Scaling Similarity Joins over Tree-Structured Data*

Yu Tang[†§], Yilun Cai[†], Nikos Mamoulis[†]
[†]The University of Hong Kong [§]EPFL Switzerland
{ytang, ylcai, nikos}@cs.hku.hk

ABSTRACT

Given a large collection of tree-structured objects (e.g., XML documents), the similarity join finds the pairs of objects that are similar to each other, based on a similarity threshold and a tree edit distance measure. The state-of-the-art similarity join methods compare simpler approximations of the objects (e.g., strings), in order to prune pairs that cannot be part of the similarity join result based on distance bounds derived by the approximations. In this paper, we propose a novel similarity join approach, which is based on the dynamic decomposition of the tree objects into subgraphs, according to the similarity threshold. Our technique avoids computing the exact distance between two tree objects, if the objects do not share at least one common subgraph. In order to scale up the join, the computed subgraphs are managed in a two-layer index. Our experimental results on real and synthetic data collections show that our approach outperforms the state-of-the-art methods by up to an order of magnitude.

1. INTRODUCTION

Tree-structured data are ubiquitous nowadays and there is a growing number of applications that require the management and search of such data. As concrete examples, consider XML/HTML data management in computer science [18], searching secondary RNA structures in biology [6], shape classes discovery in vision and graphics [26], and tree parsing in linguistics [22]. *Similarity search* is a common operation over collections of tree-structured objects, with *tree edit distance* (TED) being the typical similarity metric. TED models the distance (i.e., dissimilarity) between two trees as the minimum number of basic node edit operations required to transform one tree into another [4, 20]. Three node edit operations *insert*, *delete*, and *rename* are typically considered in the literature [11, 20, 29]. Given a query tree T_q and a distance threshold τ, a similarity search query [13, 16, 18, 27] finds in the database $\mathbb{T}$ (i.e., a collection of tree objects) all trees T_i, such that $TED(T_q, T_i) \leq \tau$.

In this paper, we study the *similarity join* operation on tree-structured data [18], which extends similarity search. We focus on self-joins, due to their increased applicability; still our solution is directly applicable for non-self joins. Formally, given a collection $\mathbb{T}$ of tree objects and a distance threshold τ, the objective is to report all pairs of trees (T_i, T_j) in $\mathbb{T} \times \mathbb{T}$, such that $TED(T_i, T_j) \leq \tau$.

The similarity join is an important operation that finds applications in data integration, near duplicate detection, and biological sequences comparison [4, 9, 14, 21]. For example, consider an online C2C shopping site; information about individual items (e.g., music albums) are often modeled as XML documents. The vendors could be interested in knowing similar items that are sold at other stores in order to find potential competitors; also, the site could use the join result to identify similar or near-duplicate items in order to diversify its recommendations to users. As another example, biologists are often interested in finding similar pairs of RNA secondary structures (which are modeled as trees) from various sources to better understand the relationships of different species. Moreover, finding sentences that have similar parsing structures would be useful in computational linguistics for semantic categorization.

Due to the importance of the similarity join over tree-structured data, its evaluation has been well studied in the past [4, 13, 16, 18, 27]. State-of-the-art techniques use nested loops to iterate all possible tree pairs (T_i, T_j). Pairs that cannot be part of the join result are pruned by effective filters [4, 18]; exact TED computation has to be performed only for pairs that survive the filters.

The tree edit distance $TED(T_i, T_j)$ between trees T_i and T_j can be computed recursively by first decomposing T_i and T_j into subtrees and subforests, and then computing the distances of the subtrees and subforests as subproblems [4]. Existing TED algorithms focus on minimizing the number of these subproblems during TED computation. However, the performance of these algorithms is sensitive to the structure of the trees. RTED [20] is the state-of-the-art algorithm for computing TED. The time complexity of RTED is $O(n^3)$, where T_i and T_j are both of size n. This makes a straightforward similarity join algorithm which applies RTED for all pairs of trees in the dataset too expensive. In view of this, existing approaches for solving the tree similarity join problem are based on transforming the trees into simpler data structures the distances between which are easier to compute and can be used to filter out tree pairs that cannot

*Supported by Grant 17205814 from Hong Kong RGC.
†Work done while the first author was with HKU.

Proceedings of the VLDB Endowment, Vol. 8, No. 11

be similar [18]. However, these methods either do not scale well [1, 13] or do not provide effective pruning [16, 27]. Furthermore, these solutions do not take the similarity threshold τ into consideration when performing the transformation from trees to other structures (i.e., the transformation is insensitive to τ) and do not build any index to reduce the number of comparisons between tree pairs (i.e., they have to examine all pairs of trees in a nested-loop fashion).

In view of this, in this paper, we propose a partition-based similarity join (PartSJ) framework over tree-structured datasets. In PartSJ, each tree is stored using its left-child right-sibling (LC-RS) binary tree representation. We divide each LC-RS tree into a set of subgraphs using a novel partitioning scheme, which balances the sizes of the resulting subgraphs. We prove that if two trees T_{R1} and T_{R2} are similar with each other, then there exists at least one subgraph s in the binary tree representation of T_{R1} such that s is also a subgraph of the binary tree representation of T_{R2}. Based on this observation, we design a filter which can effectively prune tree pairs that cannot be part of the join result. More specifically, we first build a two-layer index which divides the subgraphs into groups based on their labels and positions. Then, for each LC-RS tree T_i, we use the index to retrieve subgraphs of other trees that are also subgraphs of T_i. For each such subgraph, its container tree T_j is retrieved to compute its exact TED with T_i for further verification. Finally, after the processing of T_i, T_i is divided into a set of subgraphs using the same partitioning method, which are inserted into the two-layer index for comparison with subsequent trees. Our framework does not require any offline index construction; instead, we build the two-layer index for the subgraphs on-the-fly, while evaluating the join.

In summary, we make the following contributions:

- We propose a partition-based framework for similarity joins on tree-structured data. To the best of our knowledge, this is the first work which applies subgraph matching to facilitate tree similarity joins.
- We present a novel partitioning scheme which is sensitive to the similarity threshold τ and can generate balanced subgraphs for various shapes of trees efficiently. Based on this, a filtering method is proposed to prune tree pairs that cannot be part of the join result.
- We design a search paradigm for matching subgraphs with trees (i.e., checking whether a subgraph s of tree T_j is a subgraph of tree T_i or not). A two-layer index is introduced to organize the subgraphs for efficient matching, and a novel subgraph selection method is proposed to reduce comparisons and to avoid unnecessary matchings between subgraphs and trees.
- We conduct an extensive empirical study on real and synthetic tree collections and show that our proposed solution outperforms the state-of-the-art competitors.

The rest of the paper is organized as follows. Section 2 reviews TED and its computation, as well as state-of-the-art solutions for the evaluation of similarity queries on tree-structured data. Section 3 presents our proposed partition-based framework for efficient tree similarity joins. Our experimental evaluation is presented in Section 4. Section 5 reviews related work and Section 6 concludes the paper.

2. PRELIMINARIES

A tree-structured object organizes data items as *nodes* in a *hierarchy*. The nodes in a tree are linked via *edges*, which

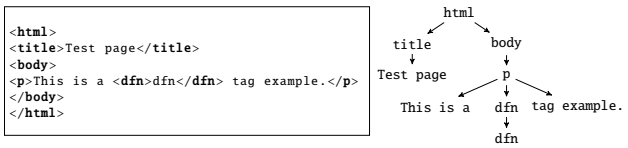


Figure 1: Tree representation of a HTML fragment

model parent-child relationships. Mathematically, a tree is an acyclic connected graph. In this paper, we focus on *rooted ordered labeled trees*, which are directed acyclic graphs with the following properties: (i) each edge points from a parent node to a child node, (ii) each node is associated with a label (two nodes can have the same label) and has at most one incoming edge, (iii) there is a unique *root* node with no incoming edge, and (iv) the children of each node are ordered. Figure 1 shows the tree representation of an HTML fragment, where tags and text are considered as labels.

Given two rooted ordered labeled trees T_1 and T_2, their tree edit distance $TED(T_1, T_2)$ is defined as the minimum number of *node edit operations* required to transform one tree into another. We consider three types of node edit operations on rooted ordered labeled trees [11, 20, 29]:

Insertion. An insert operation adds a node N_i between a parent node N_p and a subset of its consecutive children nodes (denoted as $\mathbb{N}_c$). The insertion can be viewed as adding N_i as a child node of N_p directly before the nodes in $\mathbb{N}_c$ first and then moving the nodes in $\mathbb{N}_c$ as direct children of N_i, while preserving their original order.

Deletion. A delete operation removes a node N_i from the tree by first deleting N_i from its parent node N_p and then connecting all the children nodes of N_i (if any) as children of N_p in place of N_i, while preserving their original order. Deletion and insertion are *inverse* edit operations.

Renaming. A rename operation changes the label of a node N_i into a different label.

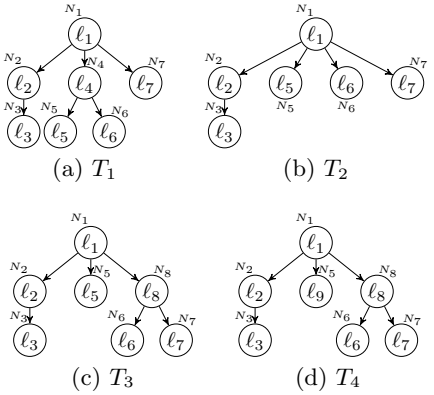


Figure 2: Tree node edit operations example

Figure 2 illustrates a sequence of node edit operations on an initial tree T_1. First, T_1 is transformed into T_2 by deleting node N_4. Then, inserting a new node N_8 between nodes N_1 and $\{N_6, N_7\}$ converts tree T_2 into T_3. Finally, renaming the label of N_5 from ℓ_5 to ℓ_9 results in the final tree T_4.

TED between two trees can be computed recursively. The worst-case optimal algorithm [11], has $O(n^3)$ time and $O(n^2)$ space complexity, for two trees of size n. For certain tree

topologies, such as balanced trees the complexity can be lowered to $O(n^2 \log^2 n)$ by an approach [29], which however has $O(n^4)$ time complexity in the worst case. A recently proposed *robust hybrid framework* (RTED) [20] dynamically chooses the best one between the algorithms of [11] and [29], based on the tree shapes (of subproblems). The time and space complexities of RTED are $O(n^3)$ and $O(n^2)$, respectively. In this paper, we use RTED for TED computations.

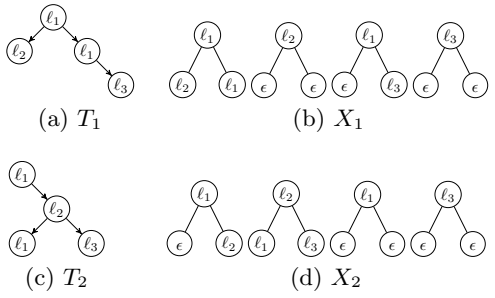


Figure 3: An example of binary branch

Since the computational cost of TED is too high, existing approaches for tree similarity search/join queries are generally based on the transformation of trees into simpler data structures, the distances between which are easier to compute and can be used to bound the TED. We now introduce the two state-of-the-art methods for similarity search queries on tree-structured data based on a recent survey [4] and experimental study [18]. Both approaches can be easily adapted to solve the similarity join problem in an indexed nested loops fashion (i.e., by issuing a similarity search query for each tree in the collection). The first method is proposed in [13], where the authors prove that the string edit distance between the preorder/postorder traversal sequences of two trees can serve as a lower bound of their TED, thus can be used to filter out dissimilar tree pairs. For example, consider trees T_1 and T_2 shown in Figure 3. It is easy to verify that $TED(T_1, T_2) = 3$. The string edit distances between the preorder traversal sequences (both are $\ell_1\ell_2\ell_1\ell_3$) and the postorder traversal sequences ($\ell_2\ell_3\ell_1\ell_1$ and $\ell_1\ell_3\ell_2\ell_1$) of the two trees are 0 and 2 respectively; both are no larger than the real $TED(T_1, T_2)$. The second solution [27], proposes to bound the tree edit distance between two binary trees by the dissimilarity between two bags of *binary branches*. More specifically, given a binary tree T_1, a binary branch of T_1 is a one-level tree structure of T_1 which consists of a node $N_i \in T_1$ and its two children (a dummy node N_ϵ with empty label ϵ is added if a child node does not exist). Thus, a tree T_i contains $|T_i|$ binary branches. Given two binary trees T_1 and T_2, whose bags of binary branches are X_1 and X_2 respectively, the binary branch distance between T_1 and T_2 is defined as $BIB(T_1, T_2) = |X_1| + |X_2| - 2|X_1 \cap X_2|$, where $X_1 \cap X_2$ is bag intersection [2]. The authors prove in [27] that $BIB(T_1, T_2) \leq 5 \cdot TED(T_1, T_2)$. Figure 3 illustrates two binary trees T_1 and T_2, as well as their corresponding bags of binary branches X_1 and X_2, respectively; it can be verified that $BIB(T_1, T_2) = 6 \leq 5 \cdot TED(T_1, T_2) = 15$.

3. PARTITION-BASED SIMILARITY JOIN

In this section we present our partition-based framework for similarity joins on tree-structured data. The main idea is based on the *binary tree representation* of a rooted ordered labeled tree and *subgraph indexing* for effective pruning.

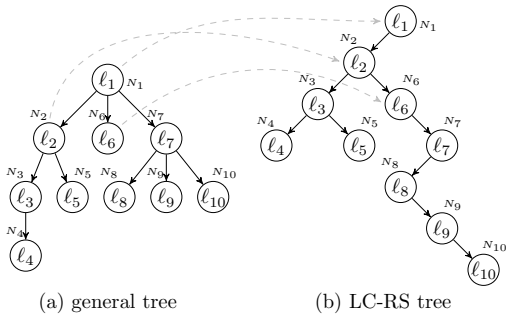


Figure 4: An example of Knuth transformation

3.1 Filtering Principle

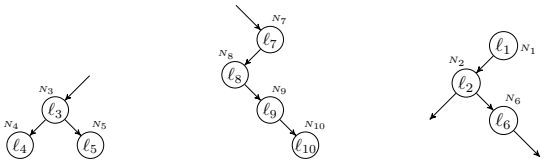
Since there is no bound on the number of children of a node in a general *rooted ordered labeled* tree, a node edit operation (e.g., the deletion of a node) may involve an arbitrary number of changes to the parent-child relationships between nodes. On the other hand, in a *binary* tree, the number of nodes affected by a node edit operation is strictly constrained, since each node is connected to at most 3 other nodes. For example, consider a general tree T_R and its equivalent binary tree representation T_B, as shown in Figures 4(a) and 4(b), respectively. The deletion of N_7 from the general tree affects 4 tree nodes and 4 edges, while on the binary tree it only affects 2 nodes/edges. Motivated by this, our join framework (candidate generation) applies on binary tree representations of the original tree objects.

One of the most popular approaches for mapping a general tree T_R to a binary tree T_B is *Knuth's transformation*, which creates a left-child right-sibling (LC-RS) (binary) tree [10]. Instead of having an array of pointers referring to each of its children, each node in an LC-RS tree has (at most) two pointers only, which suffice to access all of its children. More specifically, a node N in an LC-RS tree may contain a *left-child* pointer which points to the leftmost child of N (recall that the general tree is an ordered tree), and a *right-sibling* pointer which points to the sibling of N that is immediately to the right of N. Note that node labels remain unchanged during such a conversion. Figure 4 shows an example of a general tree and its corresponding LC-RS tree.

Before presenting the filtering principle of our partition-based join framework, we first introduce the concept of δ-partitioning, which is the basis of our framework.

Definition 1 (δ-Partitioning and Bridging Edge). *A δ-partitioning of a tree T_B consists of $(\delta - 1)$ edges of T_B; each edge in the δ-partitioning is called a bridging edge.*

Obviously, removing the *bridging edges* in any δ-partitioning of tree T_B from T_B results in δ disjoint *subgraphs* (i.e., components) of T_B. In our definition, besides the connected component, a *subgraph* also includes the *bridging edges* in the δ-partitioning that connect this subgraph to other subgraphs. Figure 5 illustrates a possible 3-partitioning for the LC-RS tree shown in Figure 4(b) (i.e., $\{\langle N_2, N_3\rangle, \langle N_6, N_7\rangle\}$). A subgraph resulting from a δ-partitioning contains at least one bridging edge (Definition 1) that belongs to that δ-partitioning. Obviously, each



(a) subgraph s_1 (b) subgraph s_2 (c) subgraph s_3

Figure 5: A 3-partitioning of the tree in Figure 4(b)

subgraph is also a *rooted binary tree* itself, if we omit the bridging edges that belong to the δ-partitioning.

In a subgraph (and in a LC-RS tree in general), at most three edges are associated with a tree node, belonging to one of the following categories: (i) right incoming edge (from the parent's left pointer), (ii) left incoming edge (from the parent's right pointer), (iii) left outgoing edge (pointing to the left child), and (iv) right outgoing edge (pointing to the right child). For example, N_3 in Figure 5(a) is associated with three edges: one right incoming edge, one left outgoing edge, and one right outgoing edge; while N_7 in Figure 5(c) is associated with a left incoming edge and a left outgoing edge. Note that each edge in a binary tree belongs to two categories (i.e., a different category for the source and destination nodes); e.g., the edge that connects N_2 and N_3 is a left outgoing edge for N_2 and a right incoming edge for N_3.

Given a δ-partitioning of a binary tree representation T_B of T_R, we show in Lemma 1 that the effects that a node edit operation has on the subgraphs of T_B are restricted.

LEMMA 1. *Let T_B be the LC-RS tree representation of a general tree T_R. For any δ-partitioning on T_B, any node edit operation on T_R changes at most 2 subgraphs of T_B.*

PROOF. We prove the lemma by enumerating the changes that each node edit operation makes.

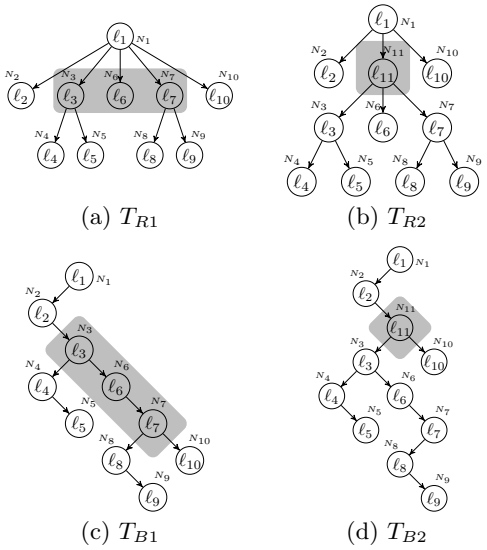
Renaming. The case of *renaming* is trivial: only the subgraph containing the node being renamed is changed.

Insertion. Assume that a new node N_x is inserted between a parent node N_p and a subset of its n consecutive children nodes $\{N_{i1}, N_{i2}, \ldots, N_{in}\}$. We show that in this case only the subgraphs containing either N_{i1} or N_{in} have to be changed. More specifically, if N_{i1} contains a right incoming edge (i.e., N_{i1} is not the leftmost child of N_p), then this edge will be changed to a left incoming edge (i.e., N_{i1} becomes the leftmost child of the newly inserted node); if N_{in} contains a right outgoing edge (i.e., N_{in} is not the rightmost child of N_p), then this edge will be eliminated (as N_{in} becomes the rightmost child of the newly inserted node). In summary, the insertion may affect only nodes N_{i1} and N_{in} (and their associated edges), which belong to at most 2 different subgraphs (w.r.t. any δ-partitioning on T_B).

Deletion. The analysis is similar to that of insertion, since they are inverse operations of each other. Specifically, assuming that N_x is the node to be deleted, the deletion of N_x may affect at most 2 subgraphs (w.r.t. any δ-partitioning on T_B). We omit the details here due to space limitations.

In conclusion, we have shown that any node edit operation changes at most 2 subgraphs of T_B. □

For example, assume that an insert operation changes T_{R1} of Figure 6(a), where N_{11} is inserted between N_1 and $\{N_3, N_6, N_7\}$. Figure 6(b) illustrates the tree after insertion,



(a) T_{R1} (b) T_{R2} (c) T_{B1} (d) T_{B2}

Figure 6: Node edit operation effects

denoted as T_{R2}. Figures 6(c) and 6(d) are the binary tree representations of T_{R1} and T_{R2} respectively. For the subgraph of T_{B1} containing N_3, the change includes altering N_3's right incoming edge into a left incoming edge; for the subgraph of T_{B1} containing N_7, the change includes eliminating N_7's right outgoing edge. In the worst case, N_3 and N_7 belong to different subgraphs, thus at most 2 subgraphs are changed by this insertion.

Lemma 1 leads to the following filtering principle:

LEMMA 2. *Consider a binary tree representation T_{B1} of a general tree T_{R1} and a similarity threshold τ. For any δ-partitioning of T_{B1} ($\delta = 2\tau + 1$) and any general tree T_{R2} with binary tree representation T_{B2}, if $TED(T_{R1}, T_{R2}) \leq \tau$, T_{B1} must contain at least one subgraph s (by the δ-partitioning) such that s is a subgraph of T_{B2}.*

PROOF. If $TED(T_{R1}, T_{R2}) \leq \tau$, then there exists a sequence of λ ($\leq \tau$) node edit operations $O = \langle o_1, o_2, \ldots, o_\lambda \rangle$ that convert T_{R1} into T_{R2}. Based on Lemma 1, any node edit operation on T_{R1} changes at most 2 subgraphs in T_{B1}. Hence at most 2λ ($\leq 2\tau$) subgraphs in T_{B1} are changed by applying O. On the other hand, T_{B1} is partitioned into $(2\tau+1)$ subgraphs, which means that at least one subgraph s in T_{B1} is untouched. Since s remains the same, its corresponding structure in T_{R1} is unaffected; such a structure will appear in T_{R2} as well (clearly, if a subgraph is not changed, its binary representation remains the same). Therefore, s must be a subgraph of T_{B2} (i.e., s is included in T_{B2}), the binary tree representation of T_{R2}. □

Lemma 2 can be used as follows. Consider the subgraphs of a tree $T_{B1} \in \mathbb{T}$ based on any δ-partitioning ($\delta = 2\tau + 1$). When processing tree $T_{B2} \in \mathbb{T}$, if there is no subgraph s of T_{B1}, which is also a subgraph of T_{B2} (i.e., appearing in T_{B2}), then it is guaranteed that (T_{B1}, T_{B2}) is not a similarity join result on $\mathbb{T}$ (with TED threshold τ); therefore the tree pair can be pruned.

3.2 Partition-based Framework

Based on Lemma 2, we propose a partition-based tree similarity join framework PARTSJ, summarized by Algorithm 1.

Algorithm 1 PARTITION-BASED SIMILARITY JOIN

```
1: procedure PartSJ(Trees 𝕋, τ)
2:     ℐ := ∅;                                   ▷ inverted index of tree size
3:     sort trees in 𝕋 in ascending order of their sizes;
4:     for each T_i ∈ 𝕋 do
5:         for n := max{1, |T_i| − τ} → |T_i| do
6:             for each node N ∈ T_i do          ▷ postorder traversal
7:                 𝕊 = ℐ_n.getSubgraphs(T_i, N);
8:                 for each s ∈ 𝕊 do
9:                     T_j := the tree which owns s;
10:                    if (T_i, T_j) has not been checked before then
11:                        if TED(T_i, T_j) ≤ τ then
12:                            report tree pair (T_i, T_j);
13:        γ := MaxMinSize(T_i, 2τ + 1);
14:        𝕊′ := Partition(T_i, 2τ + 1, γ);
15:        for each s ∈ 𝕊′ do
16:            insert s into ℐ_{|T_i|};
```

Algorithm 1 initializes an empty *inverted size index* $\mathcal{I}$ of subgraphs, which is populated on-the-fly, while processing the join. $\mathcal{I}$ contains one inverted list $\mathcal{I}_n$ for every possible tree size n; $\mathcal{I}_n$ includes the subgraphs generated from the trees that have exactly n nodes. The algorithm examines the trees in increasing order of their sizes (line 3). For each tree T_i, the sizes of trees T_j that are similar to T_i (i.e., $TED(T_i, T_j) \leq \tau$) should be in the range $[|T_i| - \tau, |T_i| + \tau]$, since each node edit operation changes the size of a tree by at most 1.[1] Still, since the trees are ordered by size, all the trees that have been processed so far cannot have size larger than $|T_i|$ (line 5), thus candidates are only the trees seen so far with size at least $|T_i| - \tau$. The subgraphs of these trees are retrieved from inverted lists $\mathcal{I}_{|T_i|-\tau}$ to $\mathcal{I}_{|T_i|}$.

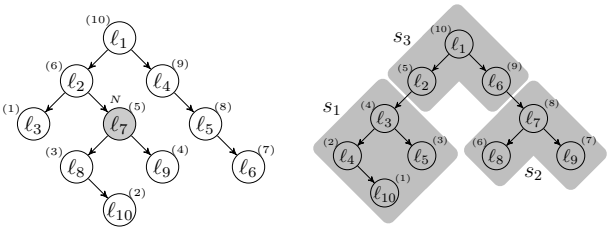


Figure 7: An example of subgraph selection

A straightforward implementation would retrieve all these subgraphs and check whether they appear in T_i; for each retrieved subgraph s, all nodes of T_i should be enumerated to decide whether s matches the subtree of T_i rooted at each node $N \in T_i$. In particular, s *matches* the subtree rooted at node N of T_i iff the root of s matches N (i.e., they have the same label) and so do the children, descendants, and bridging edges of s (i.e., s matches the structure at the top of the subtree). For example, consider tree T_j in Figure 7, which is partitioned into 3 subgraphs. Subgraph s_2 of T_j matches the subtree of T_i rooted at N, because the label of N is ℓ_7 (like the root of s_2), the two children of s_2's root have the same labels as the two children of N (note that the grandchild of N is not relevant to this matching), and both s_2 and N have a left incoming edge (bridging edge).

Such a brute-force enumeration of all nodes in T_i for each subgraph s is too time-consuming and the join becomes very slow if we have numerous large trees and/or large τ values (recall that larger τ values result in more subgraphs and larger tree sizes require checking more candidate tree nodes). To this end, we propose to select interesting subgraphs reversely, i.e., instead of matching each subgraph with the current tree T_i, we enumerate the nodes N of T_i (line 6) and select only the subgraphs that can possibly match with the subtree rooted at each node N (line 7). Consider the toy example in Figure 7, where T_i contains 10 nodes and $\tau = 1$. The number in parentheses next to each node indicates the *postorder* number of that node in the corresponding tree. Assume that $\mathcal{I}_{10}$ contains only three subgraphs s_1, s_2, and s_3 from tree T_j (note that $|T_j| = 10$). A brute-force approach would retrieve all the three subgraphs in $\mathcal{I}_{10}$ and compare each of them with all the possible subtrees (10 in total) of T_i. On the other hand, our solution selects for each node N of T_i only a subset of subgraphs from $\mathcal{I}_{10}$ (e.g., only s_2 may be selected for the subtree rooted at N in Figure 7), based on the *structure* of the subtree rooted at N and the *position* of N. In Section 3.4, we present an effective indexing approach for the inverted lists $\mathcal{I}_n$, based on which $\mathcal{I}_n.getSubgraphs(T_i, N)$ (line 7) is implemented efficiently.

For each returned subgraph $s \in \mathbb{S}$, if s matches T_i (i.e., s is a subgraph of T_i), then (T_i, T_j) becomes a candidate join pair and the TED between the two trees is computed. If this distance is no larger than τ, (T_i, T_j) is reported as a result (lines 9–12). Finally, T_i is partitioned into $\delta = 2\tau + 1$ subgraphs, which are added into inverted list $\mathcal{I}_{|T_i|}$, for comparison with all subsequently examined trees.

3.3 Partitioning Scheme

We now investigate how to obtain a good δ-partitioning in order to maximize the effectiveness of our approach. Intuitively, a good partitioning scheme should not generate very small subgraphs for any given tree, since small subgraphs have a high probability of being subgraphs of other trees. Based on Lemma 2, if more common subgraphs are identified, more candidates will be generated for which the exact TED should be computed; thus degrading the performance (pruning power) of our framework. An intuitive approach would be to divide T_B into a set of subgraphs which have (almost) the same size; this way, the minimum size of any subgraph would be maximized. However, such a partitioning may not exist due to the complexity of tree-structured data. Consider, for instance, a binary tree as shown in Figure 8, where each triangle represents a binary tree structure consisting of 50 nodes (hence the tree contains 202 tree nodes in total). Assume that $\delta = 3$ (i.e., $\tau = 1$). Ideally we want to partition this tree into 3 subgraphs each having 67 or 68 nodes respectively. However, we can easily show that any 3-partitioning of this tree will include a subgraph of at most 50 nodes and a subgraph of at least 100 nodes; thus there will be a great difference among the sizes of these subgraphs.

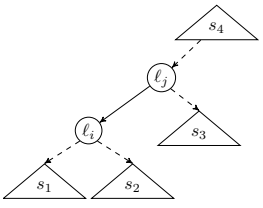


Figure 8: An example of binary tree partition

[1] An insertion (deletion) increases (decreases) the tree size by 1, while renaming does not affect it.

In view of this, we propose a partitioning approach which aims at *maximizing the minimum size* of the subgraphs for a binary tree. This objective helps in ending up with as balanced subgraphs as possible and at the same time avoids having small partitions. Before presenting our method, we first show how to solve the decision version of this problem, i.e., given a size constraint γ, find whether there exists a δ-partitioning for a given binary tree T_B such that the sizes of all the resulting subgraphs are all at least γ.

DEFINITION 2 ((δ,γ)-PARTITIONABLE). *Given a binary tree T_B and a size constraint γ, if there exists a δ-partitioning of T_B such that $\gamma \cdot \delta \leq |T_B|$ and the size of each subgraph is no less than γ, we say T_B is (δ,γ)-partitionable.*

Our solution for the (δ,γ)-partitionable problem is based on the concept of γ-subtree.

DEFINITION 3 (γ-SUBTREE). *Given a size constraint γ, a subtree s_i of which the left and right subtrees are s_{il} and s_{ir} respectively, is called a γ-subtree if and only if $|s_i| \geq \gamma$, $|s_{il}| < \gamma$ and $|s_{ir}| < \gamma$.*

For example, the subtree rooted at N_3 of T_{B2} in Figure 6(d) is a 5-subtree. It is easy to see that each binary tree T_B contains at least one γ-subtree, given that $\gamma \leq |T_B|$; we can use the following recursive approach to discover a possible γ-subtree from T_B: starting from the subtree rooted at node N_i of which the size is no less than γ (initially N_i is the root node of T_B), we check whether its two subtrees satisfy the size constraint. If yes, then we report the subtree rooted at N_i as a result; otherwise there must exist one child node (denoted as N_c) of which the size is no less than γ, and thus we search for a γ-subtree in the subtree rooted at N_c. Note that no γ-subtree can span multiple subgraphs after a legal (δ,γ)-partitioning, since none of its child branches has size that is larger than or equals to γ.

LEMMA 3. *If a binary tree T_B is (δ,γ)-partitionable, then detaching a γ-subtree from T_B will derive a binary tree which is $(\delta-1,\gamma)$-partitionable.*

PROOF. Let the γ-subtree (to be detached) in T_B be t_γ, and the residual tree after detaching t_γ from T_B be T_r. Obviously, T_r is still a rooted binary tree. Since T_B is (δ,γ)-partitionable, there exists at least one δ-partitioning $\mathcal{P}$, which cuts T_B into a set of subgraphs $\mathbb{S}$, such that for each subgraph s_i in $\mathbb{S}$, $|s_i| \geq \gamma$. Note that t_γ must be completely inside one subgraph of T_B (by applying $\mathcal{P}$), since every subtree of t_γ (except t_γ itself) has size smaller than γ, and thus must be included in the same subgraph as its parent (i.e., every node in t_γ appears in the same subgraph).

Assume that the subgraph that contains t_γ is s_γ. We denote the set of (bridging) edges in $\mathcal{P}$ that appear also in s_γ as Ω, and construct a new partitioning $\mathcal{P}'$ (for T_r) by deleting from $\mathcal{P}$ an edge $e_i \in \Omega$ (i.e., $\mathcal{P}' = \mathcal{P} \setminus \{e_i\}$). Obviously, $\mathcal{P}'$ divides T_r into $\delta-1$ subgraphs (trivial since removing $\delta-2$ edges from a tree will result in $\delta-1$ disjoint subgraphs); we now show the size of each such subgraph is not smaller than γ. Let the subgraph that connects to s_γ via e_i be s_x. We can easily see that the other $\delta-2$ subgraphs (i.e., $\mathbb{S} \setminus \{s_x, s_\gamma\}$) remain unchanged, and thus their sizes are no less than γ. Also, we connect the residual in s_γ (after detaching t_γ) with s_x to form a new subgraph s'_x (i.e., the remaining subgraph in $\mathcal{P}'$); clearly, the size of s'_x is also not smaller than γ. Therefore, T_r is $(\delta-1,\gamma)$-partitionable. □

Based on Lemma 3, we propose a greedy algorithm to solve the (δ,γ)-partitionable problem in linear time $O(|T_B|)$. In order to achieve this goal, we store at each tree node N_i two variables, *size* and *detached*: *size* stores the number of nodes in the subtree rooted at N_i, while *detached* stores the number of nodes that have been cut off in the subtree rooted at N_i. In other words, $(size - detached)$ is the actual number of nodes that are under N_i at the current stage. These two statistics will be updated while we traverse T_B. Note that in order to obtain sufficient information when processing the subtree rooted at N_i, we follow a *postorder* traversal of the binary tree to get the statistics from N_i's children first.

Algorithm 2 (δ,γ)-PARTITIONABLE TEST

```
1: procedure PARTITIONABLE(T_B, δ, γ)
2:     n := 0;                          ▷ the number of subgraphs found by far
3:     root := the root node of T_B;
4:     return RECURSIVEPARTITIONABLE(root, δ, γ);
5: procedure RECURSIVEPARTITIONABLE(N_i, δ, γ)
6:     N_i.size := 1;
7:     N_i.detached := 0;
8:     N_l := the left child node of N_i;
9:     N_r := the right child node of N_i;
10:    if N_l ≠ NIL then
11:        succ := RECURSIVEPARTITIONABLE(N_l, δ, γ);
12:        if succ then return true;
13:        N_i.size := N_i.size + N_l.size;
14:        N_i.detached := N_i.detached + N_l.detached;
15:    if Nr ≠ NIL then
16:        succ := RECURSIVEPARTITIONABLE(N_r, δ, γ);
17:        if succ then return true;
18:        N_i.size := N_i.size + N_r.size;
19:        N_i.detached := N_i.detached + N_r.detached;
20:    if N_i.size − N_i.detached ≥ γ then      ▷ γ-subtree identified
21:        n := n + 1;
22:        N_i.detached := N_i.size;
23:        if n ≥ δ then
24:            return true;
25:    return false;
```

Algorithm 2 tests whether a binary tree T_B is (δ,γ)-partitionable by greedily cutting off a γ-subtree found so far from the current residual tree (line 20). Note that we do not conduct any real detach operations in Algorithm 2, but we use the two variables *size* and *detached* to maintain the information after each detach operation. This way, we do not need to recalculate the real number of nodes that remain in the subtree rooted at the current tree node after each detach operation. As long as enough γ-subtrees are discovered (line 23), the algorithm returns true (i.e., that the tree is (δ,γ)-partitionable), otherwise it reports failure of the partitioning test (line 25). Note that Algorithm 2 not only determines whether a binary tree T_B is (δ,γ)-partitionable, but also computes (with minor modifications) a δ-partitioning when such a partitioning exists (cf. Lemma 3).

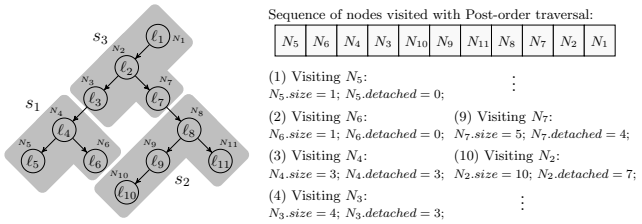


Figure 9: (δ,γ)-partitionable test example

Figure 9 illustrates the application of Algorithm 2 on a binary tree, with $\delta = 3$ and $\gamma = 3$. The processing order of tree nodes, as well as the final values of variables in each node after processing are listed on the right side of the figure. For example, after processing the left and right children of N_4, $N_4.size$ and $N_i.detached$ are updated to 3 and 0, respectively. Since $N_4.size - N_i.detached \geq 3$, a 3-subtree is identified and detached from the binary tree, resulting in a new binary tree of eight nodes. In addition, we set the value of $N_4.detached$ to $N_4.size$ to indicate that the subtree rooted at N_4 has been detached. Later, when processing N_4's parent (N_3), we obtain these statistics from N_4.

LEMMA 4. *Given a binary tree T_B, if it is (δ, γ)-partitionable, then T_B is also $(\delta, \gamma - 1)$-partitionable.*

PROOF. Trivial, since a subtree not smaller than γ is also not smaller than $(\gamma - 1)$. □

Given a binary tree T_B and the number of subgraphs δ after partitioning, according to Lemma 4, there should be a value γ, such that for all $\gamma' \leq \gamma$, T_B is (δ, γ')-partitionable; while for any $\gamma'' > \gamma$, T_B is not (δ, γ'')-partitionable. This observation motivates us to use binary search to find the maximum value of γ, for which the tree is (δ, γ)-partitionable. This value can be considered as an upper bound of all feasible γ values.

Algorithm 3 MAXIMIZING THE MINIMUM SUBGRAPH SIZE

```
1:  procedure MaxMinSize(T_B, δ)
2:      γ_max := ⌊|T_B|/δ⌋;
3:      γ_min := ⌊(|T_B| + δ − 1)/(2δ − 1)⌋;
4:      c := γ_max − γ_min + 1;
5:      while c > 1 do
6:          γ_mid := γ_min + ⌊c/2⌋;
7:          if Partitionable(T_B, δ, γ_mid) then
8:              γ_min := γ_mid;
9:              c := c − ⌊c/2⌋;
10:         else
11:             c := ⌊c/2⌋;
12:     return γ_min;
```

Algorithm 3 shows the pseudocode for finding this value of γ. Obviously if T_B is (δ, γ)-partitionable, then $\gamma \leq |T_B|/\delta$ (cf. Definition 3). In other words, the size of each subgraph cannot be larger than $\lfloor |T_B|/\delta \rfloor$ (line 2). On the other hand, the trivial lower bound of γ can be set to 1. In order to obtain a non-trivial lower bound for γ, consider the worst case where the left subtree and right subtree of each subgraph isolated in the first $\delta - 1$ iterations are both of size $(\gamma - 1)$, i.e., all these subgraphs found so far are of size $(2\gamma - 1)$. Since a tree is divided into δ subgraphs, to make sure that the size of the last subgraph is no less than γ, we should have $|T_B| - (\delta - 1) \cdot (2\gamma - 1) \geq \gamma$. Solving this inequality results in $\gamma \leq (|T_B| + \delta - 1)/(2\delta - 1)$, indicating that T_B is definitely (δ, γ)-partitionable if γ is no larger than $(|T_B| + \delta - 1)/(2\delta - 1)$ (line 3). In other words, T_B is (δ, γ_{min})-partitionable. The loop invariant for the while loop (lines 5–11) is that the largest feasible value γ is in the range $[\gamma_{min}, \gamma_{min} + c)$. If c becomes 1 (i.e., there is only one feasible value now), then γ_{min} is the desired answer. Otherwise, Algorithm 3 tests iteratively whether the value in the middle of the range (i.e., γ_{mid}) can derive a feasible partition or not (line 7). If T_B is (δ, γ_{mid})-partitionable, we can guarantee that the largest feasible value of γ is in the right half range $[\gamma_{mid}, \gamma_{mid} + c - \lfloor c/2 \rfloor)$ (lines 8–9). Otherwise it is in the left half range $[\gamma_{min}, \gamma_{min} + \lfloor c/2 \rfloor)$ (line 11).

Time complexity. The algorithm enters the while loop (lines 5–11) at most $O(\log(|T_B|/\delta))$ times, since at each iteration the search range is shrunk to half of its original length. In the loop body, we conduct a partitionable test (Algorithm 2) in $O(|T_B|)$ time, as well as some constant time operations. Therefore, the overall time complexity of Algorithm 3 is $O(|T_B| \cdot \log(|T_B|/\delta))$.

3.4 Subgraph Selection

The heart of our framework (presented in Section 3.2) is to find for each node N of the currently examined tree T_i, the subgraphs of previously accessed trees, which match with the subtree of T_i rooted at N (line 7 of Algorithm 1). For each such subgraph s, the tree T_j that contains s forms a candidate join pair with T_i. We now present two orthogonal indexing techniques for each inverted list $\mathcal{I}_n$ (containing the subgraphs of n-sized trees). These indexing techniques facilitate the efficient access of the relevant subgraphs that match N. The main idea is to partition the subgraphs in $\mathcal{I}_n$ into groups according to their *labels* and *position*, such that groups that may not contain possible candidates for T_i can be pruned. We aim at filtering out subgraphs s that (i) definitely do not match the subtree rooted at N, and (ii) although they may match the subtree, their position in their container tree T_j is not compatible with the position of N in T_i, and thus (T_i, T_j) is not a candidate pair due to s.

Label indexing. To filter out subgraphs that definitely do not match the subtree rooted at node $N \in T_i$, we can index the subgraphs in $\mathcal{I}_n$ by the labels in their topmost twig. More specifically, consider a subgraph s with root N_a and let N_b and N_c be the left and right child of N_a, respectively. If N_b (or N_c) does not exist, it becomes a dummy node (denoted as N_ϵ). Assume that the labels of N_a, N_b and N_c are are ℓ_a, ℓ_b and ℓ_c, respectively (the label of a dummy node is denoted by ϵ). By pre-ordering the three labels as $\ell_a\ell_b\ell_c$, we assign key $\ell_a\ell_b\ell_c$ to s and put s into the group corresponding to this key (we create the group if it does not already exist). For instance in Figure 7, the three subgraphs of T_j, namely s_1, s_2 and s_3, are assigned to groups corresponding to keys $\ell_1\ell_2\ell_6$, $\ell_3\ell_4\ell_5$ and $\ell_7\ell_8\ell_9$ respectively.

When retrieving subgraphs for node $N \in T_i$, we create four search keys based on N. Specifically, if N_l and N_r are the children of N and the labels of N, N_l and N_r are ℓ, ℓ_l and ℓ_r, we form keys $\ell\ell_l\ell_r$, $\ell\ell_l\epsilon$, $\ell\epsilon\ell_r$, and $\ell\epsilon\epsilon$. Only the groups in $\mathcal{I}_n$ that correspond to these keys can possibly match the subtree rooted at N. The subgraphs in group $\ell\ell_l\ell_r$ have the same topmost twig as that rooted at node N; therefore they are possible candidates for matching N and should be accessed and verified. For example, s_2 in Figure 7(b) will be selected for $N \in T_i$ since it is inside group $\ell_7\ell_8\ell_9$. For the subgraphs in the remaining three groups, although the topmost twigs are different from that rooted at node N, they can still possibly match the subtree rooted at node N; hence they should be selected and verified as well.

Postorder pruning. When a subgraph s of tree T_j matches a subtree rooted at node N of T_i, this does not necessarily mean that (T_i, T_j) is a candidate join pair. We observe that if the positions of s in T_j and N in T_i are quite different then the matching does not imply the candidature of (T_i, T_j). For example, in Figure 7, assume that $\tau = 1$ and that s_2 of tree T_j is retrieved and found to be matching with

the subtree rooted at N of tree T_i. Observe that since s_2 and N are in very different positions in T_j and T_i, respectively, we can conclude that (T_i, T_j) cannot be a join pair due to this matching. If our search for subgraphs that match node N could exclude s_2 due to its different relative position, then we could avoid having (T_i, T_j) as a candidate pair and thus avoid a necessary TED computation; we could also save the cost of checking if there is a potential match of s_2 with the subtree rooted at N in T_i. We now present the details of an indexing approach for $\mathcal{I}_n$ which partitions the subgraphs based on their *postorder* numbers in their container trees. When a node $N \in T_i$ is processed, we can use the postorder of N in T_i to search for subgraphs that may render their container trees candidate join pairs with T_i.

Consider a δ-partitioning of a tree T_j derived by our proposed partitioning scheme in Section 3.3, and let the resulting subgraphs be $s_1, s_2, \ldots, s_\delta$, and their respective identifiers in a postorder traversal of T_j be $p_1, p_2, \ldots, p_\delta$. For example, the postorder identifiers of the subgraphs s_1, s_2, and s_3 (i.e., p_1, p_2, and p_3) in Figure 7(b) are 4, 8, and 10 respectively. Let p be the postorder of $N \in T_i$.

Assume a matching is found between a subgraph s_k (of tree T_j) and a subtree rooted at $N \in T_i$. Without affecting the completeness of the results, s_k can be safely discarded if (i) we know that there exists another subgraph $s_{k'}$ ($k' \neq k$) of T_j that appears in T_i which will be selected (thus T_j is not missed if it is a true result), or (ii) the matching will definitely result in a false positive candidate (thus the exclusion of s_k does not affect the final result). Otherwise s_k should be selected by our selection methodology.

By definition, if T_j and T_i are similar, T_j can be transformed into T_i using at most τ node edit operations. Assume that among these node edit operations, Δ of them are performed on nodes in the last $\delta - k + 1$ subgraphs of T_j (i.e., $s_k, s_{k+1}, \ldots, s_\delta$). If $\Delta \geq (\delta - k + 1)/2$, the number of edit operations applied on those nodes in the first $k-1$ subgraphs (i.e., $s_1, s_2, \ldots, s_{k-1}$) is at most $\tau - (\delta - k + 1)/2 = k/2 - 1$; these node edit operations change at most $2 \cdot (k/2 - 1) = k - 2$ subgraphs of them (cf. Lemma 1). Therefore, there exists at least one subgraph $s_{k'}$ ($k' < k$) among the first $k-1$ subgraphs of T_j that is unchanged by the node edit operations; in other words, $s_{k'}$ appears also in T_i. In this case, we can discard s_k because of the existence of such $s_{k'}$, if $s_{k'}$ should be selected. On the other hand, when $\Delta < (\delta - k + 1)/2$, we cannot discard s_k based on condition (i). However, recall that according to our partitioning scheme, the last $\delta - k + 1$ subgraphs still form a *connected* binary tree T^*. For example in Figure 7(b), if we remove s_1, we still have a connected binary tree T^* of 6 nodes. In addition, as we examine subgraphs in a depth-first search (post-order) manner, all the nodes with postorder numbers greater than p_k (the postorder identifier of subgraph s_k) only appear in subgraphs after s_k. Denote these nodes as $\mathbb{N}_k$. Assume, in the extreme case, the Δ node edit operations are all applied on these nodes. Since these nodes belong to the binary tree T^* and one node edit operation changes the size of the T^* by at most 1, Δ node edit operations changes the size of the T^* by at most Δ; i.e., the size of $\mathbb{N}_k$ is changed by at most Δ as well. As a result, the postorder identifier p_k shifts at most Δ ($< (\delta - k + 1)/2$) positions in tree T_i. Let Δ' be the largest integer smaller than $(\delta - k + 1)/2$, i.e., $\Delta' = \tau - \lfloor k/2 \rfloor$. Then, $p \in [p_k - \Delta', p_k + \Delta']$. In other words, if the matchings occur between s_k and nodes whose postorder numbers do not fall into this range, s_k can be safely discarded since such matchings only provide false positive candidature results.

Based on the above reasoning, for each subgraph s_k whose postorder identifier is p_k, for each value $v \in [p_k - \Delta', p_k + \Delta']$, we assign s_k to group with key v. For example, consider the subgraph s_2 in Figure 7(b) where $\tau = 1$, $k = 2$, and $p_k = 8$; since $\Delta' = \tau - \lfloor k/2 \rfloor = 0$, the corresponding range becomes $[8, 8]$. As a result, we assign s_2 to group with key 8. When our similarity join framework (line 7 of Algorithm 1) retrieves subgraphs for node N (with postorder number p) in T_i, we only need to access subgraphs in group with key p. It is guaranteed that considering only these subgraphs is sufficient for not missing any join results. For example, in Figure 7, when selecting subgraphs for node N (with postorder number 5) in T_i, s_2 (of T_j) will not be selected since it is not assigned to the group with key 5. Thus we avoid selecting a false positive candidate T_j for T_i, although s_2 matches T_i's subtree rooted at N (by not selecting s_2 we also avoid actually verifying if s_2 matches the subtree rooted at N).

Two-layer index. In summary, the two aforementioned techniques are orthogonal and can be combined to construct a two-layer index for effective subgraph selection. More specifically, we first assign subgraphs into groups utilizing the postorder pruning technique. Then, within each such group, we split subgraphs using the label indexing technique. When selecting subgraphs for a node N (with postorder number p) of T_i, we first use p to find the group G in the first layer, and then use the twig at N to select subgroups within G.

4. EXPERIMENTS

In this section we conduct extensive experiments to demonstrate the efficiency and effectiveness of our proposed partition-based tree similarity join framework. All methods were implemented in C++ and the experiments were run on a quad-core machine running Ubuntu 12.04, with 16 GB of main memory. We evaluate our proposed method on both real and synthetic datasets.

Swissprot: Swissprot[2] contains manually annotated and reviewed protein sequences in XML format. It includes 100K flat and medium-sized trees (average tree size 62.37, number of distinct labels 84, average depth 2.65, maximum depth 4).

Treebank: Treebank[3] is an XML database containing 50K small and deep trees, where each tree tags an English sentence from the Wall Street Journal with parts of speech (average tree size 45.12, number of distinct labels 218, average depth 6.93, and maximum depth 35).

Sentiment: Sentiment[4] is used for sentiment prediction of movie reviews. It consists of 10K tagged sentences where the sentiments are computed based on long phrases (average tree size 37.31, number of distinct labels 5, average depth 10.84, and maximum depth 30).

Synthetic: We use the generator of [28] to generate a set of trees by setting the default fanout, maximum depth, number of labels, and tree size to 3, 5, 20, and 80 respectively. Moreover, we adopt the *decay factor* $\mathcal{D}_z$ as in [27] to change the generated trees. Specifically, for each node of the tree generated by the data generator, we change it with probability $\mathcal{D}_z$; the change is randomly chosen from the three

[2] http://us.expasy.org/sprot/
[3] http://www.cis.upenn.edu/~treebank/
[4] http://nlp.stanford.edu/sentiment/

node edit operations (i.e., insertion, deletion and renaming) with equal probability. We set $\mathcal{D}_z$ to 0.05 as in [27]. The synthetic dataset contains 10K trees by default.

We compare our proposed framework (denoted as PRT) with two state-of-the-art methods in the literature [4, 18], i.e., STR (adopted from [13] and [19]) and SET (adopted from [27]), as described in Section 2.

4.1 Performance Analysis

Figures 10 and 11 compare the runtimes and pruning power of STR, SET, and PRT on all four datasets for various TED thresholds τ. For experimental instances where the runtime is too high (e.g., SET for $\tau = 4$), we truncated the corresponding bars and wrote on them the total runtime (e.g., 1113). The first observation is that the generation of candidates in SET and PRT is quite fast, therefore TED computations dominate the runtime of SET, while the cost is balanced between candidate generation and verification in PRT. On the other hand, the generation of candidates by STR requires expensive string join computations. Recall that the computation of TED using RTED requires cubic time complexity. When τ is small, fewer tree pairs are similar and thus fewer candidates are generated, so the computation of string edit distance joins dominates the overall runtime. As we can see from Figure 11, STR and PRT have better pruning power compared to SET. In the plots of Figure 11, we also included the actual number of join results (series REL) for reference. The number of candidates generated by STR and PRT are very close to those of REL, indicating that these methods generate only a small number of false candidate pairs (except for the case of Sentiment), while SET requires the verification of a large number of candidates that are false hits. The difference between SET and REL increases with τ, since the binary branch structure extracted in SET is insensitive to τ and larger τ derives more candidates; the number of false candidates generated by STR and PRT only increases slightly. Therefore, although SET spends less time on candidate generation, the performance gap between SET and STR narrows with the growth of τ. SET can be worse than STR for larger thresholds, as more candidates are returned by SET for refinement.

Finally, the performance gap between PRT and STR/SET increases when smaller TED thresholds (τ) are used. For example, PRT outperforms the best competitor (i.e., SET) by an order of magnitude when $\tau = 1$. The gap decreases with increasing τ values, since more candidates are generated and the computation of TED dominates the overall runtime. When $\tau = 5$, PRT is 50%–80% faster than the best competitor (either STR or SET); this is still a substantial improvement. To summarize, our proposed framework PRT is very efficient, achieving significant performance gains over state-of-the-art solutions for similarity joins on large tree-structured datasets.

4.2 Scalability Analysis

We evaluate the scalability of all the methods by varying the cardinality of the datasets. We conducted experiments on Swissprot subsets with cardinalities from 20K to 100K, on Treebank subsets (10K to 50K), on Sentiment subsets (2K to 10K), and on Synthetic subsets (2K to 10K). In all experiments, τ is set to 3. As shown in Figure 12, the response time of each method grows as the cardinality of the dataset increases, since more pairs have to be compared and more results are identified (cf. Figure 13). The relative performance of the three methods is insensitive to the data size and the conclusions are similar to our analysis in Section 4.1: (i) PRT generates similar number of candidates as STR and this number is closer to REL compared to SET; (ii) SET spends a large percentage of its time on verifying pairs and STR spends a large percentage of its time on generating the candidates. In summary, our proposed solution PRT constantly outperforms STR and SET in all settings.

4.3 Sensitivity Analysis

Our last experiment is a sensitivity analysis for all the methods w.r.t. different tree parameters, i.e., the maximum fanout f, maximum depth d, number of labels l, and average tree size t. Table 1 summarizes these parameters with their default values in bold.

Table 1: Tree parameters

Parameter	Values
f	2, **3**, 4, 5, 6
d	4, **5**, 6, 7, 8
l	3, 5, 10, **20**, 50
t	40, **80**, 120, 160, 200

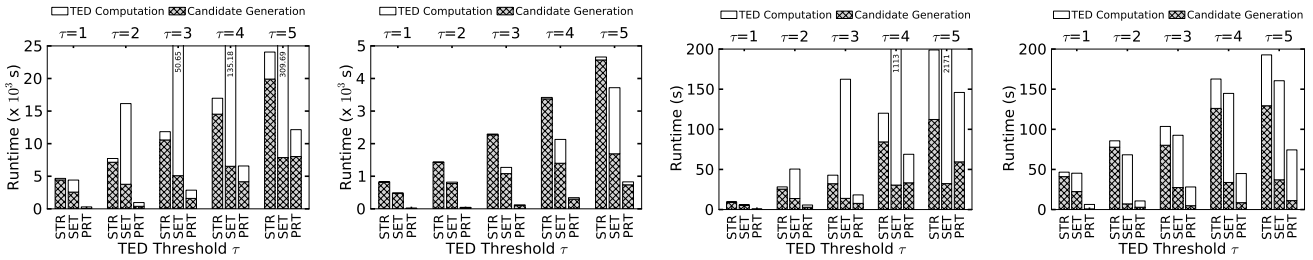
All sensitivity tests were conducted on the synthetic datasets. When investigating the effect of one parameter, we fix the values of other three parameters to defaults to diminish their effects. To this end, we generate 5 different datasets for each parameter. Each generated dataset consists of 10K trees. We set the TED threshold $\tau = 3$. Figure 14 compares all methods for various parameter settings. The results show that our proposed solution PRT outperforms STR/SET in all cases.

Figures 14(a) and 14(b) show that the runtime of STR increases when f changes from 2 to 6. The reason is that given the other parameters fixed, a small value of f increases the expected variance in the heights of the generated trees; thus, more candidates are pruned by the size constraint[5] before passing to the string join computation. When f increases to 6, the difference between the tree heights becomes smaller; hence more tree pairs are passed to STR, resulting in longer candidate generation times. On the other hand, the growth of f introduces more randomness and diversity in the tree shapes, significantly reducing the number of candidate and actual result pairs (cf. Figure 14(b)). As PRT captures the structural information of trees, more diversified tree shapes (because of higher f) lead to higher pruning effectiveness and shorter runtimes for PRT.

Figures 14(c) and 14(d) compare the performance of all methods for various tree depths d. The growth of d decreases the variance of tree sizes, thus more candidates survive the size constraint filter. As a result, the candidates generation steps of both STR and SET become more expensive with the increase of d. On the other hand, unlike the case of fanout f, the binary branch structures are irrelevant to d, hence the number of candidates generated by SET is less sensitive to d (see Figure 14(d)).

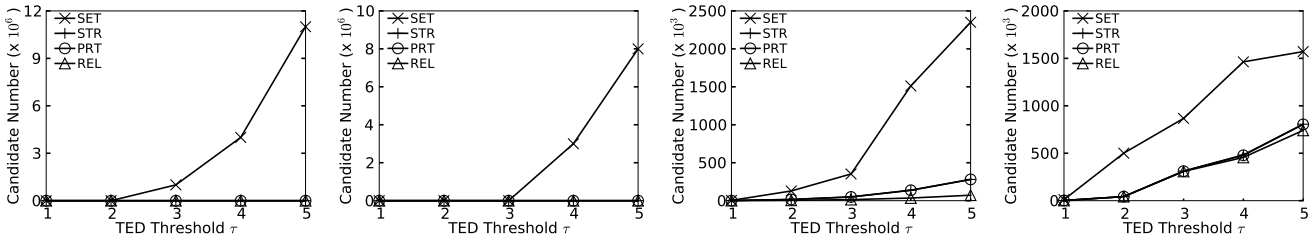
Figures 14(e) and 14(f) plot the performances of different methods when varying the number of labels l. Observe that SET is more sensitive to l than STR and PRT. When l is small, the binary branch structures of the trees are more

[5]Recall that if two trees are similar with TED threshold τ, the difference of their sizes should be no larger than τ.



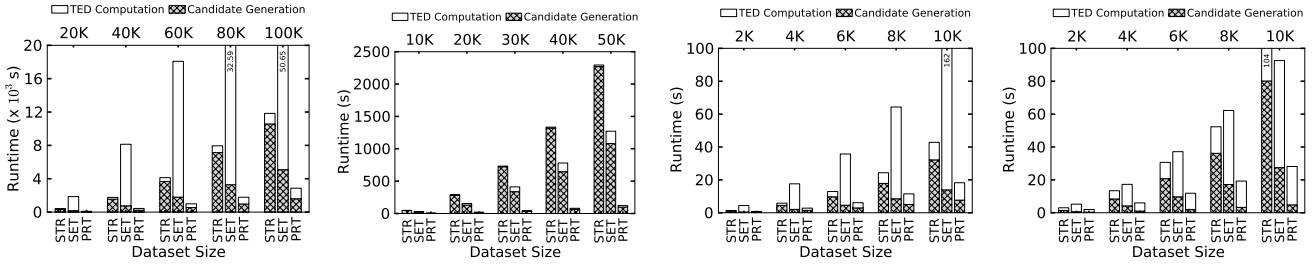
(a) Swissprot (b) Treebank (c) Sentiment (d) Synthetic

Figure 10: Runtime on all the datasets w.r.t. TED threshold τ



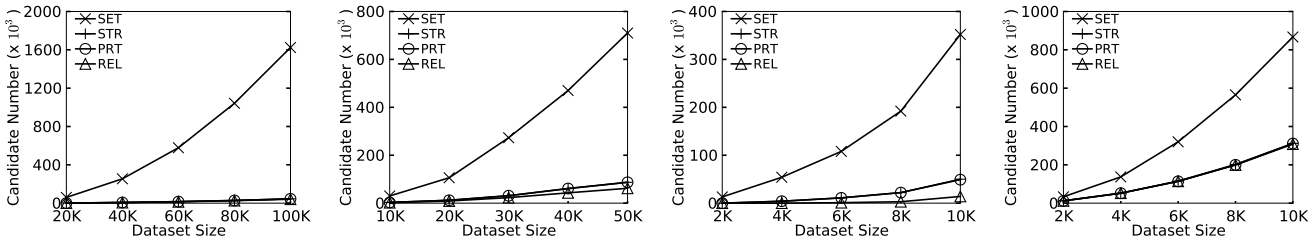
(a) Swissprot (b) Treebank (c) Sentiment (d) Synthetic

Figure 11: Number of candidates generated on all the datasets w.r.t. TED threshold τ



(a) Swissprot (b) Treebank (c) Sentiment (d) Synthetic

Figure 12: Runtime on all the datasets w.r.t. dataset cardinality



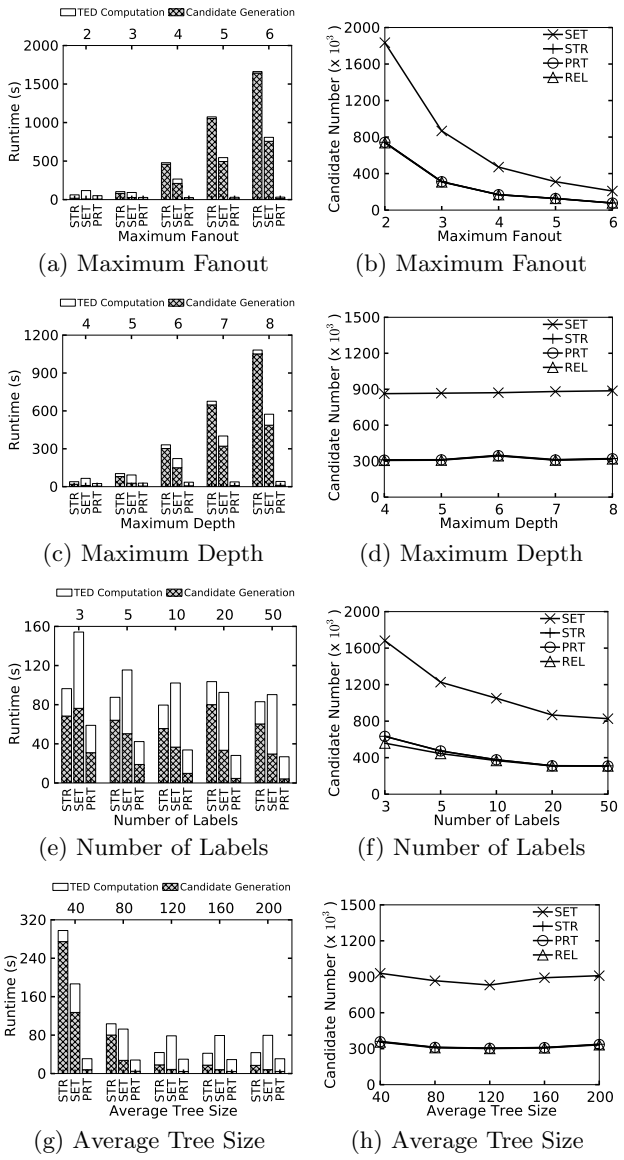
(a) Swissprot (b) Treebank (c) Sentiment (d) Synthetic

Figure 13: Number of candidates generated on all the datasets w.r.t. dataset cardinality

similar and thus SET reports more false candidates (the default tree size is 80). The runtimes of all methods stabilize when $l \geq 20$, since the labels are more sparse in these cases.

Finally, Figures 14(g) and 14(h) compare all methods for various values of the average tree size t. The runtime decreases for larger t values, since the growth of t increases the diversity of tree sizes in the dataset, and most of the tree pairs are pruned by the size constraint before they are fed to the string edit distance join computation. As a result, the time spent on generating candidates decreases. In contrast, note that the number of generated candidates by SET only slightly decreases with the growth of t; therefore, SET's cost decreases with t. PRT is less sensitive to t compared to the other methods and its runtime also decreases with t. Since τ is fixed, larger tree sizes result in larger subgraphs; therefore less subgraphs are common among trees, which increases PRT's pruning power.

To conclude, the performance of our proposed method PRT is stable w.r.t. different tree characteristics, and PRT constantly outperforms state-of-the-art approaches (STR and SET) under various settings of tree parameters. The speed-up that we achieve is important, considering applica-



(a) Maximum Fanout (b) Maximum Fanout

(c) Maximum Depth (d) Maximum Depth

(e) Number of Labels (f) Number of Labels

(g) Average Tree Size (h) Average Tree Size

Figure 14: Sensitivity to various parameters

tions that require a low response time for similarity joins, e.g., streaming workloads where tree objects (e.g., XML and HTML entities) are inserted and updated at a high rate and data collections are refreshed every few hours/minutes. As a final note, we also experimentally tested the effectiveness of our partitioning scheme in PRT (Section 3.3) and found that the general performance improvement it offers compared to performing random tree partitioning is 50%–300%. This experiment is omitted due to space constraints.

5. RELATED WORK

Tree similarity measures. Besides TED, alternative distance measures for trees have been proposed and studied. For example, the *pq*-gram [3, 5], which is defined as a small tree strucutre consisting of an anchor node with $p-1$ ancestors and q children, could be used to measure the closeness of two rooted labeled trees (either ordered or unordered); intuitively, two trees are similar (w.r.t. the *pq*-gram distance) if they share a large number of common *pq*-grams. In addition, Tatikonda and Parthasarathy [25] propose a mapping approach that transforms an unordered tree into a multiset of pivots (i.e., simple wedge-shaped structures), and then couples it with a minwise hashing framework to convert the multiset into a fixed sized signature-sketch, to estimate the similarity between trees. Despite the existence of alternative measures, TED is widely recognized as the state-of-the-art similarity measure for tree-structured objects [18, 20]. Some studies (e.g., [4, 18]) have extensively compared TED with other alternative measures and discussed its pros and cons. For example, TED is of higher quality compared to other distance measures; also, TED matches the human perception of difference between tree structures. On the other hand, TED has the highest computational complexity among existing similarity measures for rooted ordered labeled trees. In this paper, we alleviate this drawback by avoiding TED computations as much as possible in tree similarity joins.

Computation of TED. There have been several studies on reducing the time and space complexity of TED computation. Given two trees both of size n, Tai [23] proposed the first non-exponential TED algorithm, which has $O(n^6)$ time and space complexity. Later this method was improved by Zhang and Shasha [29] to an approach with $O(n^2)$ space complexity and $O(n^4)$ time complexity, which reduces to $O(n^2 \log^2 n)$ for balanced trees. Klein [17] reduced the time complexity for general trees to $O(n^3 \log n)$, at a cost of $O(n^3 \log n)$ space requirements. Based on the ideas of [17], Demaine et al. [11] further reduced the time complexity to $O(n^3)$, while keeping the $O(n^2)$ space complexity of [29]. Recently, Pawlik and Augsten [20] propose a robust hybrid framework RTED, which combines [11] and [29] and dynamically chooses the best of the two methods based on the tree shapes. The time and space complexity of RTED is $O(n^3)$ and $O(n^2)$, respectively. Some research efforts [15, 24] focus on finding a suboptimal solution with better time complexity by constraining the node edit operations; however, in this paper, we do not consider such constraints.

Similarity queries on tree-structured data. Similarity queries on tree-structured data is a well-studied topic in the database community. Guha et al. [13] study the XML data integration problem using tree similarity joins as a core module. The authors adopt TED as the distance metric for XML data objects and propose to use the preorder/postorder node traversal sequences of trees to derive lower bounds for TED, and thus facilitate the join. Akutsu et al. [1] use the strings derived from the Euler tour for a given tree to bound its TED to other trees. Both these two methods require $O(n^2)$ time to compute the string similarities for all tree pairs. On the other hand, Kailing et al. [16] propose three lower bounds for TED, based on some simple statistics (namely the distance to leaves, degrees, and labels of nodes) of the trees, which can be extracted as histograms. In [27], the authors measure the similarity among trees based on a specific pattern called *binary branch*. Each tree is transformed into a bag of binary branches and the TED between any two trees is bounded by the dissimilarity between their *binary branch vectors*. Our approach is different from all these methods, as our framework is based on tree partitioning and subgraph matching.

Another line of related work is on *subtree* similarity search [3, 7, 8]; the objective is to find in a large data tree similar *subtrees* to a given a query tree. Our focus is dif-

ferent, as we aim at finding similarity tree pairs in a large collection of tree objects. As a result, these techniques are not applicable for solving our problem.

Similarity queries for other data types. Similarity search and joins have been extensively studied on various types of complex data in recent years, e.g., strings [19], vectors [12], and graphs [30]. In specific, Li et al. [19] proposes Pass-Join for efficient string similarity joins; the main idea is to partition strings into segments and index them for generating candidate string join pairs. Zhao et al. [30] study similarity search over graph databases. The graphs are divided into variable-size non-overlapping partitions for subsequent subgraph containment tests which are used to prune unpromising candidate pairs. All the above techniques cannot be used to solve our problem, because tree-structured data have different structure and similarity measures compared to the other data types. In particular, although a tree is also a graph, the similarity measures, i.e., *graph edit distance* (GED) used in [30] and *tree edit distance* (TED), bear considerable differences in terms of definitions and semantics. For example, there are 6 types of edit operations for GED, but only 3 for TED. In addition, the insert/delete operations in GED and TED are totally different; e.g., in GED, the insertion of a node only adds it to the graph and any other nodes/edges are not affected (while in TED some existing nodes become children of the new one).

6. CONCLUSION

We proposed a novel similarity join technique for collections of tree-structured objects. Our approach is based on the decomposition of the tree objects into subgraphs, which are indexed. The decomposition is based on the join similarity threshold and it is performed dynamically during the join evaluation. Our technique prunes pairs of objects if there does not exist a subgraph of one object inside the other. We proposed an effective partitioning approach that generates as large subgraphs as possible in order to maximize the pruning effectiveness of the join framework. In addition, we proposed effective techniques for selecting which subgraphs of the current object to search in other objects, in order to obtain the join candidates. Our experiments on real and synthetic data confirmed the superiority of our join algorithm compared to the previous state-of-the-art. In the future, we plan to extend our solution to support other tree distance metrics and study also the adaption of our techniques to parallel and distributed settings (e.g., multi-core architectures, MapReduce).

7. REFERENCES

[1] T. Akutsu, D. Fukagawa, and A. Takasu. Approximating tree edit distance through string edit distance. *Algorithmica*, 57(2):325–348, 2010.

[2] J. Albert. Algebraic properties of bag data types. In *VLDB*, pages 211–219, 1991.

[3] N. Augsten, D. Barbosa, M. H. Böhlen, and T. Palpanas. TASM: Top-k approximate subtree matching. In *ICDE*, pages 353–364, 2010.

[4] N. Augsten and M. H. Böhlen. *Similarity Joins in Relational Database Systems*. Morgan & Claypool Publishers, 2013.

[5] N. Augsten, M. H. Böhlen, C. E. Dyreson, and J. Gamper. Windowed pq-grams for approximate joins of data-centric XML. *VLDB J.*, 21(4):463–488, 2012.

[6] G. Blin, A. Denise, S. Dulucq, C. Herrbach, and H. Touzet. Alignments of RNA structures. *IEEE/ACM Trans. Comput. Biology Bioinform.*, 7(2):309–322, 2010.

[7] S. Cohen. Indexing for subtree similarity-search using edit distance. In *SIGMOD*, pages 49–60, 2013.

[8] S. Cohen and N. Or. A general algorithm for subtree similarity-search. In *ICDE*, pages 928–939, 2014.

[9] W. W. Cohen. Data integration using similarity joins and a word-based information representation language. pages 288–321, 2000.

[10] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms (3. ed.)*. MIT Press, 2009.

[11] 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.

[12] S. Fries, B. Boden, G. Stepien, and T. Seidl. Phidj: Parallel similarity self-join for high-dimensional vector data with mapreduce. In *ICDE*, pages 796–807, 2014.

[13] S. Guha, H. V. Jagadish, N. Koudas, D. Srivastava, and T. Yu. Integrating XML data sources using approximate joins. *ACM Trans. Database Syst.*, 31(1):161–207, 2006.

[14] M. R. Henzinger. Finding near-duplicate web pages: a large-scale evaluation of algorithms. In *SIGIR*, pages 284–291, 2006.

[15] T. Jiang, L. Wang, and K. Zhang. Alignment of trees - an alternative to tree edit. *Theor. Comput. Sci.*, 143(1):137–148, 1995.

[16] K. Kailing, H.-P. Kriegel, S. Schönauer, and T. Seidl. Efficient similarity search for hierarchical data in large databases. In *EDBT*, pages 676–693, 2004.

[17] P. N. Klein. Computing the edit-distance between unrooted ordered trees. In *ESA*, pages 91–102, 1998.

[18] F. Li, H. Wang, J. Li, and H. Gao. A survey on tree edit distance lower bound estimation techniques for similarity join on XML data. *SIGMOD Record*, 42(4):29–39, 2014.

[19] G. Li, D. Deng, J. Wang, and J. Feng. Pass-join: A partition-based method for similarity joins. *PVLDB*, 5(3):253–264, 2011.

[20] M. Pawlik and N. Augsten. RTED: A robust algorithm for the tree edit distance. *PVLDB*, 5(4):334–345, 2011.

[21] A. Rheinlnder, M. Knobloch, N. Hochmuth, and U. Leser. Prefix tree indexing for similarity search and similarity joins on genomic data. In *SSDBM*, pages 519–536, 2010.

[22] R. Socher, A. Perelygin, J. Y. Wu, J. Chuang, C. D. Manning, A. Y. Ng, and C. P. Potts. Recursive deep models for semantic compositionality over a sentiment treebank. In *EMNLP*, 2013.

[23] K. Tai. The tree-to-tree correction problem. *J. ACM*, 26(3):422–433, 1979.

[24] E. Tanaka and K. Tanaka. The tree-to-tree editing problem. *IJPRAI*, 2(2):221–240, 1988.

[25] S. Tatikonda and S. Parthasarathy. Hashing tree-structured data: Methods and applications. In *ICDE*, pages 429–440, 2010.

[26] A. Torsello, A. Robles-Kelly, and E. R. Hancock. Discovering shape classes using tree edit-distance and pairwise clustering. *International Journal of Computer Vision*, 72(3):259–285, 2007.

[27] R. Yang, P. Kalnis, and A. K. H. Tung. Similarity evaluation on tree-structured data. In *SIGMOD*, pages 754–765, 2005.

[28] M. J. Zaki. Efficiently mining frequent trees in a forest: Algorithms and applications. *IEEE Trans. Knowl. Data Eng.*, 17(8):1021–1035, 2005.

[29] 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.

[30] X. Zhao, C. Xiao, X. Lin, Q. Liu, and W. Zhang. A partition-based approach to structure similarity search. *PVLDB*, 7(3):169–180, 2013.