Charles University in Prague

Faculty of Mathematics and Physics

BACHELOR THESIS

Egor Ivkin

Comparison of Tree Edit Distance Algorithms

Department of Software Engineering - KSI

Supervisor of the bachelor thesis:	Mgr. Martin Nečaský, Ph.D.
Study programme:	Computer Science
Specialization:	IOI

Prague 2012

Na tomto místě bych rád poděkoval vedoucímu této bakalářské práce, panu Nečaskému, za cenné rady, připomínky a podněty k zamyšlení týkající se bakalářské práce.

Název práce: Porovnání Tree Edit Distance algoritmů

Autor: Egor Ivkin

Katedra: Katedra softwarového inženýrství

Vedoucí bakalářské práce: Mgr. Martin Nečaský, Ph.D.

Abstrakt: Cílem této práce je srovnání Tree Edit Distance metod v rámci zjišťování strukturální podobnosti mezi XML Schema dokumenty. Tyto metody vyhledávají minimální počet editačních operací, pomocí kterých jeden strom je převáděn na druhý. Zanalyzovali a implementovali jsme celou řadu existujících Tree Edit Distance algoritmů. Je důležité si uvědomit, že vzdálenost počítána algoritmy je ovlivněna množinou použitých editačních operací proto kvalita při odhalování XML Schema podobnost se liší v každém postupů. První část této práce obsahuje popis použitých postupů a potřebné zápisy. Druhá část obsahuje detaily implementace a analýzu popsaných metod, která sestává teoretické analýzy a porovnávání výsledků běhů nad reálnými i syntetickými XML daty. Výsledná implementace je k dispozici ve formě Java aplikace.

Klíčová slova: XML, Tree Edit Distance, XML schema, podobnost

Title: Comparison of Tree Edit Distance Algorithms

Author: Egor Ivkin

Department: Department of Software Engineering

Supervisor: Mgr. Martin Nečaský, Ph.D.

Abstract: The aim of this thesis is the comparison of the Tree Edit Distance methods, in the context of detecting structural similarity between two XML Schema documents. The methods search the minimum number of edit operations leading from one tree to another. We have analysed and implemented a wide range of the existing tree edit distance approaches. It is important to understand that the distance computed by the algorithms is affected by the set of used edit operations, therefore the strength in detecting XML Schema similarity differs in each approach. The first part of this work contains the description of the used approaches and necessary notations. The second part provides implementation details and analysis of the described methods, which consists of theoretical comparison and empirical evaluation on real and synthetic xml data. The resulting implementation is available in the form of Java SE application.

Keywords: XML, Tree Edit Distance, XML schema, similarity

Contents

1. Introduction

XML (eXtensible Mark-up Language) [1] became recently a fundamental standard for efficient data management and exchange. This format often used to guarantee interoperability between various systems, especially to broadcast data over web. Due increasing web exploitation of XML, XML-based similarity became central issue in many researches in context of information retrieval and databases. By XML Schema similarity we underline document related similarities especially document/document comparison. There are many applications of XML/XML Schema comparison like version control, data warehousing, clustering documents gathered from the web, identifying XML documents. In this thesis we will research the so called TED (Tree Edit Distance) based methods, which computes the smallest number of edit operations that lead from one XML Schema document to another. The research offers the description and the implementation of the most interesting algorithms from this area. One of the most important part of the thesis is the comparison of the efficiency of the algorithms using real and synthetic data sets.

1.1 Background and notation

XML Schema documents represent hierarchically structured data and are generally modeled as Ordered Labeled Trees (OLT). Each node in such a tree represents a XML element and is labeled with a corresponding tag name. Each edge in this tree represents a relation between the child element and the parent element in the corresponding XML document. Attribute nodes appear as children of their encompassing element nodes, placed before all sub-element siblings. In our work we are interested only in structural properties of the XML file, so we will not represent Element/Attribute values.
The following are the frequently used definitions in this thesis.

Definition: A tree T is directed, acyclic, connected graph with nodes N(T), and edges E(T) $\subseteq$ N(T) $\times$ N(T), where each node has at most one incoming edge.

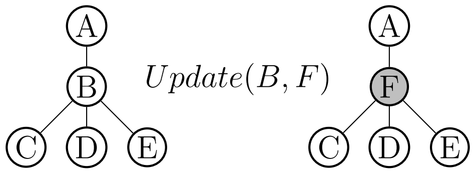
Definition: forest F is graph in which each connected component is tree.(each tree is also is forest).

Definition: Edit script is a sequence of edit operations $op_1, op_2, ..., op_n$. When applied to tree T, the resulting tree T' is obtained by applying edit operations of the Edit Script (ES) to T, following their order of appearance in the script. By associating cost with each edit operation, c_{op}, the cost of whole ES is defined as sum of the costs of its operations: $c_{ES} = \sum_{i=1}^{|ES|} c_{op}$.

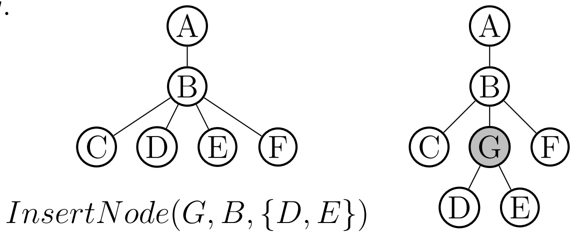
Definition: The Edit Distance between two trees A and B is defined as the minimum cost of all edit scripts that transforms A to B: $ED(A, B) = Min\,\{Cost_{ES}\}$. So, the problem of comparing two trees,i.e. evaluating their structural similarity, is defined as the problem of computing the corresponding tree edit distance[cit].

Edit operations can be classified in two groups:*atomic* operations and *complex* operations. An atomic operation is either the insertion inner/leaf node,the deletion inner/leaf node, or the updation of node. A complex tree edit operation is a set of atomic tree edit operations, treated as one operation. A complex operation is either the deletion or insertion of whole sub-tree in another tree (which is actually a sequence of atomic node insertion/deletion operations).

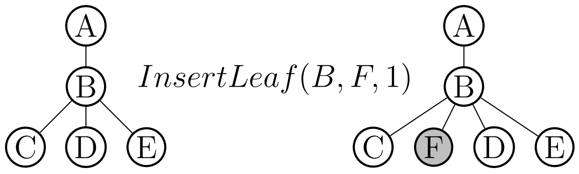
Definition: Given node x and tree T and new node y, $Update(x, y)$ is an update operation that replace x by y in the transformed tree T. New node y will have same parent and children as original node x .($Update()$ is equivalent to operation $Relabel(x, y)$ in some studies)



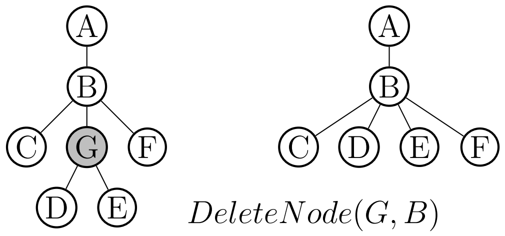
Definition: Given node x and tree T, T containing node y with children (first-level sub-trees) $\{T_1, ..., T_n\}$, operation $insertNode(x, y, \{P_i, ..., P_j\})$, inserts node x in T as i^{th} child of y making x the parent of the consecutive subsequence of sub-trees of node y.



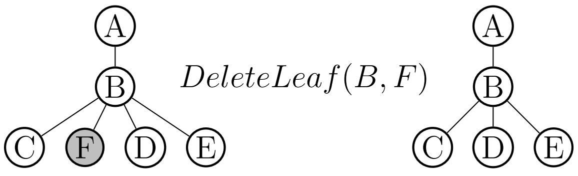
Definition: Given node x and tree T containing node y, operation $insertLeaf(x, y, i)$ inserts leaf node x as the i^{th} child of y.



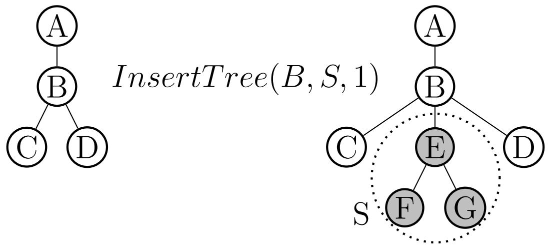
Definition: Given a node x and tree T, T encompassing node y with first level sub-trees (i.e. children) $\{T_1, ..., T_{i-1}, x, T_{i+1}, ..., T_n\}$, and x having first level sub-trees $\{X_1, ..., X_m\}$, operation $DeleteNode(x)$ deletes node x in T, making the children of x become the children of y. The children are inserted in place of original node x as a subsequence in the left-to-right order of children y.



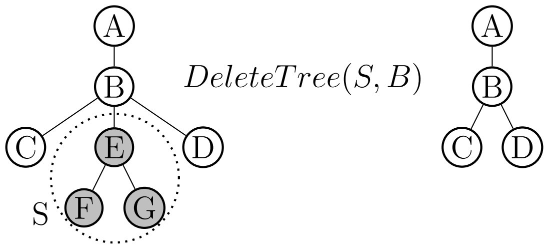
Definition: Given a leaf x and tree T, T encompassing node y with children $\{T_1, ..., T_{i-1}, x, T_{i+1}, ..., T_n\}$, operation $DeleteLeaf(x, y)$ deletes leaf x in tree T.



Definition: Given a tree A and tree T ,T including node y with first-level sub-trees $\{T_1, ..., T_n\}$, operation $InsertTree(A, x, i)$ is tree insertion applied to T, inserting A as the i^{th} sub-tree of node y.



Definition: Given a tree A and tree T, T including node y with first-level sub-trees $\{T_1, ..., T_{i-1}, A, T_{i+1}, ..., T_n\}$, operation $DeleteTree(A, y)$ deletes sub-tree in T from among children of y.

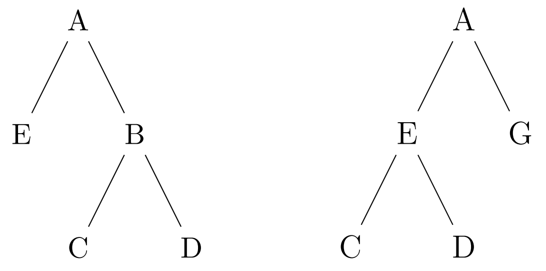


2. Algorithm description

2.1 Selkow's edit distance algorithm

Selkow is one of the early approaches in Tree Edit Distance [4]. Selkow extended the string-to-string edit distance problem solution developed by Wagner-Fischer[11], which optimality has been accredited in a broad variety of computational applications, in finding tree-to-tree edit distance. Basic operations used in this approach are UpdateNode, InsertLeaf and DeleteLeaf, so it restricts insertion and deletion of single nodes at the leaves, and relabeling of nodes anywhere in the tree. Selkow's approach is recursive. At first the algorithm is called on the root nodes of two trees, the source and the target tree. Then the algorithm is called on each child of one of the roots and each child of the other root. In each recursive step (pair of nodes), algorithm sets up a temporary n $\times$ m cost matrix, where n is the number of children of the first node in pair and m is the number of children of the second one. Each entry (i, j) in the temporary matrix is then computed by taking the minimum cost from the three available approaches to edit $A_1, A_2, ..., A_i$ to $B_1, B_2, ..., B_j$.

- Add the cost to edit $A_1, A_2, ..., A_i$ and $B_1, B_2, ..., B_{j-1}$ and the cost to insert B_j.
- Add the cost to edit $A_1, A_2, ..., A_{i-1}$ and $B_1, B_2, ..., B_j$ and the cost to delete A_i.
- Add the cost to edit $A_1, A_2, ..., A_{i-1}$ and $B_1, B_2, ..., B_{j-1}$ and the cost to edit A_i to B_i.



(a) Tree A (b) Tree B

Figure 2.1: Trees used in example computation

This is an example of computation of tree edit distance between trees in figure 2.1 .

1. At first step, algorithm sets up distance matrix D[m+1][n+1] = D[3][3] for source tree and target tree, and initialize it with values:

    - D[0][0] = cost to relabel root = 0

- D[1][0]..D[m][0] - costs to delete sequences of sub-trees E_A and E_A,B_A = 1, 4.
- D[0][1]...D[0][n] - costs to insert sequences of sub-trees B_B and B_B,G_B = 3,4.

0	1	4
3		
4		

2. To compute distance matrix for previous step, we need to solve 4 subproblems : $edit(E_A, E_B)$, $edit(E_A, G_B)$, $edit(B_A, E_B)$, $edit(B_A, G_B)$:
   - $edit(E_A, E_B)$ - cost to add nodes C_B and D_B is 2.
   - $edit(E_A, G_B)$ - cost to relabel E_A to G_B is 1.
   - $edit(B_A, E_B)$ - cost to relabel node B_A to E_B is 1.
   - $edit(B_A, G_B)$ - cost to relabel B_A to G_B, and delete C_A, D_A is 3.
3. as we have computed all subproblems we can now compute final distance matrix for source and target trees:
   - $D[1][1] = min \begin{cases} D[0][0] + edit(E_A, E_B) = 0 + 2 = 2 \\ D[0][1] + delTree(E_A) = 3 + 1 = 4 \\ D[1][0] + insTree(E_B) = 1 + 3 = 4 \end{cases}$
   - $D[2][1] = min \begin{cases} D[1][0] + edit(B_A, E_B) = 1 + 1 = 2 \\ D[1][1] + delTree(B_A) = 2 + 3 = 5 \\ D[2][0] + insTree(E_B) = 4 + 3 = 7 \end{cases}$
   - $D[1][2] = min \begin{cases} D[0][1] + edit(E_A, G_B) = 3 + 1 = 4 \\ D[0][2] + delTree(E_A) = 4 + 1 = 5 \\ D[1][1] + insTree(G_B) = 2 + 1 = 3 \end{cases}$
   - $D[2][2] = min \begin{cases} D[1][1] + edit(B_A, G_B) = 2 + 3 = 5 \\ D[1][2] + delTree(B_A) = 3 + 3 = 6 \\ D[2[1] + insTree(G_B) = 2 + 1 = 3 \end{cases}$
4. now we have final distance matrix between A and B :

0	1	4
3	2	2
4	3	3

   Edit distance between A and B is 3, and corresponds to edit script: $delTree(E_A)$,$relabel(B_A, E_B)$,$insTree(G_B)$.

Full algorithm presented in figure 2.2 .

The time complexity of algorithm is O($n \times m \times d$), where n and m are the maximum number of children of any node in each of the trees and d is the maximum depth of the trees.
By restricting insert and delete operations to the leaf nodes Selkow made his algorithm very simple and effective.

```
CostType selkowDistance(Tree A, Tree B) {
int M = Degree(A);
int N = Degree(B);
CostType[][] D = new int[0..M][0..N];
D[0][0] = $c_r(\lambda(A), \lambda(B))$;
for(int i=1;i ≤ M;i++)
  D[i][0] = D[i-1][0] + $c_d(A_i)$;
for(int j=1;j ≤ N;j++)
  D[0][j] = D[0][j-1] + $c_i(B_j)$;
for(int i=1;i ≤ M;i++)
  for (int j = 1; j ≤ N; j++)
    D[i][j] = min(
    D[i-1][j-1] + selkowDistance($A_i$,$B_j$),
    D[i][j-1] + $c_i(B_j)$,
    D[i-1][j] + $c_d(A_i)$
     );
 return D[M][N];
}
```

Figure 2.2: Selkow's Edit distance algorithm.

2.2 Chawathe's edit distance algorithm

The work provided by Chawathe [3] has been considered as a reference point in recent tree edit distance literature and has provided the basis for various XML related structural comparison studies like [2] etc. Chawathes approach the same as Selkow's solution restricts insertion and deletion operations to leaf nodes, and allows the relabeling of nodes anywhere in the tree (earlier defined operations UpdateNode(), InsertLeaf(), DeleteLeaf()). The proposed algorithm is another modification (similar to Selkow's approach) of the Wagner-Fisher algorithm for editing string [11]. It is also one of the fastest TED algorithms available. The author transforms trees into special sequences called ld-pairs. The ld-pair representation of a tree comes down to the list, in preorder, of the ld-pair representations of its nodes.

Definition: Ld-pair representation of a node is defined as the pair (l, d) where:l and d are respectively the nodes label and depth in the tree. We use $p.l$ and $p.d$ to refer to the label and the depth of an $ld - pair$ node p respectively.

For example ld-pair representations of the trees A and B in figure 2.1, are:
$A = (a, 0), (e, 1), (b, 1), (c, 2), (d, 2)$
$B = (a, 0), (e, 1), (c, 2), (d, 2), (g, 1)$
Given a tree A (in ld-pair representation) $A = (a_1, a_2, ..., a_M)$, notation $A[i]$ refers to the i^th node a_i of the tree A. Thus, $A[i].l$ and $A[i].d$ denote, respectively, the label and the depth of the node a_i of A.
Also the author defined the structure called tree edit graph:

Definition: Edit graph of two trees A and B (in ld-pair representation) consist of a $(M + 1) \times (N + 1)$ grid of nodes. There is a node at each (x, y) location for

$x \in [0...(M+1)]$ and $y \in [0..(n+1)]$ (figure 6). These nodes are connected by directed edges as follows:

- For $x \in [0, M-1]$ and $y \in [0, N-1]$, there is a diagonal edge from (x, y) to $(x+1, y+1)$ if $A[x+1].d = B[y+1].d$.
- For $x \in [0, M-1]$ and $y \in [0, N]$, there is a horizontal edge from (x, y) to $(x+1, y)$ if $y = N$ or $B[y+1].d \leq A[x+1].d$.
- For $x \in [0, M]$ and $y \in [0, N-1]$, there is vertical edge from (x, y) to $(x, y+1)$ if $x = M$ or $A[x+1].d \leq B[y+1].d$.

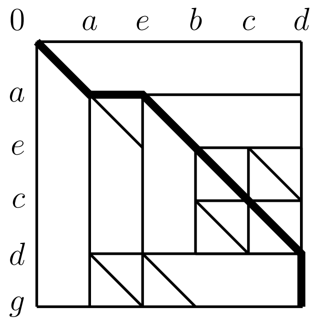
Every path from $(0,0)$ to (M, N) in the tree edit graph corresponds to an edit script that transforms A to B. Using tree edit graph and ld-pair serialization of the trees, the problem of computing minimum edit script between two trees is reduced to the problem of finding a shortest path in the edit graph of those trees. In accordance with this reduction, works the main-memory Chawathe's algorithm.

Definition: Given $(M+1) \times (N+1)$ tree edit graph G, let D be $(M+1) \times (N+1)$ matrix such that $D[x, y]$ is the length of the shortest path from $(0,0)$ to (x, y) in the edit graph. Matrix D is called *distance matrix* for G.

Distance matrix computed as follows:
For $D[x, y]$, where $0 < x \leq M$ and $0 < y \leq N$ $D[x, y] = min(m_1, m_2, m_3)$, where

$$m_1 = \begin{cases} D[x-1, y-1] + c_u(A[x], B[y]) & \text{if } ((x-1, y-1)(x, y)) \in G \\ \infty & \text{otherwise} \end{cases}$$
$$m_2 = \begin{cases} D[x-1, y] + c_d(A[x]) & \text{if } ((x-1, y)(x, y)) \in G \\ \infty & \text{otherwise} \end{cases}$$
$$m_3 = \begin{cases} D[x, y-1] + c_u(B[y]) & \text{if } ((x, y-1)(x, y)) \in G \\ \infty & \text{otherwise} \end{cases}$$

In figure 2.3, we may see examples of an edit graph and a distance matrix for the trees we have used in the selkow's example (figure 2.1).



Tree Edit Graph

	0	a	e	b	c	d
0	0	1	2	3	4	5
a	1	0	1	2	3	4
e	2	1	0	1	2	3
c	3	2	2	3	3	3
d	4	3	3	2	3	4
g	5	4	4	3	2	3

Distance Matrix

Figure 2.3: Examples of edit graph and distance matrix

Consequently, the author simplifies the problem of comparing two documents trees to that of comparing the corresponding ld-pair sequences, using a specialization of the Wagner-Fisher algorithm. In figure 2.4 there is a listing for chawathe algorithm. At first both trees are converted to their ld-pair representation, then with these sequences the algorithm computes distance matrix D, where $D[i][j]$ is the edit distance between node at position i in ld-pair representation of original tree, and node at position j in ld-pair representation of the target tree.

```
CostType chawatheDistance(Tree Original, Tree Target) {
ld-pair[] A = ldPairRepresentation(Original);
ld-pair[] B = ldPairRepresentation(Target);
int M = length(A);
int N = length(B);
CostType[][] D = new int[0..M][0..N];
D[0][0] = 0;
for(int i=1; i ≤ M; i++)
 D[i][0] = D[i-1][0] + $c_d$(A[i]);
for(int j=1;j ≤ N;j++)
 D[0][j] = D[0][j-1] + $c_j$(B[j]);
for(int i=1;i ≤ M;i++)
  for (int j = 1; j ≤ N; j++){
    int $m_1$ = ∞;
    int $m_2$ = ∞;
    int $m_3$ = ∞;
    if(A[i].d == B[j].d){
      $m_1$ = D[i-1][j-1] + $c_u$(A[i],B[j]);
    }
    if(j==N || B[j+1].d ≤ A[i].d){
      $m_2$ = D[i-1][j] + $c_d$(A[i]);
    }
    if(i==M || A[i+1].d ≤ B[j].d){
     $m_3$ = D[i][j-1] + $c_i$(B[j]);
    }
    D[i][j] = min( $m_1$, $m_2$, $m_3$);
    }
  return D[M][N];
}
```

Figure 2.4: Chawathe Edit distance algorithm.

He also extends his algorithm for external-memory computations and identifies respective I/O, RAM and CPU costs. Note that this is the only algorithm that has been extended to efficiently calculate edit distances in external memory (without any loss of computation quality/efficiency). The overall complexity of Chawathes algorithm is of $O(N^2)$ (recall that N is the maximum number of nodes in the trees being compared)

2.3 Nierman's edit distance algorithm

In work by Nierman and Jagadish [2], authors stressed the importance of identifying sub-tree structural similarities in XML document trees. Usually two XSD/XML documents could have a very different inner structure determined by the presence of repeated and optional elements in document. These elements often produce multiple occurrences of similar element/attribute sub-trees (in case of optional elements/attributes) or identical sub-trees in the same XSD document (repeated elements), which reflects the need to take these sub-tree resemblances into consideration while comparing documents. Authors extends early approaches by adding two new operators: insert tree and delete tree. To discover sub-tree similarities between trees/sub-trees, they make use of *containedin* relations.

Definition: a tree T_1 is said to be *containedIn* a tree T_2 if all nodes of T_1 occur in T_2, with the same parent/child edge relationship and node order. Additional nodes may occur in T_2 between nodes in the embedding of T_1.

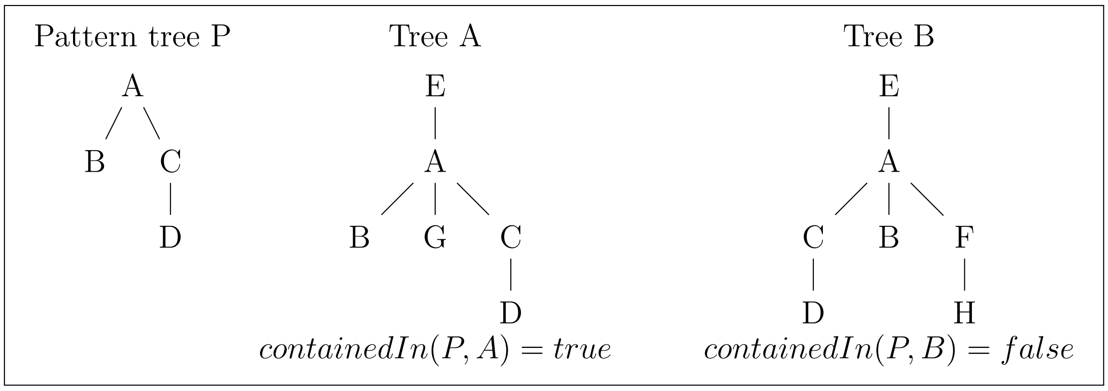


Figure 2.5: Example of containedIn relations

Authors also placed some restrictions on using tree insert/delete operations.

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.

Nierman algorithm instead of delete/insert node uses graft/prune cost which is calculated in precomputation phase. Precomputetion phase consist of two steps:

1. calculate *containedIn* relation between every sub-tree of A and tree B.
2. determine the cost of inserting every sub-tree of tree B (target tree) - c_{Graft}, and the cost of deleting every sub-tree of tree A (original tree) - c_{Prune}. At each node $v \in B$ algorithm calculates the cost of inserting single node v and add the graft cost of each child of v -called sum d_0. Then also checked if the pattern tree P (sub-tree rooted at v), is *containedIn* in the source tree A. if $containedIn(P, A)$ is true, then calculated insert tree cost for sub-tree P -called sum d_1.Then graft cost for the sub-tree rooted at v is the $min(d_0, d_1)$.Prune costs computed similarly for each node in A.

```
CostType editDistance(Tree A, Tree B) {
int M = Degree(A);
int N = Degree(B);
CostType[][] dist = new int[0..M][0..N];
dist[0][0] = CostRelabel($\lambda(A)$, $\lambda(B)$);
for(int j=1;j $\leq$ N;j++)
  dist[0][j] = dist[0][j-1] + $Cost_{Graft}(B_j)$;
for(int i=1;i $\leq$ M;i++)
  dist[i][0] = dist[i-1][0] + $Cost_{Prune}(A_i)$;
for(int i=1;i $\leq$ M;i++)
  for (int j = 1; j $\leq$ N; j++)
    dist[i][j] = min(
    dist[i-1][j-1] + editDistance($A_i$,$B_j$),
    dist[i][j-1] + $Cost_{Graft}(B_j)$,
    dist[i-1][j] + $Cost_{Prune}(A_i)$
    );
 return dist[M][N];
} //editDistance
```

Figure 2.6: Nierman Edit distance algorithm.

Full listing of main body of Nierman's algorithm presented in figure 2.6 . The overall complexity of their algorithm simplifies to $O(N^2)$. Structural clustering experiments in [Nierman] show that the proposed algorithm outperforms, in quality, that of Chawathe [Chawathe]. However, the authors show that their algorithm is conceptually more complex than its predecessor, requiring a precomputation phase for determining the costs of tree *insert/delete* operations (which complexity is of $O(2 \times N + N^2)$).

2.4 Tekli's edit distance algorithm

In order to capture the sub-tree structural similarities unaddressed by Niermans [2] approach, in [8] authors identify the need to replace the tree *containedIn* relation making up a necessary condition for executing tree insertion and deletion operations in [2], by introducing the notion of commonality between two sub-trees.

Definition: given two sub-trees $A = (a_1, ..., a_m)$ and $B = (b_1, .., b_n)$ in ld-pair representation, the structural commonality between A and B, designated by $Comsub - tree(A, B)$. is a set of nodes $N = n_1, ..., n_p$ such that $\forall n_i \in N$, n_i occurs in A and B with the same label, depth and relative node order (in pre-order traversal ranking) as in A and B. For $1 \leq i \leq p$; $1 \leq r \leq m$;$1 \leq u \leq n$:

1. $n_i.l = a_r.l = b_u.l$
2. $n_i.d = a_r.d = b_u.d$
3. For any $n_j \in N/i \leq j$, $\exists a_s \in A$ and $b_v \in B$ such as:
   - $n_j.l = a_s.l = b_v.l$
   - $n_j.d = a_s.d = b_v.d$
   - $r \leq s, u \leq v$
4. There is no set of nodes N that satisfies conditions 1, 2 and 3 and is of larger cardinality than N.

In other words, $Comsub - tree(A, B)$ identifies the set of matching nodes between sub-trees A and B, node matching being undertaken with respect to node label, depth and relative preorder ranking. In the figure 2.7, $|Comsub - tree(A_1, B_1)| = 3$ and $|Comsub - tree(A_2, B_1)| = 1$,for example.

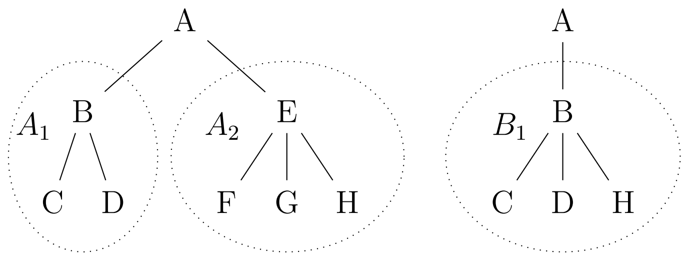


Figure 2.7: Exampled Trees

According to the above definition, the problem of finding the structural commonality between two sub-trees SbT_i and SbT_j is equivalent to finding the maximum number of matching nodes i.e. $|Comsub - tree(SbT_i, SbT_j)|$. On the other hand, the problem of finding the shortest edit distance between SbTi and SbTj comes down to identifying the minimal number of edit operations that can transform SbTi to SbTj. Those are dual problems since identifying the shortest edit distance between two sub-trees (trees) underscores, in a roundabout way, their maximum number of matching nodes.
Author introduced an algorithm CBS, based on edit distance concept, to identify

the structural commonality between pairs of sub-trees. Each sub-tree transformed into their ld-pair representation, to make them suitable for standard edit distance computation. Authors compute maximum number of matching nodes between SbT_A and SbT_B (i.e $|Comsub - tree(SbT_A, SbT_B)|$) as follows :

-Take total number of deletions - delete all nodes of SbT_A, except the nodes that have matching nodes in SbT_B:

$$\sum_{deletions} = |SbT_A| - |Comsub - tree(SbT_A, SbT_B)|$$

-Take total number of insertions - insert into SbT_A all nodes of SbT_B, except nodes that have matching nodes in SbT_A:

$$\sum_{insertions} = |SbT_B| - |Comsub - tree(SbT_A, SbT_B)|$$

-Assuming constant unit cost (=1) for node insertion and deletion operations, tree edit distance between SbT_A and SbT_B looks as follows:

$$Dist[|SbT_A|][|SbT_B|] = \sum_{deletions} \times 1 + \sum_{insertions} \times 1 =$$
$$|SbT_A| + |SbT_B| - 2 \times |Comsub - tree(SbT_A, SbT_B)|$$

- resulting:

$$|Comsub - tree(SbT_A, SbT_B)| = \frac{|SbT_A| + |SbT_B| - Dist[|SbT_A|][|SbT_B|]}{2}$$

In figure 2.8, there is a full listing for the algorithm CBS.

Similar to Nierman's approache Tekli's solution use complex precomputation phase (TOC - Tree Edit Operations costs) to calculate $insertTree/deleteTree$ costs.These costs then are exploited via Nierman's main edit distance algorithm instead of $graft/prune$ costs.
TOC consists of three main steps:

1. identifying the common sub-tree commonalities between each pair of non-leaf (sub-tree) in the source and destination trees (T_1 and T_2),assigning tree insert/delete operation costs accordingly.

2. identifying the common sub-tree commonalities between each non-leaf node sub-tree in the source tree (T_1) and the destination tree (T_2) as whole, updating delete tree operation costs.

3. identifying the common sub-tree commonalities between each non-leaf node sub-tree in the destination tree (T_2) and the source tree (T_1) as whole, updating insert tree operation costs.

Values calculated in $Comsub - tree(SbT_i, SbT_j)$ function are also normalized via corresponding tree/sub-tree cardinalities $Max(|SbT_i|, |SbT_j|)$, to be comprised between 0 and 1.

$$\frac{CBS(SbT_i, SbT_j)}{Max(|SbT_i|, |SbT_j|)} = 0$$

```
//input - sub-tree $SbT_A$ and $SbT_B$ in ld-ar representations
CostType CBS(ld-pair[] $SbT_A$, ld-pair[] $SbT_B$){
int M = |$SbT_A$|;
int N = |$SbT_B$|;
CostType Dist [][] = new CostType[0...M][0...N];
Dist[0][0] =0;
for (int i =1; i $\le$ M; i++ ){
  Dist[i][0] = Dist[i-1][0] + $Cost_{del}$($SbT_A$[i]);
}
for (int j =1; j  $\le$ N ; j++){
  Dist[0][j] = Dist[0][j-1] + $Cost_{ins}$($SbT_B$[j]);
}
for (int i =1; i $\le$ M; i++){
  for(int j=1; j $\le$ N; j++){
    CostType $m_1$= $\infty$;
    if( $SbT_A$[i].d == $SbT_B$[j].d && $SbT_A$[i].l = $SbT_B$[j].l){
      $m_1$ = Dist[i-1][j-1];
    }
    CostType $m_2$ = Dist[i-1][j] + $Cost_{del}$($SbT_A$[i]);
    CostType $m_3$ = Dist[i][j-1] + $Cost_{ins}$($SbT_B$[j]);
    Dist[i][j] = min($m_1$,$m_2$,$m_3$);
  }
}


return $\frac{M+N-Dist[M][N]}{2}$;
}
```

Figure 2.8: Common sub-tree Commonalities

When there is no commonality between SbT_i and SbT_j:$CBS(SbT_i, SbT_j) = 0$.

$$\frac{CBS(SbT_i, SbT_j)}{Max(|SbT_i|, |SbT_j|)} = 1$$

When the sub-trees are identical: $CBS(SbT_i, SbT_j) = |SbT_i| = |SbT_j|$.
Then, using normalized commonality, tree operation costs would vary as maximum insert/delete tree cost for sub-tree SbT_i:

$$Cost_{insTree/delTree}(SbT_j) = \sum_{\text{All nodes } x \text{ of } SbT_i} Cost_{ins/del}(x) \times 1$$

And minimum insert/delete tree cost for sub-tree SbT_i:

$$Cost_{insTree/delTree}(SbT_j) = \sum_{\text{All nodes } x \text{ of } SbT_i} Cost_{ins/del}(x) \times \frac{1}{2}$$

In figure 2.9, there is full listing for the algorithm *TOC* phase. The overall complexity is $O(|T_1|\,|T_2|))$ because complexity of precomputation phase limited by $O(|T_1|\,|T_2|))$, same as complexity of the main edit distance algorithm.

```
procedure TOC(Tree $T_1$, Tree $T_2$){
  for each sub-tree $SbT_i \in T_1/(|SbT_i|>1)$
  //excluding leaf nodes in $T_1$
  {
    $Cost_{DelTree}(SbT_i) = \sum_{\text{All nodes } x \text{ of } SbT_i} Cost_{del}(x)$;
  }
  for each sub-tree $SbT_j \in T_2/(|SbT_j|>1)$
  //excluding leaf nodes in $T_2$
  {
    $Cost_{InsTree}(SbT_j) = \sum_{\text{All nodes } x \text{ of } SbT_j} Cost_{ins}(x)$;
    $Cost_{DelTree}(SbT_i)$ =min{
    $Cost_{DelTree}(SbT_j)$,
        $\sum_{\text{All nodes } x \text{ of } SbT_i} Cost_{del}(x) \times \frac{1}{1+\frac{CBS(SbT_i,SbT_j)}{Max(|SbT_i|,|SbT_j|)}}$
    };
    $Cost_{InsTree}(SbT_j)$ = min{
    $Cost_{InsTree}(SbT_j)$,
        $\sum_{\text{All nodes } x \text{ of } SbT_j} Cost_{ins}(x) \times \frac{1}{1+\frac{CBS(SbT_i,SbT_j)}{Max(|SbT_i|,|SbT_j|)}}$
    };
  }
  for each sub-tree $SbT_i \in T_1/(|SbT_i|>1)$
  {
    $Cost_{DelTree}(SbT_i)$ =min{
    $Cost_{DelTree}(SbT_j)$,
        $\sum_{\text{All nodes } x \text{ of } SbT_i} Cost_{del}(x) \times \frac{1}{1+\frac{CBS(SbT_i,T_2)}{Max(|SbT_i|,|T_2|)}}$
    };
  }
  for each sub-tree $SbT_j \in T_2/(|SbT_j|>1)$
  {
    $Cost_{InsTree}(SbT_j)$ = min{
    $Cost_{InsTree}(SbT_j)$,
        $\sum_{\text{All nodes } x \text{ of } SbT_j} Cost_{ins}(x) \times \frac{1}{1+\frac{CBS(T_1,SbT_j)}{Max(|T_1|,|SbT_j|)}}$
    };
  }
}
```

Figure 2.9: Tree operation Cost algorithm

2.5 Decomposition family of TED algorithms.

Previous works restricted tree edit operations on the leaves of the trees or whole sub-trees. In Zhang and Shahsa's work [6] suggested a recursive solution to calculate the tree edit distance between two rooted ordered labeled trees, permitting tree edit operations anywhere in the trees. In fact this algorithm compares forest of trees in contrast of other TED algorithms and become the base for the wide family of decomposition algorithms ,which includes Klein's, Demain's, RTED's approaches and etc. These algorithms are based on the same recursive formula for Tree Edit Distance :

Let F and G be ordered forests,let $v \in F$ and $w \in G$ are either both rightmost or leftmost root nodes of the respective forests, $\delta(F, G)$ we denote edit distance between forests F and G. Then:
$\delta(\varnothing, \varnothing) = 0$,
$\delta(F, \varnothing) = \delta(F - v, \varnothing) + c_d(v)$,
$\delta(\varnothing, G) = \delta(\varnothing, G - w) + c_i(w)$,

If F or G is not a tree:

$$\delta(F,G) = min \begin{cases} \delta(F - v, G) + c_d(v) \\ \delta(F, G - w) + c_i(w) \\ \delta(F_v, G_w) + \delta(F - F_v, G - G_w) \end{cases}$$

If F is a tree and G is a tree:

$$\delta(F,G) = min \begin{cases} \delta(F - v, G) + c_d(v) \\ \delta(F, G - w) + c_i(w) \\ \delta(F - v, G - w) + c_r(v, w) \end{cases}$$

Figure 2.10: Recursive formula for Tree Edit Distance

The space and complexity of straight-forward algorithm, which stores distance between all pairs of subforests of two tree F and G, is $O(|F^2|\,|G^2|))$. The required storage space can be reduced to $O(|F|\,|G|))$ by computing the subproblems bottom-up and reusing storage space. This can be achieved by restricting the choice of the v and w nodes in each recursive step. Different strategies of picking these nodes produce different numbers of the recursive steps that algorithms need to compute in order to calculate tree edit distance between two trees. The subforests that result from decomposing a tree with recursive formula in figure 2.10 are called the $relevant\ subforests$. The set of all subforests that can result from decomposition is called the $full\ decomposition$.

Definition: The $full\ decomposition$ of tree F, $A(F)$, is the set of all subforests of F obtained by recursively removing the leftmost and rightmost root nodes, $r_L(F)$ and $r_R(F)$, from F and the resulting subforests:
$\mathcal{A}(\varnothing) = \varnothing$
$\mathcal{A}(F) = \{F\} \cup \mathcal{A}(F - r_L(F)) \cup \mathcal{A}(F - r_R(F))$

According to [9] a path that connects the root node of the tree to one of its leaves is called $root - leaf\ path$. The set of all root-leaf paths of F is denoted as $y^*(F)$.

There are three types of root-leaf paths used in presented algorithms:

- the left path $y^L(F)$ recursively connect a parent to its leftmost child.
- the right path $y^R(F)$ recursively connect a parent to its rightmost child.
- the heavy path $y^H(F)$ recursively connect a parent to the child which roots the largest sub-tree.

Decomposing trees using root-leaf path is essential for the explanation of the algorithms in this section.
The $relevant\ sub-trees$ of tree F for some root-leaf path y are all sub-trees that result from removing y from F, i.e., all sub-trees of F that are connected to a node on the path y.

Definition: The set of $relevant\ sub-trees$ of a tree F with respect to a root-leaf path $y \in y^*(F)$ is defined as $F - y$.

The $relevant\ subforests$ of F for some root-leaf path is F itself and all subforests obtained by removing nodes.

Definition: The set of $relevant\ subforests$ of a tree F with respect to a root-leaf path $y \in y^*(F)$ is recursively defined as:
$\mathcal{F}(\varnothing, y) = \varnothing$

$$\mathcal{F}(F, y) = \{F\} \cup \begin{cases} \mathcal{F}(F - r_R, y) & \text{if } r_L(F) \in y \\ \mathcal{F}(F - r_L, y) & \text{otherwise} \end{cases}$$

If we assume that there is also defined the root-leaf path for each of the relevant sub-trees of a tree F composition can be continued as follows:

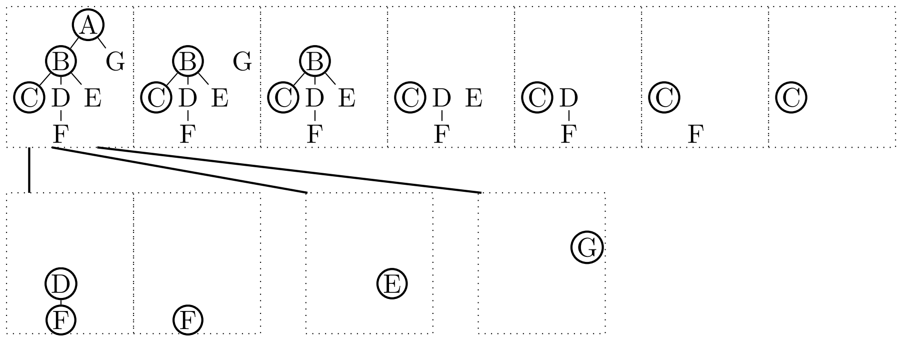
1. produce all relevant subforest of F with respect to some root-leaf path y.
2. recursively apply this procedure to all relevant sub-trees of F with respect to y.

There are examples of different ways to decompose a tree In figure 2.11 .
A decomposition strategy defined in [RTED], chooses between leftmost and rightmost root node at each recursive step, resulting in a set of relevant subforests,.

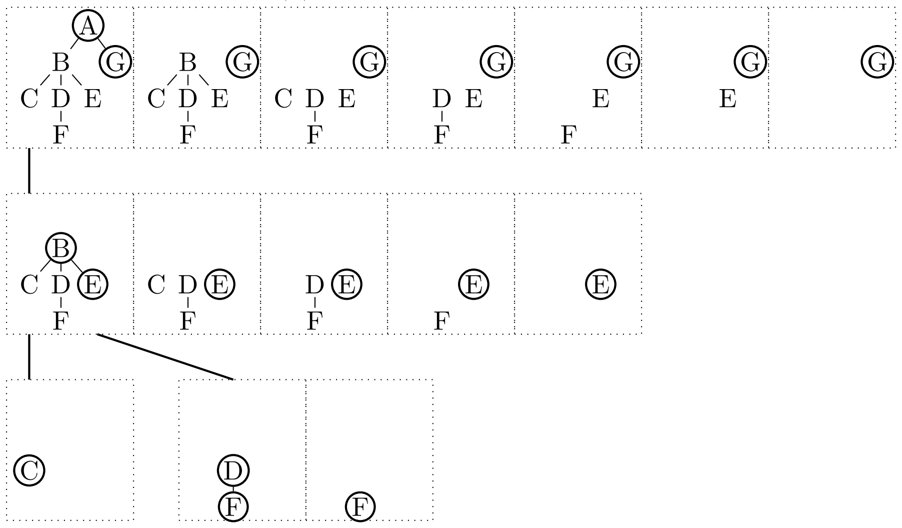
Definition: A *path strategy* S for two trees F and G maps each pair of sub-trees $(F_v, G_w), v \in F, w \in G$ to a root-leaf path in one of the sub-trees, $y^*(F_v) \cup y^*(G_w)$. An *LRH strategy* is a path strategy that maps sub-tree pairs to left, right, and/or heavy paths.

An algorithm based on an LRH strategy is an $LRH\ algorithm$. All algorithms described in this section fall into the class of LRH algorithms.
Zhang-Shasha's decomposition strategy maps all pairs of sub-trees $(F_v, G_w), v \in F, w \in G$, to the left path, $y^L(F_v)$. The symmetric strategy maps all pairs of sub-trees to the right path, $y^R(F_v)$. This strategy use only the root-leaf paths in F for the decomposition. Resulting complexity is $O(m^2n^2)$ and $O(mn)$ for space.
Demaine's approach [5] uses both trees for decomposition and maps all pairs of sub-trees $(F_v, G_w), v \in F, w \in G$ to $^H(Fv)$ if $|F_v| \geq |G_w|$ and to $y^H(G_w)$ otherwise. Thus, in each recursive step, the larger tree is decomposed using the

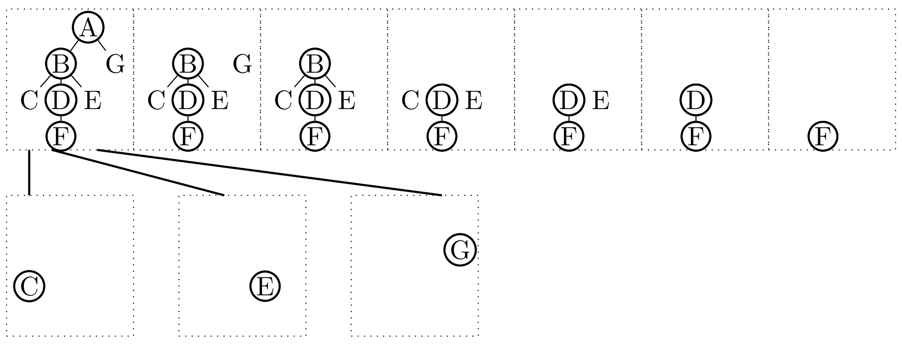
heavy path. This results in $O(n^2m(1+\log\frac{m}{n}))$ time and $O(mn)$ space complexity. This algorithm is also proved to be worst-case optimal.
In Pawlik-Augsten's approach [9], authors use precomputation phase to calculate the most optimal strategy for the choice of nodes at each step (i.e. such strategy that provides lowest number of sub-problems to solve),that results in $O(n^3)$ time and $O(n^2)$ space complexity, which has shown to be optimal among all possible strategies for the recursive formula in figure 2.10.



(a) Left-path decomposition



(b) Right-path decomposition



(c) Heavy-path decomposition

Figure 2.11: Example of decompositions

3. Algorithm implementation and Analysis

3.1 Algorithms analysis

Comparing tree edit distance approaches, to identify the best XSD structural similarity methods, is not a trivial task. Most of the methods for Tree Edit Distance were developed outside of context of XML/XSD trees, thus might produce inappropriate results. So we should compare algorithms on two criteria:

- Quality - accuracy of the XSD structural resemblances detection. It is hard to define the term quality in this context, however we expect that the algorithms will offer a result that will be equal or at least close to the real XSD structural resemblance. The results that differ from the frequent one will be analysed in detail.
- Performance - overall complexity of the algorithms. We expect that compared algorithms would return resulted distance in reasonable time. Also it is important to take into account that the complexity of some algorithms differs and this fact has an impact on the speed although the edit distance results are similar.

The main criteria while choosing the algorithms, was the ability to compute the tree edit distance between two trees. By allowable sets of tree edit distance operations, algorithms could be split in 3 main groups:

- Algorithms that allow insertion and deletion of single nodes anywhere in the tree - RTED, Demain, Zhang-Shasha.
- Algorithms that allow insertion and deletion of single nodes only at leaves of the trees - Selkow, Chawathe.
- Algorithms that allow insertion and deletion of whole sub-trees - Nierman, Tekli.

The main reason to split algorithms in groups is the fact that algorithms in the same group use similar operations while producing the edit distance for two trees A and B i.e.

$selkowEditDistance(A, B) = chawatheEditDistance(A, B)$
$zhangShashEditDistance(A, B) = rtedEditDistance(A, B)$ etc.

There is an exception in the third group, where Nierman and Tekli algorithms utilizes the same edit tree operations but produce slightly different results (edit distance), because Tekli algorithm recalculates the cost of insertion/deletion of sub-trees using Common Sub-tree commonality relations discussed earlier.

Algorithms from the first group (i.e. decomposition algorithms), were not mainly developed in XML/XSD context and thus might provide results (edit scripts and distances) that are not completely suitable for XML data. The reason of such a behavior is that by allowing deleting/inserting node in the middle of trees and moving their children up/down breaks inner logic hierarchy of XSD data. Also, these algorithms in general tend to be more complex (in implementation and asymptotic computational complexity) than algorithms in second and third group. They were developed in context of usual ordered labeled trees that try to detect distance between forests of trees, thus making too much unnecessary work in context of XML documents.
Algorithms from the second group (Selkow, Chawathe), by restricting insert/delete operation to the leaves, made their approaches simpler in both implementation and performance. It has been argued in literature [citation], that targeting leaf nodes is more appropriate, i.e more natural in context of XML documents. Hence, deletion of an internal node in XML hierarchy would require deletion of all children of this node. Equal to insertion, we need to insert this node before the insertion of any its descendant. However, their disadvantage is that these algorithms do not take into account XML document features like repeated, optional elements and sub-trees.
On other hand, algorithms in the third group, were mainly developed to use in practice with XML documents, so they could detect more accurate similarity using knowledge of XML trees (repeated sub-trees for example). Providing operations like insertTree/deleteTree, they have advantage of detecting sub-tree similarities in XML documents. For example inserting multiple identical(Nierman) or similar(Tekli) sub-trees, would result in less operations, provide more suitable edit script and edit distance. Also, these algorithms have similar asymptotic time and space bounds as algorithms in the second group (average $O(N^2)$).It should be noted that Tekli and Nierman produce slightly different edit distance script. Nierman's approach allows inserting/deleting sub-trees with constant cost only if they have the same child-node relations in the target/source trees, while Tekli approach allows inserting/deleting similar sub-trees with cost calculated in precomputation phase using common sub-tree commonalities. This results in the fact that Tekli algorithm provides more accurate result than Nierman's approach.

3.2 Implementation analysis

In this work, we have implemented algorithms described in previous sections:

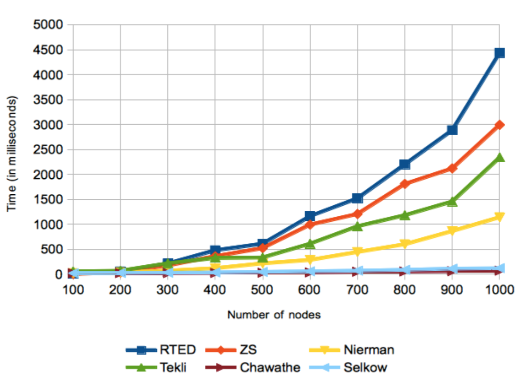
- Selkow's solution.
- Chawathe's.
- Nierman's approach.
- Tekli's approach.
- Zhang-Shasha's algorithm.
- Demaine's approach.

- Robust Tree Edit Distance algorithm.

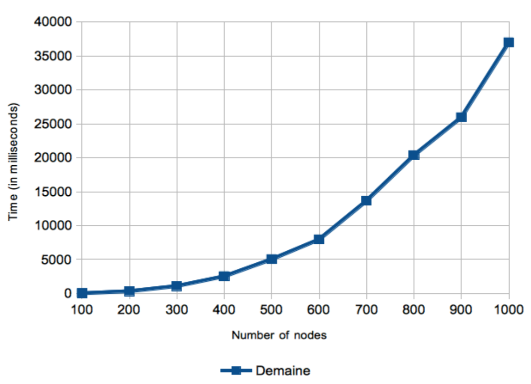
During the implementation phase we have discovered that some algorithms share the same basic edit distance approach; however, with differences in costs of tree edit operation or strategy of computation. Selkow's, Nierman's, Tekli's approaches share the same main edit distance algorithms with differences in the determination of the edit cost operation. While Selkows's approach treats sub-tree insertion/deletion as insertion/deletion of every node in the sub-trees, Nierman's approach determines the cost of sub-tree operations using $containedIn$ relations, while Tekli's algorithm uses common sub-tree similarity relations. These algorithms were implemented using the same basic tree edit distance part with additional corresponding pre-computation phases, where the corresponding costs are computed. Zhang-Shasha, Demaine, and RTED approaches also share the same basic edit distance approach. The only difference in these algorithms is the tree decomposition, as a result in our implementation we use a common edit distance module, which accepts as parameter the strategy of the tree traversal that corresponds to each approach.

3.3 Empirical evaluation

We empirically evaluated these algorithms on both real world and synthetic data-sets (which you can find on the attached CD) and compare the corresponding performance results.
In some similar papers [10], we may find comparison of algorithms using cluster evaluation. We did not implement this evaluation metric because it would require a complex generation XSD from XSD, and is not appropriate for our work.
For synthetic data-set we have generated random XSD documents with size range of 10 to 1000 nodes, with maximum depth of 15 and maximum fanout of 6.
For the real data set we have chosen $xCBL$ (XML Common Business Library) [14], which includes set of XML schema documents for business-to-business e-commerce.



(a) First part



(b) Second part

Figure 3.1: Timing results for random trees

In Figure 3.1 we can see timing results on the synthetic random trees. Each algorithm was tested on sets of documents with number of nodes in range of 100 to 1000. Timing results follows our theory analyse : decomposition algorithms (RTED,Zhang-Shasha,Demaine) shows the worst results. The main reason of this behavior is the fact that these algorithms have higher asymptotic complexity ($O(N^3)$ up to $O(N^4)$). Then goes algorithms from the third group (Nierman,Tekli).These algorithms require complex pre-computation phase to determine operations costs, while still their complexity is at most $O(N^2)$. Selkow and Chawathe algorithms clearly outperforms all other approaches,however they use only the most simple edit operations.

Algorithm	10 documents	20 documents	40 documents
Tekli	690 ms	2303 ms	7799 ms
Nierman	486 ms	854 ms	1890 ms
Selkow	348 ms	529 ms	1699 ms
Chawathe	108 ms	254 ms	1085 ms
RTED	8 s 12 ms	30 s 581 ms	129 s 490 ms
Zhang-Shasha	7 s 152 ms	29 s 499 ms	130 s 925 ms
Chawathe	7 s 127 ms	30 s 265 ms	137 s 268 ms

Figure 3.2: Timing results for real data set self-join

In practice XSD documents rarely reaches such big sizes (like thousand nodes). To test our algorithms on real data, we have selected different sets of random documents (10,20,40 documents) from $xCBL$ database(average size of 4-16 kb), and performed self join on each set T (compare each pair of trees $T_1, T_2 \in T$ and match them if distance ¡ treshold). Table in Figure 3.2 shows the runtime (average of three runs) computed in the join. Tests shows that algorithms that works on trees and sub-trees, clearly outperforms algorithms that works with forest of trees. All three decomposition approaches (RTED,Zhang-Shasha,Demaine) required up to several minutes to compute set of 40 documents, while other algorithm (Nierman,Tekli,Selkow and Chawathe) required only several seconds.

4. Comparison with related works

In this section we will compare our work with other articles that compared different Tree Edit Distance approaches, and works that researched the problem of similarity between XML/XSD documents.

4.1 A survey on tree edit distance and related problems

In this work [13] author compared some central tree edit distance algorithms along with other approaches to comparing trees (like alignment distance, tree inclusion etc.). Among described aproaches are Zhang-Shasha's,Klein's,Chawathe's algorithms that were also discussed in our work. However authors didn't gave any attention to the question of suitabilty these algorithms for the XML documents.

4.2 An overview on XML similarity:background, current trends and future directions

This work [7] focused primary on the problem of XML documents similarity/comparison. Authors made an overview of existing techniques like Tree Edit Distance (TED),Information Retrieval methods(IR) and other structure/content methods. It can be considered as most detailed and extensive work in our topic.They also noticed that some algorithms might yield results that are not completely suitable for the XML documents. In particular,it has been argued that restricting insertion and deletion operation to leaf nodes fits better in context of structural comparison of XML data. Among compared Tree Edit Distance algorithms are Zhang-Shasha's, Chawathe's,Nierman's etc. We also should note that authors of this work developed one of the algorithms that we used in our comparison (we called it Tekli's algorithm).

4.3 XS3: a prototype for XML Document and Grammar similarity evaluation

XS3 prototype [12] is an implementation of low-level algorithms to evaluate similarity between XML documents and grammars. Prototype consists of several parts like parser,validator ,xml generator and similarity evaluation component. Similar to our approach this system converts XML documents into ordered labeled trees and then allows to evaluate structural (Tree Edit Distance based) methods to compare trees. Among imlemented algorithms are Chawathe's, Dalamagas,Nierman's works .Compared to our work XS3 prototype doesn't include implementations of algorithms that allows insertion/deletion anywhere in the tree (like Zhang-Shasha,Demaine etc.), however this system has advantage providing various tools to experiment with similarity evaluation.

5. Conclusion

In our work we have researched, implemented and compared several central Tree Edit distance approaches in order to detect structural similarity of XSD (XML Schema) documents. The results demonstrate the importance of the context in which algorithms were developed. The choice of edit operations is crucial to quality and performance while comparing XML schema.
Early approaches and their extensions that use unrestricted edit model(i.e. allow to edit/insert/delete nodes anywhere in the tree), were widely proposed in the literature, but proved to be less appropriate for the XML context, due their overall complexity and breaking the hierchial logic of the XML/XSD documents. These algorithms have asymptotic complexity rarely less then $O(N^3)$, and our performance tests results shows that they are outperformed by other approaches proposed in our work, and are not very suitable while working with large XSD data sets.
On other hand, algorithms that utilize restricted edit model (Selkow's, Chawathe's) showed much better results, both in quality and performance. They work with XML trees more naturally by applying operations on the leaf nodes. Runtime complexity of the both algorithms sticks to quadratic ($O(N^2)$) by space and time. However, while these algorithms considered as starting point for recent XML tree edit distance approaches, they overlooks certain features of XML documents like frequent presence of the repeated elements/sub-trees, and left these sub-tree similarities unaddressed.
Nierman's and Tekli's approaches were developed primary in XML context. They utilize restricted model of edit distance operations and extends it with additional operations that could insert and delete whole sub-trees. It allows better detection of sub-tree structural similarities in XML/XSD document trees. Our experiments show that these algorithms provide more accurate results when comparing xml trees than other implemented approaches. In terms of performance, both algorithms do not exceed quadratic space and time complexity ($O(N^2)$). These algorithms are conceptually more complex than its predecessors, requiring a precomputation phase in order to determine the cost of the insert/delete operations, but our performance results showed that they are still very fast in use on real data sets (especially while comparing with decomposition algorithms).

Literature

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

[2] A. Nierman and H.V. Jagadish, Evaluating Structural Similarity in XML Documents. *In Proceedings of WebDB*. 2002, 61-66.

[3] S. Chawathe. Comparing Hierarchical Data in External Memory. *In Proc. of the VLDB Conference*, 1990

[4] Stanley M. Selkow: The Tree-to-Tree Editing Problem. *Information Processing Letters* 6(6): 184-186 (1977)

[5] Erik D. Demaine, Shay Mozes, Benjamin Rossman, and Oren Weimann. An optimal decomposition algorithm for tree edit distance. *ACM Trans. Algorithms*, 6:2:12:19, December 2009.

[6] K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM J. Comput.*, 18:12451262, December 1989.

[7] J. Tekli, R. Chbeir, K., Yetongnon: An overview on XML similarity: background, current trends and future directions. *Computer Science Review 3* (2009)

[8] Tekli J. et al., Efficient XML Structural Similarity Detection using Sub-tree Commonalities. *In Proc. of SBBD and SIGMOD DiSC*, Brazil, 2007.

[9] M.Pawlik and N.Augsten. RTED: A Robust Algorithm for the Tree Edit Distance. *PVLDB*. 2011.

[10] T. Dalamagas, T. Cheng, K. Winkel, and T. Sellis. A methodology for clustering XML documents by structure. *Information Systems 31, 3*, (2006) pp.187-228.

[11] J. Wagner and M. Fisher. The String-to-String correction problem. *Journal of the Association of Computing Machinery, 21, 1*, (1974) pp. 168-173.

[12] J. Tekli, R. Chbeir, and K. Ytongnon, "Semantic and Structure Based XML Similarity: The XS3 Prototype", *in Proc. COMAD*, 2006, pp.218-221. http://dbconf.u-bourgogne.fr/XS3/

[13] P. Bille, A survey on tree edit distance and related problems. *In Proceedings of Theor. Comput. Sci.*. 2005, 217-239.

[14] http://www.xcbl.org/