Synthetising Changes in XML Documents as PULs

Federico Cavalieri, Alessandro Solimando, Giovanna Guerrini
Università di Genova, Italy - {name.surname}@unige.it

ABSTRACT

The ability of efficiently detecting changes in XML documents is crucial in many application contexts. If such changes are represented as XQuery Update Pending Update Lists (PULs), they can then be applied on documents using XQuery Update engines, and document management can take advantage of existing composition, inversion, reconciliation approaches developed in the update processing context. The paper presents an XML edit-script generator with the unique characteristic of using PULs as edit-script language and improving the state of the art from both the performance and the generated edit-script quality perspectives.

1. INTRODUCTION

The ability of detecting changes in documents, through the so called *diff* algorithms, is crucial in many data management contexts. Such algorithms take as input two pieces of data and detect the differences between them, so that such differences can be analyzed or employed to tranform data. Various representations can be adopted for differences, usually referred to as *edit-scripts* or *deltas*.

The problem has been widely investigated for flat text files as well as for hierarchically-structured data and XML documents. For flat text files, the popular GNU *diff* utility, based on the *LCS* algorithm for finding the longest common subsequences in strings [15, 16], works well. More and more often, however, information is hierarchically structured and represented in XML. As a consequence, there has been a relevant number of studies devoted to the detection of changes in hierarchically-structured documents in general [7, 9] and in XML documents [11, 13, 19, 20] specifically. Such approaches take the structural properties imposed by the tag nesting into account. They differ in the assumptions on the tree model (*e.g.*, ordered or unordered), in the format of the edit-scripts, in the cost models, in the assumptions on the document contents (*e.g.*, duplicate subtrees are uncommon and can be used to identify reliable matches). Another significant difference in the proposed algorithms is the chosen trade-off between result optimality and algorithm complexity.

The standard language for updating XML documents is the W3C recommendation XQuery Update Facility (XQUF) [18]. The evaluation of an XQUF expression on a document produces an unordered list of atomic update requests, represented as a *Pending Update List* (PUL), that is then applied on the document. A decoupled PUL execution model, by which PULs can be exchanged across machines and applied on a machine different from the one where they have been produced, has been proposed [4, 5].

An XML differencing algorithm is only one of the tools in an XML engineer toolbox. In most contexts where change detection is useful, such as collaborative editing or versioning, other change manipulation operations are needed. Notable examples are change reverting, merging, reconciliation, and composition of a sequence of consecutive changes. Operators for these manipulations on PULs have been proposed in [4, 5] and implemented in the Zorba[1] XQUF engine.

PULs are thus an excellent candidate as edit-script language and we believe that a PUL-based, fast, and efficient algorithm for differencing XML documents is needed. Unfortunately, the XML diff algorithms proposed in the literature cannot be easily adapted due to the set of supported edit primitives (*e.g.*, there is no move operator in PULs, while relabelings and changes at leaves can be detected) or the assumptions they rely on. Another limitation of these sets of edit operations w.r.t. PULs is that the order of update operations of the same kind is not prescribed, and all update operations refer to the original document.

Contributions. The paper introduces PUL-Diff, an approximated diff algorithm, and brings the following contributions: (i) improvement over the state of the art with respect to edit-script quality, differencing time and robustness of the employed heuristics, as experimentally verified in Section 7 and in the extended experimental section [6]; (ii) edit-script expressed as PULs, a formally grounded W3C standard, which can be evaluated by existing XQUF engines and effectively manipulated [3, 4]. As relevant building blocks of the diff algorithm, the paper proposes: (i) an exact and optimal solution to the problem of finding, given a set of two-dimensional weighted points, one of the maximal-weight subsets defining a strictly increasing function, which is used in PUL-Diff to improve the quality of the edit-script; (ii) an adaptation of the *pq*-grams based fanout weighted tree edit-distance estimation to approximate the minimum cost of a PUL transforming a tree into another, that is, the PUL tree edit-distance.

Outline. The paper is organized as follows. Section 2 discusses relevant related work, whereas Section 3 introduces some preliminary notions. Section 4 presents the PUL-Diff algorithm. Section 5 and Section 6 detail its relevant components, namely, the identification of the heaviest increasing point subset and the PUL tree edit-distance, respectively. Section 7 provides an evaluation

[1] `http://www.zorba-xquery.com`

of PUL-Diff (*i.e.*, complexity analysis, experiments, and comparative analysis with state of the art XML differencing algorithms). Finally, Section 8 concludes the paper.

2. RELATED WORK

Given two trees and a set of operations with an associated cost model, the tree edit-distance problem is to compute the minimum-cost sequence of operations that transforms the first tree in the second one. The best known algorithms for the tree edit-distance problem decompose the trees into smaller subtrees and subforests by removing one node after the other. The classical algorithm by Zhang and Shasha [21] runs in $O(n^4)$ time, whereas the proposal of Demaine *et al.* [12] is worst-case optimal and runs in $O(n^3)$. The runtime behavior of these algorithms heavily depends on the tree shapes and on the employed decomposition strategy. At each decomposition step, they rely on a fixed strategy to choose which node to remove. Pawlik and Augsten [17] dynamically and optimally choose a different strategy. The proposed algorithm runtime complexity is in $O(n^3)$ and outperforms the previous proposals. Given the high computational cost of the exact approaches, many approximated algorithms have been proposed.

The MH-Diff algorithm [8], targeting unordered trees, is a heuristic-based approach, where the edit-distance problem is reduced to a minimum cost edge cover in a bipartite graph. Each operation is given a user-defined fixed cost, except for the relabeling operation that employs a user-provided function that compares the values of two nodes. XyDiff [11] computes hash values for all the subtrees of the analyzed trees using DOMHASH[2], an efficient hashing function specifically tailored for XML subtrees. XyDiff then searches for exact matches in a bottom-up traversal and eagerly tries to expand them looking for common ancestors for the two trees, relying on the node names. This behavior helps detecting renaming, but using hash values and bottom-up traversal prevents the detection of changes in the leaves, a very frequent kind of update. Matches are then improved through a top-down visit of the two trees, trying to propagate existing matches. X-Diff addresses change detection for XML documents modeled as unordered trees [19]. It integrates key XML structure characteristics with a standard edit-distance technique, thus resulting in an efficient algorithm. The algorithm computes hashes similarly to XyDiff but with XHash, that is suited to commutative models. This algorithm considers our set of primitive operations. Kf-Diff [20] achieves linear time-complexity transforming tree-to-tree correction problem into tree comparison without duplicated paths. It supports both ordered and unordered trees. The algorithm is label-based and thus not robust against renaming. To the best of our knowledge there are no proposals tailored to W3C's PULs.

Augsten *et al.* [2] propose an estimation of the tree edit-distance for ordered trees based on *pq*-grams, subtrees of a fixed shape corresponding to the substrings (called *q*-grams) used for string similarity evaluation. *pq*-grams are composed by a stem made of *p* elements (bound by the parent-child relation) and a base of *q* consecutive siblings: the last element of the stem is named anchor node. A tree is therefore represented by its set of *pq*-grams like a string is represented by its *q*-grams. We propose an adaptation of *pq*-grams where in each *pq*-gram the nesting depth of its anchor node is included (in order to reflect the absence of a move operation in PULs), and where different *pq*-grams for node names and values are used (not to overestimate PUL tree edit-distance when nodes have different names or values). The absence of a move primitive also requires a method for removing matches that would be expressed through this operation. This problem can be reduced to a generalization of the *Heaviest Increasing Subsequence* (HIS) problem [14]. Such generalized *Heaviest Increasing Point Subset* (HIPS) problem aims at finding, given a set of two-dimensional weighted points, one of the maximal-weight subsets defining a strictly increasing function.

3. PRELIMINARIES

In this section, we introduce the definitions and concepts used in the remainder of the paper. Specifically, the tree representation of XML documents, the XQUF language, and its dynamic representation of updates (*i.e.*, PULs) are introduced.

3.1 Tree Representation of XML Documents

The data model of XQuery is XDM[3], which describes all permissible values of expressions in the language. Every instance of the data model is a sequence, that is, an ordered collection of zero or more items. An item is either a node or an atomic value. Nodes can be nested to form a tree and are of different kinds. For brevity we consider only document, element, and text nodes. Document nodes do not have a parent, while only element and document nodes may have children. The textual contents of elements are modeled by separate text nodes. A *document* is a tree whose root node is a document node. Given a tree T, we denote its root as $\mathcal{R}(T)$, its nodes as $\mathcal{V}(T)$, and its weight, *i.e.*, the number of its nodes, as $\Omega(T)$. The weight of a node v, denoted as $\Omega(v)$, is equal to the weight of the subtree rooted at that node, denoted as $\mathcal{T}(v)$. The weight of a list of trees L, denoted as $\Omega(L)$, is equal to the sum of the tree weights. The empty list is denoted as Λ. Given a node $v \in \mathcal{V}(T)$, we denote its children as $\gamma_E(v)$, its ancestors as $\mathcal{A}(v)$, and its parent as $\mathcal{P}(v)$. The $<_p$ relation corresponds to the document order, that is, the ordering of nodes in the XML serialization of a document.

3.2 XQUF and PULs

An XQuery expression is evaluated on one or more XDM instances and returns an XDM instance. Its evaluation does not change the state (that is, the parent, children, name, or value) of any node. XQUF extends the language by introducing a new kind of expressions, named *updating expressions*, that can alter the state of nodes, and a set of low level *update operations* representing a node state change. The evaluation of an updating expression results in a single *pending update list* (PUL), that is, an unordered list of update operations. Two phases can thus be identified in the evaluation of an XQUF expression: (i) *production*, in which the expression is evaluated producing a PUL; (ii) *application*, in which the update operations in the produced PUL are applied. The application order is dictated by the operation kind and is specified in [18]. Let v and v' be nodes, $L = [T_1, \ldots, T_k]$ be a list of trees. The PUL node update operations are the following: (i) $\texttt{ins}^{\leftarrow}(v, L)$, $\texttt{ins}^{\rightarrow}(v, L)$, $\texttt{ins}^{\swarrow}(v, L)$, $\texttt{ins}^{\searrow}(v, L)$ insert the tree sequence L before/after or as first/last child of v, respectively; (ii) $\texttt{ins}^{\downarrow}(v, L)$ inserts the tree sequence L as child of node v, in an implementation dependent position; (iii) $\texttt{del}(v)$ detaches node v from its parent (if any); (iv) $\texttt{repN}(v, L)$ replaces node v with the tree sequence L; (v) $\texttt{repV}(v, s)$ replaces the value of node v with the value s; (vi) $\texttt{repC}(v, v')$ replaces the children of node v with the optional text node v'; (vii) $\texttt{ren}(v, n)$ renames node v as n. We do not consider `put` operation, as it is not a node update operation, and `insertAttributes` operation, as we do not consider node attributes. We assign to each operation a fixed unitary cost, to which we sum the weight of the removed/inserted nodes.

[2]`http://www.ietf.org/rfc/rfc2803.txt`

[3]`http://www.w3.org/TR/xpath-datamodel`

Definition 1 (Operation Cost). *Let op be an operation. Its cost, denoted as $\xi(op)$, is:*

$$\xi(op) = \begin{cases} 1 + \Omega(L) & \text{if } op = ins^d(v, L), d \in \{^{\leftarrow}, ^{\rightarrow}, ^{\swarrow}, ^{\searrow}, ^{\downarrow}\} \\ 1 + \Omega(v) & \text{if } op = \mathtt{del}(v) \\ 1 & \text{if } op \in \{\mathtt{repV}(v, s), \mathtt{ren}(v, n)\} \\ 1 + \Omega(v) + \Omega(L) & \text{if } op = \mathtt{repN}(v, L) \\ \Omega(v) + \Omega(v') & \text{if } op = \mathtt{repC}(v, v'), \\ & \text{where } \Omega(\Lambda) = 0 \end{cases}$$

△

The cost of a PUL Δ, denoted as $\xi(\Delta)$, is the sum of the costs of its operations.

3.3 Matches and Weights

The PUL-Diff algorithm (presented in Algorithm 1) compares two XML documents, named source (resp. target), and denoted as S (resp. T). Subscripts s and t denote nodes in the source and target trees, respectively, and are omitted when no confusion arises. This comparison has one main goal: decide which subtrees of the source document should be considered deleted (*i.e.*, present in the source document but not in the target one), which should be considered inserted (*i.e.*, present in the target document but not in the source one), and which should be considered *matching*, that is, present in both documents with marginal or no changes. In the former case the edit-script expresses the transformation of these matching trees.

Definition 2 (Match and Complete Match). *A match m is a pair $\langle s, t\rangle$, where $s \in \mathcal{V}(S)$ and $t \in \mathcal{V}(T)$. A match m is complete if $\mathcal{T}(s)$ is identical to $\mathcal{T}(t)$, partial otherwise.* △

Intuitively, the weight of a match $\langle s, t\rangle$ measures the advantage, in terms of edit-script cost, of considering the two nodes to match over considering them not to match. The match weight thus ranges from $1 + \Omega(s) + \Omega(t)$ when the two subtrees are identical (no operation required if we consider them to match, $\mathtt{repN}(s, t)$ if we consider them not to match) to 0 when s and t are so different that the most cost-efficient transformation is $\mathtt{repN}(s, t)$.

Definition 3 (Match Weight). *The weight of a match $\langle s, t\rangle$, denoted as $\psi(\langle s, t\rangle)$, is defined as $1 + \Omega(s) + \Omega(t) - \xi(\Delta)$, where Δ is a minimum cost edit-script transforming $\mathcal{T}(s)$ in $\mathcal{T}(t)$.* △

Given a set of matches M, function ψ returns the sum of the weights of all the matches in M. Let $\langle s, t\rangle$ be a match. Since no move operation is allowed in a PUL, it may not be possible to transform $\mathcal{T}(s)$ into $\mathcal{T}(t)$, without removing $\mathcal{T}(s)$ and inserting $\mathcal{T}(t)$. If we consider each match in isolation this situation happens whenever s and t are at different nesting depths in the respective trees.

Definition 4 (Valid Match). *A match $\langle s, t\rangle$ is valid iff s and t are at the same nesting depth.* △

If we consider a set of matches, we need to ensure that for each pair of matches the two source and target nodes are in the same structural relationship.

Definition 5 (Compatible Matches). *Let $\langle q, r\rangle$ and $\langle s, t\rangle$ be two valid matches. $\langle q, r\rangle$ and $\langle s, t\rangle$ are compatible iff all of the following conditions holds: (i) $(q <_p s \iff r <_p t)$, (ii) $(q \in \mathcal{A}(s) \iff r \in \mathcal{A}(t))$, (iii) $(\mathcal{P}(q) = \mathcal{P}(s) \iff \mathcal{P}(r) = \mathcal{P}(t))$.* △

Example 1. *Let $A = a(b(c, d), e(f, g))$ and $B = h(i(j, k))$ be two trees represented as terms (i.e., A has an a labeled root node, with b and e labeled children; node b has in turn c and d labeled children). $\langle c, j\rangle$ and $\langle d, k\rangle$ are compatible matches, whereas $\langle c, k\rangle$ and $\langle d, j\rangle$ are not.* ○

Definition 6 (Consistent Match Set). *A set M of matches is consistent iff all its elements are pair-wise compatible.* △

Algorithm 1 PUL-Diff Algorithm

```
1: function PUL-Diff ($S$,$T$)
2:     $matches \leftarrow \emptyset$
3:     identicalSubtreeMatching($\mathcal{R}(T)$, ref $matches$)
4:     if $matches[\mathcal{R}(T)] \neq \Lambda$ then
5:         return $\emptyset$
6:     end if
7:     if $matches = \emptyset$ then
8:         $matches[\mathcal{R}(T)] \leftarrow \{\langle \mathcal{R}(S), \mathcal{R}(T), 0, False\rangle\}$
9:     else
10:        refineBottomUp($\mathcal{R}(T)$, ref $matches$)
11:    end if
12:    refineTopDown($\mathcal{R}(S)$,$\mathcal{R}(T)$, $matches$, ref $matches$)
13:    if $matches = \emptyset$ then
14:        return {repN($\mathcal{R}(S)$,$\mathcal{R}(T)$)}
15:    end if
16:    return generateEditScript($\mathcal{R}(S)$, $\mathcal{R}(T)$, $matches$)
17: end function
```

4. THE PUL-Diff ALGORITHM

PUL-Diff compares a source and a target document and produces an edit-script that reflects the difference between them. The edit-script can then be used to transform the source document into the target one. In the edit-script, the subtrees which are present in only one of the two documents will be either inserted or deleted, while similar matching subtrees will be transformed one into the other (*e.g.*, by means of renaming operations). Identical matched subtrees, instead, will not require any modification. Since PULs do not allow us to "move" subtrees without deleting and re-inserting them, not all the identical or similar subtrees of the two documents should be matched. PUL-Diff first detects and matches the similar or identical subtrees of the two documents and then decide which matches should be kept and which should be discarded to obtain a consistent match set. To guide this decision, with the objective of reducing the edit-script cost, we rely on the match weight. PUL-Diff is structured in four subsequent stages: (i) identical subtree matching, (ii) bottom-up refinement, (iii) top-down refinement, and (iv) edit-script generation. Ideally, the goal of the first three stages is to detect the heaviest consistent subset of the set of all possible matches $M = \{\langle s, t\rangle \mid s \in \mathcal{V}(S), t \in \mathcal{V}(T)\}$, that is, detect the best possible matches between the source and target trees. Since this computation would be extremely expensive we employ the following heuristics. In the identical subtree matching stage (Section 4.1), the algorithm looks for complete valid matches. In this stage, we only consider subtrees appearing once in the source document. The purpose of the bottom-up refinement stage (Section 4.2) is to detect the best suitable match for any unmatched target node having at least one matched descendant, while ensuring that all detected matches are compatible. In the top-down refinement stage (Section 4.3) the algorithm tries to match unmatched descendants of matched target nodes. In the edit-script generation stage (Section 4.4) the two trees are visited bottom-up generating an edit-script according to the previously detected matches.

The detected matches are stored in a map named *matches*, which associates nodes of the target tree with their matches. Since match weights will often be estimated, the representation of a match $\langle s, t\rangle$ (Definition 2) is extended to a quadruple $\langle s, t, w, c\rangle$, where w is an estimation of $\psi(\langle s, t\rangle)$ and $c \in \{True, False\}$ specifies whether the match is complete or not.

In the presented algorithms the `ref` keyword denotes function arguments passed by reference, both in function calls and within function signatures. In what follows we detail the four stages composing the PUL-Diff algorithm.

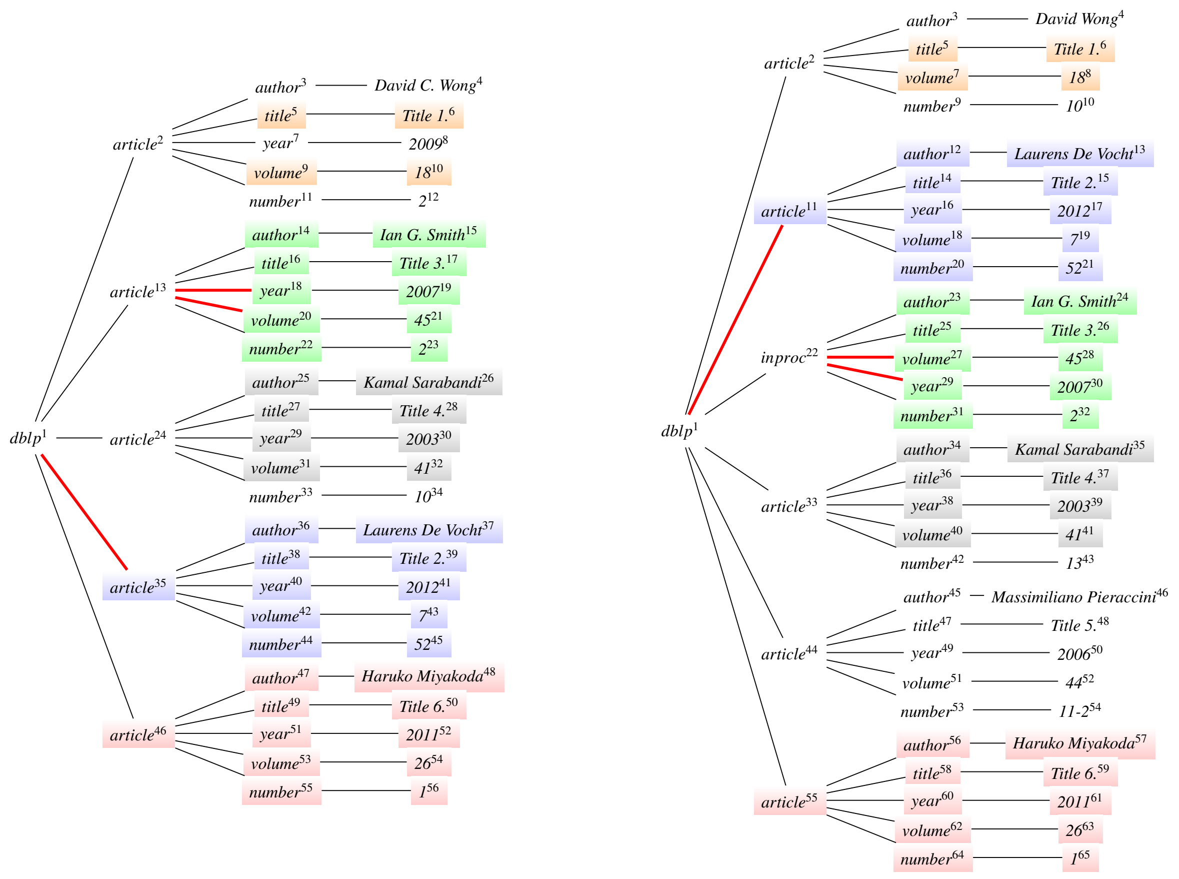


Figure 1: Example source (left) and target (right) trees. We highlight with the same color corresponding parts of the two documents.

Algorithm 2 Identical Subtree Matching

```
function IDENTICALSUBTREEMATCHING($t$, ref $matches$)
    $candidates \leftarrow$ identicalST($t$)
    if $|candidates| = 1$ then
        $matches[t] \leftarrow \langle candidates[0], t, 1 + \Omega(s) + \Omega(t), True \rangle$
    else if $|candidates| = 0$ then
        for each $d$ in $\gamma_E(t)$ do
            identicalSubtreeMatching($d$, ref $matches$)
        end for
    end if
end function
```

4.1 Identical Subtree Matching

The identical subtree matching stage, described in Algorithm 2, matches identical subtrees of the two documents. These matches will be used to limit the matches search effort in the following stages. For this reason we do not consider significant (*i.e.*, we do not add them to the matches map) those matches involving a source subtree that is repeated in the source document. Tree comparison is hash-based. Before the first invocation of Algorithm 2, we compute, for each node n in the source or target tree, a DOMHASH-like signature that identifies the subtree rooted at n and the nesting depth of n. Given a node t, the function `identicalST`(t) returns each node s of the source tree which is at the same nesting depth of t and whose subtree is identical to the subtree of t.

EXAMPLE 2. *Consider the source and target trees in Figure 1. The matches detected during the first stage of the algorithm are reported in Table 1. Note that no matches have been detected for node number$_t^{31}$, since its subtree is repeated in the source tree.* ○

4.2 Bottom-up Refinement

The bottom-up refinement stage, detailed in Algorithm 3, solves all incompatibilities among the matches detected in the previous stage and tries to match all unmatched target nodes with at least one matched descendant.

Specifically, we perform a bottom-up visit of the target tree and for every visited unmatched node t we check whether any of its children has been matched. In case none has been matched, we do not have enough information to propose a match for t at this stage. Otherwise, we consider the parent of each node matching one of the children of t to be a candidate for matching t. The algorithm then chooses one of the candidates, s, and produces a match between s and t, discarding all matches among children of other candidates and children of t. Moreover, the algorithm may also discard some of the matches among children of s and children of t, if some of them are incompatible.

Choosing the best candidate for matching t is crucial for the quality of the generated edit-script. Ideally, we would like to compare all matches among t and each candidate and choose the match with the highest weight. However, since determining match weights requires to solve the exact tree edit-distance problem, we need to approximate its computation. A first approximation is to determine, for each candidate node $cand$, the set of matches $C_{cand} = \{\langle c, d, w, r\rangle \mid \langle c, d, w, r\rangle \in matches \wedge c \in \gamma_E(cand) \wedge d \in \gamma_E(t)\}$ among its children and the children of t, and to consider the sum $W_{cand} = \sum_{\langle c,d,w,r\rangle \in C_{cand}} w$ of all the weights of the matches in C_{cand} as an estimation of the match weight between $cand$ and t. This approximation follows the observation that, if we consider $cand$ and t

Table 1: Summary of the matches detected during the three stages of PUL-Diff algorithm, when applied to source (left) and target (right) trees of Figure 1. The first column lists all target nodes, whereas the second column reports their nesting depth. The remaining columns contain the matched nodes, as detected by the PUL-Diff at the end of each of the first three stages. The ↑ symbol denotes that an ancestor of the node has a complete match

Target node	Depth	Stage 1 matches	Stage 2 matches	Stage 3 matches
$dblp^1$	0		$dblp^1$	$dblp^1$
$article^2$	1		$article^2$	$article^2$
$author^3$	2			$author^3$
$title^5$	2	$title^5$	$title^5$	$title^5$
$volume^7$	2	$volume^9$	$volume^9$	$volume^9$
$number^9$	2	$number^{33}$		$number^{11}$
$article^{11}$	1	$article^{35}$		
$author^{12}$	2	↑		
$title^{14}$	2	↑		
$year^{16}$	2	↑		
$volume^{18}$	2	↑		
$number^{20}$	2	↑		
$inproc^{22}$	1		$article^{13}$	$article^{13}$
$author^{23}$	2	$author^{14}$	$author^{14}$	$author^{14}$
$title^{25}$	2	$title^{16}$	$title^{16}$	$title^{16}$
$volume^{27}$	2	$volume^{20}$	$volume^{20}$	$volume^{20}$
$year^{29}$	2	$year^{18}$		
$number^{31}$	2			$number^{22}$
$article^{33}$	1		$article^{24}$	$article^{24}$
$author^{34}$	2	$author^{25}$	$author^{25}$	$author^{25}$
$title^{36}$	2	$title^{27}$	$title^{27}$	$title^{27}$
$year^{38}$	2	$year^{29}$	$year^{29}$	$year^{29}$
$volume^{40}$	2	$volume^{31}$	$volume^{31}$	$volume^{31}$
$number^{42}$	2			$number^{33}$
$article^{44}$	1			
$author^{45}$	2			
$title^{47}$	2			
$year^{49}$	2			
$volume^{51}$	2			
$number^{53}$	2			
$article^{55}$	1	$article^{46}$	$article^{46}$	$article^{46}$
$author^{56}$	2	↑	↑	↑
$title^{58}$	2	↑	↑	↑
$year^{60}$	2	↑	↑	↑
$volume^{62}$	2	↑	↑	↑
$number^{64}$	2	↑	↑	↑

to be matching, all matches of the children of t that are not in C_{cand} would be discarded, while at least one of the matches among the children of $cand$ and children of t would be kept. The main limitation of this estimation is that incompatible matches may be present in C_{cand} and, if this happens, the real match weight of $cand$ and t could be lower than estimated, increasing the chance of choosing a suboptimal candidate. We refine the approximation considering the weight of one of the heaviest consistent subset HC_{cand} of C_{cand}. This problem can be solved using the HIPS algorithm, as we detail in Section 5. Then we select the candidate s whose heaviest consistent subset has the highest weight (function `getBest` at line 12) and match s with t. Finally, all matches in $C_{cand} \setminus HC_{cand}$ and all matches among the children of other candidates and children of t are removed, since they are incompatible with the match of s and t.

Example 3. *Consider the source and target tree in Figure 1 and the detected matches reported in Table 1. In the bottom-up refinement stage, new matches are detected for the target nodes $dblp_t^1$, $article_t^2$, $inproc_t^{22}$ and $article_t^{33}$, whereas the match of $number_t^9$ and $year_t^{29}$ are discarded. Consider node $article_t^2$. Two candidate nodes are identified: $article_s^{24}$ (due to the match $\langle number_s^{33}, number_t^9, 2, True\rangle$) and $article_s^2$ (due to the matches $\langle title_s^5, title_t^5, 2, True\rangle$ and $\langle volume_s^9, volume_t^7, 2, True\rangle$). The algorithm matches $article_s^2$ and $article_t^2$ and removes the match $\langle number_s^{33}, number_t^9, 2, True\rangle$). Consider now $inproc_t^{22}$. A single candidate is identified ($article_s^{13}$). The set of matches among the children of node $inproc_t^{22}$ and the children of node $article_s^{13}$ however is not consistent as $\langle year_s^{18}, year_t^{29}, 2, True\rangle$ and $\langle volume_s^{20}, volume_t^{27}, 2, True\rangle$ are not compatible. The algorithm matches $article_s^{13}$ and $inproc_t^{22}$, discarding the match $\langle year_s^{18}, year_t^{29}, 2, True\rangle$.* ○

Algorithm 3 Refine Bottom-Up

```
1: function refineBottomUp($t$, ref $matches$)
2:     for each $d$ in $\gamma_E(t)$ do
3:         if $d$ is not leaf AND $matches[d] = \Lambda$ then
4:             refineBottomUp($d$, $matches$)
5:         end if
6:     end for
7:     $candidates \leftarrow \emptyset$
8:     for each $cand$ in $\{\mathcal{P}(c) | \langle c, d, w, r\rangle \in matches \wedge d \in \gamma_E(t)\}$ do
9:         $C_{cand} \leftarrow \{\langle c, d, w, r\rangle | \langle c, d, w, r\rangle \in matches \wedge \mathcal{P}(c) = cand \wedge d \in \gamma_E(t)\}$
10:        $candidates$.add($\langle cand, C_{cand}$, HIPS($C_{cand}$)$\rangle$)
11:    end for
12:    $\langle s, C_s, HC_s\rangle \leftarrow candidates$.getBest()
13:    $matches[t] \leftarrow \langle s, t, \psi(HC_s), False\rangle$
14:    $matches \leftarrow matches \setminus (C_s \setminus HC_s)$
15:    $matches \leftarrow matches \setminus \{\langle c, d, w, r\rangle | \langle c, d, w, r\rangle \in matches \wedge \mathcal{P}(d) = t \wedge \mathcal{P}(c) \neq s\}$
16: end function
```

4.3 Top-down Refinement

The top-down refinement stage, detailed in Algorithm 4, aims at improving the matches of the descendants of partially matched target nodes. Differently from the identical subtree matching stage, we look for source subtrees that are similar or identical (even if they occur more than once in the source document).

Specifically, the top-down refinement works as follows. The target tree is visited top-down and for each visited node t, which is partially matched with a node s, the algorithm tries to improve the matches among the children of s and the children of t. Ideally, we would like to consider the set $M = \{\langle c, d\rangle \mid \langle c, d\rangle \wedge c \in \gamma_E(s) \wedge d \in \gamma_E(t)\}$ of all possible matches among the children of s and those of t and to identify the heaviest consistent subset HM of M, updating the *matches* set accordingly (that is, if we discard a match among two nodes s and t, we also discard any match among their descendants).

While, as discussed in Section 4.2, identifying one of the heaviest consistent subset of a given set of matches is an efficient operation, the size of M, that is, $(|\gamma_E(s)| * |\gamma_E(t)|)$, could be significant. Thus, we consider a smaller set of candidate matches $candM$ which contains, for each child d of t, (i) the match of d in $matches$, if any; (ii) all complete matches between d and a child of s; (iii) the k best matches among d and a subset of the children of s, produced by *similarityMatches* method. Section 6, in what follows, is devoted to the complex subproblems of reducing the number of comparisons for computing the top-k matches and of estimating the tree edit-distance cost. Once the candidate matches set has been identified, we determine one of its heaviest consistent subsets, updating the *matches* map accordingly.

Example 4. *Consider the source and target trees in Figure 1 and the detected matches reported in Table 1. In the top-down refinement stage, the target nodes $author_t^3$, $number_t^9$, $number_t^{31}$, $number_t^{42}$ are matched with source nodes $author_s^3$, $number_s^{11}$, $number_s^{22}$, and $number_s^{33}$.* ○

Algorithm 4 Refine Top-Down

```
function refineTopDown($s$, $t$, $origMatches$, ref $matches$)
    $currM \leftarrow \{\langle p,q,w,r\rangle \mid \langle p,q,w,r\rangle \in matches \wedge q \in \gamma_E(t)\}$
    $candM \leftarrow \{\langle p,q,\Omega(t),True\rangle \mid q \in \gamma_E(t) \wedge p \in$ identicalST$(q) \wedge \mathcal{P}(p) = s\} \cup currM \cup$ similarityMatches$(s,t,currM)$
    $matches \leftarrow matches \cup$ HIPS$(candM) \setminus (currM \setminus$ HIPS$(candM))$
    for each $d$ in $\gamma_E(t)$ do
        $currMatch \leftarrow matches[d]$
        $origMatch \leftarrow origMatches[d]$
        if $currMatch \neq \langle\Lambda,\Lambda,\Lambda,\Lambda\rangle$ AND $!currMatch.c$ then
            if $!origMatch.c$ AND $origMatch.s \neq currMatch.s$ then
                $matches \leftarrow matches \setminus \{\langle p,q,w,r\rangle | \langle p,q,w,r\rangle \in matches \wedge q \in \mathcal{D}(d)\}$
            end if
            if $\gamma_E(currMatch.s) \neq []$ OR $\gamma_E(d) \neq []$ then
                refineTopDown$(currMatch.s, d, origMatches,$ ref $matches)$
            end if
        end if
    end for
end function
```

Algorithm 5 Edit-script Generation

```
function generateEditScript($s, t, matches$)
    $repES \leftarrow \{$repN$(s, \mathcal{T}(t))\}$
    $transES \leftarrow$ compareNode$(s,t)$
    for each unmatched node $c$ in $\gamma_E(s)$ do
        $transES \leftarrow transES \cup$ del$(c))$
    end for
    for each partially matched node $d$ in $\gamma_E(t)$ do
        $transES \leftarrow transES \cup$ generateEditScript$(matches[d].s, matches[d].t, matches)$
    end for
    for each unmatched node $u$ in $\gamma_E(t)$ do
        $transES \leftarrow transES \cup$ generateInsert$(u)$
    end for
    if $\xi(repES) \geq \xi(transES)$ then
        return $transES$
    else
        return $repES$
    end if
end function
```

4.4 Edit-script Generation

The last stage of the PUL-Diff algorithm is the edit-script generation. Since matches are propagated towards the root in the bottom-up refinement stage, the two tree roots are partially matched whenever at least one of their descendant is matched. Algorithm 5 visits the partially matched nodes bottom-up and for each visited pair of partially matched nodes s and t, the algorithm contrasts the cost of replacing $\mathcal{T}(s)$ with $\mathcal{T}(t)$, and the cost of transforming through (`ren`, `repV`, `ins`$^{\leftarrow}$, `repC`, `ins`$^{\searrow}$, `del`) operations $\mathcal{T}(s)$ in $\mathcal{T}(t)$ according to the identified matches. The less expensive alternative is chosen. Intuitively, the cost of the transformation will decrease as more and more (valid and consistent) matches are identified among the nodes in $\mathcal{T}(s)$ and those in $\mathcal{T}(t)$.

Specifically the transformation edit-script is generated as follows. First, the value and name of s and t are contrasted by means of the `compareNode` function, generating a `repV` or a `ren` operation as required. Then, a `del` (resp., `ins`$^{\leftarrow}$/`ins`$^{\searrow}$) operation is generated for every unmatched children of s (resp., t). Insertion operations are produced by means of the `generateInsert` function. Finally, the algorithm is recursively called on all the partially matched children of t. Note that, to further reduce the edit-script cost and produce deterministic edit-scripts, we might merge some operations (*e.g.*, whenever adjacent siblings have to be inserted, a single insertion or node replacement is generated).

Example 5. *Consider the source and target trees in Figure 1 and the detected matches reported in Table 1. The following edit-script is generated:*

`repV`($author^3$, '`David Wong`'),
`del`($year^7$),
`repV`($number^{11}$, '`10`'),
`ins`$^{\leftarrow}$($article^{13}$, `<article><author>Laurens De Vocht</author>`
`<title>Title 2.</title><year>2012</year><volume>7</volume>`
`<number>52</number></article>`),
`ren`($article^{13}$, '`inproc`'),
`ins`$^{\leftarrow}$($number^{22}$, `<year>2007</year>`),
`del`($year^{18}$),
`repV`($number^{33}$, '`2`'),
`repN`($article^{35}$, `<article><author>Massimiliano Pieraccini</author>`
`<title>Title 5.</title><year>2006</year><volume>44</volume>`
`<number>11 - 2</number></article>`) ○

5. HIPS: HEAVIEST INCREASING POINT SUBSET

In the bottom-up refinement stage, we want to determine the heaviest consistent subset of a set of matches, where all the source and target nodes share the same parent. We map this problem into the problem of identifying, starting from a set of two-dimensional weighted points M, one of the heaviest subsets of M that defines a strictly increasing function. Specifically, we represent each match $\langle c_i, d_j, w, r\rangle$ in C_{cand}, where i and j denote the index of c_i and d_j in the respective sibling sequences, as a weighted point $\langle i, j, w\rangle$. Since all matched source nodes and all matched target nodes are siblings in their respective trees, incompatibility (Definition 5) can be reformulated as follows.

Definition 7 (Compatible Matches). *Let $m = \langle i, j, w\rangle$ and $m' = \langle i', j', w'\rangle$ be two matches represented by means of weighted points. We say that m and m' are compatible iff either $i > i' \wedge j > j'$ or $i < i' \wedge j < j'$.* △

In the remainder of the section, we first present the problem statement along with some basic definitions and notations used in what follows, then we provide an exact $O(|M| \log |M|)$ algorithm along with the related correctness and complexity proofs.

5.1 Problem Statement

Given a sequence σ over a linearly ordered alphabet Σ, finding one of the longest subsequences of σ that is strictly increasing is known as the *Longest Increasing Subsequence* (LIS) problem. Jacobson and Vo [14] generalize the LIS problem to weighted sequences, defining the *Heaviest Increasing Subsequence* (HIS) problem. Given a sequence σ over Σ and a weight function defined on the symbols and their positions in σ, the HIS problem is to find one of the subsequences of σ with the highest sum of weights. They propose an optimal algorithm with time complexity in $O(n \log n)$.

Let M be a set of matches, represented as points. Assuming that the x component of each match is unique, we can determine the heaviest consistent subset of M by applying the HIS algorithm on the sequence consisting of the y components of the matches in M in increasing order of their x component. Since multiple matches can be present for the same source node, in this section we tackle a generalization of the HIS problem, the *Heaviest Increasing Point Subset* (HIPS), where, for each position in the sequence, multiple mutually exclusive alternatives may be present. For sake of clarity, we use the point-based notation, where a point p is a triple $\langle x, y, w\rangle$, where the x and y component of each point are chosen from two linearly ordered alphabet X and Y, respectively, and w denotes the point weight. We use $p.x$, $p.y$, and $p.w$ to denote the point components. The weight of a set of points M, denoted as $\psi(M)$, consists of the sum of the weights of its elements. Specifically, given a set M of points, the HIPS problem is to identify one of the heaviest subsets of M that defines a strictly increasing function from X to Y. More formally, a point set M is *increasing* iff

$\forall p_1, p_2 \in M, p_1 \neq p_2 \wedge (p_1.x > p_2.x \iff p_1.y > p_2.y) \wedge (p_1.x < p_2.x \iff p_1.y < p_2.y)$.

Definition 8 (Heaviest Increasing Point Subset). *Let M be a point set. An increasing subset H of M is a heaviest increasing point subset of M if no other increasing subset of M has a higher weight.* △

Let $\sigma = \sigma_1\sigma_2 \ldots \sigma_n$ be a point sequence. Following the conventions proposed in [14], we denote by σ_i the i-th element of σ. The weight of σ is the sum of its components weights, denoted as $\psi(\sigma)$.

We say that a point sequence $\tau = \tau_1\tau_2 \ldots \tau_l$ is a *subsequence* of σ if a sequence of integers $i_1 < i_2 < \ldots < i_l$ exists such that τ is equal to $\sigma_{i_1}\sigma_{i_2} \ldots \sigma_{i_l}$. We denote by $\sigma_{i\ldots j}$ the contiguous subsequence of σ consisting of the points from position i to j. We say that τ is *increasing* iff $\tau_1.x < \tau_2.x < \ldots < \tau_l.x$ and $\tau_1.y < \tau_2.y < \ldots < \tau_l.y$.

Definition 9. (Heaviest Increasing Point Subsequence). *Given a point sequence σ, an increasing point subsequence τ of σ is a heaviest increasing point subsequence if no other increasing point subsequence of σ has a higher weight.* △

Example 6. $\langle 1,2,1\rangle\ \langle 2,3,1\rangle\ \langle 3,5,1\rangle$ *is a heaviest increasing point subsequence of* $\langle 1,2,1\rangle\ \langle 2,3,1\rangle\ \langle 2,4,1\rangle\ \langle 3,5,1\rangle$. ○

In the algorithm we need to compare increasing subsequences of σ and to decide whether they can be a subsequence of a heaviest increasing point subsequence of σ. For this reason we introduce the following definition.

Definition 10 (Dominated Point Subsequence). *Consider two increasing point subequences of a point sequence σ, $\tau = \tau_1\tau_2 \ldots \tau_m$ and $\upsilon = \upsilon_1\upsilon_2 \ldots \upsilon_n$, and let ψ_τ and ψ_υ be the two sequence weights. We say that υ is dominated by τ iff $\upsilon_n.y \geq \tau_m.y \wedge \psi_\tau > \psi_\upsilon$ or $\upsilon_n.y > \tau_m.y \wedge \psi_\tau \geq \psi_\upsilon$.* △

Example 7. *Consider the point sequence $\sigma = \langle 1,2,1\rangle\ \langle 2,2,1\rangle\ \langle 2,3,1\rangle\ \langle 3,4,1\rangle$ and the two following point subsequences of σ: $\tau = \langle 1,2,1\rangle\langle 2,3,1\rangle$ and $\upsilon = \langle 2,3,1\rangle\langle 3,4,1\rangle$. According to Definition 10, υ is dominated by τ.* ○

5.2 Algorithm Description

Given the specific context in which we employ the algorithm, without loss of generality we make the following assumptions on the input data. Both X and Y alphabets are $\mathbb{N}_0$, each symbol pair represents a point in a two-dimensional space, all points are different, all weights are greater or equal to 0.

Intuitively, the algorithm considers a set of points M and sorts it in ascending order on the point x component first, and then on the point y component, producing a point sequence $\sigma = \sigma_1 \ldots \sigma_n$.

Let $Y_{1\ldots i} = \{p.y \mid p \in \sigma_{1\ldots i}\}$ be the set of all the y components of the points in $\sigma_{1\ldots i}$. The algorithm scans σ (from the first to the last element) and keeps updated an auxiliary set S. After σ_i symbol has been considered, the set S represents one of the non-dominated subsequences of σ whose last point y component is $p.y$, for each $p.y \in Y_{1\ldots i}$ (if at least one exists). When the last symbol of σ has been considered, the highest-weight subsequence represented in S (the one with the highest y component) contains all and only the points in one of the heaviest increasing point subsets of M.

Specifically, the point set S is kept sorted in ascending order on the points y component first, and on the x component second, and provides the following functions (relying on the ordering of S): (i) `pred(p)`, which, given a point p, returns the point p' in S s.t. p' is the majorant of the minorants of p. If no such point exists, the function returns Λ. (ii) `succ(p)`, which, given a point p, returns the point p' in S s.t. p' is the minorant of the majorants of p. If no such point exists, the function returns Λ. For brevity we define `succ(Λ)` as `min()`. (iii) `max()` returns the greatest element in S if S is not empty, Λ otherwise. (iv) `min()` returns the least element in S if S is not empty, Λ otherwise. More precisely, the algorithm represents non dominated subsequences of σ by means of the point set S and two additional maps, SW and P. For each subsequence τ, S stores the last considered element of τ, SW stores the subsequence weight, while P associates each point in τ with its predecessor.

Algorithm 6 HIPS Algorithm

```
1: function HIPS($M$)
2:     $\sigma = \sigma_1\sigma_2 \ldots \sigma_n \leftarrow$ the sequence of points in $M$ sorted in ascending order on the point $x$ component first and on the $y$ component second
3:     $S \leftarrow \emptyset, P \leftarrow \emptyset, SW \leftarrow \emptyset, AQ \leftarrow [], currXMaxW \leftarrow -1$
4:     for i $\leftarrow$ 1 to $n$ do
5:         $pred \leftarrow S.pred(\langle 0, \sigma_i.y, \sigma_i.w\rangle)$
6:         $SW[\sigma_i] \leftarrow SW[pred] + \sigma_i.w$
7:         $P[\sigma_i] \leftarrow pred$
8:         if $SW[\sigma_i] > currXMaxW$ then
9:             $currXMaxW \leftarrow SW[\sigma_i]$
10:            $succ \leftarrow S.succ(pred)$
11:            if $succ = \Lambda$ OR ($\sigma_i.y < succ.y$ OR $SW[succ] < SW[\sigma_i]$) then
12:                $AQ.addLast(\sigma_i)$
13:            end if
14:        end if
15:        if $i = n$ OR $\sigma_i.x \neq \sigma_{i+1}.x$ then
16:            $processQueue$(ref $S, P, SW, AQ$)
17:            $currXMaxW \leftarrow -1$
18:            $AQ.clear()$
19:        end if
20:    end for
21:    return $maximalSubset(S, P)$
22: end function
23:
24: function processQueue(ref $S$, $P$, $SW$, $AQ$)
25:    for each $\mu$ in $AQ$ do
26:        $succ \leftarrow S.succ(P[\mu])$
27:        while $succ \neq \Lambda$ do
28:            if $SW[\mu] < SW[succ]$ then
29:                break
30:            end if
31:            $S.remove(succ)$
32:            $succ \leftarrow succ(succ)$
33:        end while
34:    end for
35:    for each $\mu$ in $AQ$ do
36:        $S.add(\mu)$
37:    end for
38: end function
39:
40: function maximalSubset($S, P$)
41:    $MS \leftarrow \emptyset$
42:    $curr \leftarrow S.max()$
43:    while $curr.x \neq \tau$ do
44:        $MS \leftarrow MS \cup \{curr\}$
45:        $curr \leftarrow P[curr]$
46:    end while
47:    return $MS$
48: end function
```

Algorithm 6 starts by sorting the input set of points M in ascending order on the point x component first, and on the point y component then, obtaining a point sequence σ (line 2). Then, for each point σ_i in σ, the algorithm determines the heaviest subsequence τ (which is terminated by the *pred* point identified at line 5) in S to which σ_i can be appended. When the, possibly empty, sequence τ has been determined, the algorithm updates the SW and P maps accordingly (line 6–7). If $\tau\sigma_i$ is not dominated by another sequence ending at an element with the same x component of σ_i, it is scheduled for addition to S (line 12).

When all the points with a given x component are visited, those scheduled for addition, as well as the point set S and the maps

SW and P, are given as input to the `processQueue` function (line 15–19). This function first removes from S (line 25–34) any subsequence which is dominated by one of those scheduled for addition, then adds the scheduled subsequences to S (lines 35–37). Finally, when all the input points have been processed, the heaviest increasing point subset of M is returned using the function `maximalSubset`, that builds the heaviest increasing subsequence with a traversal of the predecessor map P, starting from the maximal point $S.max()$.

Proposition 1 (Correctness of Algorithm 6). *Given a set of points M as input, Algorithm 6 returns one of the heaviest increasing subsets of M.* ◇

Proof Sketch Proof of Proposition 1. Assume that the input point set contains no two points with the same x component and let $Y_{1...i} = \{p.y \mid p \in \sigma_{1...i}\}$ be the set of all y components of the points in $\sigma_{1...i}$. Under this assumption the proposition can be proved by induction observing that, after the i-th element of the input has been processed, the set S represents one of the non-dominated subsequences (if any) whose last point y component is $p.y$, for each $p.y \in Y_{1...i}$.

Base case: After the first element σ_1 has been considered, $Y_{1...1} = \{\sigma_1.y\}$, and the algorithm represents the only non-dominated increasing point subsequence of $\sigma_{1...1}$ whose last point y component is $\sigma_1.y$, that is, $[\sigma_1]$ in S.

Inductive case: Assume now that, after the i-th element of σ has been considered, S represents one of the non-dominated subsequences (if any) whose last point y component is $p.y$, for each $p.y \in Y_{1...i}$. When the point σ_{i+1} is considered, the algorithm ensures that at least one non-dominated increasing subsequence of σ_{i+1}, ending with a point whose y component is $\sigma_{i+1}.y$, is represented in S. Thanks to the inductive hypothesis, we can observe that either one of the highest weight subsequences of $\sigma_{1..i+1}$ is already represented in S or it ends with σ_{i+1}. To determine one of the highest weight subsequences of $\sigma_{1..i+1}$ ending with σ_{i+1}, we rely on the following observation. For any two non dominated point subsequences $\tau = \tau_1\tau_2 \dots \tau_l$ and $\upsilon = \upsilon_1\upsilon_2 \dots \upsilon_m$ of σ, where $\tau_l.y \neq \upsilon_m.y$, according to Definition 10, either $\tau_l.y > \upsilon_m.y \wedge \psi(\tau) > \psi(\upsilon)$ or $\tau_l.y < \upsilon_m.y \wedge \psi(\tau) < \psi(\upsilon)$. From this observation and the inductive hypothesis, it follows that one of the highest weight subequences ending with σ_{i+1} consists of the subsequence represented in S ending with the point with the highest y component smaller than $\sigma_{i+1}.y$ (if such an element exists in S, [] otherwise) concatenated with σ_{i+1}. This sequence, if not dominated, is represented in S by means of the `processQueue` function. Therefore, after the last point of σ has been considered, the sequence represented in S, ending with the point with the highest y component, is one of the heaviest increasing point subsequence of σ. Since σ is sorted on the points x component in ascending order first, we can easily observe that each heaviest increasing point sequence of σ contains all and only the points of one of the heaviest increasing point subset of M. The proof can then be easily extended to arbitrary input sequences, where multiple points with the same x component may be present, observing that the algorithm considers increasing sequences consisting of points with different x components, thus proving the proposition. □

Proposition 2. *Given a sequence $\sigma = \sigma_1\sigma_2 \dots \sigma_n$, the time complexity of the HIPS algorithm is in $O(n \log n)$.* ◇

Proof Sketch of Proposition 2. In Algorithm 6, at most n invocations of the `pred` and `succ` functions are required by the HIPS function. The `processQueue` function, which is invoked at most n times, requires across all invocations to evaluate n times the `succ` function (specified at line 26). The `succ` function (line 32) is also evaluated at most n times since it is only invoked when an element is removed from S, which contains at most n elements. If S is efficiently implemented, for instance using self-balancing binary search trees, both `pred` and `succ` can be evaluated in $O(\log n)$ time, proving the proposition. □

6. APPROXIMATE TREE MATCHING

In the top down-refinement stage, we consider partial matches and try to improve them, that is, to increase the weight of the matches among the children of the two partially matched nodes. Let $\langle s, t, w, False\rangle$ be one of the considered matches. Ideally, we would like to identify one of the heaviest consistent subset of all possible matches among the children of s and those of t. While identifying one of the heaviest consistent subset of a given set of matches is an efficient operation, the number of all the possible matches could be large. Moreover, exactly determining the minimum cost of a PUL transforming a tree into another is extremely expensive.

In the remainder of the section, we tackle both problems. Specifically, in Section 6.1 we introduce our *pq*-gram based technique for estimating the minimal cost of a PUL transforming a tree into another, whereas in Section 6.2 we present our approach for choosing candidate matches between two tree sequences.

6.1 Tree PUL Edit-distance

One of the most relevant problem for tree data structures is the tree edit-distance, that is, to determine, given two trees S and T, a set of edit operations, and an associated cost function, the minimum cost of a sequence of edit-operations which transform S into T.

Exact algorithms, such as those in [12, 21], have optimal time complexity in $O(n^3)$. To estimate PUL edit-distance, we extend the *pq*-gram approximated similarity measure [2]. *pq*-grams are a generalization of string *q*-grams: they are subtrees of fixed size and shape, composed by a *stem* made of p elements (bound by the parent-child relation) and a *base* of q consecutive siblings: the last element of the stem is named *anchor node* and the element of the base are children of this node. To ensure that each node of the tree appears in at least one *pq*-gram as an anchor node, the tree is extended with dummy nodes not occurring in the original tree. Moreover, the textual content of element nodes is concatenated to their name, and text nodes are removed. The obtained tree is named *extended tree representation*. The similarity between two trees can then be estimated computing the Jaccard-distance of the sets of all *pq*-grams extracted from the two trees extended representations.

In our context, since "move" operations are not allowed, identical *pq*-grams whose anchor node is at different nesting levels should not increase the similarity of the two trees under analysis. Moreover, we experimentally observed that *pq*-grams tend to under-estimate the similarity of trees where nodes have different labels or values w.r.t. to our PUL cost model, because they embed node labels and values in the same *pq*-gram [6].

To overcome these limitations, we introduce *pql*-grams, in two flavors: name *pql*-grams and value *pql*-grams. Both are generated from the same extended representation used for generating *pq*-grams, but name *pql*-grams contain only node names, whereas value *pql*-grams contain only node values. Moreover, both kinds of *pql*-grams include the nesting level of their anchor nodes. Tree similarity is computed as for *pq*-grams. Consider two trees S and T. Starting from the *pql*-gram-based similarity α between them (which ranges from 0 when two trees share no *pql*-gram to 1 when they have the same *pql*-grams), we estimate the PUL edit-distance as $(1-\alpha)(\Omega(S)+\Omega(T))/2$. An experimental evaluation of the quality of the *pql*-gram-based PUL edit-script distance estimation can be found in Section 7.2.1.

6.2 Window-based Sequence Matching

In the top-down refinement stage (see Section 4.3), we contrast the sequence of children of two partially matched nodes and aim at identifying the heaviest consistent subset of all the possible matches among nodes in the two sequences. In this section, we propose a window-based approximated approach, which uses *pql*-grams to estimate the tree PUL edit-distance between two trees.

We start by defining a search window for each unmatched node of the target sequence, exploiting existing matches among nodes in the two sequences. We partition the source and target sequences in consecutive partitions, each delimited by two currently matched nodes (first/last are delimited by the start/end of the node sequence). Since the current matches are consistent, we expect the best match for the nodes in the i-th target sequence to lie in the i-th source partition. More precisely, we consider the probability that the best match for a node of the i-th sequence can be found in the j-th source sequence is inversely proportional to the quantity $|i-j|$.

While using larger windows grants an higher probability of finding the best match for a given target node, it also increases the probability that the considered matches are incompatible with the other matches. For this reason, using large windows (or even the entire source sequence) do not usually increase the quality of the identified matches, as experimentally verified in Section 7.2.2.

Example 8. *Consider a partial match $\langle s, t, 6, False\rangle$ and let the node sequence reported at the top (resp. bottom) of Figure 2 to be the sequence of children of s (resp. t), where matches are depicted with a line. A simple heuristics for defining a search window for each target node d is to scan in both directions the target children sequence, starting from d, and to consider the window defined by the two source nodes matched by the first encountered matched node in each direction. If the start (resp. end) of the target sequence is reached, the start (resp. end) of the source children sequence is used instead. For instance, using this criteria, the search window for K is $[C..G]$, whereas for O it is $[G..H]$.* ○

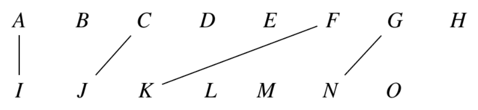


Figure 2: Source sequence (top) and target sequence (bottom).

As a further optimization, we do not perform a 1-to-1 comparison between a target node and each node in its source window. Rather, we compare all target nodes with these source window subtrees at the same time. Specifically, let $d_1, \ldots, d_n$ and $c_1, \ldots, c_m$ be a set of target nodes and their common source window, respectively. We first compute and sort the sets P_s (resp. P_t) of all source (resp. target) window *pql*-grams and tag each *pql*-gram with the subtree of the source (resp. target) window it belongs to. Then, through a scan of both sets we populate an $m \times n$ matrix M, where the $M_{i,j}$ cell contains the number of *pql*-grams in common between c_i and d_j. Starting from this matrix, we can easily compute the *pql*-gram distance. We remark that, even if the worst case complexity does not change, this approach is more efficient than 1-by-1 comparisons and yields a speedup proportional to the number of unique *pql*-grams.

Algorithm 7 accepts two partially matched nodes s and t, and first partitions the sequences of children of s and t, so that matched

Algorithm 7 Similarity Matches

```
1:  function similarityMatches($s$, $t$, $matches$)
2:      $candMatches \leftarrow \emptyset$
3:      $\{s_1, ..., s_n\} \leftarrow$ the partitions of $\gamma_E(s)$ according to the matched nodes
4:      $\{t_1, ..., t_n\} \leftarrow$ the partitions of $\gamma_E(t)$ according to the matched nodes
5:      for p = 1..n do
6:          $sGrams \leftarrow \{\langle pql, o, c\rangle \mid \langle pql, o\rangle \in$ pqlGrams(c) $\wedge\, c \in s_p\}$
7:          $tGrams \leftarrow \{\langle pql, o, d\rangle \mid \langle pql, o\rangle \in$ pqlGrams(d) $\wedge\, d \in t_p\}$
8:          $M \leftarrow$ a zero-filled $|s_p| \times |t_p|$ matrix
9:          $M[i][j] \leftarrow \sum_{\langle pql, o_1, o_2\rangle \in X} min(o_1, o_2)$
10:                 where $X = \{\langle pql, o_1, o_2\rangle \mid \langle pql, o_1, c_i\rangle \in sGrams\, \wedge$
                        $\wedge \langle pql, o_2, d_j\rangle \in tGrams\}$
11:         $candMatches$.add(heaviestMatches($M$, $s$, $t$))
12:     end for
13:     return $candMatches$
14: end function
```

nodes occur only as the first/last element of a partition (lines 3-4). Let n be the number of identified partitions. The algorithm then estimates the similarity between all the subtrees in the p-th source and target partition (lines 5-12). To compare two partitions the algorithm first computes, thanks to function `pqlGrams`, for each node c (resp. d) in the source (resp. target) partition, the set of all *pql*-grams of $\mathcal{T}(c)$ (resp. $\mathcal{T}(d)$), along with their number of repetitions in the set of *pql*-grams of $\mathcal{T}(c)$ and the node c (resp. d).

The algorithm then populates the matrix M, as described in the previous paragraph. The k-heaviest matches for each target window node are then identified and added to the candidate matches map, thanks to function `heaviestMatches` (line 11). When all the partitions have been processed, all the candidate matches are returned. We stress that, in the PUL-Diff algorithm the search window for a given target node d depends on $\Omega(d)$, and on the weights of the matches delimiting the subsequences adjacent to the one d belongs to. Additionally, we enforce an upper-bound for the number of subtrees in the target window to limit the comparison complexity.

Proposition 3 (Complexity of Algorithm 7). *Let P and Q be two trees. The time complexity of Algorithm 7, applied on P and Q, is in $O(n \log n)$, and the spatial complexity is in $O(n)$, where $n = |\mathcal{V}(P)| + |\mathcal{V}(Q)|$.* ◇

Proof Sketch of Proposition 3. As an extension of the proof given in [2], the computation of the *pql*-grams distance between two trees is in $O(n \log n)$. The number of nodes in both the source and target window is limited to a constant number. For what concerns spatial complexity, the number of *pq*-grams for a tree t is linear in the number of nodes of t [1]. For the same tree, the number of *pql*-grams is twice the number of *pq*-grams, so again linear: *pql*-grams are used only in the window-search approach, whose worst-case requires the computation of the *pql*-grams for both the source and target tree. The worst-case space complexity for *pql*-grams computation is thus in $O(n)$. □

7. ALGORITHM EVALUATION

In this section we first investigate the PUL-Diff complexity (Section 7.1). Then, we present an experimental evaluation of the PUL edit-distance estimation and provide a time and edit-script cost comparison with other state of the art XML differencing algorithms (Section 7.2).

7.1 Algorithm Complexity

The following proposition states the temporal and spatial complexity of the PUL-Diff algorithm.

Proposition 4. *Let S and T be two XML documents. The temporal complexity of PUL-Diff on S and T is in $O(n \log n)$, whereas the spatial complexity is in $O(n)$, where $n = |\mathcal{V}(S)| + |\mathcal{V}(T)|$.* $\diamond$

Proof Sketch of Proposition 4. The identical subtree matching stage (Algorithm 2) computes a hash signature of each subtree in S and T, using a bottom-up DOMHASH-like approach, with a cost linear in the number of nodes of S and T. Moreover, n hash-table insertions and up to $|\mathcal{V}(T)|$ hash-table lookups must be performed. The space required to store the node hash signatures is in $O(n)$.

The bottom-up refinement stage (Algorithm 3) performs, for each node in $|\mathcal{V}(T)|$, a (very small) constant number of hash-table insertions/lookups and deletions. Moreover, across all invocations, the HIPS algorithm processes at most $|\mathcal{V}(T)|$ matches. Let M be a set of matches. Since the time complexity of the HIPS algorithm is in $O(|M| \log |M|)$ (see Proposition 2) the worst-case complexity for the second stage is $O(n \log n)$.

The top-down refinement stage (Algorithm 4) performs, for each node in $|\mathcal{V}(T)|$, a (very-small) constant number of hash-table insertions/lookups and deletions. For reducing the complexity of this stage, the number of identical matches identified at line 3 by function `identicalST` is limited using a window-based approach. Therefore, across all invocations, the window-match algorithm (Algorithm 7) is invoked on at most $|\mathcal{V}(S)|$ source nodes and at most $|\mathcal{V}(T)|$ target nodes. If we assume a costant tree depth, according to Proposition 3, the worst-case time complexity for the third stage is in $O(n \log n)$. Moreover, the number of matches is always linear in $|\mathcal{V}(T)|$ during this stage, thus the spatial complexity of storing the matches is in $O(n)$. For what concerns *pql*-grams, according to Proposition 3, the spatial complexity is again in $O(n)$.

The edit-script generation (Algorithm 5) visits both the source and target tree and performs a small constant number of hash-set lookups for each target node. Moreover, no more than n operations can be generated and no more than $|\mathcal{V}(T)|$ nodes can be inserted. Since the number of generated operation is at most n, the spatial complexity is in $O(n)$. $\square$

7.2 Experiments

Since no suitable versioned XML document collection is available, we relied on synthetic documents, generated as follows. The source document is either produced by means of the XMark document generator[4] or selecting a random subset of the first-level children of the DBLP XML document[5]. These two kinds of source documents presents different characteristics. XMark documents have a complex organization, with subtrees representing several different entities, whereas DBLP-based documents are a simple list of subtrees describing scientific publications. Instead of computing the exact edit-script cost between the source and target documents, we randomly generate a minimal PUL with a known cost (using a complex blocking strategy for the target elements of the generated operations), and we obtain the target document by applying this PUL to the source document. These PULs contain randomly generated `del`, `ins`^{d}, `repN`, `ren`, `repV` operations. Inserted and replaced subtrees can be randomly generated, similar or equal to another subtree of the source document, or randomly selected from another XMark/DBLP document. New names and values can be randomly generated, or be randomly selected from another XMark/DBLP document. We also simulate subtree moves through pairs of deletion and insertion. The number of each operation type is roughly the same. Although we ran the experiments with many different distributions of operation types, for brevity we only report the results obtained with this distribution. No significant differences were found using XMark and DBLP-based documents. To easily combine and contrast the results obtained considering documents of different sizes, we do not reason directly on the tree edit-distance, rather we consider the *change ratio*, that is, the edit-distance divided by the source document weight.

In the experiments we considered $p = 2$, $q = 3$ in *pq*-grams and *pql*-grams generations, $k = 5$, with no limits on the approximated matching window size. The tests has been performed on a PC with an *Intel Core i7-2670QM* CPU, 16GB of RAM and *Kubuntu Linux 12.10 64-bit* O.S. For increasing statistical significancy, every time measurement is averaged over at least 50 samples.

In the remainder of the section, we first experimentally evaluate the quality of the PUL edit-distance estimation obtained by means of tree-grams, then we empirically test the soundness of our window-based approach used in top-down refinement stage, and we finally provide a time and edit-script cost comparison with other state of the art XML differencing algorithms.

7.2.1 PUL Edit-distance Estimation

This section experimentally evaluates the quality of the PUL edit-script distance estimation using tree-grams (*i.e., pq*-grams and *pql*-grams). Let S, T, and T' be three trees and let e_1 (resp. e_2) be the edit-script distance between S and T (resp. S and T'). Our goal is to maximize the confidence that if $e_1 > e_2$, then the edit-script distance between S and T is higher than the one between S and T'. Thus, we need that, independently from the kind of operations which are needed to transform a tree into another, (i) the estimated change ratio increases when the real one does and the increments are similar, (ii) for any given real change ratio, the estimated change ratio has a low variance.

As discussed in Section 6, tree-grams distance is only used to select, in a sequence of trees, the k most similar trees to a given one. Even if the k best matches, according to the estimation, are sub-optimal, the application of the HIPS algorithm often chooses a (near-)perfect set of matches, as experimented in Section 7.2.2.

We considered both homogeneous and non-homogeneous random PULs, composed by `del`, `ins`^{d}, `repN`, `ren`, `repV` or move operations. For each operation we also investigated additional aspects: (i) the deleted subtrees weight range, for `del` operations, (ii) the number of inserted subtrees, their weight range and generation algorithm for `ins`^{d} and `repN` operations, (iii) the number of adjacent sibling moved subtrees, for move operations, (iv) the name generation algorithm for `ren` PULs, (v) the value generation algorithm for `repV` PULs. We considered the following inserted tree generation algorithms: (i) *random trees*, (ii) trees which have a *similar structure* (10% of the nodes are renamed, 70% of the text values are changed) to either one of the inserted subtree sibling or to a subtree at the same nesting depth, (iii) trees which have a *similar size* to that of their new sibling/replaced node, but random structure, names, and values, (iv) *real trees*, randomly extracted from another XMark/DBLP document, among the subtrees which usually occur in the considered document, at the considered position (*e.g.,* a new `article/person` can be inserted next to another `article/person`). Random tree height is limited to 6 and the structure roughly resembles that of an XMark document. For the tests we generated roughly 1 million source documents and PULs, with different change ratios, uniformly distributed among the following values (0.01, 0.05, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6 and 0.7) and the following kinds (random, homogeneous `del`, `ins`^{d}, `repN`, `ren`, `repV` and move). Moreover, for each kind, the PUL are uniformly distributed among the different parameters of interest.

[4]`http://www.xml-benchmark.org`

[5]`http://www.informatik.uni-trier.de/~ley/db`

Table 2: PUL change ratio estimation summary, random PULs

	pq-gram estimation		*pql*-gram estimation	
Change ratio	Mean	Standard dev.	Mean	Standard dev.
0.01	0.029572	0.026947	0.016702	0.008945
0.05	0.124398	0.107564	0.074867	0.035163
0.10	0.213398	0.169629	0.135634	0.056661
0.20	0.338368	0.231525	0.231765	0.078262
0.30	0.425672	0.255584	0.305835	0.086130
0.40	0.492553	0.263406	0.365108	0.089523
0.50	0.556594	0.278110	0.418311	0.100024

For brevity, in Table 2, we only report the results for random non-homogeneous PULs. We can observe that as the real change ratio increases, both the *pq*-gram and the *pql*-gram estimated change ratios increase of a similar amount. *pq*-grams overestimate the edit-script cost when renamings or value-changes are necessary and have a much higher variance. Therefore, at least in our context, *pql*-grams provide a better estimation. Additional details can be found in the extended experimental section [6].

7.2.2 Window-based Sequence Test

In the top-down refinement stage a sequence of source subtrees is contrasted with a sequence of target subtrees aiming at finding the heaviest consistent set of matches among them. The algorithm employs an approximated approach: each target subtree is compared employing *pql*-grams with all the source subtrees in its search window. The best k matches for each target subtree are inserted into a set M. The HIPS algorithm is then invoked on M identifying the heaviest consistent subset of M.

In this experiment, we aim at replicating the window-based match settings producing subtree sequences, with a subtree size ranging from 20 to 60 nodes. Target documents are generated using random non-homogeneous PULs. The quality of the chosen matches for entire sequences of children is considered in Section 7.2.3. Here we consider the worst-case scenario: in each sequence all subtrees are very similar to each other (*e.g.*, all DBLP `article` elements or all `auction` XMark elements), and modifications are roughly uniformly distributed among subtrees. For each generated pair of sequences we contrast the cost difference between the edit-script generated by PUL-Diff ignoring all perfect matches (to simulate matching two sequences of unmatched subtrees), the edit-script generated by PUL-Diff, and an optimal edit-script.

Results for different sequence lengths (from 10 to 500) and different values for k are reported in Figure 3. Change ratio does not significantly influence the results, we present only results for change ratio 0.5. In each plot the left (resp. right) side represents the result obtained without (resp. with) perfect matches. Independently from sequence length and value of k, the selected matches are on average very precise. the median of the cost difference w.r.t. the optimal solution is always below 5%. Increasing the value of k above 3, when perfect matches are employed, has negligible impact. When perfect matches are not employed, increasing the value of k to about half the length of the sequence yields to edit-scripts as expensive as those identified with perfect matches. However, the average distance between the edit-script obtained with/without perfect matches is extremely small (1-2%) Surprisingly, increasing the length of the sequence *reduces* the distance between the computed edit-scripts and the optimal one, thanks to the HIPS algorithm.

Therefore, the proposed *pql*-gram-based distance metric, when paired with an HIPS-based top k matches pruning algorithm, is very reliable. Moreover, we stress that PUL-Diff benefits from perfect matches and thus the length of the sequences compared by the algorithm is much shorter.

7.2.3 Performance Test

This test aims at empirically determining the time complexity of PUL-Diff for different document sizes and change ratios, as well as contrasting the edit-script cost and computational time of PUL-Diff with the state of the art. The PULs generating the target documents contain random `del`, `ins`^{d}, `repN`, `ren` and `repV` operations.

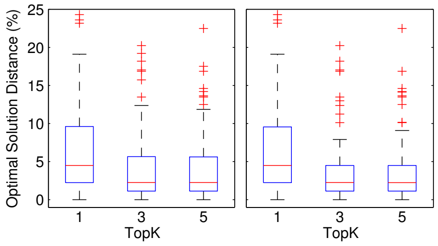
We identified several interesting XML differencing tools for ordered models in the literature, including XyDiff [11], DeltaXML [13], MMDiff [7], XMDiff [7], and RTED [17], the best performing exact algorithm. On our test machine RTED requires more than 3 minutes and 14GB of RAM for differencing two 256KB documents. Other exact algorithms (*i.e.,* MMDiff and XMDiff) require even more time. For this reason, we considered only XyDiff (C++ version) and DeltaXML 6.4.1 (Linux 14-days trial version, limited to 1 million nodes, roughly, 7MB). The edit-script produced by these algorithms can be contrasted with ours without unfair advantages. Specifically, w.r.t. our cost model, the relevant differences are that the `repC` operation exist only in PUL-Diff and that a move operation exists only in XyDiff. Since in this test we disabled the generation of `repC` operations, and since XyDiff devotes a great effort to minimize the number of moves, we just consider the cost of XyDiff move operations as the cost of the equivalent deletion and insertion operations. Moreover, we double checked that the XyDiff edit-script costs are fair by contrasting the presented results with the results obtained disabling the generation of moves in the PULs used to produce the target documents, and employing only randomly generated names, values, and inserted/replacement subtrees to almost avoid move operations in the XyDiff edit-scripts.

We report in Figure 4 the cost difference between a minimum-cost edit-script (the one used to generate the target document) and the cost of the edit-script generated by PUL-Diff, XyDiff, and DeltaXML. We consider different change ratios and document sizes. As can be observed in the figure, independently from the change ratio, both PUL-Diff and DeltaXML produce almost minimal edit-scripts, with a slight advantage for PUL-Diff. XyDiff, instead, produces far worse results. With very small documents (up to 1MB) its quality is comparable with that of the other two algorithms, but usually the generated edit-script is far from optimal. This result does not seem to depends on the XyDiff algorithm implementation as [10, 11] consider small documents and report that XyDiff can compute an edit-script 5 times more expensive than the considered reference (on average 2 times more expensive).

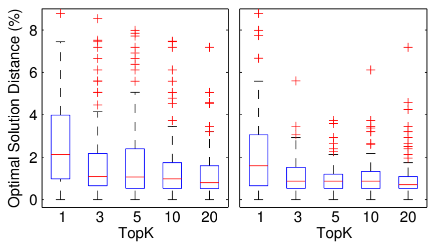
In Figure 4, we contrast also the time required for differencing two documents with different sizes and change ratios. We can observe that XyDiff is roughly 2 to 5 times slower than PUL-Diff and that both algorithms have an almost linear time complexity. As expectable, the change ratio and the two document sizes influence the expected computational time. For what concerns PUL-Diff, as the change ratio increases, the number of perfectly matched subtrees decreases, thus increasing the time spent in the top-down refinement stage. The result also shows that the quality of the DeltaXML edit-scripts is counterbalanced by an exponential computational time, which is evident for a change ratio of at least 0.3.

8. CONCLUSIONS

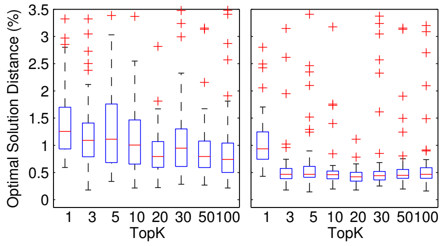
In the paper we propose an algorithm for synthetising the changes between two XML documents as PULs. The choice of PULs as edit-scripts influences the algorithm, since no move operator is considered, while internal node relabelings and changes at leaves are detected. No assumptions are made on the trees to be compared (*e.g.*, few duplicate node labels, as in [9]). The results of the experimental evaluation against state of the art approaches are very



(a) Sequence length 10.

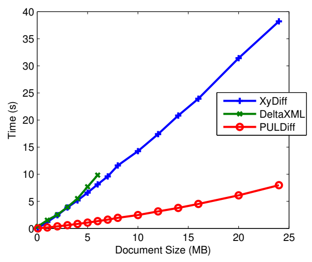


(b) Sequence length 50.

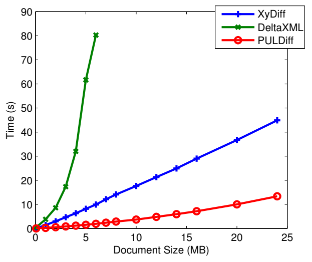


(c) Sequence length 500.

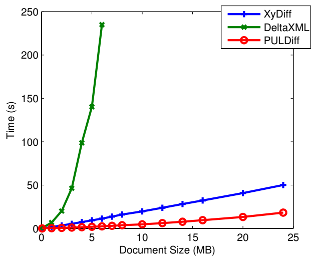
Figure 3: Window-based sequence matching with change ratio 0.5. Cost distance from an optimal solution.



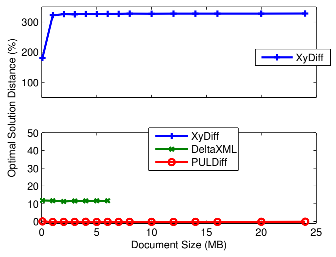
(a) Change ratio 0.1, time.



(b) Change ratio 0.3, time.



(c) Change ratio 0.5, time.



(d) Change ratio 0.5, cost.

Figure 4: Differencing time and cost-distance from an optimal solution w.r.t. document size.

good, both in terms of time and edit-script cost. As future work we plan to extend the proposal for supporting all the node kinds (*e.g.*, attributes and comments), to refine the *pql*-grams based edit-distance estimation with a deeper inquiry of its formal properties, and to perform experiments on the spatial complexity.

9. REFERENCES

[1] N. Augsten, M. Böhlen, and J. Gamper. Approximate Matching of Hierarchical Data Using pq-Grams. In *VLDB*, pages 301–312, 2005.

[2] N. Augsten, M. Böhlen, and J. Gamper. The pq-gram Distance between Ordered Labeled Trees. *ACM Transactions on Database Systems*, 35(1):4, 2010.

[3] F. Cavalieri. *Updates on XML Documents and Schemas*. PhD thesis, 2013. `http://felix.disi.unige.it/downloads/thesisFC/phdthesis.pdf`.

[4] F. Cavalieri, G. Guerrini, and M. Mesiti. Dynamic Reasoning on XML Updates. In *EDBT*, pages 165–176, 2011.

[5] F. Cavalieri, G. Guerrini, and M. Mesiti. Decoupled Execution of XML Updates. Submitted, 2012.

[6] F. Cavalieri, A. Solimando, and G. Guerrini. Synthetising Changes in XML Documents as PULs Extended Experiments. `http://felix.disi.unige.it/downloads/puldiff/experiments.pdf`.

[7] S. S. Chawathe. Comparing Hierarchical Data in External Memory. In *VLDB*, pages 90–101, 1999.

[8] S. S. Chawathe and H. Garcia-Molina. Meaningful Change Detection in Structured Data. In *SIGMOD*, pages 26–37, 1997.

[9] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom. Change Detection in Hierarchically Structured Information. In *SIGMOD*, pages 493–504, 1996.

[10] G. Cobena, T. Abdessalem, and Y. Hinnach. A Comparative Study for XML Change Detection. In *BDA*, 2002.

[11] G. Cobena, S. Abiteboul, and A. Marian. Detecting Changes in XML Documents. In *ICDE*, pages 41–52, 2002.

[12] E. D. Demaine, S. Mozes, B. Rossman, and O. Weimann. An Optimal Decomposition Algorithm for Tree Edit Distance. *ACM Transactions on Algorithms*, 6(1), 2009.

[13] R. L. Fontaine. A Delta Format for XML: Identifying Changes in XML Files and Representing the Changes in XML. In *XML Europe*, 2001.

[14] G. Jacobson and K.-P. Vo. Heaviest Increasing/Common Subsequence Problems. In *Combinatorial Pattern Matching*, pages 52–66, 1992.

[15] W. Miller and E. Myers. A File Comparison Program. *Software: Practice and Experience*, 15(11), 1985.

[16] E. Myers. An O (ND) Difference Algorithm and its Variations. *Algorithmica*, 1(2):251–266, 1986.

[17] M. Pawlik and N. Augsten. RTED: A Robust Algorithm for the Tree Edit Distance. *PVLDB*, 5(4):334–345, 2011.

[18] W3C. XQuery Update Facility 1.0. `http://www.w3.org/TR/xquery-update-10/`, 2011.

[19] Y. Wang, D. J. DeWitt, and J. Cai. X-Diff: An Effective Change Detection Algorithm for XML Documents. In *ICDE*, pages 519–530, 2003.

[20] H. Xu, Q. Wu, H. Wang, G. Yang, and Y. Jia. KF-Diff+: Highly Efficient Change Detection Algorithm for XML Documents. In *CoopIS/DOA/ODBASE*, pages 1273–1286, 2002.

[21] K. Zhang and D. Shasha. Simple Fast Algorithms for the Editing Distance Between Trees and Related Problems. *SIAM J. Comput.*, 18(6):1245–1262, 1989.