RWS-Diff: Flexible and Efficient Change Detection in Hierarchical Data

Jan Finis* Martin Raiber* Nikolaus Augsten† Robert Brunel* Alfons Kemper* Franz Färber‡

* Technische Universität München, Boltzmannstr. 3, 85748 Garching, Germany
† Universität Salzburg, Jakob-Haringer-Str. 2, 5020 Salzburg, Austria
‡ SAP AG, Dietmar-Hopp-Allee 16, 69190 Walldorf, Germany

* *firstname.lastname*@cs.tum.edu † nikolaus.augsten@sbg.ac.at ‡ franz.faerber@sap.com

ABSTRACT

The problem of generating a cost-minimal edit script between two trees has many important applications. However, finding such a cost-minimal script is computationally hard, thus the only methods that scale are approximate ones. Various approximate solutions have been proposed recently. However, most of them still show quadratic or worse runtime complexity in the tree size and thus do not scale well either. The only solutions with log-linear runtime complexity use simple matching algorithms that only find corresponding subtrees as long as these subtrees are equal. Consequently, such solutions are not robust at all, since small changes in the leaves which occur frequently can make all subtrees that contain the changed leaves unequal and thus prevent the matching of large portions of the trees. This problem could be avoided by searching for similar instead of equal subtrees but current similarity approaches are too costly and thus also show quadratic complexity. Hence, currently no robust log-linear method exists.

We propose the random walks similarity (RWS) measure which can be used to find similar subtrees rapidly. We use this measure to build the RWS-Diff algorithm that is able to compute an approximately cost-minimal edit script in log-linear time while having the robustness of a similarity-based approach. Our evaluation reveals that random walk similarity indeed increases edit script quality and robustness drastically while still maintaining a runtime comparable to simple matching approaches.

Categories and Subject Descriptors

H.3.1 [**Information Storage and Retrieval**]: Content Analysis and Indexing

Keywords

Hierarchical data; XML; similarity search; tree edit distance; tree edit script; tree diff; change detection; random walk

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CIKM'13, Oct. 27–Nov. 1, 2013, San Francisco, CA, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2263-8/13/10 ...$15.00.
http://dx.doi.org/10.1145/2505515.2505763.

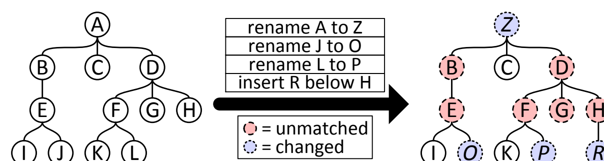


Figure 1: Slightly different trees that make top-down and bottom-up matching fail

1. INTRODUCTION

When tree data changes or versions of a data item are independently modified, it is necessary to compute the difference to reconcile or display the changes. The changes are often expressed as a so-called *edit script*: a compact sequence of operations that transforms one tree into the other. Computing edit scripts has many important applications. Consider, for example, revision control systems that deal with trees like XML data warehousing [1], source code revision control [7], or HTML warehousing [8]. The goal in these applications is to compute a compact and intuitive representation of the history. Computing compact tree diffs is also crucial for various other applications, for example, data synchronization [19], genomic and proteomic data [32, 15], RNA secondary structures [38], or image analysis [6].

Our goal is to compute compact edit scripts for very large trees, for example, two file systems with tens or hundreds of thousands of nodes. Furthermore, operations that lead to short and intuitive edit scripts are to be supported. In particular, not only edits on individual nodes (e.g., deletion of a node), but on whole subtrees should be considered (e.g., moving of a subtree). As an example, consider a diff for synchronizing a remote file system. A locally moved subdirectory requires sending all files one-by-one to be inserted into the remote file system if moves are not detected. Our approach should work for both ordered and unordered trees to be generally applicable. An example of ordered trees are HTML documents, were the order of the paragraphs matters; file systems are unordered trees.

Ideally, a tree difference algorithm computes a *minimal* edit script. Unfortunately, for unordered trees, the problem is MAX-SNP hard [37] even for a limited set of node operations; for ordered trees, exact solutions require $O(n^3)$ time in the number of nodes and thus do not scale either. Our approach is to *approximate* the minimal edit script. The result is a small (albeit not necessarily minimal) edit script that is correct, i.e., it turns the first input tree into the second.

Most previous attempts to approximate the minimal edit script run in $O(n^2)$ and consequently do not scale to large trees. The few solutions that run in $O(n \log n)$ use comparably simple matching algorithms which start either at the root (top-down) or at the leaves (bottom-up). The matching is continued as long as identical nodes or subtrees are found. These approaches rely on large subtrees that can be matched exactly and fail otherwise, as illustrated in Figure 1: Top-down cannot match any node since the root labels differ. Bottom-up subtree matching can only match single leaf nodes (I, C, K), although many inner nodes are unchanged. The changes in the leaves alter the containing subtrees and prevent them from matching. Changes in the leaf nodes are a frequent scenario, for example, files in a file system and text values in XML.

Our solution is RWS-Diff (Random Walk Similarity Diff), a novel robust algorithm for tree differences. RWS-Diff supports both node and subtree edits. It is *robust* because it does not rely on exact subtree matching, but is also able to match similar subtrees. Similarity computations are costly and the challenge is to find similar subtrees efficiently. We present a new technique which represents each subtree by a d-dimensional feature vector using random walks. This allows us to use well-established indexes for similarity search in d-dimensional space to find similar subtrees.

RWS-Diff is the first algorithm that runs efficiently in $O(n \log n)$ and deals well (due to subtree similarity matching) with all kinds of node edits for which top-down and bottom-up based approaches with the same asymptotic runtime fail. RWS-Diff does not rely on any application specific assumptions (like node identifiers) that simplify the matching. It is configurable to work on both ordered and unordered trees and supports a large set of edit operations. Our evaluation using synthetic and real-world data shows substantial gains in the matching quality (up to ten times smaller edit scripts on average compared to other quasi-linear methods) and robustness (more than 200 times smaller edit scripts for certain real-world trees) and confirms the scalability of RWS-Diff.

2. TREE EDIT SCRIPTS

For a pair of input trees A and B, a tree difference algorithm (diff) computes a sequence of edit operations (called *edit script*) that transforms A into B. Tree diffs vary in the kinds of supported operations and the underlying tree definition.

Tree Definition. Our diff algorithm works on all rooted, labeled trees with a (non-strict) order defined on the labels (i.e., the labels can be sorted and compared for equality) and a hash function that maps labels to numeric values. Our algorithm can be configured for both ordered trees (where the sibling order matters) and unordered trees. The sibling order is not related to the label order.

The labels carry application specific data. XML nodes, for example, may be labeled with element tags or text content, but also more complex labels are possible. Our flexible tree definition suites a wide range of applications such as HTML documents, XML, file systems, or RNA secondary structures, and sets us apart from many other works with restricted input trees.

Edit Operations. We allow edit operations on both *nodes* and *subtrees*. Let a and a' be two nodes in tree A; the following edit operations are allowed in an edit script:

- *rename*(a, l) Change the label of node a to l.
- *insertLeaf*(l, a, i) Insert a new leaf node with label l as a new child of node a before the i-th child of a.
- *deleteLeaf*(a) Remove leaf node a.
- *insertSubtree*(S, a, i) Insert a new subtree S before the i-th child of node a.
- *deleteSubtree*(a) Remove the subtree rooted in node a, that is, a and all its descendants.
- *move*(a, a', i) Remove the subtree rooted in a and insert it before the i-th child of a'.
- *copy*(a, a', i) Insert a copy of the subtree rooted in a before the i-th child of a'.

The child position i is omitted in the case of unordered trees. The root node is extended with a dummy parent node to allow all edit operations also on the root node. Subtree edit operations lead to more compact and intuitive edit scripts that can be applied fast. For example, moving a chapter of a document is faster and more expressive than deleting all sections and paragraphs and reinserting them at the target position individually. Node insertion and deletion are defined on leaf nodes; am inner node is deleted by first moving all its children (with their subtrees) to its parent.

Our tree difference algorithm is flexible as it can be configured to work with either the ordered or the unordered version of the edit operations. Furthermore, the operations for copying, inserting, and deleting subtrees can be switched off, in which case they are expressed by other operations.

A cost is assigned to each edit operation and the edit script is the better the lower the accumulated costs of the contained operations are. RWS-Diff does not imply a specific cost model. The only restriction is that the cost of each operation must be less than the cost of a sequence of other operations that can emulate it, because otherwise the operation would be useless. For example, deleting a subtree using *deleteSubtree* must be cheaper than deleting the same subtree node by node using *deleteLeaf*. The cost for subtree insertion should be a function of the subtree size, as it must encode the whole subtree S to be inserted; otherwise the best edit script would always consist of deleting tree A and inserting tree B if the trees are large enough.

Edit Mapping. An *edit mapping* maps nodes between two trees and is used to express the difference between the trees; intuitively, two nodes are mapped if they correspond to each other. We produce an edit mapping in the first step and infer the edit script from the mapping in a second step.

We define the edit mapping M between a tree A that should be transformed into tree B to be a function from the nodes of B to the nodes of A. The mapping function is partial and neither injective nor surjective, that is, not all nodes of B or A need to be mapped and a node of A can be the image of multiple nodes of B.

3. RELATED WORK

Edit scripts have been discussed from two points of view. Works on the edit distance compute the similarity between trees, where two trees are considered similar if a short edit script can transform one tree into the other. Tree diff algorithms are interested in the edit script itself.

3.1 Tree Edit Distance Computation

The tree edit distance is defined as the minimal cost of an edit script that transforms one tree into the other. The clas-

sical algorithm by Zhang and Shasha [38] for *ordered trees* only allows the *node* edit operations discussed in Section 2. For these operations, the exact distance for two trees T_1 and T_2 is computed in $\mathcal{O}(n_1 n_2 \min(d_1, l_1) \min(d_2, l_2))$ time and $O(n_1 n_2)$ space, where n_1 (n_2) is the number of nodes, l_1 (l_2) is the number of leaf nodes, and d_1 (d_2) the depth of T_1 (T_2). Thus, for trees with $\mathcal{O}(n)$ leaves and depth $\mathcal{O}(n)$, the runtime complexity is $O(n^4)$. Klein et al. [16] and Dulucq et al. [12] improve the runtime to $\mathcal{O}(n^3 \log n)$. Demaine et al. [11] present an algorithm that runs in $\mathcal{O}(n^3)$ time and show that this is the best worst case complexity that can be achieved. Unfortunately, the worst case is a frequent scenario in this algorithm, rendering it slower than the classical algorithm by Zhang and Shasha for many practical scenarios. Recently, the RTED [26] algorithm solved this problem; it maintains the optimal worst case complexity and runs as fast or faster than any of the previously proposed algorithms. For each of these algorithms the minimal edit script for the edit distance can be computed within the same complexity bound [38].

Overall, computing the *minimal* edit script requires $O(n^3)$ time and $\Theta(n^2)$ space for ordered trees, even when only node edit operations are allowed. An extension of Zhang and Shasha's algorithm with *deleteSubtree* and *insertSubtree* runs in $O(n^4)$ time [4]. With the *move* operation that we allow in our approximation, the edit distance problem is NP-complete even for the case of flat strings [30].

Approximations of the tree edit distance, which run more efficiently, have been proposed. Guha et al. [14] propose an upper bound for the tree edit distance by computing the *string* edit distance between the preorder (or postorder) sequences of the tree node labels in $\mathcal{O}(n^2)$ time. An edit script can be computed using this method.

With p,q-grams [3] Augsten et al. propose a concept of "q-grams for trees". The grams are constructed using the ancestor relationship (configurable by p) and the sibling relationship (q). The method decomposes the input trees into small, besom-shaped subtrees with depth p and q leaves. Each of these small subtrees is then serialized to a string and hashed. A list of these hashes represents the data in the tree and its hierarchical relationships. The algorithm calculates the p,q-grams in $\mathcal{O}(n)$ time and space. Our approach also makes use of p,q-grams for finding similar subtrees but adds a dimensional reduction step to speed up similarity search. Similar to p,q-grams, the binary branch technique [36] splits trees into small subtrees, but binary branches keep less structure information than p,q-grams [3]. Neither p,q-grams nor binary branches compute edit scripts.

For unordered trees, finding the exact tree edit distance is MAX SNP-hard [37] since the matching algorithm can not rely on the sibling order. Zhang et al. [31] propose an exact, enumeration-based algorithm for unordered trees which runs in $\mathcal{O}(n^3 16^n)$ and a heuristic solution based on searching in the enumeration space which runs in $\mathcal{O}(n^2)$. By sorting siblings lexicographically by label the concept of p,q-grams can be adapted to unordered trees. The p,q-gram approximation runs efficiently in $\mathcal{O}(n \log n)$ and is shown to work well in practice [2]. We use this technique for supporting random walk similarity on unordered trees.

3.2 Computing Diffs between Trees

Numerous approaches for computing approximately cost-minimal edit scripts have been proposed, but most of them either suffer from a prohibitive runtime of at least $\mathcal{O}(n^2)$, are restricted to very specific types of data, or do not show a robust behaviour.

Chawathe et al. propose LaDiff [10], which imposes restrictions on the hierarchical order between labels: An example are LaTeX documents, where a subsection is always within a section. This bottom-up algorithm uses a heuristic optimized for text. As a bottom-up tree edit distance, it is sensitive to changes in the leaf nodes. LaDiff combined with another method [7] is implemented in the tree-diff tool DiffXML [22]. Le et al. [17] remove the hierarchical order restriction from LaDiff. The family of algorithms based on LaDiff runs in $O(ne)$ time, where e is the size of the edit script. In the worst case, when the trees are very different, the runtime is $O(n^2)$ and the approach does not scale. In our experiments we compare to DiffXML as a representative of these algorithms.

In [18, 19] a three-way merging algorithm for XML is described, which includes an algorithm for calculating diffs between XML documents (3DM). It works in a bottom-up fashion, mapping trees using their content. It also uses the neighborhood of tree nodes to produce mappings, for example, when the left and right siblings of a node are mapped, a mapping for the node in between is inferred. The algorithm has worst-case complexity $\mathcal{O}(n^2)$ and runs in $\mathcal{O}(n \log n)$ if the changes between trees are small.

The MH-DIFF algorithm [9] allows the operations *insert*, *rename*, *delete*, *move*, and *copy*. Here, *insert* and *delete* work on single inner or leaf nodes. In the first step, all possible mappings between the nodes of two trees are considered and mappings that can only increase the cost are pruned in the second step. The problem is then solved as a bipartite weighted matching problem by assigning an approximate cost to each mapping between a pair of node. The overall runtime complexity is $\mathcal{O}(n^2 \log n)$.

Wang et al. [34] only allow *insertLeaf* and *removeLeaf*. Furthermore, only nodes with the same path to the root are mapped. The algorithm produces large edit scripts if these assumptions are not met. When the maximum number of children of all nodes is assumed to be a constant (i.e., independent of the tree size), $\mathcal{O}(n^2)$ runtime complexity is achieved. [32] improves the average runtime of this algorithm for hierarchical biological data without altering the worst case complexity.

The KF-Diff+ algorithm [35] is specific to a particular kind of XML documents, in which each node has a key that is unique between all siblings. In this case, a diff which allows *move* operations only between nodes with the same parent can be computed in $\mathcal{O}(n)$ time.

The only algorithm without strong assumptions that runs in less than quadratic time is XyDiff [21]. XyDiff uses tree hashes that are invariant to the sibling order [34] to efficiently find and map moved subtrees. In the next step, nodes in the vicinity of mapped subtrees are mapped. The overall algorithm runs in $\mathcal{O}(n \log n)$ and produces good results if large unchanged subtrees are present. In addition to the operations supported by XyDiff, our algorithm also supports subtree deletion. Subtree deletion is useful to remove surplus data from one of the trees, for example, the citation elements present in some DBLP entries that otherwise dominate the edit distance. We experimentally compare our algorithm to XyDiff and show that (even without subtree deletion) our algorithm produces significantly smaller edit scripts with a similar runtime.

4. THE RWS-DIFF ALGORITHM

A good edit mapping is one that maps as many nodes as possible and maps nodes which are very similar to each other. The better the mapping, the smaller the generated edit script will be. A perfect mapping would be one that produces a cost-minimal edit script. However, such a mapping is extremely hard to compute (MAX SNP-hard for unordered trees even if only node operations are allowed, cf. Section 3). Thus, RWS-Diff is an approximate method which tries to find a good - but not always perfect - mapping. However, our focus is on finding a better mapping than previous approximate approaches by using an elaborated similarity measure to find non-obvious mappings.

This section introduces RWS-Diff which constructs the approximate cost-minimal edit mapping and then creates an edit script from it. Our method can roughly be separated into five steps:

1. A simple matching step which tries to find obvious common structures in both versions of the tree. The nodes mapped in this step do not have to be considered in subsequent matching steps and thus significantly improve their speed.
2. Construction of feature vectors for unmapped subtrees of both trees, i.e., small fixed-length vectors which are similar if subtrees are similar. The squared euclidean distance between the vectors constitutes our random walk similarity measure.
3. Creation of appropriate index structures for nearest neighbors queries among the feature vectors.
4. Mapping of previously unmapped subtrees by looking up possible candidates using nearest neighbors queries.
5. Generation of the edit script from the edit mapping.

4.1 Finding Simple Mappings

In the first step, we try to match large parts of the trees rapidly. The goal is not to find all possible mappings but to find only the obvious ones which are easy to compute. We use methods and concepts already described in the literature and successfully applied. These are top-down matching [29, 35] and matching using subtree hashes [21].

Top-down Matching. The top-down matching starts at the roots of the trees to be compared and maps nodes with the same label to each other. If a node is mapped and is not a leaf node, the same matching method is recursively applied to its children. If more than one sibling has the same label, we do not map it in this step, since we might map the wrong pairs of nodes. When using this method, if a node's label is changed, the whole subtree rooted in this node is not mapped anymore.

Hash Matching. Hash matching between trees A and B is performed bottom up by calculating a hash value of each unmapped subtree in A and inserting it into a hash table. Then, the unmapped subtrees of B are hashed as well and the hash table is probed to find equivalent subtrees in A. For unordered trees, we have to use a position-independent hash function. Simply adding the hash of the node label and the hash values of all children multiplied with a prime number is position-independent since addition is a commutative operator (cf., Figure 2). For ordered trees it suffices to multiply the hash of each child with a prime number that is different for each child position.

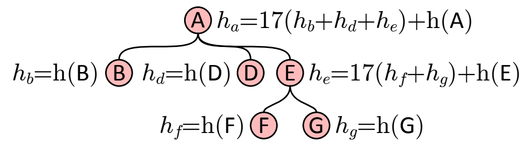


Figure 2: Sibling order invariant subtree hashes

4.2 Random Walk Similarity Matching

Using hash matching and top-down matching, we can usually map a large portion of the tree nodes if there is a moderate number of differences (which is usually the case). The top-down matching finds all paths from the root that have not changed. The hash matching finds smaller subtrees which have not changed. However, the quality of the resulting mapping is usually not sufficient because these simple matching methods are not robust at all. For example, renaming the root totally disables top-down matching and a single renaming in a node or insertion of a node disables hash matching for all subtrees that contain this node. Since even such "trivial", non-structural edit operations disable the simple matching methods, we need a method for finding trees that are similar but not equal.

Our core contribution finds trees that are not necessarily equal but similar and thus are missed by the simple matching approaches. The idea is to represent each subtree by a d-dimensional feature vector (with fixed d) that constitutes a random walk in d-dimensional space. The random walks are generated in a way that ensures that their squared euclidean distance, which we call *random walk distance (RWD)* is approximately proportional to the edit distance of the corresponding trees. How these random walks are generated in detail will be discussed in Section 5.

We find similar subtrees in trees A and B by generating all feature vectors for subtrees in A and inserting these vectors into an index structure for d-dimensional nearest neighbors queries. If the copy operation is allowed, we insert all subtrees, because even already mapped subtrees could be mapped again for a copy. Otherwise, we only insert subtrees with an unmapped root. Then, we generate feature vectors for all unmapped subtrees in B. Next, we iterate over tree B in preorder and for each unmapped subtree b, we use its feature vector to probe into the index structure to find the ℓ (with fixed ℓ) nearest neighbors which are candidates for being similar. We retrieve $\ell > 1$ neighbors, because a low RWD does not always (but often) imply a similarity in the subtrees (i.e., false positives are possible). Therefore, the ℓ nearest neighbors in the feature vector space are merely used as mapping candidates and we use the one with the least edit distance or none if all are false positives (which should happen very infrequently due to the stochastic properties of the RWD). For the similarity comparison of the ℓ mapping candidates, we use an iterative deepending top-down matching that stops after a fixed number of compared nodes and is thus in $O(1)$. Although the premature stopping might reduce approximation quality, it is important to meet the desired log-linear runtime bounds. If the copy operation is not allowed we skip candidates which have an already mapped root. Once the best candidate subtree a for subtree b is determined, we map the roots of a and b and perform an ordinary top-down matching starting from a and b to map cheaply as many descendants as possible. Afterwards, we continue the preorder iteration over B to map remaining subtrees.

We can use standard index structures to efficiently find nearest neighbors in d-dimensional space. We focus on prominent indexing schemes which are *k-d trees*, *k-means locality-sensitive hashing (KLSH)*, and *hierarchical k-means (HKM)*. These methods have been shown to be useful in practice [20, 24, 25]. However, note that any scheme for finding nearest neighbors in d-dimensional space can be used. The k-d tree is an established multidimensional index structure which repeatedly separates the space by hyperplanes. The exact nearest neighbors algorithm on the k-d tree is quite costly, so we use the approximate *best bin first (BBF)* [5] algorithm. Hierarchical k-means clustering has been successfully used to cluster high dimensional data [24]. It works by recursively finding k centroids for the data point clusters and then arranging those clusters into a tree structure. K-means locality-sensitive hashing [25] uses k-means clustering to convert space coordinates into locality-sensitive hashes.

4.3 Edit Script Generation

An edit script is a sequence of edit operations that transforms tree A into tree B. Algorithm 1 produces the edit script from an edit mapping (which maps nodes of B to nodes of A). Figure 3 shows an example for each kind of edit operation produced by the algorithm. The algorithm traverses all nodes of B in preorder, that is, parents are visited before their children. If a node b in B has no mapping (b, a) in M, b is inserted (Lines 2-3, node H in fig.). If a mapping (b, a) exists, we check whether we already have visited a node b' which maps to a as well. The node b' is already visited if its preorder rank $pre(b')$ is smaller than the one of b (Lines 4-6). If such a node exists, we already have "used" subtree a and must thus copy it (Line 7, right node E in fig.); otherwise, we check if the parents of the mapped nodes differ. If they do, we move node a to its new parent (Lines 8-9, left node E in fig.). In addition to updating the node position, we must also rename node a if the labels of the mapped nodes differ (Lines 10-11, node $B \to C$ in fig.). After the preorder iteration over B, we perform a postorder iteration over A and delete all nodes that are not mapped (Lines 12-14, node I in fig.).

Algorithm 1 *generateEditScript*(A, B, M)

1: **for all** nodes b of B in preorder **do**
2:     **if** $\nexists a.(b, a) \in M$ **then**
3:         Emit $insertLeaf(label(b), M(parent(b)), pos(b))$
4:     **else**
5:         $a \leftarrow M(b)$
6:         **if** $\exists b'.(b', a) \in M \wedge pre(b') < pre(b)$ **then**
7:             Emit $copy(a, M(parent(b)), pos(b))$
8:         **else if** $M(parent(b)) \neq parent(a)$ **then**
9:             Emit $move(a, M(parent(b)), pos(b))$
10:        **if** $label(b) \neq label(a)$ **then**
11:            Emit $rename(a, label(b))$
12: **for all** nodes a of A in postorder **do**
13:     **if** $\nexists b.(b, a) \in M$ **then**
14:        Emit $deleteLeaf(a)$

Whenever an edit operation is emitted, it is applied to tree A and subsequent edit operations are defined on the new version of A. In addition, the mapping M is updated after each *insertLeaf* and *copy* operation with mappings to the newly added node(s). After executing Algorithm 1, the sibling order is adjusted for ordered trees, which is separately discussed below.

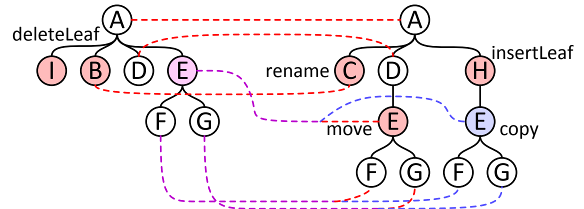


Figure 3: Edit mapping with implied edit operations

Line 8 of the algorithm requires that M maps the parent of the current node b to some node of A. This is guaranteed by traversing B in preorder and updating the mapping after each insert operation. In the second phase of the algorithm (Lines 12-14), unmapped nodes in A are removed using the *deleteLeaf* operation, which requires the nodes to be leaves at the time of being removed. This holds since (a) the mapped child a of an unmapped node in A satisfies the condition in Line 8, that is, the subtree rooted in a is moved to a mapped parent in the first phase of the algorithm; (b) the nodes of A are traversed in postorder, thus unmapped children are removed before their unmapped parents.

Algorithm 1 does not produce subtree insertions and deletions. By generating the *insertLeaf* and *deleteLeaf* operations in preorder and postorder, respectively, we ensure that all inserts and deletes that belong to the same subtree are adjacent in the edit script. We merge sequences of leaf insertions and deletions into subtree insertions and deletions in a simple postprocessing step. By omitting this step, we can switch off subtree insertion and deletion. If subtree copy is switched off, the mapping is injective, so the condition in Line 6 is never true.

After executing Algorithm 1, A is identical to B except for the sibling order, and all nodes of A and B are mapped. We use the approach of XyDiff [21] to fix the sibling order. The c children of each node in A are numbered with the sibling positions of the respective (mapped) nodes in B, that is, each child in A gets assigned a position between 1 and c. We compute the longest increasing sub-sequence X of the position numbers in $\mathcal{O}(c \log c)$ time [13]; all nodes that are not in X are moved to the right position by emitting *moveSubtree* operations.

4.4 Complexity of RWS-Diff

RWS-Diff must have an $\mathcal{O}(n \log n)$ worst case runtime complexity in order to yield a scalable solution. The simple matching methods that are also applied in existing $\mathcal{O}(n \log n)$ methods obviously fall into this bound. The generation of the random walk feature vectors for all subtrees in a tree is in $\mathcal{O}(n)$ (cf. next Section). Since there may be $O(n)$ subtrees in both trees that must be mapped by the RWS, mapping one subtree may only cost $O(\log n)$. A nearest neighbors lookup is usually in $O(\log n)$ in the index structures. We adjust the index structures to yield even worst case $O(\log n)$ behaviour by simply decreasing the approximation quality in pathological cases. For example, HKM has its height limited to $O(\log n)$ and if there are more than ℓ candidates in the final Voronoi cell, only the ℓ first are considered. Although the approximation becomes worse in some cases, our evaluation shows that the overall quality is still good. For finding the best candidate between the ℓ candidates, we use the constantly bounded iterative deepening top-down matching which is in $O(1)$, so a single RWS mapping stays in $O(\log n)$. An insertion or lookup in the mapping M is in $O(1)$ since

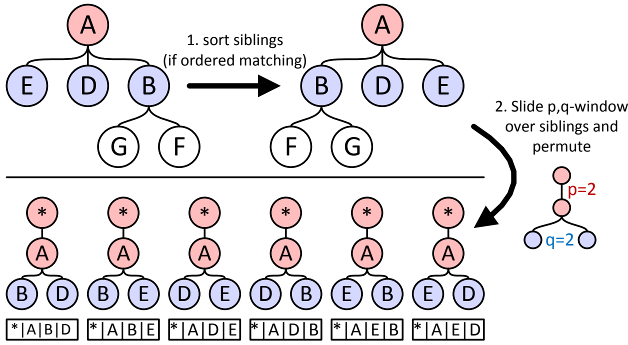


Figure 4: Partial construction of p,q-grams

dense integers can be assigned to each node in tree B and M can be implemented as an array indexed by these integers. Finally, the edit script generation loops only twice over both trees and is thus in $O(n)$, so we meet the desired overall complexity bound of $\mathcal{O}(n \log n)$.

5. RANDOM WALK SIMILARITY

To find similar subtrees rapidly, we reduce the information content of each subtree to a fixed-size d-dimensional feature vector and use indexed d-dimensional nearest neighbors queries. To obtain the feature vector, we first serialize each subtree into a bag of p,q-gram hashes. The more similar two trees are, the more hashes are equal. Each of the hashes becomes a step in a d-dimensional random walk and the final feature vector is the endpoint of the walk. Consequently, the more similar two trees are, the more steps in their random walks are equal and thus the distance of the random walk end points is the smaller the more similar two trees are. We prove certain stochastic properties of the proposed random walk to show that it is indeed a valid approximate similarity measure.

5.1 Grams for Trees

Grams (also shingles or tokens) are tree summaries that represent a tree by a set of small excerpts. Using such grams, the problem of finding similar subtrees is reduced to the problem of finding bags of grams with large intersections. This approach has been widely applied to strings before and has shown to be useful also for trees.

We use *p,q-grams*, which are besom-shaped subtrees consisting of q leaf nodes (called *base*) and a chain of p non-leaf nodes (called stem). In the original tree, the base nodes are siblings and the stem nodes their p closest ancestors. p,q-grams capture both ancestor and sibling relationships and can be made invariant to small order changes. In addition, they have already been successfully applied to tree similarity computations in various scenarios [3, 2].

The p,q-grams for all subtrees of a tree of size n can be generated in $\mathcal{O}(n)$ time. The p,q-gram construction is illustrated in Figure 4 for an example tree ($p = q = 2$). In the first step, the tree is sorted lexicographically by labels (the sort order of identical labels is irrelevant for the p,q-gram construction). Next, for each node a in the tree, a window of size $w \geq q$ is slided over the children of the node and p,q-grams are produced. The bases are formed by the first node of each window and any sub-sequences of the remaining nodes in the window. If a node does not have enough ancestors or enough children, dummy nodes (labeled with an asterisk in the figure) are used to produce the p,q-grams. In Figure 4, the sorting changes the order of the children of the node with label "A". The window size is $w = 3$ and six bases are formed for three window positions (the window is wrapped around at the right border).

Invariance to order changes is obtained through sorting siblings. Augsten et al. show in [2] that this way the permutation of a constant amount of siblings changes only a constant amount of p,q-grams. The construction of the base using a window makes the p,q-grams robust to modifications that change the sort order of children, called "children error" in [2], while still capturing sibling relationships. The "stem" captures ancestor relationships in the p,q-grams. If sibling permutations should not be allowed (ordered trees), the trees are not sorted and windows of size q are used.

The p,q-grams are finally serialized into arrays of size $p+q$, which is straightforward due to the fixed shape of the p,q-grams. The bottom of Figure 4 shows the serializations of the respective p,q-grams.

The similarity of two trees can now be expressed over their bags of p,q-grams. Let b_A and b_B be the bags of grams of tree A and B, respectively, then the symmetric bag difference $D(A, B)$ is defined as $|S_A \uplus S_B| - 2|b_A \cap b_B|$. This difference directly reflects the number of elements we have to remove from A and add to B if we want to transform A to B and as such approximately reflects the required edit operations. It is a distance measure, that is, the distance $D(A, B)$ between identical sets is zero while the distance between entirely different sets is $|A| + |B|$.

5.2 Random Walk Distance

Even though the comparison of trees is now easier, the actual size of the tree representations has gone up. If there are a and b unmapped subtrees in tree A and B, respectively, we have to compute $a \times b$ bag differences to find the best matches. Each of these computations has linear runtime in the bag sizes, which would clearly violate the $\mathcal{O}(n \log n)$ runtime bound. To speed up the similarity search, we do not explicitly calculate the bag difference between any two bags. Instead, we compress each bag of grams to a fixed-size d-dimensional vector and then use a nearest neighbors search in the d-dimensional space to find mapping candidates.

The d-dimensional feature vector for a tree A which has a bag of grams b_A is generated as follows: First, compute a hash value h_g for each p,q-gram g in the bag b_A. Then, use h_g to generate a random point v_g on the d-dimensional unit sphere (e.g., use h_g as seed for a random number generator that generates the vector components). To get the final feature vector v_A for A, add up all the vectors v_g. To approximate the symmetric bag difference, we use the d-dimensional squared euclidean distance. The vector v_A constitutes the end point of an d-dimensional random walk with $|b_A|$ steps of length one. Therefore, we call the resulting distance *random walk distance (RWD)*:

$$D(A, B) \approx RWD(A, B) = ||v_A - v_B||^2 = v_A \cdot v_B$$

By using the random walk distance, we reduce the problem of finding similar subtrees to the problem of finding points which are close in euclidean space. It is intuitive that this is a valid similarity measure: The more grams differ between the bags b_A and b_B, the more steps from which v_A and v_B are assembled differ.

All grams that are in both bags b_A and b_B yield the exact same steps in the random walk. Consequently, these steps

do not alter the distance at all. The number of remaining grams is $x = |b_A \setminus b_B|$ and $y = |b_B \setminus b_A|$ which constitute the two random walks whose squared euclidean distance is the RWD. The distance between the end points of two random walks with x and y steps is equal to the distance between the origin and an end point of a random walk with $z = D(A, B) = x + y$ steps. Figure 5 shows the transformation of two bags of grams b_A and b_B into corresponding two-dimensional random walks v_A and v_B. The numbers below the grams show their hash values. In this example, the trees have 4 common grams and $2 + 4 = 6$ different grams. Thus, the expected euclidean distance is $\sqrt{6}$.

Of course, the RWD is only an approximation since random walks with totally different steps might end up at points that are close to each other. To argue that the measure is useful indeed, we examine its stochastic properties: Let $v = v_1 + \ldots + v_z$ be the endpoint of a random walk starting from the origin and taking z steps in d-dimensional space and let RWD_z^d be the squared euclidean distance between the origin and v. For each step v_i, we have $E[v_i] = \vec{0}$, $\|v_i\|^2 = 1$ with all v_i being independent random variables. Then we have $\mathrm{RWD}_z^d = \|v\|^2 = v \cdot v$. By expansion of the dot product we obtain $\mathrm{RWD}_z^d = \sum_{i=1}^{z}(v_i)^2 + \sum_{k \neq \ell} v_k \cdot v_\ell$. With $E[(v_i)^2] = 1$ and $E[v_k \cdot v_\ell] = E[v_k] \cdot E[v_\ell] = 0$ for every $k \neq \ell$, we have

$$E[\mathrm{RWD}_z^d] = z = D(A, B) \qquad (1)$$

Thus, the RWD is indeed an approximation of the symmetrical bag distance regardless of the number of dimensions. The squared RWD is as follows:

$$(\mathrm{RWD}_z^d)^2 = \underbrace{z^2}_{a} + \underbrace{2z \sum_{k \neq \ell} v_k \cdot v_\ell}_{b} + \underbrace{(\sum_{k \neq \ell} v_k \cdot v_\ell)^2}_{c}$$

and thus for the variance:

$$\mathrm{Var}[\mathrm{RWD}_z^d] = \underbrace{E[a]}_{=z^2} + \underbrace{E[b]}_{=0} + E[c] - \underbrace{E[\mathrm{RWD}_z^d]^2}_{=z^2} = E[c]$$

and for $E[c]$:

$$E[c] = \sum_{k \neq \ell} \sum_{i \neq j} E[(v_k \cdot v_\ell)(v_i \cdot v_j)]$$

All terms with $\{k, \ell\} \neq \{i, j\}$ in this summation are zero. The remaining $2z(z-1)$ terms have a mean of $m = E[(v_i \cdot v_j)^2]$. Let the ℓ-th component of vector v_i be v_i^ℓ. Due to independence of v_i and v_j and because v_i and v_j are equally distributed, we can expand the dot product and simplify to $m = \sum_{\ell=1}^{d} E[(v_i^\ell)^2]^2$. Since all d components v_i^ℓ are identically distributed with $\|v_i\|^2 = 1$, we have $E[(v_i^j)^2] = \frac{1}{d}$ by symmetry and thus $m = \frac{1}{d}$. Consequently

$$\mathrm{Var}[\mathrm{RWD}_z^d] = 2z(z-1)m = \frac{2z(z-1)}{d} \qquad (2)$$

The first consequence of Equation 2 is that the number of dimensions reduces the variance and thus makes the approximation more precise. For an infinite number of dimensions, the RWD would even be exactly the symmetric bag difference. This implies that a high number of dimensions is desirable. However, a high number of dimensions makes computations more costly and renders the index structures that we use for the nearest neighbors search ineffective due to the curse of dimensionality. Hence, too many dimensions are prohibitive as well. We obtained best results with $10 \leq d \leq 20$. The second consequence of Equation 2 is that the variance is proportional to $z(z-1)$. Thus, the larger the distance, the less reliable the approximation is. In contrast, the RWD is a very reliable approximation for small distances. This fact is extremely beneficial for our application: As we want to execute a nearest neighbors search, we are especially interested in points with a small distance. For these points, the RWD is very precise, so there are no false negatives. As mentioned, the problem of false positives is mitigated by choosing the best of ℓ nearest neighbors. In conclusion, the stochastic properties of the d-dimensional random walk make the RWD an excellent approximate distance measure for our purposes. Note that while the RWD is defined as the squared euclidean distance since this is a direct approximation for the bag distance, the index structures use the usual euclidean distance. As we are not interested in the value of the RWD itself but only in the nearest neighbors, this is not an issue.

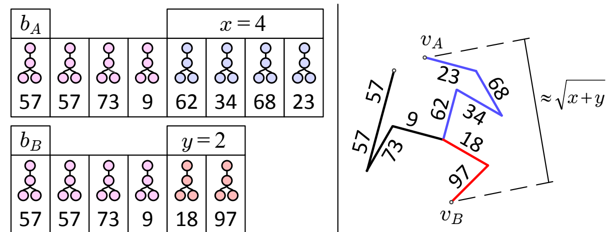


Figure 5: Transforming bags of grams into corresponding two-dimensional random walks

5.3 Weighting Grams

Until now, we assumed that each step in the random walk has a length of one. However, we can also weight the grams and multiply the step length by that weight to give certain grams more or less significance. A general assumption we can make is that having less frequent grams in common is more significant than having frequent grams in common. If we look, for example, at HTML documents, two subtrees having a *br* element in common are not that rare, while having a long text node in common which appears infrequently in the document is a strong indication of a correct mapping. Therefore, we use the *inverse gram frequency*, that is, the reciprocal value of the number of times a gram occurs in both trees as weight. Although this is a quite simple heuristic, it improved the edit script quality noticeably in our tests. Of course, more elaborate heuristics could be used as for example proposed in [28]. What we want to emphasize here is not the concrete choice of heuristic, but the fact that the random walk similarity can be tuned easily by such a heuristic.

6. EVALUATION

Index Structure Comparison. In order to find out which of the feature vector indexes presented in Section 4.2 is best suited we compare them with test data. We generate that data by taking 10,000/100,000 random subtrees with 10 to 100 nodes from various freely available XML files [33]. For each subtree S_i, we generate a feature vector f_i with $d = 10$ dimensions, modify the tree randomly, and calculate another feature vector f_i'. We then insert the feature vectors

Method	Precision	AVG dist	Runtime	Setup time
Linear scan	1	3.45	625 ms	0 ms
Exact k-d tree	1	3.45	1197 ms	8 ms
BBF k-d tree	0.46	3.85	54 ms	8 ms
KLSH	0.74	3.60	30 ms	131 ms
HKM	0.54	3.75	13 ms	261 ms

Table 1: Comparison of indexes with 10,000 points

Method	Precision	AVG dist	Runtime	Setup time
Linear scan	1	3.02	6454 ms	0 ms
Exact k-d tree	1	3.02	15041 ms	160 ms
BBF k-d tree	0.33	3.55	88 ms	160 ms
KLSH	0.76	3.15	470 ms	1988 ms
HKM	0.40	3.39	21 ms	3461 ms

Table 2: Comparison of indexes with 100,000 points

f_i into an index. Afterwards, we issue l-nearest neighbors queries with $l = 10$ for 1000 randomly chosen f_i's. We measure *precision*, that is, the fraction of returned nodes which are correct l-nearest neighbors, *average distance* to query point, *setup time* (i.e., time for generating the index) and *runtime* for the 1000 queries. All figures are averaged over 20 runs on identical hardware (Core i5 M460).

We assess exact approaches and approximate approaches. The exact approaches are a *linear scan* (comparison to all points) and a *k-d tree* with *exact* querying. The approximate approaches are *KLSH*, *HKM*, and a *k-d tree with best-bin-first (BBF)* querying.

Table 1 shows the result for 10,000 subtrees and Table 2 for 100,000 subtrees. Of course, the exact methods have a precision of 1. They also show that the "perfect" average distance, i.e., the distance of the real 10-nearest neighbors is 3.44 & 3.02 (for 10,000 nodes & 100,000 nodes). The runtime of the exact methods is prohibitively long even for a small dataset consisting of only 10000 points. The approximate methods, in contrast, show good results. Although the precision is not too good (46%–74% & 33%–76%), the average distance of 3.60–3.85 & 3.15–3.55 shows that the wrongly selected nodes are still close to the perfect average distance and thus are still good mapping candidates (in comparison, the distance of randomly selected points was 8.51 & 8.63). In conclusion, the approximate indexing methods, especially HKM and KLSH, are well suited for RWS-Diff. HKM is very fast even for larger datasets while KLSH has very precise results close to the exact solution. Since HKM is much faster than KLSH while its average distance is not much worse, we use HKM for the further experiments.

Comparison to Other Methods. To evaluate the quality of our method we compare the results with other proposed solutions. Here, the problem is that most methods have only a limited set of operations or they work only on unordered or ordered data. Even when they are comparable they often suffer from excessive runtime or enormously large edit scripts. Other comparative studies have found these shortcomings as well [27, 15]. Both studies have also found that XyDiff [21] is the only serious contender. In addition to XyDiff, we measure DiffXML [22] as an example for a widely used open source tool for XML change detection [23] with a worst-case complexity of $\mathcal{O}(n^2)$.

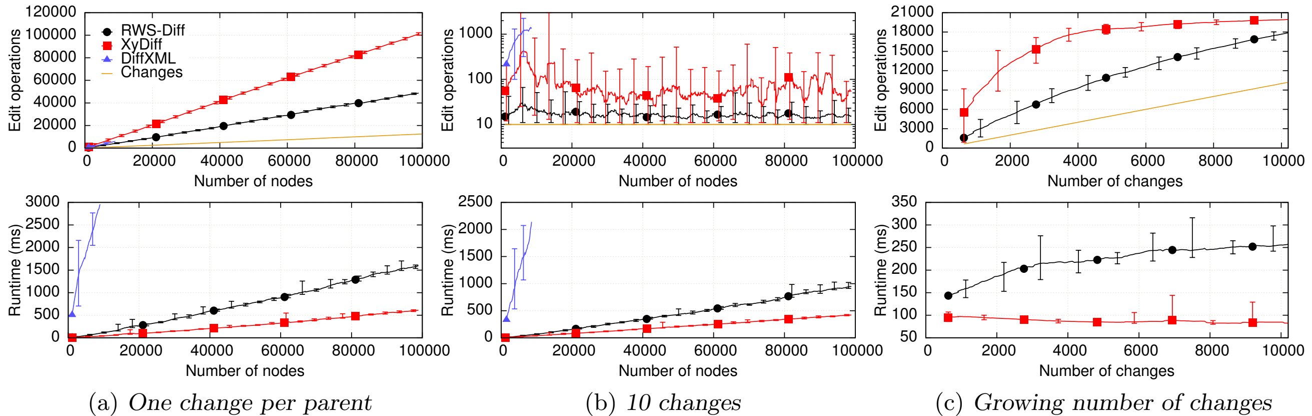
To make the comparison fair we switch off *insertSubtree*, *deleteSubtree*, and *copy* for our method as XyDiff does not support them. DiffXML uses *deleteSubtree*, but we grant that advantage. We use a uniform cost model, so the cost of the resulting edit script is equal to its length. Both contenders work in ordered mode. Since this can introduce additional work and edit operations, we use our method in ordered mode as well. The measured time does not include reading and parsing the XML/HTML trees or writing the edit script to a file.

Synthetic Changes. To show the runtime and quality for different tree sizes and change patterns, we first measure the results for synthetically changed trees. The trees are generated by extracting an increasing amount of nodes from the real-world dataset *nasa1* [33], which contains astronomical XML data, and modifying the extracted tree. The size of the extracted tree ranges from 100 to 100,000 nodes, but we stop DiffXML early after 10,000 nodes because of its tremendous runtime. The modification consists of (a) either random renames of one child of every node (one change per parent) or (b) of 10 random inserts, deletes, renames, or moves within the tree. The former change pattern represents a scenario where many small changes are introduced across the whole tree. Since one child of every node is modified simple methods might fail in this scenario. The latter change pattern of 10 random changes represents a scenario where only a comparably small part of the tree is changed. Of course, a very short edit script is anticipated in this case. Finally (c), we also measure an increasing amount of changes on a tree of constant size (20,000 nodes). This change pattern shows how the methods behave for a growing number of changes.

Figure 6 shows the number of edit operations (top) and the runtime (bottom) for the three different change patterns on the *nasa1* dataset. The data points are smoothed by calculating the moving average of 20 points. The error bars depict the minima and maxima smoothed away. For the two change patterns with a growing number of nodes in Figure 6(a,b) the horizontal axis shows the tree size, while for the change pattern with a growing number of changes in Figure 6(c) the horizontal axis shows the number of changes. Note the logarithmic vertical axis for the number of edit operations in Figure 6(b).

The runtime plots at the bottom of the figure show that RWS-Diff is around two to three times slower than a simple matching approach like XyDiff. The runtime of RWS-Diff always stays in the same order of magnitude as XyDiff and therefore qualifies for the same application scenarios. The runtime of both XyDiff and our method is almost linear which backs up the claimed $\mathcal{O}(n \log n)$ runtime bound. As anticipated, the $\mathcal{O}(n^2)$ runtime of DiffXML is infeasible for larger scenarios.

The plots for the number of edit operations at the top of the figure depict the quality gain of the similarity-based matching: For the case with one change per parent, the edit script of RWS-Diff is only around half as long as the one of XyDiff. Surprisingly, it is also slightly smaller than the more complex approach of DiffXML. Especially the scenario with only 10 changes shows the quality and robustness gains of our solution: XyDiff emits 74.5 changes on average while RWS-Diff emits only 16.6 on average which is only about 50% more than the exact solution and around 4.5 times less than XyDiff. In addition, our method is very robust as it never emits more than 68 operations, that is, 6.8 times more than the exact solution. In comparison, XyDiff sometimes yields comparably good results but often creates edit scripts with 100–1000 edit operations. Its largest edit script even consists of 3255 operations which is more than 300 times longer than the exact solution. Such an edit



(a) *One change per parent* (b) *10 changes* (c) *Growing number of changes*

Figure 6: Edit operations and runtime for the *nasa1* data set.

script is almost useless and demonstrates the huge robustness problem of simple matching approaches. DiffXML is even much worse than XyDiff for these few changes: It generated the astronomical amount of 850.8 changes on average and a peak number of 2477 changes. Consequently, the experiment shows that especially for common scenarios with few changes per version, similarity-based matching can lower the edit script size and increase robustness significantly—not only in comparison to simple matching methods but also in comparison to more expensive methods like DiffXML.

When examining a growing number of changes in Figure 6(c), our method always shows a considerable edit script quality gain. For a very large number of changes (in this case 8000+ changes for 20000 nodes), the edit script quality of both methods becomes more similar. This is due to the large amount of changes that alter the subtrees so much that they are no longer similar (i.e., do not share any p,q-grams anymore) and thus also render similarity-based approaches like ours ineffective. However, using edit scripts for such a large number of changes (almost half the tree size!) is very likely to be inferior to saving all versions explicitly anyway. Note that we did not measure DiffXML here as the tree consisted of too many nodes to yield a reasonable runtime.

Website Data. We inspected the two news websites *www.bbc.co.uk/news* and *www.tagesschau.de* in 20-minute intervals. These websites were selected because they change frequently. After collecting 900 different versions, we used our approach, XyDiff, and DiffXML to calculate the difference between each consecutive pair of versions. DiffXML was sometimes not up to the task and aborted with an exception; the following averages thus only considered the runtimes and results where it did not crash. While our previous experiment only revealed the behavior on a synthetic set of changes, this experiments shows how the methods perform in a real scenario. Table 3 and 4 show the results for the two datasets. In this real-world scenario, the quality gain by using our similarity approach is even higher than for the synthetic changes: While XyDiff is between 4 and 5 times faster than our method because the number of changes is comparably high, the quality of its edit script is highly inferior to our method: For the bbc data set, our method produces on average an edit script that is almost 13 times smaller than the one of XyDiff. In addition, our edit script is also around 4.5 times smaller than the one of DiffXML—even though DiffXML is an $\mathcal{O}(n^2)$ method and should therefore yield better results. For the tagesschau data set, the edit script length of

Method	AVG(#Edits)	σ(#Edits)	AVG(Runtime)	σ(Runtime)
RWS-Diff	81.41	42.41	120.93 ms	9.61 ms
XyDiff	1034.14	405.46	37.12 ms	6.32 ms
DiffXML	359.12	278.37	2649.49 ms	244.99 ms

Table 3: Result for the *bbc* data set

Method	AVG(#Edits)	σ(#Edits)	AVG(Runtime)	σ(Runtime)
RWS-Diff	61.32	87.37	126.58 ms	10.76 ms
XyDiff	381.02	736.58	25.15 ms	8.39 ms
DiffXML	313.39	299.90	1399.85 ms	309.64 ms

Table 4: Result for the *tagesschau* data set

our method is around 6 times smaller than those of XyDiff and DiffXML. The low standard deviation of our method for both data sets shows that similarity-based matching drastically increases the robustness of the method and thus leads to a more constant edit script quality. In contrast, the methods without similarity sometimes produce very inflated edit scripts: XyDiff produced 2083 edit operations between two versions of the bbc dataset while our method produced only 7 for these versions which is around 238 times less.

Our experiments revealed that the theoretical advantage of similarity matching is indeed also a practical one. The edit script quality is considerably increased (even in comparison to an $\mathcal{O}(n^2)$ method) while the runtime stays comparable to simple matching approaches. The similarity also increases the robustness by drastically reducing the variance of the edit script quality. Consequently, similarity-based edit script generation is a viable tool for scenarios where a short edit script is desired but runtime is still important.

Although the runtimes of our algorithm and XyDiff are comparable, XyDiff is still faster which was to be expected since similarity computations are more expensive than simple matching computations. In contrast, edit script quality and especially robustness is consistently improved a lot by the similiarity matching. In almost all applications, this quality/runtime tradeoff is in favor RWS-Diff, since an edit script is read more often than it is generated: A smaller edit script makes applying that script faster. Since applying an edit script to go back to a former version is the core of most version control systems that use edit scripts, the additional generation runtime will pay off by a reduced runtime for applying the script. Also when changes are used to reconcile or visualize changes, a shorter script uses less bandwidth and yields a better visualization of the changes and is thus always preferable. In conclusion, similarity-based matching is usually worth its increased runtime.

7. CONCLUSION

We proposed a method for rapidly generating an approximately cost-minimal edit script between two trees. Our approach uses subtree similarity for finding a comparably good mapping in cases where simple matching methods like top-down or hash matching fail. Nevertheless, it retains the quasi-linear runtime of such simple methods. The similarity matching is executed by first summarizing each unmapped subtree by a bag of p,q-grams, hashing each of the grams, and generating a random d-dimensional vector from each hash. Then, the vectors are added generating a random walk feature vector. The random walks possess the property that the squared euclidean distance of two walks approximates the symmetric bag distance of the corresponding bags and is therefore a suitable similarity measure. By using index structures, we perform a rapid nearest neighbors search on the feature vectors to complete the edit mapping. The proposed algorithm is flexible as it can handle various types of edit operations and works for ordered and unordered trees.

Our evaluation has shown that the similarity search is able to decrease the length of the edit scripts up to an order of magnitude while the runtime stays similar to previously published simple matching approaches. This constitutes an important advancement, since a short edit script is extremely beneficial for all applications. In addition to the overall decrease in edit script length, the chance that the matching fails—leading to a huge edit script—is drastically reduced by the similarity matching, so the quality of the generated edit script is far more constant than the quality of previous contributions. RWS-Diff is thus the first generally applicable robust tree diff algorithm with log-linear runtime complexity.

8. REFERENCES

[1] S. Abiteboul, V. Aguilera, S. Ailleret, B. Amann, F. Arambarri, S. Cluet, G. Cobena, G. Corona, G. Ferran, A. Galland, et al. Xyleme, a dynamic warehouse for XML data of the Web. In *IDEAS*, 2001.

[2] N. Augsten, M. Böhlen, C. Dyreson, and J. Gamper. Approximate joins for data-centric XML. In *ICDE*, 2008.

[3] N. Augsten, M. Böhlen, and J. Gamper. The *pq*-gram distance between ordered labeled trees. *TODS*, 35(1), 2005.

[4] D. Barnard, G. Clarke, and N. Duncan. Tree-to-tree correction for document trees. Technical report, Queen's University, Kingston, 1995.

[5] J. S. Beis and D. G. Lowe. Shape indexing using approximate nearest-neighbour search in high-dimensional spaces. In *CVPR*, 1997.

[6] L. Boyer, A. Habrard, and M. Sebban. Learning metrics between tree structured data: Application to image recognition. In *ECML*, 2007.

[7] S. S. Chawathe. Comparing hierarchical data in external memory. In *VLDB*, 1999.

[8] S. S. Chawathe, S. Abiteboul, and J. Widom. Representing and querying changes in semistructured data. In *ICDE*, 1998.

[9] S. S. Chawathe and H. Garcia-Molina. Meaningful change detection in structured data. *SIGMOD*, 1997.

[10] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom. Change detection in hierarchically structured information. *SIGMOD*, 1996.

[11] E. D. Demaine, S. Mozes, B. Rossman, and O. Weimann. An optimal decomposition algorithm for tree edit distance. *TALG*, 6(1), 2009.

[12] S. Dulucq and H. Touzet. Analysis of tree edit distance algorithms. In *CPM*, 2003.

[13] M. L. Fredman. On computing the length of longest increasing subsequences. *DM*, 11(1), 1975.

[14] S. Guha, H. V. Jagadish, N. Koudas, D. Srivastava, and T. Yu. Approximate XML joins. In *SIGMOD*, 2002.

[15] C. Hedeler and N. W. Paton. A comparative evaluation of XML difference algorithms with genomic data. In *SSDBM*, 2008.

[16] P. N. Klein. Computing the edit-distance between unrooted ordered trees. In *ESA*, 1998.

[17] K.-H. Lee, Y.-C. Choy, and S.-B. Cho. An efficient algorithm to compute differences between structured documents. *TKDE*, 16(8), 2004.

[18] T. Lindholm. XML three-way merge as a reconciliation engine for mobile data. In *MobiDE*, 2003.

[19] T. Lindholm. A three-way merge for XML documents. In *DocEng*, 2004.

[20] D. G. Lowe. Object recognition from local scale-invariant features. In *ICCV*, 1999.

[21] A. Marian. Detecting changes in XML documents. In *ICDE*, 2002.

[22] A. Mouat. XML diff and patch utilities. BSc thesis, Heriot-Watt University, Edinburgh, Scotland, 2002. `http://prdownloads.sourceforge.net/diffxml/dissertation.ps`.

[23] A. Mouat. DiffXML, 2013. `http://diffxml.sourceforge.net/`.

[24] D. Nister and H. Stewenius. Scalable recognition with a vocabulary tree. In *CVPR*, 2006.

[25] L. Paulevé, H. Jégou, and L. Amsaleg. Locality sensitive hashing: A comparison of hash function types and querying mechanisms. *PRL*, 31(11), 2010.

[26] M. Pawlik and N. Augsten. RTED: a robust algorithm for the tree edit distance. *PVLDB*, 5(4), 2011.

[27] S. Rönnau, J. Scheffczyk, and U. M. Borghoff. Towards XML version control of office documents. In *DocEng*, 2005.

[28] G. Salton and C. Buckley. Term-weighting approaches in automatic text retrieval. *IPM*, 24(5), 1988.

[29] S. M. Selkow. The tree-to-tree editing problem. *IPL*, 6(6), 1977.

[30] D. Shapira and J. A. Storer. Edit distance with move operations. In *CPM*, 2002.

[31] D. Shasha, J. T. L. Wang, K. Zhang, and F. Y. Shih. Exact and approximate algorithms for unordered tree matching. *TSMC*, 24(4), 1994.

[32] Y. Song and S. S. Bhowmick. BioDIFF: an effective fast change detection algorithm for genomic and proteomic data. In *CIKM*, 2004.

[33] D. Suciu. XML data repository, 2012. `http://www.cs.washington.edu/research/xmldatasets/www/repository.html`.

[34] Y. Wang, D. J. DeWitt, and J. yi Cai. X-Diff: An effective change detection algorithm for XML documents. In *ICDE*, 2003.

[35] H. Xu, Q. Wu, H. Wang, G. Yang, and Y. Jia. KF-Diff+: Highly efficient change detection algorithm for XML documents. In *ODBASE*, 2002.

[36] R. Yang, P. Kalnis, and A. K. H. Tung. Similarity evaluation on tree-structured data. In *SIGMOD*, 2005.

[37] K. Zhang and T. Jiang. Some MAX SNP-hard results concerning unordered labeled trees. *IPL*, 49(5), 1994.

[38] K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM J. of Computing*, 18(6), 1989.