}_{a\to b} - k^{\sigma}_{a\to b}$ with $k^{\sigma}_{a\to b}$ being only the number of relations "lost additionally". Since we cannot retain more than $r_{a\to b}$ anyway, only the additional losses $k^{\sigma}_{a\to b}$ are interesting and should influence costs.

Example: In Figures 12 and 13, the credit values are when using the child relation: $c_{a\to b} = -1$, $c_{a\to c} = -1$, $c_{b\to b} = -1$, $c_{b\to c} = 0$, $c_{b\to d} = 0$, $c_{b\to e} = -1$, $c_{c\to b} = 0$, $c_{c\to e} = 0$ and when using the descendant relation $c_{a\to b} = -2$, $c_{a\to c} = -1$, $c_{a\to d} = 0$, $c_{a\to e} = -1$, $c_{b\to b} = -1$, $c_{b\to c} = 0$, $c_{b\to d} = 0$, $c_{b\to e} = -1$, $c_{c\to b} = 0$, $c_{c\to e} = 0$.

First improved cost function: By using this credits calculation it is sufficient to track one document completely for cost calculation. This shows that the problems 3 and 4 referenced earlier are resolved – the process is now symmetric and the negative quantifier causing the high amount of costs incomputable until the last step is completed.

By assigning costs only to relations when exceeding the "credits" value of $c^+_{a \to b}$, we improve the situation for problem 2 much: when D_1 is identical to a subtree of D_2, credits will exactly cover all additional relations in D_2, resulting in a cost of 0 for the "perfect" solution of mapping D_1 to the subtree.

The new cost function looks like the following:

$$\sum_{\overline{a},\overline{b} \in \overline{V}} \left(\max \left(0, \left(\sum_{a_1 \to b_1 \in E_1 \cap (\overline{a},\overline{b})} \begin{cases} 0 & \text{if } \exists_{a_2 \to b_2 \in E_2} \sigma(a_1) = a_2 \wedge \sigma(b_1) = b_2 \\ \mu(\overline{a \to b}) & \text{if } \sigma(a_1) = \perp \vee \sigma(b_1) = \perp \\ 0 & \text{if } a_1 \text{ or } b_1 \text{ not yet processed} \\ \mu(a_1 \to b_1) & \text{otherwise} \end{cases} \right) - c^+_{a \to b} \right) \right)$$

It can be shown that this differs from the value given before exactly by the factor of 2 and by the credits sum $\sum_{\overline{a},\overline{b} \in \overline{V}} |c_{a \to b}|$, i.e. independently of the actual mapping σ and a strictly monotone transformation. Therefore, the new cost function is still optimistic and admissible.

This calculation no longer has the negative quantifier which showed negative effects in the original cost function.

Example: Assuming we want to map the b_1 node of the first document (see Figure 12) to a b node in the second document (see Figure 13), we need to calculate costs for matching with different nodes. We'll calculate the costs exemplary for the nodes marked 3, 4, 5 and 6 in Figure 13 in the case that no other nodes have yet been matched.

The node in the first document has three relations in classes $\overline{a \to b}$, $\overline{b \to c}$ and $\overline{b \to d}$.

The node b_3 has three relations in classes $\overline{a \to b}$, $\overline{b \to b}$ and $\overline{b \to d}$, the node b_4 has $\overline{a \to b}$, $\overline{b \to c}$ and $\overline{b \to e}$, node b_5 has $\overline{a \to b}$ (in descendant relation only) and $\overline{b \to b}$, the node b_6 has $\overline{a \to b}$ (in descendant relation only) and $\overline{c \to b}$.

Assuming that we have not yet made other matches, when matching the node from first document with node b_3 (see Figure 14), we lose one relation of class $\overline{b \to c}$ in first document and one of class $\overline{b \to b}$ in second document. We do have a negative credit we can use for $\overline{b \to b}$, but we haven't got a positive credit for $\overline{b \to c}$, so we have a cost of 1 for this move. This can be interpreted as: "when doing this matching, we can no longer retain a $\overline{b \to c}$ relation we expected to retain."

Expecting to be able to match a $\overline{b \to c}$ relation is not incorrect: when choosing the node b_4 (see Figure 15), we can retain a $\overline{b \to c}$ relation. In that case we have $d_{a \to b} = 0$, $d_{b \to c} = 0$, $d_{b \to d} = +1$, $d_{b \to e} = -1$, resulting in a cost of 1, too, since $c_{b \to e} = -1$.

But when choosing the node b_5 or b_6 (for b_6 see Figure 16), we can only retain $\overline{a \to b}$, losing $\overline{b \to c}$, $\overline{b \to d}$ and (unless using descendant relation) $\overline{a \to b}$ in the first document and losing $\overline{c \to b}$ in the second, i.e. $d_{a \to b} = +1$ (or 0), $d_{b \to c} = +1$, $d_{b \to d} = +1$ and $d_{c \to b} = -1$, resulting in a cost of 4 (or 3 for descendant relation).

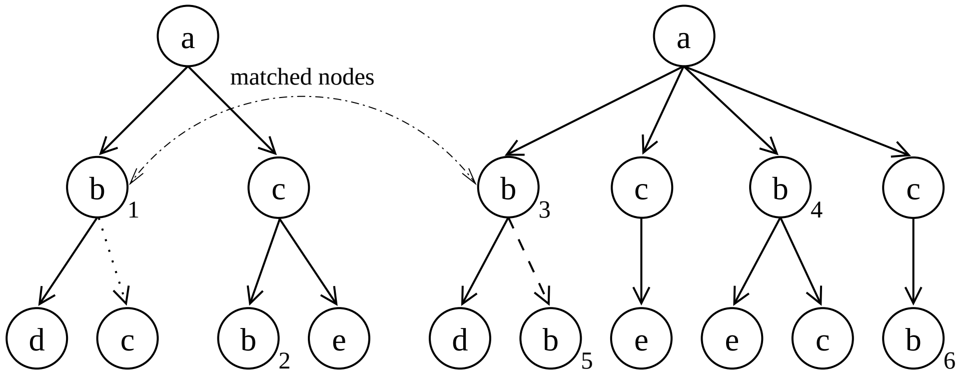


Figure 14: Trying to match b_1 with the b_3.
A dotted line is an expected loss, a dashed line is causing costs.

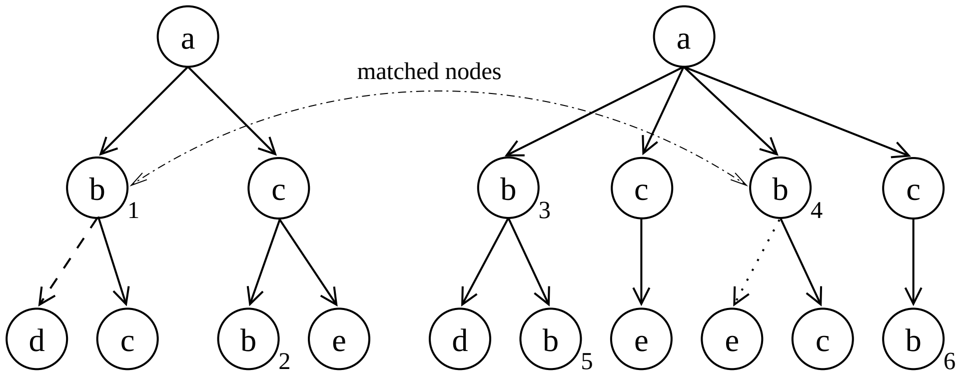


Figure 15: Trying to match b_1 with the node marked b_4.
A dotted line is an expected loss, a dashed line is causing costs.

The A^*-Search algorithm therefore prefers choosing either b_3 or b_4 to match with b_1. Still options b_5 and b_6 are not discarded completely, but will be placed later in the processing queue.

When other nodes have been matched before, for example the d node, costs may be different: matching the parent of the fixed d node will be cheaper than choosing a different b, even when that has a d child, too.

Second improved cost function: But there is an additional improvement that can be made, which will improve problem 1 listed before. Currently, we do not yet assign costs when either a_1 or b_1 is not yet processed. But we can do better if one of them is already processed: If a_1 is related to $d^+_{a_1 \to b}$ unmapped nodes of class $\overline{b}$, $\sigma(a_1) = a_2$ and a_2 is related to $d^-_{a_1 \to b}$ unmapped nodes of $\overline{\pi(b)}$, then at most $\min(d^+_{a_1 \to b}, d^-_{a_1 \to b})$ relations of these can be kept. Therefore, $|d^+_{a_1 \to b} - d^-_{a_1 \to b}|$ relations will be lost and we can – earlier than before – predict additional costs of $\mu(\overline{a \to b}) \cdot |d^+_{a_1 \to b} - d^-_{a_1 \to b}|$ for them. An equation similar to the one above can be constructed for this case. The equation will not be given explicitly here, since the focus

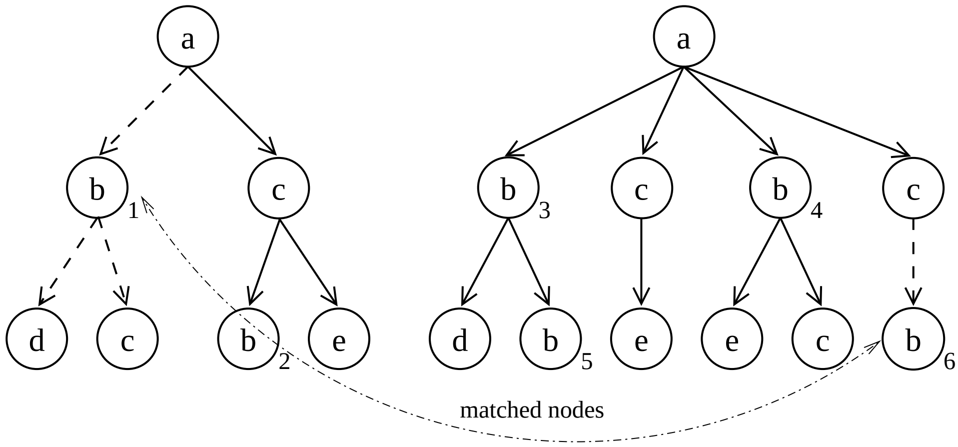


Figure 16: Trying to match b with the node b_6.
A dotted line is an expected loss, a dashed line is causing costs.

lies on incremental calculation.

Incremental calculation: Up to now, the credits value was fixed for the whole process. This will change now – the credits value becomes part of the state information and will be updated with current "predictions". This is better to handle for incremental calculation – especially since A^*-Search will continue processing into different directions.

In each step, a new node a_1 is either mapped to a node a_2 in the second document or dropped.

To calculate the cost increase to the previous step we have to iterate over all nodes b_1 the node a_1 is related to – both as $a_1 \to b_1$ and $b_1 \to a_1$ – and, unless a_1 is being dropped, over all nodes b_2 the node a_2 is related to – both as $a_2 \to b_2$ and $b_2 \to a_2$. For these we have to consider the different cases already listed before in the equations and the new case from the second improvement:

1. $\sigma(a_1) = \perp$

   For each node b_1 that a_1 is related to:

   (a) $\sigma(b_1)$ unmapped or $\sigma(b_1) = b_2 \neq \perp$
   This means we can not retain this $\overline{a \to b}$ relation (and that this is new information), therefore the credits value $c_{a\to b}$ is decreased by 1. If now $c_{a\to b} < 0$, this means we will not be able to retain an additional relation in both documents we had expected to retain, increasing the cost by $2 \cdot \mu(\overline{a}, \overline{b})$. If $c_{a\to b} \geq 0$, this was an expected loss, not yielding costs.

   (b) $\sigma(b_1) = \perp$
   While this still means we cannot retain this $\overline{a \to b}$ relation, this is not new information: it was already handled when b_1 was mapped to $\perp$ as in above case.

2. $\sigma(a_1) = a_2 \neq \perp$

   For each node b_1 that a_1 is related to:

(a) $\sigma(b_1)$ unmapped
This is an $\overline{a \rightarrow b}$ relation we can still retain. Increase the local credits counter $d_{a\rightarrow b}$ by 1.

(b) $\sigma(b_1) = \bot$
While we cannot retain this $\overline{a \rightarrow b}$ relation, this has already been handled when b_1 was mapped to $\bot$.

(c) $\sigma(b_1) = b_2 \neq \bot$

i. $a_2 \rightarrow b_2 \in E_2$ – this is the good case, we have successfully retained a relation.

ii. $a_2 \rightarrow b_2 \notin E_2$ – we have lost the $\overline{a_1 \rightarrow b_1}$ relation, the credits value $c_{a\rightarrow b}$ is increased by 1. If now $c_{a\rightarrow b} > 0$, this means we will not be able to retain an additional relation in both documents we had expected to retain, increasing the cost by $2 \cdot \mu(\overline{a}, \overline{b})$. If $c_{a\rightarrow b} \leq 0$, this was an expected loss, not yielding costs.

For each node b_2 that a_2 is related to:

(a) $\nexists_{b_1} \sigma(b_1) = b_2$ (unmapped)
This is a $\overline{a \rightarrow b}$ relation we can still retain. Decrease the local credits counter $d_{a\rightarrow b}$ by 1.

(b) $\exists_{b_1} \sigma(b_1) = b_2$

i. $a_1 \rightarrow b_1 \in E_2$ – this is the good case, we have actually retained a relation. No costs generated, no further calculation necessary

ii. $a_1 \rightarrow b_1 \notin E_2$ – we have lost the $\overline{a_2 \rightarrow b_2}$ relation, the credits value $c_{a\rightarrow b}$ is decreased by 1. If now $c_{a\rightarrow b} < 0$, this means we will not be able to retain an additional relation in both documents we had expected to retain, increasing the cost by $2 \cdot \mu(\overline{a}, \overline{b})$. If $c_{a\rightarrow b} \geq 0$, this was an expected loss, not yielding costs.

Afterwards, the global credits value $c_{a\rightarrow b}$ is increased or decreased by adding $d_{a\rightarrow b}$, resulting in the new credits value $c'_{a\rightarrow b} = c_{a\rightarrow b} - d_{a\rightarrow b}$. Costs are generated if positive credits are further increased or negative credits are further decreased (i.e. not existent credits are used): if $c'_{a\rightarrow b} > 0$ and $c'_{a\rightarrow b} > c_{a\rightarrow b}$, then costs are $(c'_{a\rightarrow b} - c_{a\rightarrow b}) \cdot \mu(\overline{a}, \overline{b})$ and if $c'_{a\rightarrow b} < 0$ and $c'_{a\rightarrow b} < c_{a\rightarrow b}$ then costs are $(c_{a\rightarrow b} - c'_{a\rightarrow b}) \cdot \mu(\overline{a}, \overline{b})$ resulting in the total formula $(\max(0, \min(0, c_{a\rightarrow b}) - c'_{a\rightarrow b}) + \max(0, \min(0, -c_{a\rightarrow b}) + c'_{a\rightarrow b})) \cdot \mu(\overline{a}, \overline{b})$.

This process needs to be done for both the $a \rightarrow b$ and $b \rightarrow a$ directions.

Algorithm: Calculate step costs for matching node a_1 to a_2
node1 := *node a_1 to match in first document*
node2 := *node a_2 to match in second document*
matchedNodes := *list of previously matched nodes in current context*
creditsMap := *map of equivalence class pairs to credits in current context*
stepCost := 0
localCreditsMap := *map of equivalence class pairs to* 0
foreach downNode1 **in** *nodes* node1 *is related to in "down" direction*
    eqClass := *equivalency class of* node1 *and* downNode1
    if matchedNodes *contains* downNode1

```
        downNode1Match := the node downNode1 is matched to
        if node2 is related to downNode1Match in “down” direction
            /* no costs, retained relation */
        else
            /* lost relation, using global credits, eventually generating costs */
            if credits in creditsMap for class eqClass are not sufficient (≥ 0)
                stepCost := stepCost + 1
            update creditsMap for class eqClass by +1
    else
        /* count relation we still can retain */
        update localCreditsMap for class eqClass by +1
foreach downNode2 in nodes node2 is related to in “down” direction
    eqClass := equivalency class of node2 and downNode2
    if matchedNodes contains downNode2
        downNode2Match := the node downNode2 is matched to
        if node1 is related to downNode2Match in “down” direction
            /* no costs, retained relation */
        else
            /* lost relation, using global credits, eventually generating costs */
            if credits in creditsMap for class eqClass are not sufficient (≤ 0)
                stepCost := stepCost - 1
            update creditsMap for class eqClass by -1
    else
        /* count relation we still can retain */
        update localCreditsMap for class eqClass by -1
/* [ ... same for relations in “up” direction ... ] */
foreach eqClass in classes in localCreditsMap
    creditsNeeded := credits in localCreditsMap for class eqClass
    if creditsNeeded for first document (creditsNeeded ≥ 0)
        if credits in creditsMap for class eqClass are not sufficient
            /* that is (credits + creditsNeeded ≥ 0) */
            stepCost := stepCost + min( creditsNeeded, credits + creditsNeeded )
    if creditsNeeded for second document (creditsNeeded ≤ 0)
        if credits in creditsMap for class eqClass are not sufficient
            /* that is (credits + creditsNeeded ≤ 0) */
            stepCost := stepCost + |max( creditsNeeded, credits + creditsNeeded )|
    update creditsMap for class eqClass by creditsNeeded
```

3.5 Worst-Case Situations

The improved algorithm makes use of differences between nodes – having different labels, different numbers of relations and such.

If these differences are not present, the algorithm can only do an exhaustive search.

An example situation is given in Figure 17, with each document containing 5 nodes with the same label a. While both documents have 4 relations of type $\overline{a \to a}$ each, at most one can be retained.

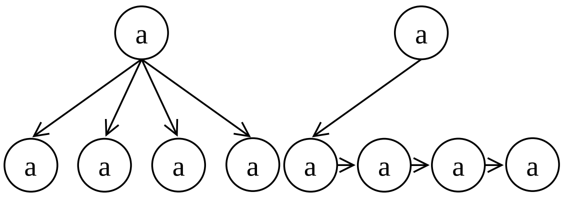


Figure 17: Worst-Case scenario

The resulting search tree is an almost completely expanded graph (except for the last level) of around 500 nodes. And while a human can easily see the best solution, this algorithm (i.e. this cost function) can not. This shows room for extending the cost function.

4 Experimental Results

4.1 First Example – Nesting Inversion:

For the example given in figure 2, the output generated with the "descendant" relation is as follows:

```
<xupdate:modifications xmlns:xupdate="http://www.xmldb.org/xupdate" version="1.0">
<xupdate:variable name="m0" select="/r[1]/a[1]/b[1]"/>
<xupdate:remove select="/r[1]/a[1]/b[1]"/>
<xupdate:insert-after select="/r[1]/a[1]"><xupdate:element name="dummy"/>
</xupdate:insert-after>
<xupdate:variable name="m1" select="/r[1]/a[1]"/>
<xupdate:remove select="/r[1]/a[1]"/>
<xupdate:insert-before select="/r[1]/text()[1]"><xupdate:value-of select="$m0"/>
</xupdate:insert-before>
<xupdate:append select="/r[1]/b[1]"><xupdate:value-of select="$m1"/>
</xupdate:append>
<xupdate:remove select="/r[1]/dummy[1]"/>
</xupdate:modifications>
```

This output can be interpreted as follows:

1. cut the `<b/>` element out of the document into a variable
2. insert a `<dummy/>` node after the `<a/>` element
3. cut the `<a/>` element out of the document into a second variable
4. insert the `<b/>` element cut at step 1 at its place
5. insert the `<a/>` element cut at step 3 into the `<b/>` element
6. remove the `<dummy/>` node inserted at step 2

Therefore, the operation performed actually is exchanging the two nodes. The edit script is not yet fully optimized by the reference implementation: the insertion of the `<dummy/>` node (to avoid text collapsing) would not be necessary, if the `<b/>` node is inserted before the `<a/>` tag is deleted.

4.2 Second Example – Node Movements

The following two example documents with the only difference being a moved `<b/>` tag (with the contents staying in place)

```
<?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>
```

result in the following output by the reference application when run with default parameters (XUpdate output, context defaulted to '`./node() | ./*/*`', that is all children and grand-children (which are regular nodes) are considered relations worth keeping)

```
<?xml version="1.0"?>
<xupdate:modifications xmlns:xupdate="http://www.xmldb.org/xupdate" version="1.0">
<xupdate:variable name="m0" select="/root[1]/sub[1]/node[2]/b[1]/text()[1]"/>
<xupdate:remove select="/root[1]/sub[1]/node[2]/b[1]/text()[1]"/>
<xupdate:variable name="m1" select="/root[1]/sub[1]/node[2]/b[1]"/>
<xupdate:remove select="/root[1]/sub[1]/node[2]/b[1]"/>
<xupdate:append select="/root[1]/sub[1]/node[2]"><xupdate:value-of select="$m0"/>
</xupdate:append>
<xupdate:variable name="m2" select="/root[1]/sub2[1]/node[2]/text()[1]"/>
<xupdate:remove select="/root[1]/sub2[1]/node[2]/text()[1]"/>
<xupdate:append select="/root[1]/sub2[1]/node[2]"><xupdate:value-of select="$m1"/>
</xupdate:append>
<xupdate:append select="/root[1]/sub2[1]/node[2]/b[1]">
<xupdate:value-of select="$m2"/></xupdate:append>
</xupdate:modifications>
```

Or when described in natural language:

1. copy and remove the contents of the `<b/>` element
2. copy and remove the `<b/>` element
3. reinsert the contents removed in step 1
4. copy and remove the contents of the destination `<node/>`
5. insert the `<b/>` tag removed in step 2 at destination `<node/>`
6. insert the old contents of the destination node into the `<b/>` tag

In comparison, the output of Logilab xmldiff is (whitespace reformatted, XUpdate output):

```
<?xml version="1.0"?>
<xupdate:modifications version="1.0" xmlns:xupdate="http://www.xmldb.org/xupdate">
  <xupdate:rename select="/root[1]/sub[1]/node[2]/b[1]" > node </xupdate:rename>
  <xupdate:remove select="/root[1]/sub[1]/node[2]/node[1]" />
  <xupdate:insert-after select="/root[1]/sub[1]/node[1]" >
    <xupdate:element name="node">1</xupdate:element>
  </xupdate:insert-after>
  <xupdate:insert-after select="/root[1]/sub2[1]/node[1]" >
    <xupdate:element name="node">
    </xupdate:element>
  </xupdate:insert-after>
  <xupdate:rename select="/root[1]/sub2[1]/node[3]" > b </xupdate:rename>
  <xupdate:remove select="/root[1]/sub2[1]/b[1]" />
  <xupdate:append select="/root[1]/sub2[1]/node[2]" >
    <xupdate:element name="b">2</xupdate:element>
  </xupdate:append>
  <xupdate:remove select="/root[1]/sub[1]/node[3]" />
</xupdate:modifications>
```

The first thing to be noted here is that there are two redundant statements. The node renamed is removed afterwards. This is due to the XUpdate output by Logilab xmldiff which does not use variable to achieve moves. When run with the Logilab output format, the remove and the following insert are a "move" statement.

Transcribed to natural language – fixing the output problems – this is:

1. rename the `<b/>` element to `<node/>`
2. move the just renamed `<node/>` element a level up
3. remove the `<node/>` the (now renamed) `<b/>` was in
4. rename the other modified `<node/>` element to `<b/>`
5. add a new element `<node/>` below it
6. move the `<b/>` element into the newly generated element

These changes are counterintuitive: instead of removing the `<b/>` tags, they are renamed from and to `<node/>` tags, and a `<node/>` tags is removed and added instead. This probably is a consequence of the algorithm used, which appears to work by bottom-up extending identical subtrees.

4.3 Third Example – Configurability

The following example, consisting of two imaginary documents, shows the benefits of being able to specify which relations are important.

From the same two documents, three different outputs will be generated, all of which make sense and are desirable in certain situations.

```
<?xml version="1.0"?>
<root>
  <part>
    <title>Title text</title>
    <subtitle>First subtitle</subtitle>
    <content>
      <text>
        <par>First text content, first paragraph</par>
        <par>First text content, second paragraph</par>
      </text>
    </content>
  </part>
  <part>
    <title>Second title</title>
    <subtitle>Second subtitle</subtitle>
    <content>
      <text>
        <par>Second text content, first paragraph</par>
        <par>Second text content, second paragraph</par>
      </text>
    </content>
  </part>
</root>
```

```
<?xml version="1.0"?>
<root>
  <part>
    <title>Title text</title>
    <subtitle>First subtitle</subtitle>
    <content>
      <text>
        <par>Second text content, first paragraph</par>
        <par>Second text content, second paragraph</par>
      </text>
    </content>
  </part>
  <part>
    <title>Second title</title>
    <subtitle>Second subtitle</subtitle>
    <content>
      <text>
        <par>First text content, first paragraph</par>
        <par>First text content, second paragraph</par>
      </text>
    </content>
  </part>
</root>
```

The output format used in the following is the “merged” output format of the reference implementation. It uses attributes and additional nodes in the “merged-diff” namespace (using the “`di:`” prefix here) to mark changes in the document. This is usually easier to read than XUpdate statements, and the files are often still valid documents. The output has been slightly reformatted by using additional whitespace for printing.

When using “`./node() | ./*/*`” – direct children of any type and grand children of normal (a default setting that has shown useful), the result is that the text contents are removed and inserted at their new location. You can tell this from the `<di:removed-text>` element wrapped around the old text with the attribute “`di:following=''moved-here''`” to signal the following node – that is the new text – was moved there.

```
<?xml version="1.0"?>
<root xmlns:di="merged-diff">
  <part>
    <title>Title text</title>
    <subtitle>First subtitle</subtitle>
    <content>
      <text>
        <par>
          <di:removed-text di:following="moved-here">First text content, first paragraph</di:removed-text>
          Second text content, first paragraph</par>
        <par>
          <di:removed-text di:following="moved-here">First text content, second paragraph</di:removed-text>
          Second text content, second paragraph</par>
      </text>
    </content>
  </part>
  <part>
    <title>Second title</title>
    <subtitle>Second subtitle</subtitle>
    <content>
      <text>
        <par><di:removed-text di:following="moved-here">Second text content, first paragraph</di:removed-text>
          First text content, first paragraph</par>
        <par><di:removed-text di:following="moved-here">Second text content, second paragraph</di:removed-text>
          First text content, second paragraph</par>
      </text>
    </content>
  </part>
```

```
</root>
```

When using “descendant::node()” as XPath statement to define relations – any descendant node – the deeper nesting of the text nodes comes into effect. Therefore, instead of replacing the text paragraphs, the title lines are replaced, and the <part/>s are reordered.

The output is a bit trickier to read this time, since we have nested operations. Since the first <part> is “di:n=''moved-away''”, only removals are written inside – the title texts being moved away. In the second <part> element, which is not moved, the title texts are replaced as in the previous example. Then at the new location of the first <part> element, marked by “di:n=''moved-here''”, only insertions are printed (since the removals were done at the old location). Therefore, the new text is only marked by the attribute “di:content=''moved-here''” on the containing title elements.

```
<?xml version="1.0"?>
<root xmlns:di="merged-diff">
  <part di:n="moved-away">
    <title><di:removed-text>Title text</di:removed-text></title>
    <subtitle><di:removed-text>First subtitle</di:removed-text></subtitle>
    <content>
      <text>
        <par>First text content, first paragraph</par>
        <par>First text content, second paragraph</par>
      </text>
    </content>
  </part>
  <part>
    <title><di:removed-text di:following="moved-here">Second title</di:removed-text>
      Title text</title>
    <subtitle><di:removed-text di:following="moved-here">Second subtitle</di:removed-text>
      First subtitle</subtitle>
    <content>
      <text>
        <par>Second text content, first paragraph</par>
        <par>Second text content, second paragraph</par>
      </text>
    </content>
  </part>
  <part di:n="moved-here">
    <title di:content="moved-here">Second title</title>
    <subtitle di:content="moved-here">Second subtitle</subtitle>
    <content>
      <text>
        <par>First text content, first paragraph</par>
        <par>First text content, second paragraph</par>
      </text>
    </content>
  </part>
</root>
```

A third variant explicitly detaches the <text/>-<par/> relation using the XPath expression “self::*[not(name()='text')]/node()”, therefore not only the text contents are moved, but the whole paragraphs containing them, resulting in the output below.

This time, the output is very easy to read – nodes with the attribute “di:n=''moved-away''” only exist in the first version, nodes with “di:n=''moved-here''” only in the new version.

```
<?xml version="1.0"?>
```

```
<root xmlns:di="merged-diff">
  <part>
    <title>Title text</title>
    <subtitle>First subtitle</subtitle>
    <content>
      <text>
        <par di:n="moved-away">First text content, first paragraph</par>
        <par di:n="moved-away">First text content, second paragraph</par>
        <par di:n="moved-here">Second text content, first paragraph</par>
        <par di:n="moved-here">Second text content, second paragraph</par>
      </text>
    </content>
  </part>
  <part>
    <title>Second title</title>
    <subtitle>Second subtitle</subtitle>
    <content>
      <text>
        <par di:n="moved-away">Second text content, first paragraph</par>
        <par di:n="moved-away">Second text content, second paragraph</par>
        <par di:n="moved-here">First text content, first paragraph</par>
        <par di:n="moved-here">First text content, second paragraph</par>
      </text>
    </content>
  </part>
</root>
```

By assigning weights to the relations – for example a high weight to the `<par/>`-text relation, to keep these together – instead of having to un-select other relations as shown in the previous example.

For reference, the following four "replace text content by" statements is the output as generated by Logilab XmlDiff (in the native Logilab XmlDiff output format):

```
[update, /root[1]/part[1]/content[1]/text[1]/par[1]/text()[1], Second text content, first paragraph]
[update, /root[1]/part[1]/content[1]/text[1]/par[2]/text()[1], Second text content, second paragraph]
[update, /root[1]/part[2]/content[1]/text[1]/par[1]/text()[1], First text content, first paragraph]
[update, /root[1]/part[2]/content[1]/text[1]/par[2]/text()[1], First text content, second paragraph]
```

While this diff is exceptionally short (using the "update" statements, which are not yet supported by the reference implementation used to produce above outputs), it does not contain the information, that the text parts "updated" here have actually been in the file before. For big amounts of text content, this makes a big difference.

4.4 Fourth Example – Queries

A different use of this difference notion is for querying documents. Albeit there was no notion introduced for "must" type relations and there is no support for multiple results, the following example can show the importance for queries of specifying which relations are important.

When we apply the algorithms to the following two documents, one being called the "query" document and the second the "database" document, we can achieve very different mappings by choosing different relations.

```
<?xml version="1.0"?>
<query>
  <data id="5">
    <a/>
    <b/>
    <c/>
    <d/>
    <e/>
  </data>
</query>
```

```
<?xml version="1.0"?>
<database>
  <data id="1">
    <c/>
    <a/>
    <d/>
    <b/>
    <e/>
  </data>
  <data id="2">
    <e/>
    <d/>
    <c/>
    <b/>
    <a/>
  </data>
  <data id="3">
    <c/>
    <d/>
    <e/>
    <a/>
    <b/>
  </data>
  <data id="4">
    <a/>
    <d/>
    <b/>
    <c/>
    <e/>
  </data>
  <data id="5">
    <a/>
    <b/>
    <c/>
    <d/>
  </data>
</database>
```

By default, using only "child" and "descendant" relations, all `<data>` elements (except for the last) are equally good for the algorithm. A typical implementation will then choose the `<data>` element with `id="1"` and rearrange the elements `<a/>`, `<b/>`, `<c/>`, `<d/>` and `<e/>`. This is also what the reference implementation does.

But when we choose the XPath expression "`child::* | following-sibling::*[1]`" – that is child relations and the first following sibling –, the `<data>` element with `id="3"` will come out best.

When choosing the XPath expression "`child::* | following-sibling::*`" – child relations and all following siblings – the `<data>` element with `id="4"` is chosen instead.

By using "`child::* | preceding-sibling::*[1] | following-sibling::*[1]`" – child relations, the directly preceding and directly following sibling – we can make the `<data>` element with `id="2"` come out best: here the algorithm can match the forward ordering of the query document with the backward ordering of the database elements.

Finally, by using the XPath expression "`child::* | parent::*/@id[1]`" – child relations, and the id attribute of the container – we can have the document match to the last `<data>` element, even if it does not contain a `<e/>` node: it has the same `id="5"` as the query document.

This example showed that also for querying XML documents with fuzzy queries – for example without requiring elements to appear in the same order – the result depends onto the relations specified, and that choosing the correct relations is important for giving better results.

5 Outlook

There are different possibilities to increase the speed of the algorithm or enhance its usage possibilities:

5.1 Optimized Traversal

Both the speed of the algorithm and the memory usage depend on the order the nodes are processed. Finding the optimal sequence needs further investigation of good and bad cases for the algorithm. Using the document order, as currently employed in the implementation, is likely to not be optimal.

Strategies might include sorting the nodes by similarity class cardinality, or by the number of relations the node participates in.

The general approach should be to try to keep the number of low-cost choices low at the beginning, to avoid a too broad search.

5.2 Reducing the Number of Equivalence Classes

The memory usage is up to quadratic in the number of equivalence classes. This can be reduced by merging equivalence classes. Depending on the document, such merges may not make a difference to the quality of the cost estimation function.

5.3 Search Tree Pruning

To reduce memory usage, "dead ends" of the search tree should be pruned from the tree. For example, if there is a partial solution already retaining 30 of 50 possible relations, any partial solution with a cost of 20 (that would never be considered again) can be dropped. This frees memory for storing new open ends of the search tree.

5.4 Divide and Conquer Strategies

Currently, the algorithm searches for a path through the whole document. To find a possibly better path it might be necessary to step back much and redo the same search for the next nodes again. A typical case is when processing the next subtree of a document – finding the optimal match for one subtree is already a complex search task; having to step back from the second subtree to try an alternative in the first subtree is "expensive", usually meaning to have to redo the second subtree from scratch (when processing in document order, processing in classes or any other order has similar troubles).

The alternative is to process similarity classes independently (or subtrees, but similarity classes are easier to separate independently on the used relation), then use a second level search, trying to combine the previous results. Basically the same algorithm can be used, just that the steps made are not additions of single node correspondences, but node set correspondences covering a whole class as returned by earlier searches. When using this approach, the algorithm must not stop after having found the first complete solution, rather looking at less optimal solutions for each class might be required to get the best global result.

5.5 Approximative Solutions

A major drawback of the presented algorithm is the amount of memory needed, and the expensive lookups whether a node has already been matched earlier.

When not caring about finding the optimal solution but only an approximation, a possible approach would be to assume in each step that no nodes from a different class have yet been fixed (i.e., only check which "similar" nodes are unmatched to only return valid mappings, but not calculating the real costs but just using the credit information).

This is similar to the previous "divide and conquer" approach, basically taking the best result in each class and combining them to form the final solution without looking at combinations with secondary results in the classes, but eventually applying local optimizations techniques.

This approach can further be optimized for trees and DAGs by using index tables.

5.6 Solution Simplification

Currently, the algorithm will even match two nodes if no relation is retained at all, as long as there is no better choice. There might be cases where single relations are kept, such that the total number is maximal, still the solution is counterintuitive. Such cases can probably not be detected automatically, but the user might want to force the program to behave differently. For example, extra costs might be assigned to nodes that keep none of their relations or only a small number or small percentage of their relations. This can be used to prevent "stray matches".

5.7 Using DTD Information and DTD Learning

DTD ("Document Type Definition", language for describing the format of an XML document) information could be used to optimize the search process. For example to divide nodes with the same label into different classes when they occur in different contexts (that have a different meaning).

Also, automatic learning of DTD descriptions when given a set of example documents is an interesting application.

5.8 Database Applications

Databases need a way to easily narrow down candidates for matching. X-Trees and similar high-dimensional data structures could be used to store the number of relations a node participates in (or a subtree contains) to easily narrow down the amount of data to search in.

On the data indexing side, the A^*-Search cost function used here makes use of a value to estimate costs that provides an ordering between elements that could show useful in databases to partition the data space. On the query side, the example given in 4.4 consists of a query document and a database and shows how the result can depend on the used relations.

6 Appendix: Implementation Notes

The reference implementation of the algorithm was done in C++, using operator overloading to achieve an easier-to-read source and templates to achieve type safety in most situations. It is far from complete and still has bugs.

XML processing is done using the DOM interface of the libxml C library. The C++ API of this library was not used due to its limited preinstalled availability on the test systems and the little benefit to be gained. XML namespace support is rudimentary.

A few optimization techniques have been employed to speed up the program and reduce memory usage – still, the application is not suited for larger data sets. C++ features such as operator overloading are used to hide these optimizations. Further optimizations that require bigger changes are suggested in the next section.

The XUpdate output generation has shown to be more difficult than the core algorithm itself; sometimes it is necessary to insert "dummy" nodes and remove them later to avoid text node collapsing by the DOM model. Earlier test output formats are also included.

Not all output writers are complete, especially the output of attribute changes has not been implemented in each, since they require special handling at the output level but they are not handled differently by the algorithm.

Only label equality is supported, and there is currently no support for retainment quality functions, all relations have a fixed weight of 1.

Here is the call reference for the application:

```
Usage:
  xmldiff [-t trace.dat] [-p xpath] [-w] document1.xml document2.xml
    -t trace.dat        Dump search information for analysis
    -w                  Process whitespace
    -m                  Use 'merged' output format
    -a                  Use 'marked' output format
    -u                  Use 'xupdate' output format
    -f                  Use fast mode (approximative)
    -p xpath            Use a different xpath statement for structure
    -p './/node()'      Use descendant relation
    -p './node()'       Use child relation
```

The "trace" output can be used to visualize the search progress as well as obtaining statistics. The "fast approximative" mode is still very limited.

The source code of the reference implementation will presumably be published on the Debian projects Alioth platform, the URL will presumably be `http://ssddiff.alioth.debian.org/`.

References

[BGea89] Brian Berliner, Dick Grune, and Jeff Polk et al. Concurrent Versions System (CVS). CVS Homepage, `http://cvshome.org/`, April 1989.

[CAH02] Grégory Cobéna, Talel Abdessalem, and Yassine Hinnach. A comparative study for XML change detection. `ftp://ftp.inria.fr/INRIA/Projects/gemo/gemo/GemoReport-221.pdf`, April 2002.

[CAM02] Grégory Cobéna, Serge Abiteboul, and Amelie Marian. Detecting changes in XML documents. In *ICDE*, 2002.

[EHH+88] P. Eggert, M. Haertel, D. Hayes, R. Stallman, and L. Tower. GNU Diffutils. GNU Homepage, `http://www.gnu.org/software/diffutils/`, 1988.

[FMM+02] S. Flesca, G. Manco, E. Masciari, L. Pontieri, and A. Pugliese. Detecting structural similarities between xml documents, 2002.

[IBM99] IBM AlphaWorks. XML Diff and Merge Tool. Homepage `http://www.alphaworks.ibm.com/tech/xmldiffmerge`, January 1999.

[IET04] Extensible Messaging and Presence Protocol (XMPP): Core. IETF RFC, `http://ietf.org/rfc/rfc3920.txt` 3920, The Internet Society (ISOC), October 2004.

[JAX97] Java Technology: XML. Java XML website, `http://java.sun.com/xml/`, 1997.

[Lib03] Davide Libenzi. Libxdiff. Library Homepage `http://www.xmailserver.org/xdiff-lib.html`, 2003.

[Log] Logilab xmldiff. Application `http://www.logilab.org/projects/xmldiff`.

[Mon00] Deltaxml. Application `http://www.deltaxml.com/`, Monsell EDM Ltd, 2000.

[MT01] Nicolas Mailhot and Joachim Tordjman. xdiff, un diff pour xml en java. Documentation `http://perso.enst.fr/~talel/projets/xdiff/`, 2001.

[OAS04] Open Office Specification 1.0. Committee draft 1, `http://www.oasis-open.org/committees/office`, Organization for the Advancement of Structured Information Standards (OASIS), March 2004.

[W3C98] Extensible Markup Language (XML) 1.0. W3C Recommendation, `http://w3.org/XML/`, World Wide Web Consortium (W3C), February 1998.

[W3C00] XHTML™1.0: The Extensible HyperText Markup Language. W3C Recommendation, `http://www.w3.org/TR/xhtml1/`, World Wide Web Consortium (W3C), January 2000.

[WDC03] Y. Wang, D. DeWitt, and J. Cai. X-diff: A fast change detection algorithm for xml documents. Paper, Application `http://www.cs.wisc.edu/~yuanwang/xdiff.html`, 2003.