A General Algorithm for Subtree Similarity-Search

Sara Cohen [1], Nerya Or [2]

The Rachel and Selim Benin School of Computer Science and Engineering
The Hebrew University of Jerusalem
Jerusalem, Israel
[1] sara@cs.huji.ac.il
[2] nerya.or@mail.huji.ac.il

***Abstract*—Determining similarity between trees is an important problem in a variety of areas. The *subtree similarity-search problem* is that of finding, given a tree Q and a large set of trees $\Gamma = \{T_1, \ldots, T_n\}$, the subtrees of trees among Γ that are most similar to Q. Similarity is defined using some tree distance function. While subtree similarity-search has been studied in the past, solutions mostly focused on specific tree distance functions, and were usually applicable only to ordered trees. This paper presents an efficient new algorithm that solves the subtree similarity-search problem, and is compatible with a wide family of tree distance functions (for both ordered and unordered trees). Extensive experimentation confirms the efficiency and scalability of the algorithm, which displays consistently good runtime even for large queries and datasets.**

I. Introduction

Tree structured data is used extensively in many fields. Comparing two given trees to determine their similarity is highly important in a variety of contexts, including applications in natural language processing [1], document similarity [2], medical image processing [3], comparison of RNA secondary structures [4], quantifying neuronal morphology [5], discovering and comparing shape classes [6], [7], character recognition [8], similarity joining and querying of XML documents [9]–[11], and information extraction [12].

In past work, many different functions have been devised which take two trees and return a measure of their distance. The cost of calculating these functions varies, ranging from simple linear time algorithms, to algorithms which must solve NP-hard problems. One way of measuring tree similarity is *tree edit distance* [13]. The tree edit distance between two trees is the minimal cost of applying edit operations (node insertion / deletion / renaming) in order to transform one tree to the other, where each edit operation is associated with a certain cost. As such, tree edit distance is NP-complete for unordered trees, and computable in polynomial time for ordered trees. Other tree distance functions were developed which are quicker to evaluate (often yielding near-linear runtime), such as *pq*-gram distance [14] or binary branch distance [15], among many others. Such functions usually summarize certain tree features and then compare them to derive a distance measure. Methods that run in linear time for comparing trees were proposed in [16].

An important problem that arises when dealing with tree structured data, is to locate fragments of trees in a database which are similar to a given sample tree (according to some distance function). Usually, one of the following variations is considered:

- *Tree-to-tree similarity-search:* Given a tree Q and a database of trees $\Gamma = \{T_1, T_2, \ldots, T_n\}$, find the trees T_i most similar to Q.
- *Subtree similarity-search*[1]*:* Given a tree Q and a database of trees $\Gamma = \{T_1, T_2, \ldots, T_n\}$, find the *subtrees* of trees T_i, which are most similar to Q.

We note that both variations are also useful in solving the *approximate join problem*, in which, given two sets of trees Γ_1, Γ_2, we wish to output a join of their members based on some upper bound on the distance between joined trees.

Subtree similarity-search is generally harder to solve than tree-to-tree similarity-search (as it must consider many more trees) and is the focus of this paper. Specifically, we will be focusing on finding the top-k most similar subtrees in the database, to a given query tree.

There has been substantial research on tree-to-tree similarity-search, particularly in the context of approximate joins. Past work includes [15], [17], [18], which use the tree edit distance measure, but leverage upper and lower bounds to perform filtering, in order to reduce the number of edit distance computations. A different tree distance function for ordered trees, called *pq-grams*, was proposed in [19], as a possible replacement to tree edit distance, with a significantly lower runtime. This measure was extended to unordered trees in [20].

There has also been some previous work on subtree similarity-search, specifically for tree edit distance over ordered trees. In particular, [9] solves subtree similarity-search using tree edit distance by presenting a clever algorithm that traverses the tree in post-order, while attempting to minimize the number of edit distance computations needed (with memory use depending only on the size of input from the user). Using a different approach, [21] solves the subtree similarity-search problem for unit-cost tree edit distance, by leveraging structural regularities in the database trees and heavily relying on index structures. We note that both of these methods require non-linear computations, due to the nature of the tree edit distance function.

It is noteworthy that all of the aforementioned solutions (other than [16]) are tailored for a specific tree distance function.[2] On the other hand, in this paper we present an algorithm for solving the subtree similarity-search problem,

[1] This problem is sometimes called *approximate subtree matching* [9], [10].

[2] We note that [18] gives a general algorithm but focuses on tree edit distance in the paper.

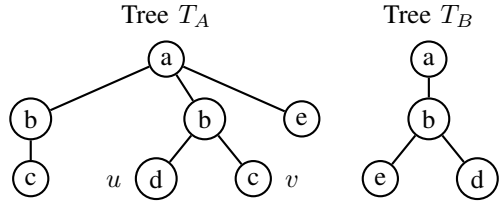


Fig. 1: Two trees T_A and T_B.

which is compatible with a wide family of tree distance functions, which we call *composite profile distance functions*. As composite profile distance functions are generally of lower complexity than tree edit distance, our algorithm is highly efficient in practice. In addition, in our solution, the trees may be ordered or unordered, depending only on limitations imposed by the distance function used. Our algorithm traverses over the database in near-linear time, with small memory usage dominated mostly by the size of the user query (up to a factor determined by the distance function used) and by the maximal height of a tree in the database.

Our algorithm is applicable to any distance function that can be formulated as a composite profile distance function. In particular, as this is the case for pq-grams [14], windowed pq-grams [20] and binary branch distance [15], our algorithm is also the first solution for subtree similarity-search for these measures. To summarize, the main contributions of this paper are:

- We introduce a coherent framework for the subtree similarity-search problem, capable of expressing a large general family of tree distance functions, called composite profile distance functions.
- We show how several tree distance functions considered in the past can be cast within our framework.
- We present an efficient algorithm for solving the subtree similarity-search problem for composite profile distance functions, that runs in time that is close to linear.
- We experimentally prove the efficiency and scalability of our algorithm over both real and synthetic datasets.

II. Methods for Measuring Tree Distance

A variety of methods for computing the distance between trees have been developed. One well-known measure is tree edit distance, which is computable in polynomial time for ordered trees, and is NP-complete for unordered trees [13]. Unfortunately, even for ordered trees, computing tree edit distance is not always feasible. If the trees are very large, the runtime becomes prohibitively high.

For this reason, many other functions that measure distance between trees have been proposed, which have better runtime — often linear in the size of the input. Next, we will review a few such functions informally. As will become apparent later, our algorithm solves the subtree similarity-search problem for a large family of functions, including those discussed here.

A. pq-gram Distance

First presented in [14], pq-gram distance is a tree distance function for ordered trees, in which the main idea is to break down trees into constant-sized fragments called pq-grams, which represent both tree structure and content.

Given a tree T and two integers $p, q > 0$, the tree is extended with new nodes having special new labels (denoted $*$) according to the parameters p, q, by *(1)* adding $p-1$ ancestors to the root, *(2)* adding $q-1$ children before each first child and after each last child and *(3)* adding q children to each leaf node. Then, the extended tree is decomposed into many small connected subgraphs having $p+q$ nodes $(v_1, \ldots, v_p; v'_1, \ldots, v'_q)$ where $v_1, \ldots, v_p$ is a path of parent-child nodes and $v'_1, \ldots, v'_q$ are consecutive children of v_p. These subgraphs are called *pq-grams*. Finally, the multiset of string representations of these pq-grams is used to represent the content and structure of the original tree. We denote this multiset by $\varphi^{\mathrm{t}}_{p,q}(T)$. Next, given two trees, [14] measures their similarity by comparing their pq-gram multisets using the formula in the following example.

Example 2.1: Consider the trees T_A, T_B shown in Figure 1. Their corresponding pq-gram multisets (for $p = 2, q = 1$) are:

$$\begin{aligned}\varphi^{\mathrm{t}}_{2,1}(T_A) &= \{\!\{(*,a;b),(*,a;b),(*,a;e),(a,b;c),(a,b;d),\\ &\quad (a,b;c),(a,e;*),(b,c;*),(b,d;*),(b,c;*)\}\!\}\\ \varphi^{\mathrm{t}}_{2,1}(T_B) &= \{\!\{(*,a;b),(a,b;e),(a,b;d),(b,e;*),(b,d;*)\}\!\}\end{aligned}$$

Using the following normalized pq-gram distance function [14] to calculate the distance between T_A, T_B,

$$\frac{|\varphi^{\mathrm{t}}_{2,1}(T_A) \uplus \varphi^{\mathrm{t}}_{2,1}(T_B)| - 2|\varphi^{\mathrm{t}}_{2,1}(T_A) \uplus \varphi^{\mathrm{t}}_{2,1}(T_B)|}{|\varphi^{\mathrm{t}}_{2,1}(T_A) \uplus \varphi^{\mathrm{t}}_{2,1}(T_B)| - |\varphi^{\mathrm{t}}_{2,1}(T_A) \uplus \varphi^{\mathrm{t}}_{2,1}(T_B)|}$$

we obtain a distance value of $\frac{15-2\cdot 3}{15-3} = 0.75$. Note that this distance formula is normalized to the range $[0, 1]$. A non-normalized version has also been defined in [14]. □

As shown in [14], the pq-gram distance can be used as a lower bound on fanout weighted tree edit distance, which is a variant of tree edit distance that is defined in the same paper.

Several variations to the pq-gram distance have been proposed in [22]. These attempt to give special attention to text nodes in the context of XML document trees, by defining the tree's multiset elements for text data differently.

B. Windowed pq-gram Distance

Recall that pq-gram distance is well-defined only for ordered trees. This function has been extended to deal with unordered trees, with the introduction of *windowed pq-grams* in [20]. In this variation, sibling nodes in the tree are first sorted, and then a procedure which is reminiscent of pq-gram generation is performed. For instance, in the tree T_A from Figure 1, the sibling nodes u, v are first sorted lexicographically by their labels so that v will appear before u, and then a procedure which extends the tree with new nodes (labeled $*$) takes place, followed by extraction of tuples of nodes from this extended tree. The result is a multiset of fragments which provide an order-agnostic representation of the tree. Once again, [20] measures tree similarity by comparing the multisets associated with two given trees.

C. Binary Branch Distance

A different tree distance function for ordered trees, called *binary branch distance*, was introduced by [15]. Intuitively, to compare trees T and S using binary branch distance, one first transforms T and S into binary trees, and then compares subcomponents of these binary trees one with another.

To be more precise, each node v in T will correspond to a node v' in the binary version T' of T. The left child of v' is the node u' corresponding to the first child u of v in T. The right child of v' is the node w' corresponding to the next right sibling w of v in T. (If v has no children or no right sibling, a $*$-labeled node is added to the binary tree instead.)

Next, [15] associates a *binary branch* with each node v in T, which is simply a tuple $(l(v); l(v_l), l(v_r))$ where $l(v)$ is the label of a node v, and v_l, v_r are the left and right children of v, respectively, in the binary version T' of T. The multiset of all binary branches of all of nodes in T is denoted $\varphi^{\mathrm{t}}_{BiB}(T)$. Similarly to pq-distance, [15] calculates the similarity between two given trees by comparing their associated multisets.

In [15] it is shown that the binary branch distance can be used as a lower bound on tree edit distance.

Example 2.2: Consider the trees T_A, T_B shown in Figure 1 and their binary branch multisets:

$$\begin{aligned}\varphi^{\mathrm{t}}_{BiB}(T_A) &= \{\!\{(c;*,*),(b;c,b),(d;*,c),(c;*,*),\\ &\quad (b;d,e),(e;*,*),(a;b,*)\}\!\}\\ \varphi^{\mathrm{t}}_{BiB}(T_B) &= \{\!\{(e;*,d),(d;*,*),(b;e,*),(a;b,*)\}\!\}\end{aligned}$$

To calculate the distance between T_A and T_B, [15] uses a procedure equivalent to obtaining the symmetric difference of the multisets,

$$|\varphi^{\mathrm{t}}_{BiB}(T_A) \uplus \varphi^{\mathrm{t}}_{BiB}(T_B)| - 2|\varphi^{\mathrm{t}}_{BiB}(T_A) \pitchfork \varphi^{\mathrm{t}}_{BiB}(T_B)|$$

thereby deriving a value of $11 - 2 \cdot 1 = 9$. Note that unlike in Example 2.1, the formula used to calculate distance between the two multisets is not normalized. □

D. Commonalities Among Similarity Functions

In this section we briefly surveyed 4 functions. The first, tree edit distance, makes a rather complicated analysis of two given trees in order to measure their similarity. The rest of the functions significantly reduce the complexity of measuring similarity, by using a two-phase method:

- *Phase 1:* Summarize the interesting features of each tree using a multiset.
- *Phase 2:* Measure the distance between multisets of different trees using some multiset distance function.

The aim of this paper is to find a general algorithm for subtree similarity-search that is suitable for functions that work in two such phases. We define the precise framework in the next section.

III. Formal Framework

We present the necessary formal definitions. We start by introducing some notation for trees and multisets. Then, we present two notions needed to compare trees: (1) *tree profile functions*, used to summarize a tree as a multiset, and (2) *multiset-distance functions* used to compare multisets. Using these two types of functions, we will be able to formally define the problem of interest.

A. Trees and Multisets

We use Σ to denote an infinite alphabet of labels. If $T = (V, E)$ is a directed labeled tree, we use $r(T)$ to denote the root of T, and $l(v) \in \Sigma$ to denote the label of a node $v \in V$. We use $height(T)$ to denote the height of T, i.e., the number of nodes on the longest path from the root of T to a leaf node. We will use $\mathcal{V}(T)$ to denote the nodes of T. We denote the number of nodes $|\mathcal{V}(T)|$ in the tree by $|T|$. Given a node v, we denote by T_v the *subtree* whose root is v. Finally, we denote by $SubTrees(T)$ the set of all subtrees of T, i.e., $SubTrees(T) = \{T_v \mid v \in \mathcal{V}(T)\}$. Note that $T \in SubTrees(T)$.

In this paper we often use multisets. We will usually use the letter x to denote elements in a multiset. We denote the number of times an element x appears in a multiset M by $|M|_x$. We denote the *multiset union* and *multiset intersection* between two multisets M_1, M_2 by $M_1 \uplus M_2$ and $M_1 \pitchfork M_2$, respectively. For each element x that appears at least once in either M_1 or M_2, it holds that x appears $|M_1|_x + |M_2|_x$ times in the multiset $M_1 \uplus M_2$, and x appears $\min\{|M_1|_x, |M_2|_x\}$ times in the multiset $M_1 \pitchfork M_2$.

B. Tree Profile Functions

Similarity between trees can be defined in a variety of ways, as demonstrated in Section II. Many similarity measures have been defined that compare trees by utilizing some type of structural summary of the tree. We call such a structural summary a *profile*, and it is defined formally next.

Definition 3.1: A tree profile function φ^{t} *is a function that takes a tree T as its input, and returns a multiset of objects. This multiset $\varphi^{\mathrm{t}}(T)$ is called the* profile *of T.*

Example 3.2: Given a tree T, consider the tree profile function $\varphi^{\mathrm{t}}_{lab}(T)$ which returns the multiset of all labels of nodes in T, i.e.,

$$\varphi^{\mathrm{t}}_{lab}(T) = \{\!\{l(v) \mid v \in \mathcal{V}(T)\}\!\} .$$

Consider the tree T_A in Figure 1. It holds that $\varphi^{\mathrm{t}}_{lab}(T_A) = \{\!\{c, b, d, c, b, e, a\}\!\}$. Note that comparing the multisets $\varphi^{\mathrm{t}}_{lab}(S)$ and $\varphi^{\mathrm{t}}_{lab}(T)$ for two trees S and T disregards all structural information in the trees, as such information is ignored by $\varphi^{\mathrm{t}}_{lab}$.

Other tree profile functions, which do give importance to structural information in the tree, include $\varphi^{\mathrm{t}}_{p,q}$ (the pq-gram profile of a tree [14]), $\varphi^{\mathrm{t}}_{BiB}$ (the binary branch profile of a tree [15]), and the windowed pq-grams profile of a tree [20], among others. These functions were introduced in Section II. □

In principle, tree profile functions can be defined arbitrarily. We note that many previously studied tree profile functions

display a certain type of locality, which can be very helpful when devising algorithms that compare the values of tree profile functions for multiple trees. The simplest type of locality is when the tree profile function can be computed by locally considering each node in the tree, as follows.

Definition 3.3: Let T be a tree. A function φ^{v} is a vertex profile function *if it takes as input a node $v \in \mathcal{V}(T)$ and returns a multiset of objects.*

Let V be a set of nodes and φ^{v} be a vertex profile function. For the sake of readability, we will use the following notation throughout the paper: $\varphi^{\mathrm{v}}(V) = \biguplus_{v \in V} \varphi^{\mathrm{v}}(v)$.

Definition 3.4: A tree profile function φ^{t} is simple *if there exists a vertex profile function φ^{v} such that $\varphi^{\mathrm{t}}(T) = \varphi^{\mathrm{v}}(\mathcal{V}(T))$.*

It is quite easy to see that $\varphi^{\mathrm{t}}_{lab}$ is simple. However, there are tree profile functions of interest that are not simple. As we will also be interested in such functions, we define a wider class of tree profile functions, called *composite tree profile functions*.

Definition 3.5: Let T be a tree. A tree profile function φ^{t} is composite *if there exist two vertex profile functions $\varphi^{\mathrm{v}}_{local}, \varphi^{\mathrm{v}}_{global}$ such that:*

$$\varphi^{\mathrm{t}}(T) = \varphi^{\mathrm{v}}_{local}(r(T)) \uplus \varphi^{\mathrm{v}}_{global}(\mathcal{V}(T) \setminus \{r(T)\}) \,. \quad (1)$$

Thus, a composite tree profile function uses one function to summarize the root $r(T)$ of the tree and a different function to summarize the remainder of the nodes. Observe that every simple tree profile function is also a composite tree profile function, but the opposite is not always the case.

Example 3.6: The function which maps a tree to its pq-gram profile [14] is composite. Let T be a tree, and let $v \in \mathcal{V}(T)$. We define $\varphi^{\mathrm{v}}_{global}(v)$ as all pq-grams of T that have v as their root, and we define $\varphi^{\mathrm{v}}_{local}(v)$ as the multiset union of $\varphi^{\mathrm{v}}_{global}(v)$ with the multiset of all pq-grams of the subtree T_v that contain a prefix of $*$-labeled nodes. See Figure 4 for an example.

In a similar fashion, the function mapping a tree to its profile of windowed pq-grams [20] can be shown as composite.

The function which maps a tree to its profile of binary branches [15] is also composite. For each node v we define $\varphi^{\mathrm{v}}_{global}(v) = \{\!\{(l(v); l(v_l), l(v_r))\}\!\}$ where v_l is the first (left-most) child of v (or a $*$-labeled node if v is a leaf), and v_r is leftmost right sibling of v (or a $*$-labeled node if v has no right siblings). We also define $\varphi^{\mathrm{v}}_{local}(v) = \{\!\{(l(v); l(v_l), *)\}\!\}$, since in the subtree rooted in v, v does not have a right sibling. □

Remark 3.7: We note that while the difference between simple and composite profile functions may not be clear at first, the definition of composite profile functions is helpful in the context of subtree similarity-search. For a given tree T, each node $v \in \mathcal{V}(T)$ is the root of the subtree T_v, and since a composite profile function associates a different $\varphi^{\mathrm{v}}_{local}(v)$ for each node in the tree, this allows us to define a unique profile multiset for every subtree $T_v \in SubTrees(T)$, rather than just using a multiset-union over all multisets of descendant nodes, as in simple profile functions.

C. Profile Distance Functions

Given trees S and T, and a specific tree profile function φ^{t}, we can derive the profiles $\varphi^{\mathrm{t}}(S)$, $\varphi^{\mathrm{t}}(T)$, of these trees. Recall that profiles are simply multisets. Next, we define multiset distance functions, that are used to compute the distance between multisets, and thus, can be used to compare the profiles $\varphi^{\mathrm{t}}(S)$ and $\varphi^{\mathrm{t}}(T)$ of trees S and T.

Definition 3.8: Given multisets M_1 and M_2, a multiset-distance function *$d(M_1, M_2)$ returns a number signifying the distance (i.e., the dissimilarity) of M_1 and M_2. In addition, we require that $d(M_1, M_2)$ can be computed from the sizes $|M_1|$, $|M_2|$ and the multiset intersection size $|M_1 \Cap M_2|$. In other words, there exists a function d_* which takes 3 integer arguments, such that $d(M_1, M_2) = d_*(|M_1|, |M_2|, |M_1 \Cap M_2|)$.*

Requiring our multiset-distance functions to be computable using $|M_1|, |M_2|, |M_1 \Cap M_2|$ may seem rather limiting at first. However, as the following example demonstrates, well-known measures for comparing multisets can be defined in this manner.

Example 3.9: Different functions for comparing multisets have been defined in the past, and used in a variety of areas. One well known example is Dice difference [23] , defined as

$$Dice(M_1, M_2) = 1 - \frac{2|M_1 \Cap M_2|}{|M_1 \uplus M_2|} \,.$$

Another well known example is the Tversky index [24], defined as

$$Tversky(M_1, M_2) = 1 - \frac{|M_1 \Cap M_2|}{|M_1 \Cap M_2| + \alpha|M_1 \setminus M_2| + \beta|M_2 \setminus M_1|}$$

where $\alpha, \beta \geq 0$ are parameters, usually defined such that $\alpha + \beta = 1$. The parameters α, β let us assign weights that determine, in a sense, how important it is that $M_1 \Cap M_2$ contains elements from each of the two multisets, in an asymmetric manner.

We note that it holds that $|M_1 \uplus M_2| = |M_1| + |M_2|$, and $|M_1 \setminus M_2| = |M_1| - |M_1 \Cap M_2|$. Hence, both $Dice$ and $Tversky$ are multiset-distance functions. Many other multiset distance functions exist. □

Combining the notions of tree profile functions and multiset distance functions, we define a profile distance function.

Definition 3.10: A profile distance function *$\mathcal{D}$ is a tuple $(\varphi^{\mathrm{t}}, d)$ where φ^{t} is a tree profile function, and d is a multiset-distance function. Given two trees S, T, we define $\mathcal{D}(S, T) = d(\varphi^{\mathrm{t}}(S), \varphi^{\mathrm{t}}(T))$.*

D. Problem of Interest

Given a profile distance function $\mathcal{D}$, a quantity bound $k \in \mathbb{N}$, a corpus of (large) trees $\Gamma = T_1, \ldots, T_n$ and a (small) tree of interest Q, we are interested in finding the k subtrees of trees in Γ most similar to Q. In the following, we use $SubTrees(\Gamma)$ to denote $\cup_{T \in \Gamma} SubTrees(T)$.

Problem 1: The *top-k subtree similarity-search problem* is to return a set of trees $Top_k(Q, \Gamma)$ which satisfies the following two conditions:

- Size is k (if possible):

$$|Top_k(Q,\Gamma)| = \min\{k, |SubTrees(\Gamma)|\}$$

- Contains the best subtrees: for every pair of trees $S_1, S_2 \in SubTrees(\Gamma)$, if
  - $S_1 \in Top_k(Q,\Gamma)$ and
  - $S_2 \notin Top_k(Q,\Gamma)$,

  then $\mathcal{D}(Q,S_1) \leq \mathcal{D}(Q,S_2)$.

Specifically, we are interested in solving the subtree similarity-search problem for cases in which the profile distance function uses a composite tree profile function.

In the remainder of this paper, we will refer to Q as the *query* and Γ as the *database*. To simplify the presentation, we will assume that Γ contains a single tree. Lifting this assumption is straightforward.

The subtree similarity-search problem for a composite profile distance function $\mathcal{D} = ((\varphi^{\mathrm{v}}_{local}, \varphi^{\mathrm{v}}_{global}), d)$ can trivially be solved in polynomial time, if $\varphi^{\mathrm{v}}_{local}$, $\varphi^{\mathrm{v}}_{global}$ and d are computable in polynomial time. To see how, recall that Γ defines a linear number of subtrees—one for each node of each tree T in Γ. For each such subtree S, we can compute the values $\varphi^{\mathrm{v}}_{global}$ for all its nodes, and the value $\varphi^{\mathrm{v}}_{local}$ for its root. Given a similar computation of the profile for Q, we can then compute the multiset-distance function d for the profiles of each subtree. Finally, we choose the k subtrees with the smallest distance value.

While the above procedure is polynomial in time, it is highly inefficient in practice as it repeatedly considers trees and their contained subtrees. In the following sections we present an algorithm that is considerably faster, and hence, is feasible in practice.

IV. A Dynamic Programming Algorithm

We start by considering a dynamic programming solution to the subtree similarity-search problem for composite tree profile functions. The algorithm *DynamicSearch* in Figure 2 receives as input a query Q, a (large) tree T, and a quantity bound k. Additionally, the algorithm is given a profile distance function $\mathcal{D} = (\varphi^{\mathrm{t}}, d)$. We require φ^{t} to be a composite tree profile function, and thus we denote it as $(\varphi^{\mathrm{v}}_{local}, \varphi^{\mathrm{v}}_{global})$.

As we show in Section V, the runtime of this algorithm can be improved significantly. Nevertheless, we start off by explaining this simplified algorithm in order to help clarify the operation of the more sophisticated one later, as the two algorithms operate similarly.

A. Data Structures and Conventions

The algorithm *DynamicSearch* uses both hash-maps and a bounded priority queue as basic data structures. We review the use of these items, as well as the notation employed, next.

Hash-maps: For a hash-map object m, we denote the value which is associated by the map to the key x, by $m[x]$. The set of all valid keys in a map m is denoted by $m.keys$. The algorithm *DynamicSearch* uses two types of hash-maps:

Algorithm *DynamicSearch*$(Q, T, k, \mathcal{D})$
1. $QueryQueues \leftarrow$ getQueryQueues (Q)
2. $Sizes \leftarrow$ new HashMap
3. $Results \leftarrow$ new BoundedPriorityQueue(k)
4. **for** each $v \in \mathcal{V}(T)$ in post-order:
5. $\quad global \leftarrow$ *IntersectGlobal*$(QueryQueues, v, \varphi^{\mathrm{t}})$
6. $\quad local \leftarrow$ *IntersectLocal*$(QueryQueues, v, \varphi^{\mathrm{t}})$
7. $\quad intersect \leftarrow global + local$
8. $\quad vProfileSize \leftarrow Sizes[v] + |\varphi^{\mathrm{v}}_{local}(v)|$
9. $\quad \delta \leftarrow d(|\varphi^{\mathrm{t}}(Q)|, vProfileSize, intersect)$
10. $\quad Results$.boundedAdd$(\langle \delta, v \rangle)$
11. $\quad$ **for** each $x \in \varphi^{\mathrm{v}}_{global}(v)$ s.t. $x \in \varphi^{\mathrm{t}}(Q)$:
12. $\quad\quad u \leftarrow QueryQueues[x]$.dequeue()
13. $\quad\quad QueryQueues[x]$.enqueue$(v)$
14. $\quad$ add $(Sizes[v] + |\varphi^{\mathrm{v}}_{global}(v)|)$ to $Sizes[v.parent]$
15. $\quad$ free the memory allocated for $Sizes[v]$
16. **return** $Results$

Algorithm *IntersectGlobal*$(QueryQueues, v, \varphi^{\mathrm{t}})$
1. $count \leftarrow 0$
2. **for** each distinct $x \in \varphi^{\mathrm{t}}(Q)$:
3. $\quad i \leftarrow |\varphi^{\mathrm{t}}(Q)|_x$
4. $\quad$ **while** $(i > 0$ and
$\quad\quad QueryQueues[x]$.elemAt$(i) \neq \perp$ and
$\quad\quad$ lca$(QueryQueues[x]$.elemAt$(i), v) = v)$:
5. $\quad\quad$ decrement i
6. $\quad\quad$ increment $count$
7. **return** $count$

Algorithm *IntersectLocal*$(QueryQueues, v, \varphi^{\mathrm{t}})$
1. $count \leftarrow 0$
2. **for** each distinct $x \in \varphi^{\mathrm{v}}_{local}(v)$ s.t. $x \in \varphi^{\mathrm{t}}(Q)$:
3. $\quad i \leftarrow 1$
4. $\quad$ **while** $i \leq \min\{|\varphi^{\mathrm{t}}(Q)|_x, |\varphi^{\mathrm{v}}_{local}(v)|_x\}$ and
$\quad\quad (QueryQueues[x]$.elemAt$(i) = \perp$ or
$\quad\quad$ lca$(QueryQueues[x]$.elemAt$(i), v) \neq v)$:
5. $\quad\quad$ increment i
6. $\quad\quad$ increment $count$
7. **return** $count$

Fig. 2: A dynamic algorithm for subtree similarity-search.

- *Profile Values to Queues:* The algorithm uses a hash-map $QueryQueues$. This map takes each unique value $x \in \varphi^{\mathrm{t}}(Q)$, and maps it to a fixed-size queue $QueryQueues[x]$ whose capacity is $|\varphi^{\mathrm{t}}(Q)|_x$. Each such queue is initialized to contain $|\varphi^{\mathrm{t}}(Q)|_x$ members, all of which are *null* placeholder items, denoted $\perp$. See Figure 3 for an example. In Line 1, we assume the existence of a function *getQueryQueues* that receives as input a tree Q, and returns such a hash-map.
  We assume that it is possible to perform random-access reads over the elements of the queues in $QueryQueues$, using the queue's *elemAt* function. A queue's head (the next item to be returned when the queue's *dequeue* operation is performed) is always assumed to be at index 1, and subsequent queued elements reside at the indices $2, 3, \ldots, |\varphi^{\mathrm{t}}(Q)|_x$.
- *Nodes to Counters:* The hash-map *Sizes* takes a node as key, and maps it to an integer value. This value is used in calculating the size $|\varphi^{\mathrm{t}}(T_v)|$ for all nodes $v \in \mathcal{V}(T)$. To simplify the pseudocode presented here,

given a node v, if $v \notin Sizes.keys$ and we attempt to access the value $Sizes[v]$, then a value of 0 is automatically allocated and assigned to the key v in the map. In other words, for all $v \notin Sizes.keys$, v is implicitly mapped to a default value of 0.

Bounded Priority Queue: In Line 3, *Results* is a priority queue, limited to hold at most k entries. Each entry in the queue is a tuple $\langle \delta, v \rangle$, where $v \in \mathcal{V}(T)$ is a node, and $\delta = \mathcal{D}(Q, T_v)$ is the distance between the tree Q and the subtree T_v rooted at v, as determined by the profile distance function $\mathcal{D}$. Entries with a lower distance value are given precedence in this priority queue, and the function *boundedAdd* ensures only the best k entries seen so far are stored.

B. Overview of the Algorithm

After initializing the necessary data structures (Lines 1–3), *DynamicSearch* iterates in post-order[3] over all nodes $v \in \mathcal{V}(T)$. For each node $v \in \mathcal{V}(T)$ we wish to obtain the value $\mathcal{D}(Q, T_v)$, and add v to *Results*, if this value is among the k best that we have seen so far.

As $\mathcal{D}$ uses a multiset-distance function to calculate distance between trees, in order to calculate the value $\mathcal{D}(Q, T_v)$ we need to know the sizes of the intersection $|\varphi^{\mathrm{t}}(Q) \uplus \varphi^{\mathrm{t}}(T_v)|$ and the profile sizes $|\varphi^{\mathrm{t}}(Q)|, |\varphi^{\mathrm{t}}(T_v)|$. *DynamicSearch* assumes that $|\varphi^{\mathrm{t}}(Q)|$ is already computed before the algorithm begins.

Following Equation (1), due to the fact that φ^{t} is a composite tree profile function, for all $w \in \mathcal{V}(T)$ it holds[4] that:

$$|\varphi^{\mathrm{t}}(T_w)| = |\varphi^{\mathrm{v}}_{local}(w)| + \sum_{u \in \mathcal{V}(T_w) \setminus \{w\}} |\varphi^{\mathrm{v}}_{global}(u)|.$$

The hash-map *Sizes* is used to perform this calculation for all $w \in \mathcal{V}(T)$, as we traverse the tree T in post-order. At the end of each iteration (Lines 14–15) we add $Sizes[w] + |\varphi^{\mathrm{v}}_{global}(w)|$ to $Sizes[w.parent]$ and release the memory for $Sizes[w]$. Thus, it holds that when we reach the algorithm's iteration over the parent v of w, in Line 8 we have $Sizes[v] = \sum_{u \in \mathcal{V}(T_v) \setminus \{v\}} |\varphi^{\mathrm{v}}_{global}(u)|$, and therefore the variable $vProfileSize$ gets the correct value of $|\varphi^{\mathrm{t}}(T_v)|$.

The remainder of the algorithm *DynamicSearch* is dedicated to computing $\mathcal{D}(Q, T_v)$ by deriving the sizes of the intersection $|\varphi^{\mathrm{t}}(Q) \uplus \varphi^{\mathrm{t}}(T_v)|$, for each node v, as described next.

Due to the process which takes place in Lines 11–13 (this process will be explained later), at the beginning of each iteration over v, it holds that $QueryQueues$ contains all the information needed to compute the size of the multiset intersection $|\varphi^{\mathrm{t}}(Q) \uplus \varphi^{\mathrm{t}}(T_v)|$. To see why, consider some $x \in \varphi^{\mathrm{t}}(Q)$. The queue $QueryQueues[x]$ contains, among other elements, nodes from the subtree rooted at v that have x in their global profile (this is achieved by Line 13). The contents of the queue are ordered in a fashion that enables us to easily infer data regarding the size of the intersection. More specifically, the queue $QueryQueues[x]$ contains, from head to tail, a possibly empty prefix of $\perp$ elements, followed by a suffix of nodes $u \in \mathcal{V}(T) \setminus \{v\}$, ordered by increasing post-order. These nodes u all have x in their $\varphi^{\mathrm{v}}_{global}$ profile. Since the nodes are sorted by post-order, we obtain that any nodes u in the queue such that $u \notin \mathcal{V}(T_v)$ must all reside in a prefix of the non-$\perp$ elements in the queue. Thus, $QueryQueues[x]$ contains a prefix of $\perp$ and nodes from outside the current subtree, followed by a suffix of the nodes from the current subtree that have x in their profile. In case more than $|\varphi^{\mathrm{t}}(Q)|_x$ instances of x exist in $\varphi^{\mathrm{v}}_{global}(\mathcal{V}(T_v) \setminus \{v\})$, we would only keep the last $|\varphi^{\mathrm{t}}(Q)|_x$ instances we have seen.

The procedures *IntersectGlobal* and *IntersectLocal* extract information from the queues, to calculate the intersection size. Note that for a node u such that u comes before v in post-order (or $u = v$), it holds that $u \in \mathcal{V}(T_v)$ if and only if $\mathrm{lca}(u, v) = v$, where $\mathrm{lca}(u, v)$ is the lowest common ancestor of u and v. This fact is used in both *IntersectGlobal* and *IntersectLocal*.

IntersectGlobal goes over all distinct elements $x \in \varphi^{\mathrm{t}}(Q)$. Each unique element x appears in at most one iteration here, as we only iterate over distinct elements of the multiset. Each such element x is a key of the hash-map $QueryQueues$. For each queue in $QueryQueues$, searching from its tail to its head, *IntersectGlobal* finds where the suffix of nodes $u \in \mathcal{V}(T_v) \setminus \{v\}$ starts. In Line 7 of *IntersectGlobal*, the sum of all suffix sizes is exactly $count = |\varphi^{\mathrm{t}}(Q) \uplus \varphi^{\mathrm{v}}_{global}(\mathcal{V}(T_v) \setminus \{v\})|$.

IntersectLocal goes over all distinct elements x in $\varphi^{\mathrm{v}}_{local}(v)$ such that $x \in \varphi^{\mathrm{t}}(Q)$. For each such x, the size of the prefix of the queue $QueryQueues[x]$ that contains nodes $u \notin \mathcal{V}(T_v)$ (or $\perp$), is exactly the number of appearances of x in $\varphi^{\mathrm{t}}(Q)$ that are not already assigned to some node $u \in \mathcal{V}(T_v) \setminus \{v\}$. We add to a counter, the minimum between the size of this prefix and $|\varphi^{\mathrm{v}}_{local}(v)|_x$. Thus, in Line 7 of *IntersectLocal*, we return $count = |(\varphi^{\mathrm{t}}(Q) \setminus \varphi^{\mathrm{v}}_{global}(\mathcal{V}(T_v) \setminus \{v\})) \uplus \varphi^{\mathrm{v}}_{local}(v)|$.

Hence, in Line 7 of *DynamicSearch*, we have

$$\begin{aligned} intersect = {} & |\varphi^{\mathrm{t}}(Q) \uplus \varphi^{\mathrm{v}}_{global}(\mathcal{V}(T_v) \setminus \{v\})| + \\ & |(\varphi^{\mathrm{t}}(Q) \setminus \varphi^{\mathrm{v}}_{global}(\mathcal{V}(T_v) \setminus \{v\})) \uplus \varphi^{\mathrm{v}}_{local}(v)| \end{aligned} \quad (2)$$

which is precisely $|\varphi^{\mathrm{t}}(Q) \uplus \varphi^{\mathrm{t}}(T_v)|$.

Line 9 of *DynamicSearch* computes the distance of T_v from Q using the values $|\varphi^{\mathrm{t}}(Q)|, |\varphi^{\mathrm{t}}(T_v)|, |\varphi^{\mathrm{t}}(Q) \uplus \varphi^{\mathrm{t}}(T_v)|$. Finally, Line 10 adds the new result to the priority queue of results.

In Lines 11–13, we iterate over the values x in $\varphi^{\mathrm{v}}_{global}(v)$ that are common with $\varphi^{\mathrm{t}}(Q)$. Note that we will iterate several times for the same value x if it appears several times in $\varphi^{\mathrm{v}}_{global}(v)$. In Lines 12–13, we dequeue the node u at the head of the queue for x (it is possible that $u = \perp$), and insert v to the queue's tail, indicating that v has the value x in its $\varphi^{\mathrm{v}}_{global}$ profile. This process ensures that the properties of $QueryQueues$ which were mentioned earlier persist in the algorithm's next iterations.

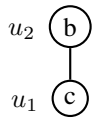
After iterating over all of $\mathcal{V}(T)$, the main loop finishes. In Line 16, the best results are returned.

C. A Sample Run of the Algorithm DynamicSearch

We demonstrate a sample run of the algorithm *DynamicSearch* using the query Q from Figure 3 and the database T

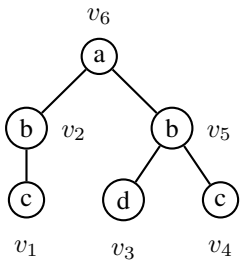
[3] We note that if the profile distance function that is used is compatible with unordered trees, then we arbitrarily assign an order on sibling nodes.

[4] The equation holds since we are using multiset operations.



x		$QueryQueues[x]$
(*, b; c)	→	$\boxed{\perp}$
(b, c; *)	→	$\boxed{\perp}$

Fig. 3: A tree Q, and a table which shows for each pq-gram label tuple $x \in \varphi^{\mathrm{t}}_{2,1}(Q)$, the queue $QueryQueues[x]$. Each queue is initialized with placeholder $\perp$ elements, and is filled up to its maximal capacity $|\varphi^{\mathrm{t}}_{2,1}(Q)|_x$. In this example, Q is very simple and results in only 2 queues, both having a capacity of 1.



$v \in \mathcal{V}(T)$	$\varphi^{\mathrm{v}}_{local}(v)$	$\varphi^{\mathrm{v}}_{global}(v)$
v_1	(*, c; *)	
v_2	(*, **b**; **c**) (**b**, **c**; *)	(**b**, **c**; *)
v_3	(*, d; *)	
v_4	(*, c; *)	
v_5	(*, b; d) (*, **b**; **c**) (b, d; *) (**b**, **c**; *)	(b, d; *) (**b**, **c**; *)
v_6	(*, a; b) (*, a; b) (a, b; c) (a, b; d) (a, b; c)	(a, b; c) (a, b; d) (a, b; c)

Fig. 4: A tree T with nodes $\mathcal{V}(T) = \{v_1, \ldots, v_6\}$ (numbered by post-order). The global and local multisets of the tree nodes appear in the table, for $\varphi^{\mathrm{t}}_{2,1}(T)$. Elements that also appear in the profile of the tree Q from Figure 3 are in bold font.

from Figure 4. We will show how subtree similarity-search is performed for the pq-grams distance (of [14]), using our algorithm. For this, we use the profile function $\varphi^{\mathrm{t}}_{2,1}$ (pq-grams for $p = 2, q = 1$), and the multiset distance function

$$dist^{p,q}_{norm}(M_1, M_2) = \frac{|M_1 \uplus M_2| - 2|M_1 \uplus M_2|}{|M_1 \uplus M_2| - |M_1 \uplus M_2|} .$$

Both are defined in [14]. Figures 3 and 4 provide the necessary information regarding the contents of the pq-gram profiles associated with the trees. We choose $k = 3$, and thus, find the top-3 subtrees of T, which are most similar to Q.

In Line 1, $QueryQueues$ is created, mapping each $x \in \varphi^{\mathrm{t}}_{2,1}(Q)$ to a queue of fixed size, and each such queue is filled with $\perp$ elements (see Figure 3). Note that $|\varphi^{\mathrm{t}}_{2,1}(Q)| = 2$, and this value is calculated before the algorithm begins.

As will be apparent, throughout the iterations of the algorithm, when iterating over a node $v \in \mathcal{V}(T)$, in Line 7 it holds that $intersect = |\varphi^{\mathrm{t}}_{2,1}(Q) \uplus \varphi^{\mathrm{t}}_{2,1}(T_v)|$, and in Line 8 it holds that $vProfileSize = |\varphi^{\mathrm{t}}_{2,1}(T_v)|$.

We begin iterating over nodes in $\mathcal{V}(T)$ in post-order, starting with $v = v_1$. In Lines 5–6, both calls to *IntersectLocal* and *IntersectGlobal* return 0 and so $intersect$ gets a value of 0 (note that indeed, $|\varphi^{\mathrm{t}}_{2,1}(Q) \uplus \varphi^{\mathrm{t}}_{2,1}(T_{v_1})| = 0$). In Line 8 we set $vProfileSize = 1$ (and indeed, it holds that $|\varphi^{\mathrm{t}}_{2,1}(T_{v_1})| = 1$). In Line 9, we calculate $dist^{p,q}_{norm}(Q, T_{v_1}) = \frac{2+1-2\cdot 0}{2+1-0} = 1$ and add this to $Results$. In Line 11 we start iterating over $x \in \varphi^{\mathrm{v}}_{global}(v_1)$, but since $\varphi^{\mathrm{v}}_{global}(v_1) = \emptyset$, we do nothing. We add $Sizes[v_1] + |\varphi^{\mathrm{v}}_{global}(v_1)| = 0$ to the parent's counter (whose value is 0 too), $Sizes[v_2]$.

We now iterate over $v = v_2$. As $QueryQueues$ is still unchanged from the previous iteration, the call to *IntersectGlobal* returns 0. During *IntersectLocal* we first consider $x = (*, \mathrm{b}; \mathrm{c}) \in \varphi^{\mathrm{v}}_{local}(v_2)$. The queue $QueryQueues[x]$ contains an entry $\perp$, and so we increment $count$. Next, we consider $x = (\mathrm{b}, \mathrm{c}; *) \in \varphi^{\mathrm{v}}_{local}(v_2)$ and once again for a similar reason increment $count$. Thus, *IntersectLocal* returns 2, and back in *DynamicSearch*, $intersect$ gets a value of 2. In Line 8 we set $vProfileSize = 2$. In Line 9, we calculate $dist^{p,q}_{norm}(Q, T_{v_2}) = \frac{2+2-2\cdot 2}{2+2-2} = 0$ and add this to $Results$. Note that the tree T_{v_2} is identical to Q, which explains the distance value of 0. In Line 11 we start iterating over $x \in \varphi^{\mathrm{v}}_{global}(v_2)$. We see $x = (\mathrm{b}, \mathrm{c}; *)$, dequeue an element $\perp$ from $QueryQueues[x]$, and enqueue v_2 instead. We add $Sizes[v_2] + |\varphi^{\mathrm{v}}_{global}(v_2)| = 1$ to the parent's counter (whose value is 0), $Sizes[v_6]$.

In the next 2 iterations we consider nodes v_3, v_4, and in each of those iterations the variable $intersect$ gets a value of 0, so we try and add 2 more entries to $Results$, each with distance 1. Since the priority queue $Results$ is bounded to contain only the top-3 best results, when adding the entry for v_4, $Results$ remains unchanged. Additionally, the counter $Sizes[v_5]$ for the parent of v_3, v_4, remains with a value of 0 after these iterations, and the contents of $QueryQueues$ do not change.

Now, we iterate over $v = v_5$. During the call to *IntersectGlobal*, while looking at the contents of both queues in $QueryQueues$ we only see $v_2 \notin \mathcal{V}(T_{v_5})$ and $\perp$, and so *IntersectGlobal* returns 0. Next, *IntersectLocal* is called. During the iteration over $x = (*, \mathrm{b}; \mathrm{c}) \in \varphi^{\mathrm{v}}_{local}(v_5)$, we see $\perp$ in $QueryQueues[x]$ and increment $count$. During the iteration over $x = (\mathrm{b}, \mathrm{c}; *) \in \varphi^{\mathrm{v}}_{local}(v_5)$, we see v_2 in $QueryQueues[x]$, and since $\mathrm{lca}(v_2, v_5) = v_6$, we increment $count$. Thus, *IntersectLocal* returns 2, and back in *DynamicSearch*, $intersect$ gets a value of 2. In Line 8 we set $vProfileSize = 4$. In Line 9, we calculate $dist^{p,q}_{norm}(Q, T_{v_5}) = \frac{2+4-2\cdot 2}{2+4-2} = 0.5$ and add this to $Results$; an entry with distance 1 (either v_1 or v_3) is removed from $Results$ to make room for the new, better entry. In Line 11 we start iterating over $x \in \varphi^{\mathrm{v}}_{global}(v_5)$. During the iteration over $x = (\mathrm{b}, \mathrm{c}; *) \in \varphi^{\mathrm{v}}_{global}(v_5)$, we dequeue v_2 and enqueue v_5. We add $Sizes[v_5] + |\varphi^{\mathrm{v}}_{global}(v_5)| = 2$ to the parent's counter (whose value is 1), $Sizes[v_6]$.

Finally, we iterate over $v = v_6$. During *IntersectGlobal* we find v_5 in one of the queues in $QueryQueues$, and return 1. The call to *IntersectLocal* returns 0 since no elements exist in $\varphi^{\mathrm{v}}_{local}(v_6)$ that are also in $\varphi^{\mathrm{t}}(Q)$. The variable $intersect$ gets a value of 1, and we note that this is indeed the size of $|\varphi^{\mathrm{t}}_{2,1}(Q) \uplus \varphi^{\mathrm{t}}_{2,1}(T_{v_6})|$, as $\varphi^{\mathrm{t}}_{2,1}(T_{v_6})$ contains two instances of $(\mathrm{b}, \mathrm{c}; *)$, but $\varphi^{\mathrm{t}}_{2,1}(Q)$ contains only one such instance. Since $Sizes[v_6] = 3$, in Line 8 we set $vProfileSize = 3 + 5 = 8$. In Line 9, we calculate $dist^{p,q}_{norm}(Q, T_{v_6}) = \frac{2+8-2\cdot 1}{2+8-1} = 0.889$ and add this to $Results$, in place of an existing entry with a higher distance value of 1. The rest of this iteration over v_6

will not have any effect, as v_6 is the last node in the tree. We reach Line 16 and return the results. The top-3 results returned in *Results* are $\langle 0, v_2\rangle, \langle 0.5, v_5\rangle, \langle 0.889, v_6\rangle$.

D. Analysis of DynamicSearch

We present a brief analysis of the correctness and complexity of the algorithm *DynamicSearch*.

Theorem 4.1: DynamicSearch returns $Top_k(Q, T)$.

Intuitively, the correctness of the algorithm follows from Equation 2. For each $v \in \mathcal{V}(T)$ we have computed $|\varphi^{\mathrm{t}}(Q) \uplus \varphi^{\mathrm{t}}(T_v)|$, $|\varphi^{\mathrm{t}}(Q)|$, $|\varphi^{\mathrm{t}}(T_v)|$, and we calculate $d(\varphi^{\mathrm{t}}(Q), \varphi^{\mathrm{t}}(T_v))$ using these values. The priority queue ensures only the top-k best results are saved. Precise details of the proof were omitted due to space limitations.

We use $t_{lca}(T)$ to denote the amount of time it takes to perform *lca* queries over a tree T, to obtain the lowest common ancestor of two given nodes in the tree. Using an approach as described in [25], we have $t_{lca}(T) = \mathcal{O}(1)$, assuming we performed additional $\mathcal{O}(|T|)$ work in a pre-processing stage. However, in this paper we use *Dewey identifiers* [26] to simplify the implementation, and with this approach we have $t_{lca}(T) = \mathcal{O}(height(T))$.

To simplify the analysis, we will assume that obtaining $\varphi^{\mathrm{v}}_{local}(v)$ and $\varphi^{\mathrm{v}}_{global}(v)$ for all $v \in \mathcal{V}(T)$ takes $\mathcal{O}(|\varphi^{\mathrm{v}}_{local}(\mathcal{V}(T))|)$ and $\mathcal{O}(|\varphi^{\mathrm{v}}_{global}(\mathcal{V}(T))|)$ respectively, and that obtaining $\varphi^{\mathrm{t}}(Q)$ takes $\mathcal{O}(|\varphi^{\mathrm{t}}(Q)|)$. In case these runtimes are higher (depending on the function $\mathcal{D}$), the appropriate factor should be added to the following complexity analysis.

Theorem 4.2: The algorithm *DynamicSearch* runs in time:

$$\mathcal{O}(|\varphi^{\mathrm{t}}(Q)| + |T| \log k + t_{lca}(T)|T||\varphi^{\mathrm{t}}(Q)| + |\varphi^{\mathrm{v}}_{global}(\mathcal{V}(T))| + t_{lca}(T)|\varphi^{\mathrm{v}}_{local}(\mathcal{V}(T))|).$$

To see why, observe:

- The call to *getQueryQueues* (Line 1) takes $\mathcal{O}(|\varphi^{\mathrm{t}}(Q)|)$.
- In each iteration over v, adding elements to the bounded priority queue (Line 10) takes $\mathcal{O}(\log k)$. Using amortized analysis over all iterations, this contributes a total of $\mathcal{O}(|T| \log k)$.
- The iteration over $\varphi^{\mathrm{v}}_{global}(v)$ for a node v (Lines 11–13) contributes $\mathcal{O}(|\varphi^{\mathrm{v}}_{global}(v)|)$. Since we iterate over all $v \in \mathcal{V}(T)$, in total we have $\mathcal{O}(|\varphi^{\mathrm{v}}_{global}(\mathcal{V}(T))|)$.
- In each call to *IntersectGlobal* (Line 5), we perform $\mathcal{O}(t_{lca}(T)|\varphi^{\mathrm{t}}(Q)|)$ operations. As we make $|T|$ such calls, this contributes $\mathcal{O}(t_{lca}(T)|T||\varphi^{\mathrm{t}}(Q)|)$.
- In each call to *IntersectLocal* (Line 6), we perform $\mathcal{O}(t_{lca}(T)|\varphi^{\mathrm{v}}_{local}(v)|)$ operations, for the current node v. Using amortized analysis, in total over all iterations, this contributes $\mathcal{O}(t_{lca}(T)|\varphi^{\mathrm{v}}_{local}(\mathcal{V}(T))|)$.

Note the large factor of $t_{lca}(T)|T||\varphi^{\mathrm{t}}(Q)|$, requiring $\mathcal{O}(t_{lca}(T)|\varphi^{\mathrm{t}}(Q)|)$ work for each node in T. As T and Q get larger, this is significant overhead. In the next section we present an algorithm that avoids this overhead.

Algorithm *ProfileSimSearch*$(Q, T, k, \mathcal{D})$

```
1.  QueryQueues ← getQueryQueues (Q)
2.  Isects ← new HashMap
3.  Sizes ← new HashMap
4.  Results ← new BoundedPriorityQueue(k)
5.  for each v ∈ V(T) in post-order:
6.      local ← IntersectLocal(QueryQueues, v, φ^t)
7.      intersect ← Isects[v] + local
8.      vProfileSize ← Sizes[v] + |φ^v_local(v)|
9.      δ ← d(|φ^t(Q)|, vProfileSize, intersect)
10.     Results.boundedAdd(⟨δ, v⟩)
11.     for each x ∈ φ^v_global(v) s.t. x ∈ φ^t(Q):
12.         u ← QueryQueues[x].dequeue()
13.         if u ≠ ⊥:
14.             w ← lca(v, u)
15.             decrement Isects[w]
16.         QueryQueues[x].enqueue(v)
17.         increment Isects[v]
18.     add Isects[v] to Isects[v.parent]
19.     add (Sizes[v] + |φ^v_global(v)|) to Sizes[v.parent]
20.     free the memory allocated for Isects[v]
21.     free the memory allocated for Sizes[v]
22. return Results
```

Fig. 5: A more efficient algorithm for subtree similarity-search.

Now consider the memory requirements of our algorithm. We will assume that elements x of the multisets $\varphi^{\mathrm{v}}_{global}(v), \varphi^{\mathrm{v}}_{local}(v)$ are generated as we iterate over them, and that once we are finished iterating over such an element, the memory used for the element is released. In addition, to simplify, assume all multiset elements x have a constant size.

Theorem 4.3: The memory usage of *DynamicSearch* is:

$$\mathcal{O}(|\varphi^{\mathrm{t}}(Q)| + height(T) + k)\,.$$

- We use $\mathcal{O}(|\varphi^{\mathrm{t}}(Q)|)$ memory for $QueryQueues$.
- We allocate and de-allocate memory to the $Sizes$ map throughout the algorithm. We always keep in memory a counter for the current node $v \in \mathcal{V}(T)$ and possibly some of its ancestors. Whenever we finish iterating over the node v, its counter $Sizes[v]$ is deleted from the memory, which implies that at any point in time, we retain at most $\mathcal{O}(height(T))$ elements in $Sizes$.
- Naturally, the top-k results take $\mathcal{O}(k)$ space.

V. The *ProfileSimSearch* Algorithm

The algorithm *DynamicSearch* has a significant drawback; it must iterate over all of $\varphi^{\mathrm{t}}(Q)$ for each node in T. As T is assumed to be very large, this drastically harms performance.

To overcome this inefficiency, we present the algorithm *ProfileSimSearch* in Figure 5, which solves the subtree similarity-search problem for composite tree profile functions, while achieving near linear performance. *ProfileSimSearch* does not make calls to *IntersectGlobal*, and instead obtains the value $|\varphi^{\mathrm{t}}(Q) \uplus \varphi^{\mathrm{v}}_{global}(\mathcal{V}(T_v) \setminus \{v\})|$ for each node v, by adding some operations to the iterations over the multiset $\varphi^{\mathrm{v}}_{global}(v)$. Further details appear in the discussion below.

A. Data Structures and Conventions

In addition to the data structures used by *DynamicSearch*, the algorithm *ProfileSimSearch* uses a hash-map $Isects$ that maps nodes to counters. Its behavior is similar to that of the hash-map $Sizes$, i.e., attempting to access a value associated with a key not appearing in $Isects$ causes the value 0 to be allocated and assigned to the key. This hash-map is used in calculating the sizes of the multiset intersection between $\varphi^{\mathrm{t}}(Q)$ and $\varphi^{\mathrm{v}}_{global}(\mathcal{V}(T_v) \setminus \{v\})$ for all subtrees $T_v \in SubTrees(T)$.

B. Overview of the Algorithm

The general structure of the algorithm *ProfileSimSearch* is similar to that of *DynamicSearch*. During the algorithm's iteration over nodes $v \in \mathcal{V}(T)$, the hash-map $Isects$ is used to calculate the size of the intersection $|\varphi^{\mathrm{t}}(Q) \uplus \varphi^{\mathrm{v}}_{global}(\mathcal{V}(T_v) \setminus \{v\})|$, by storing pertinent information from descendants of v. A value of 0 for $Isects[v]$ indicates that no values in the global vertex profiles of descendants of v are in common with $\varphi^{\mathrm{t}}(Q)$.

While we iterate over all $x \in \varphi^{\mathrm{v}}_{global}(v)$ that are in common with $\varphi^{\mathrm{t}}(Q)$, we dequeue the node u at the head of the queue for x, and if u is not a placeholder (i.e., $\perp$), we compute the lowest common ancestor w of v and u (Line 14). Now, we must update the counter for w with the necessary information about its descendants, as we have removed u from the queue, and thus, may lose information. Interestingly, this update takes the form of decrementing the value in $Isects$ for w (Line 15). Intuitively, a decrement is performed, as we have discovered that w has an extra copy of x in the profiles of its descendants, more than the number of appearances of x in $\varphi^{\mathrm{t}}(Q)$. Since, we will eventually be adding the number of appearances of x in the profiles of the descendants of w to the counter of w, this decrement ensures that the total count will be no larger than the intersection size. Finally, in Lines 16–17, we insert v to the queue of x (indicating that v has the value x in its profile), and increment the $Isects$ value of v, to reflect this. One can prove that this process (in addition to the fact that we update the value $Isects[v.parent]$ in Line 18) ensures that in Line 7 it holds that $Isects[v] = |\varphi^{\mathrm{t}}(Q) \uplus \varphi^{\mathrm{v}}_{global}(\mathcal{V}(T_v) \setminus \{v\})|$.

Thus, the call to *IntersectGlobal* that we had in *DynamicSearch* is not needed, as $Isects$ provides the same functionality. The remainder of the algorithm *ProfileSimSearch* works the same as *DynamicSearch*. In Line 6, *local* is assigned the value $|(\varphi^{\mathrm{t}}(Q) \setminus \varphi^{\mathrm{v}}_{global}(\mathcal{V}(T_v) \setminus \{v\})) \uplus \varphi^{\mathrm{v}}_{local}(v)|$, and thus in Line 7, we assign

$$intersect = |\varphi^{\mathrm{t}}(Q) \uplus \varphi^{\mathrm{v}}_{global}(\mathcal{V}(T_v) \setminus \{v\})| + |(\varphi^{\mathrm{t}}(Q) \setminus \varphi^{\mathrm{v}}_{global}(\mathcal{V}(T_v) \setminus \{v\})) \uplus \varphi^{\mathrm{v}}_{local}(v)| \quad (3)$$

which is precisely $|\varphi^{\mathrm{t}}(Q) \uplus \varphi^{\mathrm{t}}(T_v)|$, just as we had in *DynamicSearch*.

C. A Sample Run of the Algorithm ProfileSimSearch

Due to the similarity between *ProfileSimSearch* and *DynamicSearch*, we revisit our sample run of *DynamicSearch* from Subsection IV-C and point out the important differences. The key point here is that during the iteration over a node $v \in \mathcal{V}(T)$, the value of $Isects[v]$ in Line 7 of *ProfileSimSearch*, is always equal to the value which would be returned by *IntersectGlobal* when running *DynamicSearch*. We shall now see that this is indeed the case.

In the first iteration, $v = v_1$, and $Isects[v_1] = 0$. Since $\varphi^{\mathrm{v}}_{global}(v_1) = \emptyset$, no changes are made to $QueryQueues$ at this iteration. Next, we iterate over $v = v_2$, and $Isects[v_2] = 0$. We iterate over $\varphi^{\mathrm{v}}_{global}(v_2)$, and when we encounter $x = (\mathrm{b}, \mathrm{c}; *) \in \varphi^{\mathrm{v}}_{global}(v_2)$, we increase $Isects[v_2]$ by 1. We now update the parent's counter $Isects[v_6]$ to a value of 1. During the next two iterations of *ProfileSimSearch*, the nodes v_3, v_4 have $Isects$ values of 0, and no other counters are changed. Next, we iterate over $v = v_5$, and $Isects[v_5] = 0$. We iterate over $\varphi^{\mathrm{v}}_{global}(v_5)$, and when we encounter $x = (\mathrm{b}, \mathrm{c}; *) \in \varphi^{\mathrm{v}}_{global}(v_5)$, we dequeue the previously enqueued v_2 from $QueryQueues[x]$. The lowest common ancestor $w = \mathrm{lca}(v_5, v_2)$ is v_6, and so we decrease $Isects[v_6]$ back to 0. We increase $Isects[v_5]$ by 1, and at the end of the iteration over v_5 we propagate the $Isects$ value up to the parent v_6 of v_5, so that once again $Isects[v_6] = 1$. Intuitively, during this iteration we handled the fact that there are more instances of x in $\varphi^{\mathrm{t}}_{2,1}(T_{v_6})$ than there are in $\varphi^{\mathrm{t}}_{2,1}(Q)$, by decreasing $Isects[v_6]$ and then increasing it back again. This has prevented a potential double-counting in the size of the intersection $|\varphi^{\mathrm{t}}_{2,1}(Q) \uplus \varphi^{\mathrm{t}}_{2,1}(T_{v_6})|$ later on. Finally, *ProfileSimSearch* reaches $v = v_6$, and $Isects[v_6] = 1$.

D. Analysis of ProfileSimSearch

In the following complexity and memory analysis, we continue with the assumptions stated in Subsection IV-D.

Theorem 5.1: *ProfileSimSearch* returns $Top_k(Q, T)$, using memory of size:

$$\mathcal{O}(|\varphi^{\mathrm{t}}(Q)| + height(T) + k)\,.$$

The correctness of the algorithm follows from Equation 3. Precise details of the proof are omitted due to space limitations.

We shall now analyze memory use, which is of the same order of magnitude as that of *DynamicSearch*. Note that other than having the additional hash-map $Isects$, the memory usage of *ProfileSimSearch* is identical to that of *DynamicSearch*. Due to reasons analogous to those which we have seen when analyzing the memory usage of $Sizes$, the hash-map $Isects$ also stores $\mathcal{O}(height(T))$ elements.

While correctness and memory usage were similar to that of *DynamicSearch*, the algorithm *ProfileSimSearch* is significantly more efficient.

Theorem 5.2: The algorithm *ProfileSimSearch* runs in:

$$\mathcal{O}(|\varphi^{\mathrm{t}}(Q)| + |T| \log k + t_{lca}(T)(|\varphi^{\mathrm{v}}_{global}(\mathcal{V}(T))| + |\varphi^{\mathrm{v}}_{local}(\mathcal{V}(T))|)).$$

The analysis is as follows:

- A factor of $\mathcal{O}(|\varphi^{\mathrm{t}}(Q)| + |T| \log k + t_{lca}(T)|\varphi^{\mathrm{v}}_{local}(\mathcal{V}(T))|)$ is taken by all parts of the algorithm except the loop of Lines 11–17. This follows from a similar analysis to that of *DynamicSearch*.

- Since we perform *lca* queries during the loop over $\varphi^{\mathrm{v}}_{global}(v)$ (Lines 11–17) for all $v \in \mathcal{V}(T)$, additional $\mathcal{O}(t_{lca}(T)|\varphi^{\mathrm{v}}_{global}(\mathcal{V}(T))|)$ operations are needed.
- Notice that *ProfileSimSearch* does not make calls to *IntersectGlobal*, and thus we do *not* pay the additional $\mathcal{O}(t_{lca}(T)|T||\varphi^{\mathrm{t}}(Q)|)$ that was needed in *DynamicSearch*. This has a dramatic positive effect on the runtime of *ProfileSimSearch*, as $|T|$ is usually very large.

For many practical applications, $t_{lca}(T)$ can be considered constant. (This is certainly true if *lca* is computed in the manner suggested in [25], where $t_{lca}(T) = \mathcal{O}(1)$. Even if Dewey identifiers are used to calculate *lca*, we have $t_{lca}(T) = \mathcal{O}(height(T))$, and commonly trees are wide, but not excessively tall, resulting in a very small value of $height(T)$.) The value k can also be thought of as a constant, as the number of results of interest is usually a small number with respect to the total size of the database (e.g., the user may be interested in hundreds of results, while the database has millions of nodes). When taking $t_{lca}(T)$ and k as constant, the algorithm *ProfileSimSearch* has linear runtime.

We note that for many composite profile functions, including pq-grams, windowed pq-grams, and binary branch distance (and also $\varphi^{\mathrm{t}}_{lab}$ from Example 3.2), for a tree S it holds that the sizes $|\varphi^{\mathrm{v}}_{global}(\mathcal{V}(S))|$ and $|\varphi^{\mathrm{v}}_{local}(\mathcal{V}(S))|$ are both $\mathcal{O}(|S|)$. In this case, the analysis can be further simplified.

VI. Implementation

Both *ProfileSimSearch* and *DynamicSearch* were implemented in Java, and work with trees in an XML input format. We chose to pre-process the tree T, and generate in advance the multisets $\varphi^{\mathrm{v}}_{global}(v), \varphi^{\mathrm{v}}_{local}(v)$ for all $v \in \mathcal{V}(T)$, as these multisets remain the same, regardless of the query.

We assign each element x, appearing in at least one of the profile multisets of $\mathcal{V}(T)$, a unique identifier. Instead of storing the actual multisets $\varphi^{\mathrm{v}}_{global}(v), \varphi^{\mathrm{v}}_{local}(v)$, for each v, we store the multisets of identifiers for $\varphi^{\mathrm{v}}_{global}(v)$ and $\varphi^{\mathrm{v}}_{local}(v)$. To be precise, the file which we generate when pre-processing T contains an entry for each node $v \in \mathcal{V}(T)$. Node data is stored sequentially in post-order. Each entry for a node $v \in \mathcal{V}(T)$ contains the Dewey identifier of v, and the identifiers of all elements of $\varphi^{\mathrm{v}}_{global}(v)$ and of $\varphi^{\mathrm{v}}_{local}(v)$.[5] Note that we use Dewey identifiers for nodes, and hence, we have $t_{lca}(T) = \mathcal{O}(height(T))$. In our implementation, the assumptions from Subsection IV-D hold, i.e., we can efficiently enumerate the profile elements of T.

Regardless of the precise user input Q, this pre-processed version of T will always be read by our algorithm, in a sequential manner, thus avoiding disk seek operations. Finally, we note that the file for T is compressed using the Snappy compression library[6], to further reduce read times.

As part of the pre-processing of T, we use the Berkeley DB to store a hash table of profile-elements to identifiers. This hash table is later utilized when we are given a query Q, to find the identifiers associated with the elements in the profile of Q. Thus, all comparisons of profile elements in the algorithm are actually integer comparisons, instead of complex element comparisons, thereby significantly saving time.

[5] We note that for functions such as pq-grams or $\varphi^{\mathrm{t}}_{lab}$, in which it holds that $\varphi^{\mathrm{v}}_{global}(v) \subseteq \varphi^{\mathrm{v}}_{local}(v)$ for all $v \in \mathcal{V}(T)$, in our implementation we avoid storing the elements of $\varphi^{\mathrm{v}}_{global}(v)$ twice in the input file, by applying a slightly different order of operations in each iteration of the algorithm.

[6] http://code.google.com/p/snappy/

TABLE I: Dataset statistics.

Name	Nodes (millions)	XML Size (MB)	2,3-grams (MB)	BiB (MB)	Labels (MB)
XMark100	3.6	111	70	43	32
XMark200	7.2	226	142	88	66
XMark400	14.5	454	288	177	132
XMark900	28.9	910	585	358	267
XMark1800	57.8	1770	1140	718	535
DBLP	17.6	311	322	198	153
SProt	9.4	109	150	97	77

VII. Experimentation

The algorithms *DynamicSearch* and *ProfileSimSearch* are both evaluated throughout this section, and are compared with previously studied algorithms.

All experimentation was performed on a desktop machine running Windows 7 64-bit with an i7-2600k CPU clocked at 3.4GHz. The machine has 8GB of RAM, out of which 3GB were dedicated to running a Java 6 VM. A standard consumer-grade 7200 RPM hard drive was used.

For our datasets, we mostly use the XMark benchmark [27], which generates XML files of varying sizes. In addition, two non-synthetic datasets[7] were considered: the DBLP dataset, containing bibliographic information, and the SwissProt dataset, containing annotated protein sequences.

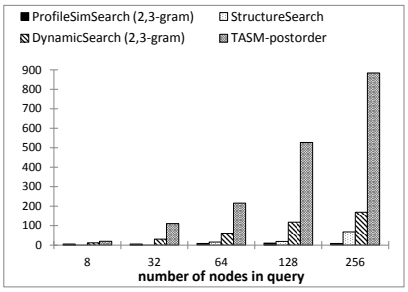
Table I lists details regarding the datasets we have used. Numbers have been rounded for readability. The table lists for each dataset the number of nodes it contains, the size of its XML file in megabytes, and the sizes of our pre-processed sequential database file containing profile elements, for the three profile functions that we tested (as will be detailed later). Sizes of the on-disk hash-tables mapping multiset elements to identifiers are not included in the table.

The queries used in the experimentation were chosen randomly from the subtrees of the datasets used. In addition, each point appearing in our charts actually reflects the result of running four experiments (using randomly chosen queries of the same size), and returning the average of the runtimes. All of the charts in Figure 6 present runtimes in seconds.

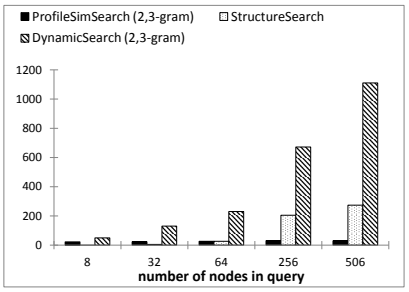
Increasing Query Size. We compare the runtime of our algorithms *ProfileSimSearch* and *DynamicSearch* using the pq-gram profile function (with $p = 2, q = 3$) and Dice difference of multisets, with the algorithms *TASM-postorder* from [9], [10], and *StructureSearch* from [21]. *TASM-postorder* and *StructureSearch* solve the subtree similarity-search problem for tree edit distance.[8] (Recall that one of the motivations for pq-

[7] *DBLP*: http://dblp.uni-trier.de/xml, *SwissProt*: http://www.expasy.ch/sprot

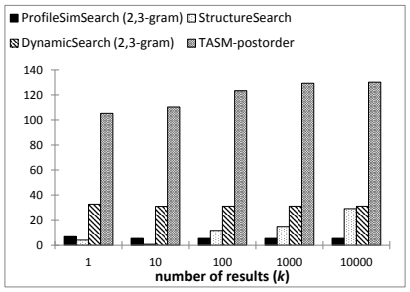
[8] The algorithm *StructureSearch* receives an extra input parameter m which is an upper bound on the edit distance of the results from the query. As was shown in [21], this value does not impose a serious performance impact, so we chose a default value of $m = |Q|$.



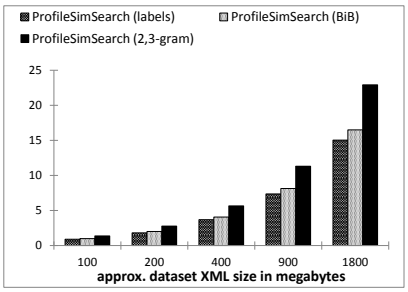
(a) Varying $|Q|$, XMark400



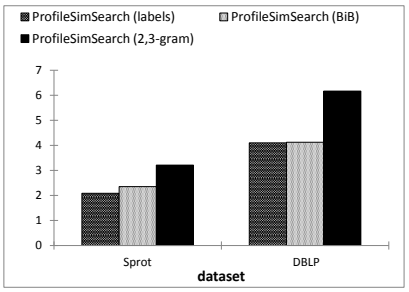
(b) Varying $|Q|$, XMark1800



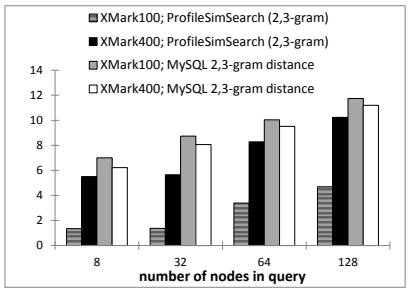
(c) Varying k, XMark400



(d) Varying dataset size



(e) Non-synthetic datasets



(f) Comparison of *ProfileSimSearch* and the tree-to-tree pq-gram distance from [14]

Fig. 6: Execution times for varying query sizes, number of results and dataset sizes. All times are in seconds.

grams is that it is more efficiently computed than tree edit distance.) For this comparison, the authors of [9], [10], [21] kindly provided us with their source code.

Figures 6a and 6b show the results of running queries with increasing sizes $|Q|$, on the datasets XMark400 and XMark1800, respectively. We set k to be 10, i.e., we computed the top-10 results. The performance of *TASM-postorder* does not appear in Figure 6b due to its excessive runtime.

As is apparent in the figures, *ProfileSimSearch* is almost unaffected by the size of the query, and thus, as the query size increases, the runtime remains almost constant. This is easily explained, as the size $|\varphi^{\mathrm{t}}_{p,q}(Q)|$ of the pq-grams profile is linear in the size $|Q|$ of the query, and even large queries are very small in comparison to the size of the dataset $|T|$. Thus the difference in runtimes for larger queries is negligible, for *ProfileSimSearch*.

For the other algorithms considered, the performance degrades (sometimes quite significantly) as the query gets larger. Note that *DynamicSearch* is very sensitive to the size of the queries, due to the extra $\mathcal{O}(t_{lca}(T)|T||\varphi^{\mathrm{t}}(Q)|)$ factor in its runtime complexity. The algorithm *StructureSearch* heavily relies on index structures, and therefore gives good performance for small queries. However, as queries grow larger it must read many more index entries, yielding slower performance. Note that even for small queries where *StructureSearch* has a better runtime than *ProfileSimSearch*, the time difference is not very large. As for the algorithm *TASM-postorder*, it has non-linear runtime complexity, which can explain its sensitivity to the query size.

Increasing Output Size. Figure 6c shows the result of running queries of constant size ($|Q| = 32$), over the dataset XMark400, with different increasing values for k, i.e., with increasing output size. Once again, we compare *ProfileSimSearch* and *DynamicSearch* with *TASM-postorder* and *StructureSearch*. We note that similar results for varying k values were observed in all datasets, including XMark1800.

The runtime of *ProfileSimSearch* is nearly unaffected by the parameter k. This was apparent from the theoretical runtime complexity, and is borne out in practice. For similar reasons, *DynamicSearch* is also not sensitive to changes in k (but, as expected, it fares significantly worse than *ProfileSimSearch*, due to the extra operations that it performs).

StructureSearch takes more time to run as k grows larger. This can be attributed to the fact that the algorithm can derive less effective tree distance bounds for large k, which entails additional operations. *TASM-postorder* also appears to be somewhat affected by changes in k.

Increasing Dataset Size. Figure 6d shows the runtime of *ProfileSimSearch* on increasingly large datasets. As before, queries are of size 32, and $k = 10$. Performance of *ProfileSimSearch* was measured using three different profile functions: pq-grams

with $p = 2, q = 3$, binary branches (abbreviated *BiB*), and the label profile $\varphi^{\mathrm{t}}_{lab}$ from Example 3.2. All three profile functions result in profile multisets whose sizes are linear in the size of the database tree. Figure 6d clearly shows that our algorithm runs in time that is close to linear in the profile multiset size (as it is close to linear in the size of the the dataset). Figure 6e shows that similar runtimes are obtained in the non-synthetic datasets, DBLP and SProt.

Comparison With Tree-to-Tree pq-gram Distance. In this experiment, we compared *ProfileSimSearch* with the implementation of tree-to-tree pq-gram distance from [14]. Recall that a tree-to-tree distance computation simply computes the distance between the query and the tree(s) in the database, but does not compute the distances for any subtrees. Thus, in our experimentation, the algorithm from [14] returns a single distance value.

The code used to measure pq-gram distance was provided by the authors of [14], and has been slightly modified to fit into our experiments, as follows: the database tree and its pq-grams were first indexed in a MySQL 5.6 database (as per their implementation). Then, we measured the time it takes, given a query, to index the query and its pq-grams in the database and calculate the distance between it and the single, entire tree of the database.

Figure 6f shows the runtime of *ProfileSimSearch* (with $k = 10$) and the implementation of tree-to-tree pq-gram distance given in [14]. Experiments were run on the XMark100 and XMark400 datasets. Note that due to variance in runtimes of [14], its measured times for XMark100 were actually higher than the times for XMark400. Interestingly, *ProfileSimSearch* seems to perform better than the algorithm from [14], despite the fact that *ProfileSimSearch* solves the subtree similarity-search problem (i.e., considers all subtrees in the database), while [14] only calculates one value, which is the distance between the query and the entire database tree.

VIII. Conclusion

In this paper we introduced a class of tree distance functions, to which several previously studied functions belong. We presented an algorithm to efficiently solve the subtree similarity-search problem using distance functions in this class, and experimentally showed that it performs well.

Further work on the subject will include utilizing upper bounds on the sizes or other attributes of subtrees considered by our algorithm, in order to avoid excess computations. Other future work will involve using our algorithm to solve the subtree similarity-search problem for tree edit distance, by obtaining bounds on edit distance using a composite profile function.

Acknowledgment

The authors were partially supported by the Israel Science Foundation (Grant 143/09) and the Ministry of Science and Technology (Grant 3-8710).

References

[1] M. Kouylekov and B. Magnini, "Combining lexical resources with tree edit distance for recognizing textual entailment," in *MLCW*, 2005.

[2] P. Lakkaraju, S. Gauch, and M. Speretta, "Document similarity based on concept tree distance," in *HT*, 2008.

[3] W. Tang and A. C. Chung, "Cerebral vascular tree matching of 3D-RA data based on tree edit distance," in *MIAR*, 2006, vol. 4091.

[4] S. Dulucq and L. Tichit, "RNA secondary structure comparison: exact analysis of the Zhang-Shasha tree edit algorithm," *Theor. Comput. Sci*, vol. 306, no. 1-3, 2003.

[5] H. Heumann and G. Wittum, "The tree-edit-distance, a measure for quantifying neuronal morphology," *Neuroinformatics*, vol. 7, 2009.

[6] P. Klein, S. Tirthapura, D. Sharvit, and B. Kimia, "A tree-edit-distance algorithm for comparing simple, closed shapes," in *SODA*, 2000.

[7] A. Torsello, A. Robles-Kelly, and E. R. Hancock, "Discovering shape classes using tree edit-distance and pairwise clustering," *International Journal of Computer Vision*, vol. 72, 2007.

[8] J. R. Rico-Juan and L. Micó, "Comparison of AESA and LAESA search algorithms using string and tree-edit-distances," *Pattern Recognition Letters*, vol. 24, no. 9-10, 2003.

[9] N. Augsten, D. Barbosa, M. Böhlen, and T. Palpanas, "TASM: top-k approximate subtree matching," in *ICDE*, 2010.

[10] N. Augsten, D. Barbosa, M. H. Böhlen, and T. Palpanas, "Efficient top-k approximate subtree matching in small memory," *IEEE Trans. Knowl. Data Eng.*, vol. 23, no. 8, 2011.

[11] M. Garofalakis and A. Kumar, "Correlating xml data streams using tree-edit distance embeddings," in *PODS*, 2003.

[12] D. C. Reis, P. B. Golgher, A. S. Silva, and A. F. Laender, "Automatic web news extraction using tree edit distance," in *WWW*, 2004.

[13] P. Bille, "A survey on tree edit distance and related problems," *Theor. Comput. Sci*, vol. 337, 2005.

[14] N. Augsten, M. Böhlen, and J. Gamper, "The pq-gram distance between ordered labeled trees," *ACM Trans. Database Syst.*, vol. 35, no. 1, Feb. 2008.

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

[16] S. Helmer, N. Augsten, and M. Böhlen, "Measuring structural similarity of semistructured data based on information-theoretic approaches," *The VLDB Journal*, vol. 21, no. 5, 2012.

[17] K. Kailing, H.-P. Kriegel, S. Schoenauer, and T. Seidl, "Efficient similarity search for hierarchical data in large databases," in *In Proc. of EDBT*, 2004.

[18] S. Guha, H. V. Jagadish, N. Koudas, D. Srivastava, and T. Yu, "Approximate xml joins," in *SIGMOD*, 2002.

[19] N. Augsten, M. Böhlen, and J. Gamper, "Approximate matching of hierarchical data using pq-grams," in *VLDB*, 2005.

[20] N. Augsten, M. Böhlen, C. Dyreson, and J. Gamper, "Windowed pq-grams for approximate joins of data-centric xml," *The VLDB Journal*, vol. 21, no. 4, 2012.

[21] S. Cohen, "Indexing for subtree similarity-search using edit distance," in *SIGMOD*, 2013.

[22] L. Ribeiro and T. Härder, "Evaluating performance and quality of xml-based similarity joins," in *ADBIS*, 2008.

[23] L. R. Dice, "Measures of the amount of ecologic association between species," *Ecology*, vol. 26, no. 3, 1945.

[24] A. Tversky, "Features of similarity," *Psychological review*, vol. 84, no. 4, 1977.

[25] B. Schieber and U. Vishkin, "On finding lowest common ancestors: simplification and parallelization," in *VLSI Algorithms and Architectures*, 1988.

[26] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram, "Xrank: Ranked keyword search over xml documents," in *SIGMOD*, 2003.

[27] A. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse, "Xmark: A benchmark for xml data management," in *VLDB*, 2002.