Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE

Aleš Wojnar
Similarity of XML Data

Katedra softwarového inženýrství
Vedoucí diplomové práce: *RNDr. Irena Mlýnkova, Ph.D.*
Studijní program: *Informatika*

Na tomto místě bych rád poděkoval vedoucí své diplomové práce RNDr. Ireně Mlýnkové, Ph.D., za cenné rady, připomínky a náměty které nezanedbatelným dílem přispěly k dokončení této práce.

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

V Praze dne 17. dubna 2008

Aleš Wojnar

Contents

Název práce: *Podobnost XML dat*

Autor: *Aleš Wojnar*

Katedra: *Katedra softwarového inženýrství*

Vedoucí diplomové práce: *RNDr. Irena Mlýnková, Ph.D.*

e-mail vedoucího: *irena.mlynkova@mff.cuni.cz*

Abstrakt: *Jazyk XML se v dnešní době stává stále důležitějším formátem pro uchování a výměnu dat. Porovnávání podobnosti XML dat hraje zásadní roli v efektivním ukládání, zpracovávání a manipulaci s daty.*

Tato práce se zabývá možnostmi jak zjišťovat podobnost mezi DTD. Napřed je navržena vhodná reprezentace DTD stromů. Následně je navržen také algoritmus, který je založený na editační vzdálenosti stromů. Nakonec se zaměřujeme na různé aspekty podobnosti, jako jsou například strukturální a lingvistické informace, a snažíme se je zahrnout do naší metody.

Klíčová slova: *XML, XML schémata, DTD, porovnávání podobnosti DTD, editační vzdálenost stromů*

Title: *Similarity of XML Data*

Author: *Aleš Wojnar*

Department: *Department of Software Engineering*

Supervisor: *RNDr. Irena Mlýnková, Ph.D.*

Supervisor's e-mail address: *irena.mlynkova@mff.cuni.cz*

Abstract: *Currently, XML is still more and more important format for storing and exchanging data. Evaluation of similarity of XML data plays a crucial role in efficient storing, processing and manipulating data.*

This work deals with possibility to evaluate similarity of DTDs. Firstly, suitable DTD tree representation is designed. Next, the algorithm based on tree edit distance technique is proposed. Finally, we are focusing on various aspects of similarity, such as, e.g., structural and linguistic information, and integrate them into our method.

Keywords: *XML, XML schema, DTD, evaluating DTDs similarity, tree edit distance*

Chapter 1

Introduction

Exchanging of a wide variety of data plays an increasingly important role on the Web. Nowadays, it is not convenient to exchange data in a format that needs a special software to handle them.

In consequence of this fact, in recent years, the eXtensible Markup Language (XML) [BPSM00b] has gained increasing relevance as a standard for data representation and manipulation. Currently, XML is recommended by the World Wide Web Consortium and a support of XML can be found in various applications such as database systems, programming languages, or even in word processors.

However, for sharing data between two subjects it is necessary to use the same structure of documents. For this purpose various XML schemes were proposed for description of a type of XML documents. The Document Type Definition (DTD) [BPSM00a] language is one of standards expressing XML schemes. XML Schema [Fal01] is other very popular, more expressive, XML schema language.

Increasing numbers of data available on the Web invoke new tasks, such as e.g., document validation, query processing, data transformation, storage strategies based on clustering, data integration, etc. Evaluation of similarity of XML documents or XML schemes plays a crucial role in all of these fields, especially for the purpose of optimization For example, a similarity measure can be exploited for grouping together data containing the same type of objects or for integration of different schemes describing the same kind of information.

1.1 Aims of this Work

The first aim of this work is an analysis of existing approaches to XML similarity. The core of the work is a proposal of an efficient method for similarity evaluation among XML schemes. Our aim is to focus on various aspects of similarity evaluation, such as, e.g., linguistic and structural in-

formation, and integrate them into our method. The last part of the work is an experimental implementation of the proposed algorithms.

This work is based on so-called edit distance method which are used for computing the distance between two structures, i.e. strings or trees. DTD is chosen as a language for definition of schemes of XML documents especially for its popularity and simplicity.

In the implementation, we exploited some existing solutions for side components of the algorithm, e.g. the library for searching the thesaurus.

Finally, we get experimental results of our implementation.

1.2 Contents of the Work

In the first chapter, the motivation of the work is described and aims, that we want to realize, are determined.

In the second chapter, a brief summary of basic technology used in the work is described. Selected basic terms are defined in this chapter as well.

Categorization and analysis of related works is described in the third chapter.

The fourth chapter focuses on the proposal and a detailed description of our own algorithms and discussion of their advantages and disadvantages.

In the fifth chapter we describe used technologies and architecture of our implementation. Minor restriction of our implementation is mentioned at the end of the chapter.

Experimental results of various tests with our implementation are described in the sixth chapter.

And finally, the seventh chapter evaluates the proposed method. Possibilities for future extension of the work are mentioned as well.

Chapter 2

Technology and Definition

In this chapter, basic technologies used in the work are described - languages XML and DTD and edit distance technique which is used for comparing similarity between two structures.

2.1 XML – eXtensible Markup Language

The Extensible Markup Language (XML) [BPSM00b] is a markup language for representation of structured data. It is standardized and recommended by the World Wide Web Consortium (W3C). XML is a subset of the Standard Generalized Markup Language (SGML). In contrast to HTML, that is other popular subset of SGML, XML does not have fixed specified set of tags.

XML documents are composed of markup and content. There are several kinds of markup that can occur in an XML document: elements, entities, comments, processing instructions, marked sections, and document type declarations.

Elements

Elements are the most common form of markup. Most elements identify the nature of the content they surround. However, some elements may be empty, in which case they have no content. An element begins with a start-tag, `<element>`, and ends with an end-tag, `</element>`, except an empty element, which can have only one tag, `<element/>`. Elements may be nested but must not overlap. Each non-root element must be completely contained in another element.

```
<USER>
<FirstName>James</FirstName>
```

```
   <Username>bart</Username>
</USER>
```

Attributes

Attributes are name-value pairs used to describe XML elements, or to provide additional information about elements. Attributes are always contained within the start tag of an element after its name. All attribute values must be quoted.

```
   <USER ID="21">
   <FirstName>James</FirstName>
   <Username>bart</Username>
</USER>
```

Entities

Some characters cannot be easily entered on the keyboard. In order to insert these characters into a document, entities are used to represent these special characters. Entities are also used to refer to often repeated text. An entity reference is a placeholder that represents the entity. It consists of the entity's name preceded by an ampersand "&" and followed by a semicolon ";". Each entity must have a unique name. For instance, "&ls;" represents left angle bracket "<".

Comments

XML comments start with <!– and end with –>. Two dashes – may not appear anywhere in the text of the comment. Comments may be placed between markup anywhere in a document. They are not part of the textual content of an XML document.

Processing Instructions

Processing instructions provides information to an application. Like comments, they are not of the textual content of an XML document. Processing instructions have the form: <?`identifier data`?>

CDATA Sections

A CDATA section is suitable if we want to insert a text with special characters, such as e.g., "<" or "&". Inside section, `<![CDATA[` *text* `]]>`, all special characters are ignored.

Well-formed Document

A well-formed document conforms to rules of XML syntax. A document that is not well-formed is not considered to be XML. We say that a document is well-formed if:

- all not empty elements have a closing tag,
- opening and closing tags are written with the same case (XML tags are case sensitive),
- all elements are properly nested,
- document have a root tag,
- all attribute values are quoted.

2.2 DTD – Document Type Definition

Document Type Definition is a SGML and XML schema language. The DTD describes a type of a XML document by defining the constraints on the structure of an XML document. It declares the allowable set of elements within the document. It also declares children element types, and their order and number, attributes, entities, processing instructions and comments in a document.

Associating DTDs with Documents

A DTD is associated with an XML document via a Document Type Declaration.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
```

The declaration can be internal or it can reference an external file. The internal document type declaration must be placed between the XML declaration and the root element. The keyword DOCTYPE must be followed by the name of the root element in the XML document. External declaration are useful for creating a common DTD that can be shared between multiple documents.

Element Type Declaration

Element type declarations specifies the rules for the type and number of elements that may appear in an XML document, what elements may appear inside each other, and what order they must appear in. An element type

cannot be declared more than once. Following contents of an element type is allowable:

- EMPTY - refers to tags that are empty
- ANY - refers to any valid content
- children element types - refers to any number of element types within another element type
- mixed content - refers to a combination of (#PCDATA) and children elements. PCDATA represents text that is not markup

```
<!ELEMENT element0 (element1, element2)>
<!ELEMENT element1 (element3)>
<!ELEMENT element2 (#PCDATA)>
<!ELEMENT element3 (#PCDATA)>
```

Attribute Declaration

Attributes are additional information associated with an element type. Attributes are declared via the keyword ATTLIST. The ATTLIST declarations identify which element types may have attributes, what type of attributes they may be, and what the default value of the attributes are. There are three types of attributes:

- CDATA - represents text that is not markup
- Tokenized attribute type:
  - ID is a unique identifier of the attribute.
  - IDREF is used to establish connections between elements. The IDREF value of the attribute must refer to an ID value.
  - ENTITY are used to reference data that act as an abbreviation or can be found at an external location.
- Enumerated attribute types allow you to make a choice between different attribute values.

```
<!ELEMENT image EMPTY>
<!ATTLIST image height CDATA #REQUIRED>
```

Constraints

All attributes have one of the following constraints:

- #REQUIRED - The attribute must always be included

- #IMPLIED - The attribute does not have to be included.
- #FIXED "Value" - The attribute must always have the default value that is specified

Entities

Entities reference data that act as an abbreviation or can be found at an external location.

Notations

Notations are used to identify the format of unparsed entities (non-XML data), elements with a notation attribute, or specific processing instructions.

2.3 Edit Distance Algorithms

Edit distance algorithms were originally used for comparing similarity between two strings, [Ham50, Lev66]. They are based on the idea to find the cheapest sequence of edit operations that can transform one string into another.

Edit operations can be defined variously. For example, [Ham50] uses only one operation - *substitution* of a single character, therefore this algorithm can be used only for strings of the same length. Currently used algorithms are usually based on three edit operations defined in *Levenshtein distance* algorithm [Lev66]. It uses one-step operations - *insertion*, *deletion*, and *substitution* of a single character. A non-negative constant cost is associated with each operation. In the simplest version all the operations cost one unit except for substitution of identical characters, in which case the cost is zero.

Levenshtein algorithm, based on dynamic programming, is depicted in Figure 2.1. At the beginning (line 4) a matrix for storing computed costs of edit operations is defined. At lines 5-7 costs of insertions characters of string B are initialized, similarly costs of deletions characters of string A are initialized at lines 8-10. At lines 11-18 the optimal operation for each pair of characters is found and previously computed values are incremented by the cost of this optimum operation. Finally, the minimum distance between strings A and B is stored in to position $dist[M][N]$.

2.3.1 Tree Edit Distance

Edit distance technique is also used for finding similarity between two trees. Most algorithms in this category are direct descendants of the Levenshtein

```
   Input: stringA, stringB
   Output: Edit distance between string_A and string_B
 1 begin
 2 |  M = Length(A);
 3 |  N = Length(B);
 4 |  int[][] dist = new int[0..M][0..N];
 5 |  for j=0 to N do
 6 |  |  dist[0][j] = j; //generally: dist[0][j] = Cost_Insert(B[j])
 7 |  end
 8 |  for i=0 to M do
 9 |  |  dist[i][0] = i; //generally: dist[i][0] = Cost_Delete(A[i])
10 |  end
11 |  for i=1 to M do
12 |  |  for j=1 to N do
13 |  |  |  dist[i][j] = Min(
14 |  |  |  dist[i-1][j-1] + Cost_Substitution(A[i], B[j]), //substitution
15 |  |  |  dist[i][j-1] + 1, //insertion
16 |  |  |  dist[i-1][j] + 1); //deletion
17 |  |  end
18 |  end
19 |  return distance[M][N];
20 end
```

Figure 2.1: Levenshtein distance algorithm

algorithm, e.g. [SZ97], [NJ]. All of them use three basic edit operations *insertion*, *deletion*, and *substitution*. In contrast to string edit distance, these operations are applied on a (single) node of a tree instead of a character in a string.

[NJ] defines two additional edit operations that allow transforming of more complex structures of a tree. This algorithm is described in the section 3.1.1. In our work we use the same five operations therefore they should be described formally. However, at first, some basic terms have to be defined.

Definition 2.3.1. ***[Ordered Tree]*** *An ordered tree is a rooted tree in which the children of each node are ordered. If a node x has k children then these children are uniquely identified, left to right, as $x_1, x_2, \ldots, x_k$.*

Definition 2.3.2. ***[First-Level Subtree]*** *Given an ordered tree T with a root node r of degree k, the first-level subtrees, $T_1, T_2, \ldots, T_k$ of T are the subtrees rooted at $r_1, r_2, \ldots, r_k$.*

For a given tree T with a root node r of degree m and first-level subtrees $T_1, T_2, \ldots, T_m$, the tree transformation operations are defined formally as follows:

Definition 2.3.3. ***[Substitution]*** *$Substitution_T(r_{new})$ is a node substitution operation applied to T that yields the tree T' with root node r_{new} and first-level subtrees $T_1, ..., T_m$.*

Definition 2.3.4. ***[Insertion]*** *Given a node x with degree 0, $Insert_T(x, i)$ is a node insertion operation applied to T at i that yields the new tree T' with root node r and first-level subtrees $T_1, ..., T_i$, x, $T_{i+1}, ..., T_m$.*

Definition 2.3.5. ***[Deletetion]*** *If the first-level subtree T_i is a leaf node, $Delete_T(T_i)$ is a delete node operation applied to T at i that yields the tree T' with root node r and first-level subtrees $T_1, ..., T_{i-1}, T_{i+1}, ..., T_m$.*

Definition 2.3.6. ***[Insertion Tree]*** *Given a tree A, $InsertTree_T(A, i)$ is an insert tree operation applied to T at i that yields the tree T' with root node r and first-level subtrees $T_1, ..., T_i$, $A, T_{i+1}, ..., T_m$.*

Definition 2.3.7. ***[Deletion Tree]*** *$DeleteTree_T(T_i)$ is a delete tree operation applied to T at i that yields the tree T' with root node r and first-level subtrees $T_1, ..., T_{i-1}, T_{i+1}, ..., T_m$.*

Chapter 3

Related Works

Measuring similarity of XML data is a very general issue and we can use several classifications of it. Probably the most frequently used classification is based on the level on which the data are measured. Similarity of XML data can be measured among XML documents, XML schemes, or between the two groups.

The task of measuring similarity can be also classified according to the kind of its application or according to the approach that is chosen for solving the problem. For example, some works are focusing on structural similarity of data whereas others measure similarity based on meaning of words, i.e. semantics.

Of course various other classifications can be established. We can consider required precision of similarity or we can divide methods according to the amount of information that is taken into account.

3.1 Similarity Among Documents

At present there are many works that focus on measuring similarity among XML documents. A lot of them exploit the fact that XML documents can be represented as labeled trees. Two types of different approaches are discussed in this chapter. The first one measures similarity by transforming one tree to another. In contrast, the second type does not rely on graph representation of XML documents and its approach is completely different.

3.1.1 Tree Edit Distance

Tree edit distance is one of the most popular approaches for computing similarity between two XML documents. The most of works in this area use operations for transformation only on a single node [ZS89].

Beside three basic operations for transforming tree - *Substitute node*, *Insert node* and *Delete node*, two new edit operations are used in [NJ].

Their definitions were formally described in 2.3.1. Since XML documents usually contain structures that occur repeatedly, these repeating structures can be transformed with single operation, (*Insert tree* or *Delete tree*), if they are contained in both trees. In this case the cost for their transforming is lower than if sequence of a single-node operations is applied and it invokes higher similarity of the trees. *Insert tree* or *Delete tree* operations can not be applied if the structures are not contained in both trees.

A non-negative constant cost is associated with each of these five operations. The algorithm works with general costs that can be specified by user. For experimental evaluation of this method the costs were set to 1 for each operation.

Transformation of one tree to another tree can be done by a lot of different sequences of edit operations. Finding the optimal variant of all these sequences is a time-consuming task. Instead of that so-called *allowable* sequences are defined in the proposal.

Definition 3.1.1. ***[Allowable]*** *A sequence of edit operations transforming a source tree A to a destination tree B is allowable if it satisfies the following two conditions:*

1. A tree P may be inserted only if P already occurs in the source tree A. A tree P may be deleted only if P occurs in the destination tree B.

2. A tree that has been inserted via the InsertTree operation may not subsequently have additional nodes inserted. A tree that has been deleted via the DeleteTree operation may not previously have had (children) nodes deleted.

Without the first restriction on allowable sequences of edit operations, the whole source tree could be deleted in the first step and destination tree could be inserted in the second step. The second restriction enables to compute the costs for inserting and deleting subtrees efficiently.

To satisfy the first restriction, i.e. for inserting a subtree, we have to find out if this subtree is *contained in* the source tree A. This is realized with pre-created $ContainedIn$ lists for each node of destination tree B. If a subtree of tree B rooted at node v_B is involved also in A then a pointer on corresponding root node v_A of the subtree of tree A is added to $ContainedIn$ list of node v_B. Hence, we can simply find out if a subtree rooted at any node of destination tree can be inserted via $InsertTree$ operation.

In Figure 3.1 is depicted an example of *containedIn* relationship. Pattern tree P is *contained* in tree A and B but is not *contained* in tree C. That is why $InsertTree$ operation for tree P would be applied only for trees A and B.

Having $ContainedIn$ lists, we can now calculate the cost of inserting every subtree of B or deleting every subtree of A. $Cost_{Graft}(T_i)$ (for inserting) and $Cost_{Prune}(T_i)$ (for deleting) are produced for this purpose.

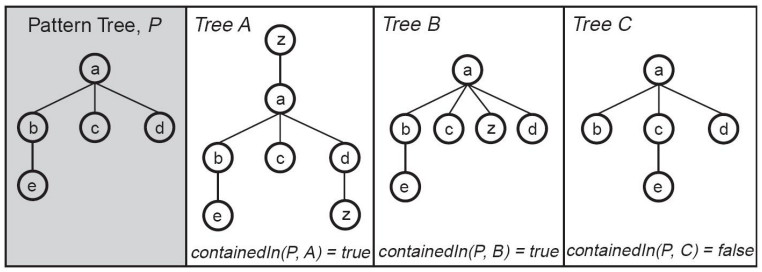

Figure 3.1: Examples of $ContainedIn$ Relationship

$Cost_{Graft}(T_i)$ carries minimum of the costs of all allowable sequences of operations necessary for inserting subtree T_i into tree B. $Cost_{Prune}(T_i)$ is similar for deleting subtree T_i from tree A.

Graft cost is computed by a simple bottom-up procedure. For each node $v \in B$ we consider two possibilities. If subtree P rooted at v is not contained in tree A ($ContainedIn$ list of node v is empty) then $Cost_{Graft}$ for this subtree is calculated recursively as the sum of inserting the single node v and the $Cost_{Graft}$ of each child of v - this sum can be called d_0. If subtree P is contained in tree A then we compute also the $InsertTree$ cost for P - we can call this d_1. Then the $Cost_{Graft}$ for the subtree rooted at v is the minimum of d_0 and d_1. Prune costs are computed similarly for each node in A.

If we have graft cost (resp. prune cost) for each subtree in B (resp. A) then we can determine the minimum cost for transforming tree A into tree B by the following dynamic algorithm depicted in Figure 3.2.

The algorithm dynamically computes costs of different sequences of operations to transform tree A to B. At line 4, the $M + 1 \times N + 1$ matrix for storing computed costs is created, where M is the degree of the root node of tree A and N is the degree of the root node of tree B. At line 5 the first element of the matrix is initialized with the cost of substitution of the root node of A to the root node of B. At lines 7-8, the value from previous step is subsequently increased by the costs of inserting each subtree of the root node of B to the root node of A. Consequently, we have evaluated the cost of inserting all subtrees of B at position $dist[0][N]$. Similarly, evaluated cost of deleting all subtrees of the root node A is stored at position $dist[M][0]$ after executing lines 9-10.

The remaining values of the matrix are computed dynamically at lines 12-18, where the procedure considers also *Substitution* edit operation. In other words, at these lines the procedure decides among inserting or deleting the subtree or substituting nodes of the subtree. At line 15, the procedure is called recursively for evaluating single-node operations. In recursive calling is the main difference between this procedure and the procedure depicted

```
1.  private int editDistance(Tree A, Tree B) {
2.      int M = Degree(A);
3.      int N = Degree(B);
4.      int[][] dist = new int[0..M][0..N];
5.      dist[0][0] = Cost_Relabel(λ(A), λ(B));
6.
7.      for (int j = 1; j ≤ N; j++)
8.          dist[0][j] = dist[0][j-1] + Cost_Graft(B_j);
9.      for (int i = 1; i ≤ M; i++)
10.         dist[i][0] = dist[i-1][0] + Cost_Prune(A_i);
11.
12.     for (int i = 1; i ≤ M; i++)
13.         for (int j = 1; j ≤ N; j++)
14.             dist[i][j] = min{
15.                 dist[i-1][j-1] + editDistance(A_i, B_j),
16.                 dist[i][j-1] + Cost_Graft(B_j),
17.                 dist[i-1][j] + Cost_Prune(A_i)
18.             };
19.     return dist[M][N];
20. } //editDistance
```

Figure 3.2: Edit Distance Algorithm

in Figure 2.1. Finally, the value of the optimum variant of edit operations is returned at line 19.

Due to lines 12-18 the algorithm runs in quadratic time in the combined size of elements of the two documents which are compared.

3.1.2 Time Series Comparing

As mentioned above, most of existing techniques for measuring structural similarity of XML documents concern with tree representation of documents. In contrast, the algorithm described in [FMM+02] represents XML documents as time series where each tag occurrence corresponds to an impulse. The degree of similarity between documents is given by analyzing frequencies of the Fourier Transform of such series.

The algorithm has two phases. In the first one, called *document encoding*, the structure of documents is encoded into time series. In the second one, called *similarity measures*, the similarity of such time series is calculated.

In the first phase the tree structure of an XML document is traversed in a depth-first, left-to-right way. During the visit an impulse is produced

when a start tag is found. The impulse is hold until the corresponding end tag is reached.

More precisely, during the the traversing of the tree structure a unique number is assigned to each node (tag) by a function γ. For a set D of XML documents this function, called *direct tag encoding*, associates each start tag el_s with its position in the sequence $tnames(D)$, where $tnames(D)$ is the set of all the distinct tag names appearing in D. End tags el_e can be associated with the symmetric value $\gamma(el_e) = -\gamma(el_s)$.

A *document encoding* is a function $enc(d)$ that associates an XML document $d \in D$ with a time series representing the structure of d. For a set D of XML documents function enc associates each $d \in D$ with a sequence of integers, i.e. $enc(d) = h_0, h_1, ..., h_n$.

A document encoding function can be defined by several ways. In [FMM+02] authors use encoding strategy called *multilevel encoding*. To describe it following terms have to be defined. For a given set D of XML documents,

Definition 3.1.2. ***[nest$_d$(t)]*** *is set of start tags el_s in $d \in D$ occurring before tag t and for which there is no end tag el_e appearing before t.*
$l_t = |nest_d(t)|$ is denoted as nesting level.

Definition 3.1.3. ***[maxdepth(D)]*** *denotes the maximum nesting level of tags appearing in documents in D.*

Multilevel encoding weights each tag t using its nesting level l_t and $maxdepth(D) - l_t$ as an exponent of a fixed factor B, so that elements appearing at higher levels of the document tree have higher weights. B is usually set as $B = |tnames(D)| + 1$. Then, a multilevel encoding of d is a sequence of impulses $[h_0, h_1, ..., h_n]$, where:

$$h_i = \gamma(t_i) \times B^{\text{maxdepth}(D)-l_{t_i}} + \sum_{t_j \in \text{nest}_d(t_i)} \gamma(t_j) \times B^{\text{maxdepth}(D)-l_{t_j}}$$

In the second phase of the algorithm the resulting similarity is calculated from the signals produced in the first phase. However, comparing such two signals can be as difficult as comparing the original documents because comparing documents having different lengths requires to combine resizing and alignment.

Hence, better way is to apply $Discrete\,Fourier\,Transform\,(DFT)$ on the signals h_1 and h_2 and compute the integral of the magnitude difference of their transforms. In order to approximate the integral, the result of DFT is linearly interpolated and a new $\widetilde{DFT}$ is produced, having $M = N_{d_1} + N_{d_2} - 1$ points, where N_{d_i} is the number of tags in document d_i.

Now we get new two signals $\widetilde{h}_1 = \widetilde{DFT}(h_1)$, $\widetilde{h}_2 = \widetilde{DFT}(h_2)$ (with M points) and the distance of the documents is defined as the approximation of the difference of the magnitudes of these new signals:

$$dist(d_1, d_2) = \left(\sum_{k=1}^{M/2} \left(|\widetilde{h}_1(k)| - |\widetilde{h}_2(k)| \right)^2 \right)^{\frac{1}{2}}$$

The main advantage of this complex approach is that it is not so computationally expensive as the most of algorithms based on transforming a document into another. Time complexity is $O(N \, log \, N)$, where N is number of tags in both XML documents. So, it is better than complexity of the algorithm mentioned in the previous section.

3.2 Similarity Among Data and Schema

Measuring similarity between an XML document and an XML schema is another area of investigation. Works interested in this area are usually used for approximate validation of XML documents or for clustering of XML data. However, not so many approaches for solving these problems has benn described yet.

3.2.1 Common, Plus, and Minus Elements

The algorithm proposed in [BGM04] measures similarity between an XML document and a DTD. Both of them are represented as labeled trees. The matching algorithm is based on identification of:

- *common* elements appearing both in the document and in the DTD,
- *plus* elements appearing in the document but not in the DTD, and
- *minus* elements appearing in the DTD but not in the document.

Obviously, the number of *common* elements must be higher than *plus* and *minus* elements to achieve a high degree of similarity.

However, not only the ratio between *common* and *plus* and *minus* elements is relevant. The algorithm takes into account also the structure of the trees, the presence of DTD operators, and the fact that elements at higher *levels* are more relevant than elements at lower *levels*. For this purpose function f_{eval} is defined that evaluates all possible matches between the document and the DTD based on the level of common, plus and minus elements.

3.3 Similarity Among Schemes

Exploitation of similarity among XML schemes is mainly connected with integration of heterogeneous data or with clustering of XML schemes. A huge number of works have been proposed in this area as well.

Schema matchers are often classified according to their approach. There are a lot of *individual* matchers that use a single matching criterion for computing mapping. Recently, automatic matchers that combine individual matchers were proposed. They differ in the method of using individual matchers. *Hybrid* [MBR01, LYHY02] matchers directly integrate several matching approaches to determine the mapping based on multiple matching criteria. *Composite* [DDH01, DR02] matchers use individual matchers for computing separated results and then combine their results.

In this section three different matchers are outlined. Only the first one is described in more detail.

3.3.1 XClust

XClust is a hybrid matcher that integrates several matching approaches. It considers semantics, immediate descendant, and leaf-context similarity of DTD elements. It analyzes element by element in order to identify possible matches among direct subelements, considering the cardinality of the elements (optional, repeatable, or mandatory) and similarity of their tags.

Element Similarity

The first phase of computing similarity of two DTDs is evalution of element similarity. It considers the semantics, structure, and context information of elements.

The similarity of a pair of element nodes $ElementSim(e_1, e_2)$ is defined as the weighted sum of three components:

(1) Semantic Similarity $SemanticSim(e_1, e_2)$

(2) Immediate Descendants Similarity $ImmediateDescSim(e_1, e_2)$

(3) Leaf Context Similarity $LeafContextSim(e_1, e_2)$

The whole algorithm is shown in Figure 3.3.1. Particular procedures are explained in the following text.

(1) Semantic Similarity

The semantic similarity considers similarity between the names, constraints, and path context of two elements. The similarity is computed using several algorithms:

Algorithm: ElementSim
Input: elements e_1,e_2; matching threshold $Threshold$; weights α, β, γ
Output: element similarity
Step 1. Compute recursive nodes similarity
if only one of e_1 and e_2 is recursive nodes
then return 0; //they will not be matched;
else if both e_1 and e_2 are recursive nodes
then return $ElementSim(R-e_1, R-e_2, Threshold)$;
// $R-e_1, R-e_2$ are the corresponding reference nodes.
Step 2. Compute leaf-context similarity (LCSim)
if both e1 and e2 are leaf nodes
then return $SemanticSim(e_1, e_2, Threshold)$;
else if only one of e_1 and e_2 is leaf node
then LCSim = $SemanticSim(e_1, e_2, Threshold)$;
else //Compute leaf-context similarity
LCSim =$LeafContextSim(e_1, e_2, Threshold)$;
Step 3. Compute immediate descendants similarity(IDSim)
IDSim=$ImmediateDescSim(e_1, e_2, Threshold)$;
Step 4. Compute element similarity of e_1 and e_2
return $\alpha \times SemanticSim(e_1, e_2, Threshold) + \beta \times IDSim + \gamma \times LCSim$;

Figure 3.3: Algorithm to Compute Element Similarity

BasicSim - The basic similarity of two elements is defined as a weighted sum of $OntologySim$ and $ConstraintSim$:

$$\begin{aligned} BasicSim(e_1, e_2) &= w_1 * OntologySim(e_1, e_2) \\ &+ w_2 * ConstraintSim(e_1, e_2), \end{aligned}$$

where weights $w_1 + w_2 = 1$ and

- *OntologySim* - A recursive algorithm which determines ontology similarity between two words w_1 and w_2 by comparing w_1 with synonyms of w_2. It exploits procedure $SynSet(w)$ that searches a thesaurus and returns the set of synonyms of a word w. At the beginning of the algorithm $OntologySim$ a set S is initialized as $S = \{w_2\}$ and the *depth* of algorithm is 0. If $w_1 \notin S$, then $S = \bigcup_{w \in S} SynSet(w)$ and $depth+ = 1$, until $w \in S$ or *depth* is higher than $Maxdepth$, where $Maxdepth$ is a threshold to avoid infinite searching of thesaurus. If no synonym is found, then $OntologySim$ is 0, otherwise it is defined as 0.8^{depth}.

- *ConstraintSim* - An algorithm to compute similarity of cardinality constraints of two elements. ConstraintSim is computed from constraint compatibility table depicted in Figure 3.4.

	*	+	?	none
*	1	0.9	0.7	0.7
+	0.9	1	0.7	0.7
?	0.7	0.7	1	0.8
none	0.7	0.7	0.8	1

Figure 3.4: Cardinality Compatibility Table

Path Context Coefficient - An algorithm to determine the degree of similarity of the paths of two elements. For two elements' path contexts (list of elements on the path from one element to another) we compute their similarity by first determining the *BasicSim* between each pair of elements in the contexts. Then the pairs of elements with highest similarity are returned as a list of one-to-one mapping. Finally, resulting *Path context coefficient* (PCC) is obtained by taking the average *BasicSim* from the mapping list.

Let $Root_1$, $Root_2$ are the roots of e_1, e_2 respectively. Semantic similarity now can be defined as:

$$\begin{aligned} SemanticSim(e_1, e_2) &= PCC(e_1, e_1.Root_1, e_2, e_2.Root_2) \\ &\times BasicSim(e_1, e_2), \end{aligned}$$

(2) Immediate Descendants Similarity

$ImmediateDescSim$ is obtained by comparing immediate descendants (attributes and subelements) of an element. For element e_1 with immediate descendants $c_{11}, ..., c_{1n}$, and element e_2 with immediate descendants $c_{21}, ..., c_{2m}$, basic similarity is computed at first between each pair of descendants in the two sets. The pairs of descendants with highest similarity are selected. The resulting $ImmediateDescSim$ of e_1 and e_2 is finally determined taking the average *BasicSim* of their descendants.

(3) Leaf Context Similarity

In contrast to $ImmediateDescSim$, $LeafContextSim$ of elements e_1 and e_2 considers leaf nodes of the subtrees rooted at these elements. Leaf similarity is calculated as follows:

$$\begin{aligned} LeafSim(l_1, e_1, l_2, e_2) &= PCC(l_1, e_1, l_2, e_2) \\ &\times BasicSim(l_1, l_2). \end{aligned}$$

Then, analogous to Immediate Descendants Similarity, the pairs of leaf nodes with highest similarity are returned and the resulting $LeafContextSim$ is determined taking the average $LeafSim$ of leaf nodes.

The resulting element similarity can be obtained as follows:

$$\begin{aligned} ElementSim(e_1, e_2) &= \alpha \times SemanticSim(e_1, e_2) \\ &+ \beta \times LeafContextSim(e_1, e_2) \\ &+ \gamma \times ImmediateDescSim(e_1, e_2), \end{aligned}$$

where $\alpha + \beta + \gamma = 1$ and $(\alpha, \beta, \gamma) \geq 0$.

Having $ElementSim$ for each pair of elements of two DTDs, we can finally evaluate similarity of the DTDs. Analogous to above procedures, the pairs of elements with highest $ElementSim$ are found and the resulting similarity of two DTDs is determined taking the average of $ElementSim$ of found elements.

3.3.2 Cupid

Cupid [MBR01] is other variant of a hybrid matcher. The matcher focuses on computing similarity coefficients, in the [0,1] range, between elements of the two schemas and then deducing a mapping from those coefficients. The algorithm has three phases:

1. *Linguistic matching* - In this phase the algorithm matches schema elements on the basis of their names, data types, domains, etc. Linguistic matching consists of three particular steps:

- $Normalization$ - element names are normalized by *tokenization* (parsing names into tokens based on punctation, case, etc.), *expansion* (identifying abbreviations and acronyms) and *elimination* (discarding preposition, articles, etc.). A thesaurus is used for each of these steps.

- $Categorization$ - elements are clustered into *categories*. This is based on linguistic meaning of element names or on data types.

- $Comparsion$ - tokens of elements (obtained in $Normalization$ phase) are compared using a thesaurus. Tokens are compared on the basis of synonymy and hypernymy relationship. Only elements in the same category (produced in $Categorization$ phase) are compared in order to reduce amount of comparisons.

The result is a linguistic similarity coefficient, $lsim$, between each pair of elements of two schemas.

2. *Structural matching* - The second phase transforms the original schema into a tree and then performs a bottom-up structure matching, resulting in a structural similarity between pairs of element. The result is a context similarity coefficient, $ssim$.

The algorithm of this phase is based on the following observations:

- Two leaves are similar if they are individually (linguistic and data type) similar, and if their ancestors and siblings are similar.
- Two non-leaf elements are similar if they are linguistically similar, and the subtrees rooted at the two elements are similar.
- Two non-leaf elements are structurally similar if their leaf sets are highly similar, even if their immediate descendants are not.

These observations are exploited in the $TreeMatch$ algorithm depicted in Figure 3.5. It is noticeable that the structural similarity is mainly based on a similarity of leaf nodes. The focus on leaves is based on the assumption that most of the information content is represented in leaves and that leaves have less variation between schemes than the internal structure.

```
TreeMatch(SourceTree S, TargetTree T)
  for each s ∈S, t ∈T where s,t are leaves
      set ssim (s,t) = datatype-compatibility(s,t)
  S' = post-order(S), T' = post-order(T)
  for each s in S'
    for each t in T'
      compute ssim(s,t) = structural-similarity(s,t)
      wsim(s,t) = w_struct.ssim(s,t) + (1-w_struct).lsim (s,t)
      if wsim(s,t) > th_high
        increase-struct-similarity(leaves(s),leaves(t),c_inc)
      if wsim(s,t) < th_low
        decrease-struct-similarity(leaves(s),leaves(t),c_dec)
```

Figure 3.5: The TreeMatch algorithm

3. *Mapping* - Weighted sum, $wsim$, of linguistic and structural similarity of pairs of elements is calculated in this phase.

$$wsim = w_{struct} \times ssim + (1 - w_{struct}) \times lsim,$$

where w_{struct} is in the range [0,1]. Mapping is then decided on the base of that sums.

After weighted sums are computed a mapping of elements is created by choosing pairs of schema elements with maximal weighted similarity.

3.3.3 LSD

In contrast to the previous two methods, LSD (Learning Source Description) [DDH01] belongs to matchers based on composite approach. The LSD system semi-automatically creates semantic mappings of schema elements.

As a composite matcher, LSD integrates several individual matchers. Each of them uses a special technique called *machine learning*. That technique enables to match a new data source against a previously determined set of data. The system has two phases. In the first phase, called *training phase*, the matcher is learned on sample data sources where the mapping is given by a user. In the second phase, the rules gained from patterns are applied to a new data sources.

Finally a global matcher is used to merge the lists of results from individual matchers into a combined list of match candidates for each schema element. The rate of efficiency of mapping depends on the amount of examined schemas during the *training phase*.

Chapter 4

Proposed Method

4.1 Method Overview

The method described in this work proposes to exploit the edit distance and adjust it so it can be used to compute similarity of DTDs. The algorithm is based mainly on the work presented in [NJ] which focuses on comparing XML documents. The main contribution of this algorithm is in introducing two special edit operations $Insert - Tree$ and $Delete - Tree$. These operations allow manipulating more complex structures than a single node. This is profitable for XML documents where some structures can be found repeatedly due to cardinality of elements of DTD. For example, in Figure 4.1 two trees representing XML documents are depicted. The difference between them is only in the number of *Product* elements. Using $Delete-Tree$ operation the whole subtree of *Product* element can be removed from tree B in one step.

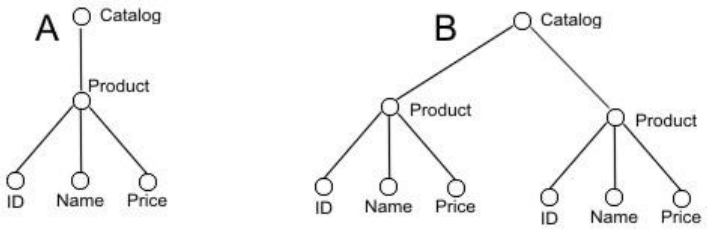

Figure 4.1: Applying *Delete-Tree* operation

Some repeated structures can be found in DTD trees too, especially if DTD contains some shared or recursive elements. That is why these new edit operations are suitable for our method as well. However, some procedures for computing edit distance need to be modified in order to use the algorithm for DTDs.

In addition, the semantic aspect of elements is often very important. Therefore, this work concerns also with semantic similarity [LYHY02].

4.2 Parts of Method

The whole method can be divided into 3 main parts as shown in Figure 4.2.

```
   Input: DTD_A, DTD_B
   Output: Edit distance between DTD_A and DTD_B
1  begin
2  |  TreeA = ParseDTD(DTD_A);
3  |  TreeB = ParseDTD(DTD_B);
4  |  Cost_Graft = ComputeCost(TreeB);
5  |  Cost_Prune = ComputeCost(TreeA);
6  |  return EditDistance(TreeA, TreeB, Cost_Graft, Cost_Prune);
7  end
```

Figure 4.2: Main parts of the algorithm

At first step the input DTDs are parsed (line 2 and 3) and their trees are constructed. Next, costs for tree-inserting (line 4) and tree-deleting (line 5) are computed. In the final step (line 6) we compute edit distance using dynamic programming.

4.3 Tree Construction

Many variants of transformation DTD into graph representation have already beena described. One of the most widely used, so-called *DTD graph*, was presented in [STZ+99]. The method transforms a DTD into an oriented graph, where nodes represent elements, attributes, and their cardinality constraints and edges represent relationships among element and its sub-elements, attributes, or cardinality constraints. However, this representation is not suitable for our method, since it can contain also auxiliary nodes: *OR* node for choice among elements, *AND* node for sequence of elements. It is difficult to compare similarity of DTD graphs containing both of these types of nodes, as they can generate totaly different XML documents, although they can have very similar structure. Therefore, we will use other representation of a DTD graph. However, we need to simplify DTD at first.

4.3.1 Simplification of DTDs

The DTD content model can be very complex and complicated, but it can be simplified. Some transformation rules for simplifying DTDs are described, e.g. in [STZ+99]. But these simplifications are too strong, because e.g. all "+" operators are transformed to "*" operators. Hence, we extend the rules to preserve "+" operators. The resulting rules are shown in Figures 4.3 and 4.4. Note that also transformation rules could be applied, but we do not need any other for our method.

I-a) $(e_1 \mid e_2)^* \rightarrow e_1^*, e_2^*$
I-b) $(e_1, e_2)^* \rightarrow e_1^*, e_2^*$
I-c) $(e_1, e_2)? \rightarrow e_1?, e_2?$
I-d) $(e_1, e_2)^+ \rightarrow e_1^+, e_2^+$
I-e) $(e_1 \mid e_2) \rightarrow e_1?, e_2?$

Figure 4.3: Flattening transformation rules

II-a) $e_1^{++} \rightarrow e_1^+$
II-b) $e_1^{**} \rightarrow e_1^*$
II-c) $e_1^*? \rightarrow e_1^*$
II-d) $e_1?^* \rightarrow e_1^*$
II-e) $e_1^{+*} \rightarrow e_1^*$
II-f) $e_1^{*+} \rightarrow e_1^*$
II-g) $e_1?^+ \rightarrow e_1^*$
II-h) $e_1^+? \rightarrow e_1^*$
II-i) $e_1?? \rightarrow e_1^?$

Figure 4.4: Simplification transformation rules

Rules *I-a* and all *II-a* to *II-i* are information preserving and therefore they have high priority.

An example of element simplification is depicted in Figure 4.5. Note that only the third step leads to information loss. As mentioned above, another transformation could be used, e.g. for grouping elements with the same name (e.g. "Para" elements could be grouped in our example), but we do not need these rules.

Rules *I-a* to *I-e* convert a nested definition into a flat representation. Rules *II-a* to *II-i* reduce combination of cardinality operators. These transformation rules are important for correct transforming DTDs into trees. Their usage provides a logical foundation for DTD transformation and minimizes information loss. Other solution to avoid using "OR" nodes in DTD

Rule I-a: `Sections (Title?, (Para | (Title?, Para+)+)*)`
⇒ `Sections (Title?, (Para*, ((Title?, Para+)+)*))`

Rule II-e: `Sections (Title?, (Para*, ((Title?, Para+)+)*))`
⇒ `Sections (Title?, (Para*, (Title?, Para+)*))`

Rule I-b: `Sections (Title?, (Para*, (Title?, Para+)*))`
⇒ `Sections (Title?, (Para*, Title?*, Para+*))`

Rules II-d and II-e: `Sections (Title?, (Para*, Title?*, Para+*))`
⇒ `Sections (Title?, Para*, Title*, Para*)`

Figure 4.5: Example of transformation rules

tree is, for example, to avoid such nodes at all. But transformation rules preserve more semantic information.

Other reason for using these rules is to enable converting all elements definitions so that each cardinality constraint operator will be connected only to one element. If we then join the constraint operator directly with the element we can avoid nodes representing cardinality constraint operators.

4.3.2 DTD Tree

After transformation of a DTD, its tree can be defined as follows:

Definition 4.3.1. ***[DTD Tree]*** *is an ordered rooted tree* $T(V, E)$ *where*

- V *is a finite set of nodes. For* $v \in V$, $v = (v_{Type}, v_{Name}, v_{Cardinality})$, *where* v_{Type} *is a type of node (attribute, element, #PCDATA),* v_{Name} *is a name of an element or attribute, and* $v_{Cardinality}$ *is cardinality constraint operator of an element or an attribute,*
- $E \subseteq V \times V$ *is a set of edges representing relationships between element and its attributes or sub-elements.*

For example, the DTD in Figure 4.6 can be transformed after simplification into DTD tree depicted in Figure 4.7.

Note that the DTD definition may contain other constructs, such as, e.g., entities, notations or EMPTY and ANY types of elements, enumerated and tokenized attribute type. These constructs are not used in our implementation for simplicity. However, it is very simple to extend our tree definition to include them and it would not affect complexity of the algorithm.

```
<!ELEMENT Article (Title, Author+, Section+)>
<!ELEMENT Sections (Title?, (Para|(Title?, Para+)+)*)>
<!ELEMENT Title (#PCDATA)>
<!ELEMENT Para (#PCDATA)>
<!ELEMENT Author (#PCDATA)>
<!ATTLIST Author CDATA Name REQUIRED>
```

Figure 4.6: DTD example

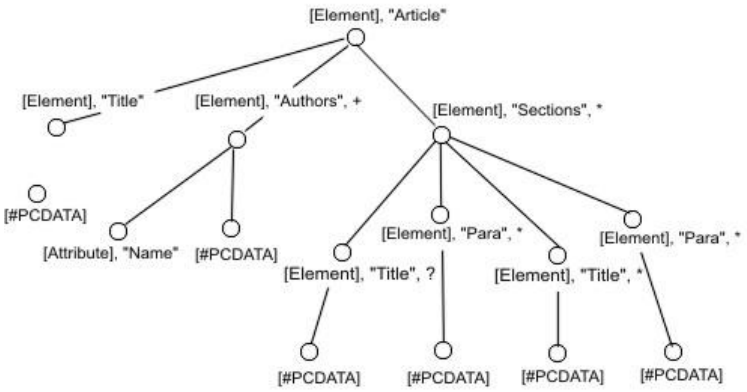

Figure 4.7: DTD Tree representation after transformation

4.3.3 Shared and Repeating Elements

In general, the structure of a DTD does not have to be purely tree-like. Some sub-elements may be shared by more than one element. In this case edges in graph would violate the tree structure. Therefore, each appearance of a shared element is represented using a single node (including its subtree).

On the other hand, if one of element's ancestors appears in its definition then recursive inclusion of elements occurs. And applying the previous rule for shared element it would lead to an infinite branch of tree. The majority of works concerning XML schema processing ignore this possibility at all. But the mentioned inconvenience can be solved, e.g., by simplification of such branch and repeating only several occurrences of this structure. From statistical analysis of real data [MTP06] we know that 10 occurrences are enough for approximation. Actually, it is not very important for our method how many occurrences we use because each of them can be transformed using single operation. Transformation of tree structures will be explained in the following text more precisely.

4.4 Tree Edit Operations

As mentioned above our method is based on the edit distance algorithm for XML documents proposed in [NJ]. They use five edit operations for transformation of XML trees. Due to our representation of DTD trees we can use exactly the same operations. They were described in definitions 2.3.3 - 2.3.7.

However, we need to modify definition 3.1.1 of allowable sequences.

Definition 4.4.1. ***[Allowable]*** *A sequence of edit operations transforming a source tree A to a destination tree B is allowable if it satisfies the following two conditions:*

1. A tree P may be inserted only if tree similar to P already occurs in the source tree A. A tree P may be deleted only if tree similar to P occurs in the destination tree B.

2. A tree that has been inserted via the InsertTree operation may not subsequently have additional nodes inserted. A tree that has been deleted via the DeleteTree operation may not previously have had (children) node deleted.

The meaning of *similar node* and *similar tree* will be explained in section 4.5.1. The reason for using only allowable sequence of edit operations is the same as in the original algorithm. We only do not insist on occurrence of exactly the same tree, but we allow only similar trees to be inserted or deleted.

Each of the edit operations is associated with a non-negative cost. The algorithm works with arbitrary costs. In our experimental implementation constant unit costs are set for operation *Insert* and *Delete*. Costs for insertion and deletion tree are parametrized. So, for example, we can simply avoid using *InsertTree* operation by setting high value to its cost.

The original algorithm does not consider similarity of nodes. Since in our method we want to take into account also cardinality, semantic and syntactic similarity of elements, we compute cost for operation *Substitution* as follows. Let x be the root node of tree A, y be the root node of B and ϵ be similarity of these nodes. Then cost for $Substitute_A(y) = 1 - \epsilon$.

4.5 Computing Costs for Inserting and Deleting Trees

Inserting a subtree T_i can be done with a single operation *InsertTree* or with some combination of *InsertTree* and *Insert* operations. To find the optimal variant the algorithm uses pre-computed cost for inserting tree

T_i, $Cost_{Graft}(T_i)$, and deleting tree T_i, $Cost_{Prune}(T_i)$. The procedure for computing these costs can be divided into two parts. In the first part *containedIn* list is created for each subtree of T_i. In the second part $Cost_{Graft}$ and $Cost_{Prune}$ are computed for T_i. This procedure is described in detail in [NJ] and it is also summarized in the chapter 3.1.1. In our approach the procedure is modified to involve similarity. Before the changes are applied, the similarity of DTD elements must be described.

4.5.1 Element Similarity

Similarity of elements can be evaluated using various criteria. Our method focuses on semantic and syntactic similarity and also on similarity of cardinality constraints of elements.

The first step to determine similarity focuses on the semantics of words. Semantic similarity is a score that reflects the semantic relation between the meanings of two words. Computing the score between two words w_1 and w_2 can be handled by searching synonyms of these words in a thesaurus. For this purpose we can reuse the procedure *OntologySim* described in section 3.3.1.

Secondly, we focus on syntactic similarity of elements. It is determined by computing the edit distance between labels of two elements. For our purpose the Levenshtein algorithm id used (see Figure 2.1). It uses three common edit operations: *Substituting*, *Inserting* and *Deleting* of a single character where each operation is associated with constant unit costs.

Finally, we consider similarity of cardinality constraints of elements. For our purpose we use cardinality compatibility table depicted in Figure 3.4.

In our experimental implementation this table is used also for attributes where the $IMPLIED$ constraint is associated with ? cardinality constraint and $REQUIRED$ constraint is associated with *none* cardinality constraint.

Now, the overall similarity, *Sim*, can be computed. However, since two words with a similar meaning can have small syntactic similarity, we use only maximum of this two scores. Hence, the overall similarity of elements e_1 and e_2 is computed as follows:

$$\begin{aligned} Sim(e_1, e_2) &= Max(SemanticSim(e_1, e_2), SyntacticSim(e_1, e_2)) \times \alpha \\ &+ CardinalitySim(e_1, e_2) \times \beta, \end{aligned}$$

where $\alpha + \beta = 1$ and $\alpha, \beta >= 0$.

Since two different words can have relatively small edit distance it is appropriate to use a threshold for the similarity. It can have general non-negative value <= 1. If this value is set to 1, then only exactly same elements will be marked as similar.

4.5.2 *ContainedIn* Lists Creating

The procedure for determining element similarity is used for creating *ContainedIn* lists which are used for computing $Cost_{Graft}$ and $Cost_{Prune}$. The list is created for each node of the destination tree and contains pointers to similar nodes in the source tree.

The sketch of procedure for creating *ContainedIn* lists is shown in Figure 4.8. Since creating of lists starts from leaves and continues to root, there is recursive calling of procedure at line 4. At line 6 we find all similar nodes of *nodeB* in tree *A* and add them to a list temporary. If *nodeB* is a leaf, we have a *ContainedIn* list created. Otherwise, for non-leaf nodes we have to filter the list with lists of node's descendants (line 8).

In this step each descendant of *nodeB* has to be found at the corresponding position in descendants of nodes in created *ContainedIn* list. More precisely, let $v_A \in nodeB_{ContainedInList}$, $children_{v_A}$ is the set of v_A descendants, and *childB* is a child of *nodeB*, then $childB_{ContainedInList} \cap children_{v_A} \neq \varnothing$, otherwise v_A is removed from $nodeB_{ContainedInList}$. Due to this step only whole subtrees remain in the *ContainedInList*.

```
   Input: tree A, root of tree B
   Output: CointainedInLists for all nodes in tree B
 1 CreateContainedInLists(treeA, nodeB);
 2 begin
 3 |  foreach child in nodeB do
 4 |  |  CreateContainedInLists(treeA, child)
 5 |  end
 6 |  nodeB_ContainedInList = FindSimilarNodes(treeA, nodeB);
 7 |  foreach child in nodeB do
 8 |  |  nodeB_ContainedInList = FilterLists(nodeB_ContainedInList,
   |  |  child_ContainedInList);
 9 |  end
10 |  Sort(nodeB_ContainedInList);
11 end
```

Figure 4.8: *ContainedIn* Lists Creating

4.5.3 Costs for Inserting Trees - $Cost_{Graft}$

When *ContainedIn* lists with corresponding nodes are created for node r, the cost for inserting the tree rooted at r can be assigned. The procedure is shown in Figure 4.9. The *ForEach* loop computes sum, sum_{d0}, for inserting node r and all its subtrees. If *InsertTree* operation can be applied

(*ContainedIn* list of r is not empty), sum_{d1}, is computed for this operation at line 10. The minimum of these costs are finally denoted as $Cost_{Graft}$ for node r.

```
   Input: root of tree B
   Output: Cost_Graft for tree B
 1 ComputeCost(r);
 2 begin
 3 |  sum_d0 = 1;
 4 |  foreach child in r do
 5 |  |  ComputeCost(child);
 6 |  |  sum_d0 += Cost_Graft(child);
 7 |  end
 8 |  sum_d1 = ∞;
 9 |  if root_ContainedInList is not empty then
10 |  |  sum_d1 = ComputeInsertTreeCost(r);
11 |  end
12 |  Cost_Graft(root) = Min(sum_d0,sum_d1);
13 end
```

Figure 4.9: Computing $Cost_{Graft}$

4.5.4 Costs for Deleting Trees - $Cost_{Prune}$

Since rules for deleting a subtree T from source are same as rules for inserting a subtree T into destination tree, costs for deleting trees are obtained by exactly the same procedures. We only switch tree A to tree B in procedures *CreateContainedInLists* and *ComputeCost*.

4.6 Computing Edit Distance

The last part of the algorithm for computing the edit distance is based on dynamic programming. At this step the procedure decides which of the operations defined in section 4.4 will be applied for each node to transforming source tree A to destination tree B. This part of algorithm does not have to be modified for DTDs so the original procedure presented in [NJ] is used. The procedure is depicted in figure 3.2.

4.7 Advantages and Disadvantages of the Proposed Method

The method proposed in this thesis evaluates the minimal edit distance cost for transforming one tree to another. Due to edit operations for inserting and deleting trees this method is appropriate for DTD trees as well.

It takes into account semantic and syntactic similarity and cardinality constraints too. The method is also very flexible due to possibility of using parameters for weights of particular similarities. For example we can avoid computing structural similarity between elements and determine only semantic similarity. In this case we reflect only similar meanings of labels of elements. We can also avoid computing similarity all together. Then this method would be similar to [NJ].

This method uses transformation rules for eliminating alternative operators and complexity of element type definition. These rules are necessary for effectiveness of computing the edit distance. But, on the other hand, these rules can lead to some information loss, which could be a disadvantage for some applications. A possible solution for including OR operators in DTD trees is to split it into a forest of AND trees. But it can lead into a huge number of trees and, hence, to higher complexity of the algorithm.

The method can be also extended with some other DTD constructs which can appear in DTD, such as, e.g., entities or attribute types . Although this is not difficult, task these constructs were not implemented only for the purpose of transparency of this text.

4.8 Complexity

In [NJ] was shown that the overall complexity is $O(|A||B|)$ for algorithm transforming tree A into tree B without determining similarity between their nodes. In our method we have to consider additional procedures for constructing DTD trees and mainly for computing similarity between elements.

Constructing a DTD tree is simple operation which can be done in $O(|A|)$ for tree A. The complexity of finding similarity depends on three procedures: *SemanticSimilarity*, *SyntacticSimilarity* and *CardinalityConstraintSimilarity*.

Syntactic similarity (edit distance of elements' labels) is computed for each pair of elements in tree A and B. So the complexity is $O(|A||B||\omega|)$ where ω is a maximal length of element's label.

Similarity of cardinality operators is also computed for each pair of elements. However, it is an operation with constant complexity. Hence, its complexity is $O(|A||B|)$.

The complexity of finding semantic similarity depends on the size of the thesaurus and on the number of steps (depth) we want to search synonyms. Since it is reasonable to search synonyms only for a few steps, we do not consider depth for computing complexity. So the overall complexity is $O(|A||B||\Sigma|)$, where Σ is the set of words in the thesaurus. Without any doubt this is the most time-consuming procedure in our method.

Chapter 5

Implementation

Experimental implementation of the proposed method is a part of this work. The application is called *DTDEditDistance*. An overview of used technologies, the architecture of the implementation, and user manual can be found in this chapter.

5.1 Used Technology and Libraries

The application is written as a .NET Framework 2.0 Windows Application. C# was used as programming language for the implementation. .NET was chosen for its rich support of libraries for working with XML documents and for author's good experience of this language.

The special libraries [Sem05] for computing semantic similarity of two words were used in our implementation. This library is also written using .NET framework and it uses a lexical database, called *WordNet* [Wor07], which is available online and provides a large repository of English lexical items. WordNet library (version 2.1) has to be installed on computer, unless semantic similarity is not enabled.

5.2 Overview of Architecture

The architecture composes of several modules and objects.

- *UserInterface* - this module is an interface between user and logical parts of application
- *EditDistanceMetric* - the main module containing basic procedures for computing edit distance
- *DTDTree* - the class representing DTD tree object, it also contains some methods for processing DTD trees, e.g., depth-first-searching of tree

- *DTDParser* - the module for parsing DTD definitions, simplifying of definitions and DTD trees constructing
- *CostComputer* - the module for computing costs of edit operations
- *ElementManager* - the module for processing elements (e.g., elements similarity is computed by this module)

The communication among these modules is depicted in Figure 5.1.

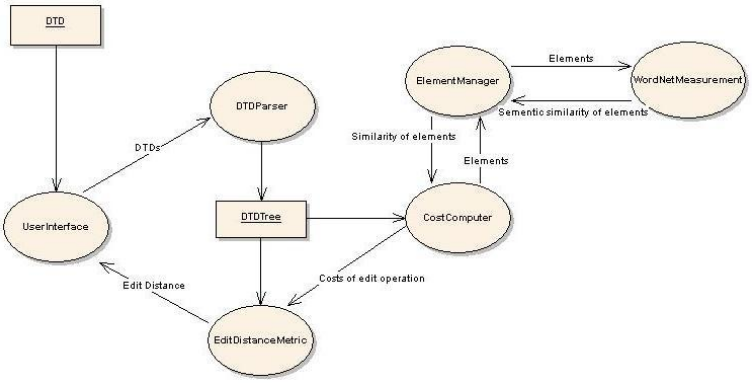

Figure 5.1: Application Architecture

5.3 User Manual

A communication between user and application is enabled only through graphic interface that is shown in Figure 5.2.

After starting of the application a windows form appears. At first two separate files with DTD definitions have to be selected. Then, the process for computing edit distance between the two DTDs can be started by pressing button "Edit Tree".

A user can also influence the process for computing edit distance by modifying its parameters or by selecting types of similarities that should be used during the process. For example, *Node similarity threshold* is a minimum value of a similarity between two elements. If the similarity is lower than threshold, than it is decreased to zero. The meaning of the other parameters are obvious from their description on the form.

The resulting value of edit distance between two DTDs is displayed on the form as well. The value of similarity is displayed together with the result of edit distance.

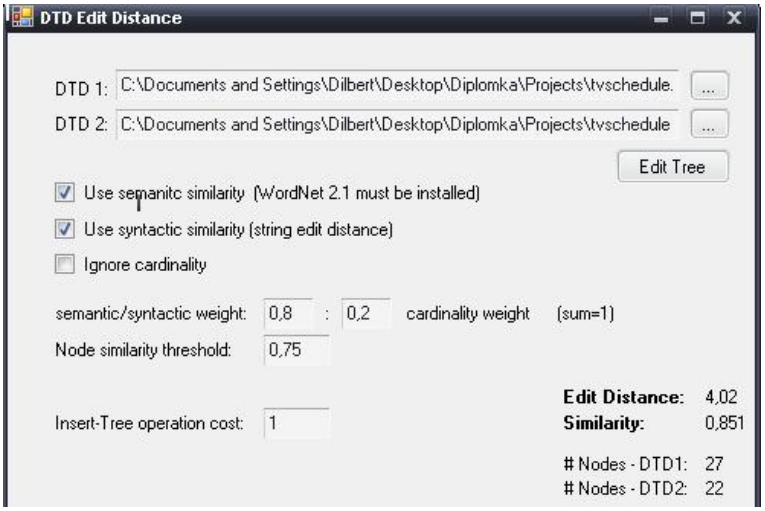

Figure 5.2: DTDEditDistance application

5.4 Restriction of Implementation

The application *DTDEditDistane* was implemented only for experimental purpose. It should demonstrate behaviour of the main edit distance procedure and influence of various parameters. The main restriction of application is that it requests DTDs with simplified definition, described in section 4.3.1. The transformation rules are not implemented.

Chapter 6

Experiments

In this chapter some experimental results of our implemented application are reviewed. We made several types of experiments for evaluating our method. In first one we collected some real DTDs and compared them each other. Then DTDs were categorized according to their similarities. Next experiments were focused on influence of some parameters on the algorithm. For this purpose we used artificial DTDs.

6.1 Real Data Comparing

In this experiment we used 7 different DTDs. All of them are stored on the supplied CD in "DATA" directory.

First 5 DTDs represent a $CUSTOMER$ object. Next two DTDs represent other objects: $TV\ SCHEDULE$ and $NEWSPAPER$. We used default values of parameters. Results of this test are depicted in Figure 6.1. The value in each cell is the resulting similarity of two DTDs for which both semantic and structural similarity were used. We can see that DTDs, representing the same object $CUSTOMER$, have higher similarities among themselves (the average is 0.44) than similarities among DTDs representing different objects (the average for $NEWSPAPER$ DTD is 0.13 and average for $TVSCHEDULE$ DTD is only 0.03). The only one exception is between $CUSTOMER1$ DTD and $NEWSPAPER$ DTD. Their similarity is relatively high.

In the second test we used the exactly same DTDs but we computed similarities without focusing on semantic similarity of elements. The resulting values are a little lower as we can see in Figure 6.2. However, the trend between same and different objects is same as in the previous test.

	c1	c2	c3	c4	c5	tv	news
customer1.dtd	1	0.57	0.43	0.19	0.71	0.08	0.42
customer2.dtd	0.57	1	0.57	0.45	0.48	0.10	0.11
customer3.dtd	0.43	0.57	1	0.39	0.36	0.01	0.13
customer4.dtd	0.19	0.45	0.39	1	0.21	0.00	0.00
customer5.dtd	0.71	0.48	0.36	0.21	1	0.00	0.11
tvschedule.dtd	0.08	0.10	0.01	0.00	0.00	1	0.00
newspaper.dtd	0.42	0.11	0.13	0.00	0.11	0.00	1

Figure 6.1: Real DTDs comparing

	c1	c2	c3	c4	c5	tv	news
customer1.dtd	1	0.45	0.23	0.09	0.57	0.00	0.13
customer2.dtd	0.45	1	0.50	0.42	0.32	0.00	0.00
customer3.dtd	0.23	0.50	1	0.30	0.15	0.00	0.00
customer4.dtd	0.09	0.42	0.30	1	0.20	0.00	0.00
customer5.dtd	0.57	0.32	0.15	0.20	1	0.00	0.00
tvschedule.dtd	0.00	0.00	0.00	0.00	0.00	1	0.00
newspaper.dtd	0.13	0.00	0.00	0.00	0.00	0.00	1

Figure 6.2: Real DTDs comparing without semantic similarity

6.2 Semantic Similarity Comparing

In this section we focused on some parameters of the application. At first, we are concerned with semantic similarity. We defined three DTDs, see Figure 6.3, which have exactly the same tree structure. They differs only in their element names. The element names of first and second DTDs have similar meaning while the element names of the third DTD have no lexical meaning.

The resulting values of comparing these DTDs are depicted in Figure 6.4. As we can see, there is a significant difference in comparing first two DTDs. They were identified as almost similar when we used semantic similarity. Despite comparing semantic similarity of DTD elements is time-consuming task, it can be very useful for identifying similar DTDs.

6.3 Edit Distance Operations

The last experiment is focused on two special edit operations using for transforming DTD trees, $InsertTree$ and $DeleteTree$. They are proposed for transforming repeating structures of a tree. We defined two similar DTDs

```
<!ELEMENT PERSON (DOMICILE, WORK)>
<!ELEMENT DOMICILE (STATE, TOWN)>
<!ELEMENT STATE (#PCDATA)>
<!ELEMENT TOWN (#PCDATA)>
<!ELEMENT WORK (#PCDATA)>
<!ATTLIST PERSON SURNAME CDATA #REQUIRED>
```

```
<!ELEMENT USER (RESIDENCE, JOB)>
<!ELEMENT RESIDENCE (COUNTRY, CITY)>
<!ELEMENT COUNTRY (#PCDATA)>
<!ELEMENT CITY (#PCDATA)>
<!ELEMENT JOB (#PCDATA)>
<!ATTLIST USER LASTNAME CDATA #REQUIRED>
```

```
<!ELEMENT AAA (BBB, DDD)>
<!ELEMENT BBB (EEE, FFF)>
<!ELEMENT DDD (#PCDATA)>
<!ELEMENT EEE (#PCDATA)>
<!ELEMENT FFF (#PCDATA)>
<!ATTLIST AAA CCC CDATA #REQUIRED>
```

Figure 6.3: DTDs definition for semantic similarity comparing

	With semantic similarity	Without semantic similarity
PERSON x USER	0.92	0.40
PERSON x AAA	0.33	0.33

Figure 6.4: Influence of semantic similarity

depicted in Figure 6.5. One of them has shared elements. As mentioned in the section 4.3.3, shared element is duplicated for each of its ancestors in our DTD tree representation.

We made four comparison of these DTDs with different costs of edit operations $InsertTree$ and $DeleteTree$. We can see in Figure 6.6 that in first two cases these special operations were really used. In the last two comparisons the costs for the operations were too high and the repeating tree structures were transformed by sequence of other single-node edit operations.

The DTDs were correctly identified as similar only when costs of these special operations were set sufficiently low.

```
<!ELEMENT USER (CUSTOMER, EMPLOYEE)>
<!ELEMENT CUSTOMER (USERDATA, ORDERS)>
<!ELEMENT EMPLOYEE (USERDATA, POSITION)>
<!ELEMENT USERDATA (ID, NAME, BIRTHDAY)>
<!ELEMENT ORDERS (#PCDATA)>
<!ELEMENT POSITION (#PCDATA)>
<!ELEMENT ID (#PCDATA)>
<!ELEMENT NAME (#PCDATA)>
<!ELEMENT BIRTHDAY (#PCDATA)>
```

```
<!ELEMENT USER (CUSTOMER)>
<!ELEMENT CUSTOMER (USERDATA, ORDERS)>
<!ELEMENT USERDATA (ID, NAME, BIRTHDAY)>
<!ELEMENT ORDERS (#PCDATA)>
<!ELEMENT ID (#PCDATA)>
<!ELEMENT NAME (#PCDATA)>
<!ELEMENT BIRTHDAY (#PCDATA)>
```

Figure 6.5: DTDs definition for special edit operations

	Cost=1	Cost=5	Cost=10	Cost=100
USER1 x USER2	0.92	0.74	0.52	0.52

Figure 6.6: Comparing different costs of edit operations

Chapter 7

Conclusion

The aim of this work was a proposal and implementation of own method for similarity evaluation among XML schemes.

Firstly, existing solutions were analyzed and their advantages and disadvantages were discussed. Then, on the basis of the analysis and found disadvantages of analyzed solutions, the algorithms of our new method were proposed.

DTD language was chosen as a language for definition of schemes of XML documents. DTD is commonly used for declaration of XML documents on the web and it is also suitable language for its simplicity. These reasons were decisive for selecting DTD. The tree edit distance technique was chosen for similarity evaluation. This technique has approved itself for similarity evaluation of XML documents and we wanted to extend it also on DTDs.

The proposed algorithms were implemented in the practical part of this work. However, at first we researched some existing possible components useful for our algorithms, such as WordNet working with the thesaurus, that we used for searching synonyms of words.

The main contribution of this work is the extension of the edit distance algorithm to processing DTD trees. In many related works this technique was previously used, however, only for comparing similarity of XML documents. Other contribution is in focusing on various aspects of similarity evaluation and using several individual solutions (i.e. edit distance of elements names, or searching a thesaurus to evaluate their semantic similarity) together in one complex method. We also proposed our own representation of DTD trees.

Finally, we can declare that our primary aims were realized in principle. However, the proposed solution can be still extended. For example, further edit operations can be added to our algorithm, e.g., moving a node or a subtree, or deleting a non-leaf node. Other scope for future work is in using another type of edit distance algorithm. There are quite a lot of types of such algorithms dealing with XML documents. Using XML Schema instead

of DTD is another interesting alternative for implementing tree edit distance algorithm. XML Schema is another language for description of type of XML document and it can be also represented as a tree, partly due to the fact that it is based on XML language.

Bibliography

[BGM04] Elisa Bertino, Giovanna Guerrini, and Marco Mesiti, *A matching algorithm for measuring the structural similarity between an xml document and a dtd and its applications*, Inf. Syst. **29** (2004), no. 1, 23–46.

[BPSM00a] Tim Bray, Jean Paoli, and C. M. Sperberg-McQueen, *Document type declaration*, 2000.

[BPSM00b] ———, *Extensible Markup Language (XML) 1.0 - W3C recommendation 10-february-1998*, 2000.

[DDH01] AnHai Doan, Pedro Domingos, and Alon Y. Halevy, *Reconciling schemas of disparate data sources: A machine-learning approach*, SIGMOD Conference, 2001.

[DR02] H. Do and E. Rahm, *Coma - a system for flexible combination of schema matching approaches*, 2002.

[Fal01] D. C. Fallside, *Xml schema part 0: Primer*, 2001.

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

[Ham50] Richard W. Hamming, *Error detecting and error correcting codes*, Bell System Technical Journal 26(2), 1950, pp. 147–160.

[Lev66] V. I. Levenshtein, *Binary Codes Capable of Correcting Deletions, Insertions and Reversals*, Soviet Physics Doklady **10** (1966), 707–+.

[Lev05] *http://codeproject.com/kb/recipes/improvestringsimilarity.aspx*, 2005.

[LYHY02] Mong L. Lee, Liang H. Yang, Wynne Hsu, and Xia Yang, *Xclust: clustering xml schemas for effective integration*, CIKM '02: Proceedings of the eleventh international conference on Information and knowledge management (New York, NY, USA), ACM Press, 2002, pp. 292–299.

[MBR01] Jayant Madhavan, Philip A. Bernstein, and Erhard Rahm, *Generic schema matching with cupid*, The VLDB Journal, 2001, pp. 49–58.

[MP06] Irena Mlýnková and Jaroslav Pokorný, *Exploitation of similarity and pattern matching in XML technologies*, Tech. Report 13, Charles University, Prague, Czech Republic, 2006.

[MTP06] I. Mlynkova, K. Toman, and J. Pokorny, *Statistical Analysis of Real XML Data Collections*, COMAD'06: Proc. of the 13th Int. Conf. on Management of Data (New Delhi, India), Tata McGraw-Hill Publishing Company Limited, 2006, pp. 20–31.

[NJ] Andrew Nierman and H. V. Jagadish, *Evaluating structural similarity in xml documents*, Proceedings of the Fifth International Workshop on the Web and Databases (WebDB 2002), pp. 61–66.

[RB01] Erhard Rahm and Philip A. Bernstein, *A survey of approaches to automatic schema matching*, The VLDB Journal **10** (2001), no. 4, 334–350.

[Sem05] *http://www.codeproject.com/kb/string/semanticsimilaritywordnet.aspx*, 2005.

[STZ+99] Jayavel Shanmugasundaram, Kristin Tufte, Chun Zhang, Gang He, David J. Dewitt, and Jeffrey F. Naughton, *Relational databases for querying xml documents: Limitations and opportunities*, VLDB '99: Proceedings of the 25th International Conference on Very Large Data Bases (San Francisco, CA, USA), Morgan Kaufmann Publishers Inc., 1999, pp. 302–314.

[SZ97] Dennis Shasha and Kaizhong Zhang, *Approximate tree pattern matching*, Pattern Matching Algorithms, Oxford University Press, 1997, pp. 341–371.

[Wor07] *http://www.ebswift.com/opensource/wordnet.net/*, 2007.

[ZS89] K. Zhang and D. Shasha, *Simple fast algorithms for the editing distance between trees and related problems*, SIAM J. Comput. **18** (1989), no. 6, 1245–1262.

Appendix A

Contents of CD-ROM

Contents of CD-ROM

The enclosed CD-ROM is a part of this thesis. It contains the text of the work, source code of the implemented application and executable files of the application DTDEditDistance. CD-ROM contains following files and directories:

- text - directory with the text of the thesis in PDF format
- src - directory with source codes of the application
- app - directory containing binaries files of the application
- data - directory with data files that were used for experiments