SISAP 2013, 6th International Conference on Similarity Search and Applications, LNCS, 2-4 OCT 2013 (to appear)

Faster Algorithms for Tree Similarity Based on Compressed Enumeration of Bounded-Sized Ordered Subtrees

Kunihiro Wasa[1], Kouichi Hirata[2], Takeaki Uno[3], and Hiroki Arimura[1]

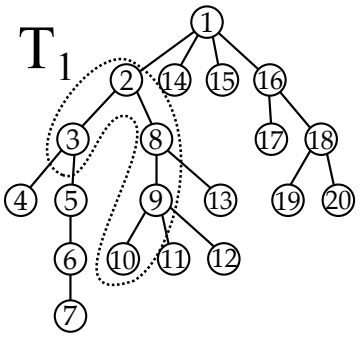
[1] Hokkaido University, N14 W9, Sapporo 060-0814, Japan
{wasa,arim}@ist.hokudai.ac.jp
[2] Kyushu Institute of Technology, Kawazu 680-4, Iizuka 820-8502, Japan
hirata@ai.kyutech.ac.jp
[3] National Institute of Informatics, 2-1-2 Hitotsubashi, Tokyo 101-8430, Japan
uno@nii.jp

Abstract. In this paper, we study efficient computation of tree similarity for ordered trees based on compressed subtree enumeration. The *compressed subtree enumeration* is a new paradigm of enumeration algorithms that enumerates all subtrees of an input tree T in the form of their compressed bit signatures. For the task of enumerating all compressed bit signatures of k-subtrees in an ordered tree T, we first present an enumeration algorithm in $O(k)$-delay, and then, present another enumeration algorithm in constant-delay using $O(n)$ time preprocessing that directly outputs bit signatures. These algorithms are designed based on bit-parallel speed-up technique for signature maintenance. By experiments on real and artificial datasets, both algorithms showed approximately 22% to 36% speed-up over the algorithms without bit-parallel signature maintenance.

1 Introduction

Similarity search is a fundamental problem in modern information and knowledge retrieval [14]. In particular, we focus on *tree similarity* between two trees, which plays a key role in a number of information and knowledge retrieval problems from semi-structured data such as similarity search, clustering, recommendation, and classification for structured data in the real world [2–4, 8, 13].

In this paper, we study the efficient computation of the frequency-based tree similarities using classes of ordered trees. *Ordered trees* are rooted trees which have total ordering among siblings. They are useful for modeling semi-structured documents such as HTML and XML, chemical compounds, natural language data, and Web access logs. In one direction, ordered tree similarities based on substructures have been extensively studied [9–11], where the focuses are on substructures of restricted forms such as paths [9] and q-grams [10]. In other direction, Kashima and Koyanagi [8] presented an efficient dynamic



i	Subtree S_i	Signature B_i	Freq. f_i
1	$S_1 = \{1, 2, 14, 15, 16\}$	()()()()	1
2	$S_2 = \{1, 2, 14, 16, 17\}$	()()(())	6
3	$S_3 = \{2, 3, 8, 9, 13\}$	()(()())	5
4	$S_4 = \{2, 3, 8, 9, 10\}$	()((()))	10
5	$S_5 = \{1, 2, 3, 14, 15\}$	(())()()	6
6	$S_6 = \{2, 3, 5, 8, 13\}$	(())(())	8
7	$S_7 = \{2, 3, 4, 5, 8\}$	((())())	7
8	$S_8 = \{8, 9, 10, 11, 12\}$	((()()()))	1
9	$S_9 = \{1, 2, 3, 8, 9\}$	((()(())))	5
10	$S_{10} = \{1, 2, 3, 16, 17\}$	((()))()	13
11	$S_{11} = \{1, 2, 3, 4, 8\}$	((())())	5
12	$S_{12} = \{1, 2, 3, 16, 18\}$	((()()))	6
13	$S_{13} = \{1, 2, 8, 9, 11\}$	(((())))	5

Fig. 1. An ordered tree T_1, the corresponding k-subtrees, and the feature vector $\phi^{\mathcal{S}_k}(T)$ for T_1, where $k = 5$. The set of nodes surrounded by a dashed circle indicates the subtree $S_4 = \{2, 3, 8, 9, 10\}$, which has occurrences $S_4^1 = \{1, 15, 16, 18, 19\}$ and $S_4^2 = \{2, 3, 8, 9, 12\}$ in T isomorphic to itself.

programming algorithm to compute the *ordered subtree kernel* of two ordered trees T_1 and T_2 in $O(|T_1|\times|T_2|)$ time using general ordered subtrees of *unbounded size*. Besides the efficient DP algorithms for ordered trees of unbounded size [8], some authors pointed out the usefulness of the semi-structured features using bounded sized substructures [12]. However, it does not seem easy to extend the DP algorithm [8] for *bounded sized* ordered trees.

The *enumeration-based approach* [2,12,16] is another way of computing such a tree distance based on a general class of substructures, which is a simple and flexible approach that one uses a *pattern enumeration algorithm* [2, 17, 19], to find all substructures contained in an input data to construct a feature vector, and then to solve a variety of tasks for information retrieval, data mining, and machine learning using similarity measure obtained from the constructed feature vectors. One problem in this approach is the high computational complexity of enumerating all small substructures. Hence, our goal is to devise efficient algorithms for frequency-based similarity by employing the recent development of efficient enumeration and mining algorithms for semi-structured data [2,17,19].

In this paper, we study efficient computation of tree similarity between two ordered trees using as features the class of *bounded sized ordered subtrees in unrestricted shape*. We present two new efficient algorithms for enumerating the *compressed bit-signatures* of all ordered k-subtrees in an input ordered tree using bit-parallel speed-up technique. The first one runs in $O(k)$ time per signature, and the second one runs in constant time per signature using $O(n)$ time preprocessing [1, 7]. From these compressed signatures, we can quickly compute the tree similarity between two ordered trees. We note that these algorithm are the first *compressed pattern enumeration algorithms* [18] for a subclass of trees and graphs. They directly enumerate the compressed representation of all substruc-

tures by incrementally constructing their compressed form on-the-fly without encoding/decoding.

Finally, we ran the experiments on real and artificial datasets to evaluate the proposed methods. We observed that the improved versions of the algorithms equipped with our bit-parallel speed-up technique showed around 36% speed-up from the algorithm without speed-up.

Organization of this paper: In Sec.2, we give the definitions of the tree similarities with ordered k-subtrees. In Sec. 3, we present the first algorithm using at most k-subtrees, and in Sec.4, the second algorithm using exactly k-subtrees. In Sec.5, we ran experiments to evaluate these algorithms, and in Sec.6, we conclude this paper.

2 Preliminaries

In this section, we give basic definitions and notation on the tree similarities over ordered trees. We denote by $|A|$ the number of elements in a set A. For every integers $i \leq j$, we denote by $[i, j] = \{i, i+1, \ldots, j\}$.

Trees and k-subtrees: An *ordered tree* is a rooted tree $T = (V, E, r)$ with a node set $V = V(T) = \{1, \ldots, n\}$, an edge set $E = E(T)$, and the root $r = \mathrm{root}(T)$ such that there exists a fixed ordering among children $Ch(u)$ of each internal node $u \in T$. The *size* of T is $|T| = |V|$. We assume the standard definitions of the parent, children, leaves, and paths [5]. We denote by $pa(v)$ the parent of node v, $Ch(v)$ the set of all children of v, and $Lv(T) \subseteq V(T)$ the set of all leaves of T. The *border set* is the set $Bd(S) = \{\, y \in Ch(x) \,|\, x \in S, y \notin S \,\}$, that is, the set of all nodes that are not contained in S, but are children of some nodes in S. We denote by $\mathcal{T}$ the countable set of all ordered trees

A *subtree with size k* of T, or simply a *k-subtree*, is any connected subgraph $T[S] = (S, E(S), \mathrm{root}(S))$ of T induced in a subset $S \subseteq V(T)$ consisting of exactly k nodes of T. Since such a subtree can be completely specified by a connected node set S, we identify any subset $S \subseteq V(T)$ with the subtree $T[S]$ if it is clear from the context. We denote by $\mathcal{S}_k(T)$ the set of all distinct k-subtrees appearing in T modulo isomorphism. A k-subtree $S \in \mathcal{S}_k(T)$ *appears in* T if there is some subset $U \subseteq V(T)$ such that $T[U]$ is isomorphic to $T[S]$. We assume that the nodes are ordered as $v_1 \leq \cdots \leq v_n$ by the preorder traversal of T [5].

Bit signatures: As the succinct representation of ordered trees, the *balanced parentheses representation* (BP) [1] of an ordered tree T of k nodes is a bit sequence $BP(T) = b_{2k-1} \cdots b_0$ defined by the depth-first traversal of T starting from $\mathrm{root}(T)$ the left and right parentheses, "(" = 0 and ")" = 1, when it visits a node at the first and last times, respectively. We call $BP(T)$ the bit signature of T. For example, the BP of a tree $S_3 = \{2, 3, 8, 9, 13\}$ of size 5 is "(()(()()))".

For each node $v \in T$, $lpos(v)$ and $rpos(v) \in [0, 2k-1]$ denotes the bit positions for the left and right parentheses in $BP(S)$ corresponding to v, respectively. For any subset $R \subseteq S$, $RPOS(R)$ denotes the bit-vector $X \in \{0,1\}^{2k}$ such that, for every $v \in S$, $X[rpos(v)] = 1$ iff $v \in R$. The *compressed subtree enumeration* on T is the task of enumerating all subtrees in T in the form of bit signature.

Tree similarity: In this subsection, we give the tree similarity for ordered trees [8,11]. Let T be an input ordered tree and $\mathcal{S} = \{S_1, \ldots, S_M\} \subseteq \mathcal{T}$, $M \geq 1$, be a class of possible subtrees in T. Each elements of $\mathcal{S}$ are called *a subtree-feature*. The *subtree-feature vector* of T based on $\mathcal{S}$ is the vector $\phi^{\mathcal{S}}(T)$ of the number of counts that subtrees of $\mathcal{S}$ appear in T. Below, we will omit the superscript $\mathcal{S}$ if it is clear from context. Formally, the *subtree-feature vector* $\phi(T)$ for T based on $\mathcal{S}$ is defined by

$$\phi(T) = (f_1(T), \ldots, f_M(T)) \in \mathbb{N}^M, \tag{1}$$

where for every $i \in [1, M]$, $f_i(T) = Occ(S_i, T)$ is the number of all occurrences of the i-th subtree S_i in T. Then, we consider the the tree similarity $Sim(T, T')$ between T and T' in one of the following forms [14]:

– L_p-tree distance for every $p = 1, 2, \ldots$:

$$Sim(T, T') = ||\phi(T) - \phi(T')||_p = \left(\sum_i |f_i(T) - f_i(T')|^p \right)^{1/p}. \tag{2}$$

– Cosine-tree distance:

$$Sim(T, T') = Cosine(\phi(T), \phi(T')) = \frac{\sum_i f_i(T) \times f_i(T')}{(\sum_i f_i(T))^{\frac{1}{2}} \times (\sum_i f_i(T'))^{\frac{1}{2}}} \tag{3}$$

Once the feature vectors $\phi(T)$ and $\phi(T')$ are computed by a subtree enumeration algorithm for class $\mathcal{S}$, $Sim(T, T')$ can be computed in linear time in the length of the vectors. In Fig.1, we show examples of an ordered tree T_1, the corresponding k-subtrees, and the feature vector $\phi^{\mathcal{S}_k}(T_1)$ for T_1 based on all k-subtrees for $k = 5$.

Model of Computation: We assume the Word RAM [1,7] with standard bit-wise Boolean and arithmetic operations ("+" and "$*$") on $w = \Theta(\log n)$ bits registers including *bitwise* AND "&", *bitwise* OR "|", *bitwise* NOT "$\sim$", *left shift* "$\ll$", and *right shift* "$\gg$", where n is an input size. We write a constant variable-length bit-vector as "1011". In this paper, a bit vector of length L is written as $X = b_{L-1} \cdots b_0 \in \{0,1\}^L$, where the MSB b_{L-1} and LSB b_0 come in this order. For every i, we define the length and i-th bit of B by $|B| = L$ and $B[i] = b_i$, respectively.

An *enumeration algorithm* $\mathcal{A}$ receives an instance of size n and outputs all of m solutions without duplicates (See, e.g. [6]). For a polynomial $p(\cdot)$, $\mathcal{A}$ is of $O(f(n))$-delay using preprocessing $p(n)$ if the *delay*, which is the maximum computation time between two consecutive outputs, is bounded by $f(n)$ after preprocessing in $p(n)$ time.

3 Enumeration of at-most k-subtrees in a tree

In Sec.3 and Sec.4, we present algorithms for computing the subtree-feature vector of T based on efficient compressed subtree enumeration.

Algorithm 1 The algorithm EnumAtMost for computing the feature vector H for the bit signatures of all subtrees with at most k nodes in an input tree T

```
 1: procedure EnumAtMost(T, k)
 2:     H ← ∅;                          // A hash table H representing a feature vector
 3:     for r ← 1, ..., n do
 4:         Initialize bit-vectors B and X;
 5:         RecAtMost({r}, T, k);
 6:     return H;
 7: procedure RecAtMost(S, T, k)
 8:     If H[S] is defined, then H[S] ← H[S] + 1 else H[S] ← 1;
 9:     Output BS(S);
10:     If |S| = k, then return;
11:     for each extension point v on RMB(S) do
12:         Attach a new leaf u to v as the youngest child;
13:         Let S ∪ {u} be the resulting subtree;
14:         RecAtMost(S ∪ {u}, T, k);
```

The first algorithm that we present in this section is the compressed enumeration version of the constant delay enumeration algorithm for uncompressed subtrees of at most size k in an ordered tree by [2, 15, 19].

The outline of the enumeration algorithm: In Algorithm 1, we show our algorithm EnumAtMost for at most k-subtrees and its subprocedure RecAtMost. This is a simple backtrack algorithm, which starts from a singleton tree as 0-subtree, and recursively expands the current $(i-1)$-subtree by attaching a new node u to some node $v = pa(u)$ on the current rightmost branch $RMB(S)$ to generate a new i-subtrees, until its size i becomes k (See [2]). Then, we say that some node $v \in T \setminus S$ *can be added to* S as a child of a node $u = pa(v)$ on $RMB(S)$ if v is the younger than any child of u contained in S. Such a parent node v on $RMB(S)$ is called the *extension point* and u is called the associated new child. If there is no such a node u, then the algorithm backtracks to the parent subtree. The *extension point set* is the set $XP(S) \subseteq RMB(S)$ of all extension points of S. The next lemma gives the characterization of $XP(S)$.

Lemma 1. *For any $u \in RMB(S)$, $u \in XP(S)$ iff there exists some $v \in T \setminus S$ such that (i) $v > \max(S)$ and (ii) v is younger than the youngest child of u in S.*

Example 1. For the 5-subtree $S_3 = \{2, 3, 8, 9, 10\}$ in Fig.1, $Lv(S_4) = \{3, 10\}$ and $Bd(S_4) = \{4, 5, 11, 12, 13\}$, $RMB(S_4) = \{2, 8, 9, 10\}$, and $XP(S_4) = \{8, 9\}$.

We will show how to incrementally maintain the extension set $XP(S)$ by growing S. For a singleton tree S consisting with the root $r = root(S)$ only, if r has a child in T then $XP(S) = \{r\}$, and otherwise $XP(S) = \emptyset$. For a subtree with more than one nodes, we have the next lemma.

Lemma 2. *Let S be any k-subtree of T with $k \geq 2$. Suppose that $k \geq 2$ and a k-subtree $R = S \cup \{v\}$ is obtained from a $(k-1)$-subtree S by attaching a new*

child v to its extension point $u = pa(v) \in XP(S)$. Then, $XP(R)$ is the set of vertices that satisfies the following (a)–(c):

(a) For the parent, $pa(v) \in XP(R)$ iff v has a properly younger sibling in T.
(b) For the child, $v \in XP(R)$ iff v has some child in T.
(c) For any old extension point $x \in XP(S)$ other than $pa(v)$, $x \in XP(R)$ iff x is an ancestor of $pa(v)$.

In condition (c) of the above lemma, we note that any extension point x is either an ancestor or a descendant of $pa(v)$ in $XP(S)$ since $XP(S)$ is a subset of $RMB(S)$, a branch in S.

Fast update of bit signatures: In our bit-parallel implementation of EnumAtMost, for each k-subtree, we maintain two bit-vectors of length $2k$, $B = BP(S)$ and $X = RPOS(XP(S))$, that represent the current subtree S and its extension point set $XP(S)$, respectively. For simplicity, we first describe the algorithm with bit-vectors whose length is no larger than the word length $w = \Theta(\log n)$. We efficiently update the bit-vectors B and X as follows.

Let $i \geq j$ and $ONE^{\ell}_{i:j} = 0^{i-1}1^{j-1+1}0^{\ell-j} \in \{0,1\}^{\ell}$ be the bit-mask of length ℓ whose i to j bits are filled with 1 bits and the other bits are filled with 0 bits, which can be computed by shift and subtraction in constant time for $i, j = O(\log n)$.

First, we initialize the bit-vectors B and X for the sets $S = \{r\}$ and $XP(S)$ by the following code: $B \leftarrow$ "01"; **if** r has a child on T **then** $X \leftarrow$ "01" **else** $X \leftarrow$ "00";

Next, the following code correctly updates the bit-vectors B and X when we compute the extension point set $XP(S \cup \{v\})$ for the new subtree $S \cup \{v\}$ from $XP(S)$ for the old one S, where $q = rpos(v)$ and $\ell = len(B)$:

- B is updated as follows.
  $B \leftarrow (B \ \& \ ONE^{\ell}_{\ell-1:q+1}) \ll 2 \ \mid \ (\text{"01"} \ll q) \ \mid \ (B \ \& \ ONE^{\ell}_{q:0})$;
- X is updated as follows.
  $X \leftarrow \{ (\text{"1"} \ll q-1)$ **if** v has a properly younger sibling $\}$ // a parent
  $\mid \ \{ (\text{"1"} \ll q)$ **if** v has some child $\}$; // a child
  $\mid \ (X \ \& \ ONE^{\ell}_{q-1:0}) \gg 2$ // others

From Lemma 2, we have the following lemma.

Lemma 3 (Update in the small subtree case). If $2k \leq w$, the above codes correctly updates the bit-vectors B and X in constant time using $O(1)$ words.

Lemma 4 (Update in the large subtree case). If $k = O(2^w)$, the bit-vectors B and X can be correctly updated in constant time using $O(k/w)$ words.

Proof. Proof sketch: We represent bit-vectors B and X as a doubly linked list of $b = O(\log n)$-bits blocks, each of which are maintained to store consecutive bits of length $\lceil b/2 \rceil < \ell \leq b$ (bits) similarly to [7]. In the update of B and X, we need to update only constant number of blocks, and thus, takes constant time. □

Theorem 1 (Compressed enumeration of at most k-subtrees). *For every $k \geq 1$, Algorithm 1 enumerates all compressed representations of the at most k-subtrees appearing in an ordered tree T in $O(1)$ time per solution, generated on the bit-vector $B \in \{0,1\}^{2k}$, using $O(k/w)$ words of space in addition to the space for an enumeration algorithm.*

From the above theorem, we observe that for every $k \geq 1$, all compressed representations of exact k-subtrees in T can be enumerated in $O(k)$ time per compressed representation using the same amount of space as above.

4 Enumeration of exact k-subtrees in a tree

The second algorithm that we present in this section is the compressed enumeration version of the constant delay enumeration algorithm for uncompressed k-subtrees in an ordered tree by Wasa *et al.* [17].

The outline of the algorithm: The basic idea of Wasa *et al.'s* algorithm is as follows: Given a k-subtree S in an input tree T, we can obtain the other k-subtree S' from S by deleting one node from S and adding one node to S.

By repeating this process recursively using backtracking, for each node r in T, starting from the lexicographically least k-subtree with r as its root, we can enumerate all k-subtrees with root r appearing in T by recursively transforming the current k-subtree by the above process. This algorithm runs in $O(1)$ time per k-subtree by maintaining the node lists $DL(S) \subseteq Lv(S)$ and $AL(S) \subseteq Bd(S)$ to delete and to add, respectively. In Algorithm 2, we show our algorithm EnumExact and its subprocedure RecExact.

Fast update of bit signatures: In the implementation with bit-operations, we use three bit-vectors B, L, and $A \in \{0,1\}^*$, where B is the *BP-vector* as defined in the previous section, L is the *leaf-vector* representing the set of leaves to delete, and A is the *add-vector* representing the set of nodes to add.

In this section, we give the efficient method for updating bit-vectors using bit parallel technique. A node v is an *exact extension point* in S if one of children of v can be attached to S. We define the sets $AL(S)$ and $DL(S)$ of nodes to add and to delete, and the set $EXP(S)$ of exact extension points by

$$DL(S) = \{\, x \in Lv(S) \mid x < minbord(S) \,\}. \tag{4}$$

$$AL(S) = \{\, x \in Bd(S) \mid x > maxleaf(S) \,\}. \tag{5}$$

$$EXP(S) = \{\, x \in S \mid x = pa(v), v \in AL(S) \,\}. \tag{6}$$

We give the following recurrence relation for $Lv(S)$, $Bd(S)$, and $EXP(S)$. In this subsection, $\leq$ denotes the DFS-ordering on $\mathcal{T}$.

Lemma 5. *Then, set is defined for any subset S.*

(a) If $S = \mathcal{I}_k$ is an initial k-subtree rooted at u, then $AL(\mathcal{I}_k)$ is the set $XP(\mathcal{I}_k)$ of all extension points, and $DL(\mathcal{I}_k)$ is the set $Lv(\mathcal{I}_k)$ of all leaves.

Algorithm 2 The algorithm EnumExact for computing the feature vector H for the bit signatures of all subtrees with exactly k nodes in an input tree T based on constant delay enumeration

```
 1: procedure EnumExact(T, k)
 2:     H ← ∅;                          // A hash table H representing a feature vector
 3:     Number the nodes of T by the DFS-numbering;
 4:     Compute the initial k-subtree I_k;
 5:     Initialize the related lists and pointers;
 6:     RecExact(I_k, B, L, X; T, k);
 7:     return H;

 8: procedure RecExact(S, B, L, X; T, k)
 9:     Output BS(S);   p ← MSB(L);
10:     for each ℓ ∈ DelList(S) do
11:         for each β ∈ AddList(S) such that β ∉ Ch(max(Lv(S)))  do
12:             S ← Child_1(S, ℓ, β) by updating the related lists and pointers;
13:             Update bit sequences B, L, X;
14:             RecExact(S, B, L, X; T, k);
15:             S ← P^1(S) by restoring the related lists and pointers;
16:             Restore bit sequences B, L, X;
17:             Modify X;
18:         Proceeds p;                          // to the next leaf position in L
19:     if S is a k-pre-serial tree then
20:         S ← Child_2(S) by updating the related lists and pointers;
21:         Update bit sequences sig(S), A, L;
22:         RecExact(S, B, L, X; T, k);
23:         S ← P^2(S) by restoring the related lists and pointers;
24:         Restore bit sequences sig(S), A, L;
```

(b) Let S be any k-subtree, $v \in AL(S)$, and $u \in DL(S)$ such that v is not a child of u. For any node x, the following conditions hold:

(i) $S' = (S \setminus \{u\}) \cup \{v\}$.

(ii) $Lv(S') = \{\, x \in Lv(S) \mid x \neq u \,\} \cup \{\, x \in T \setminus Lv(S) \mid Ch(x) \cap S = \{u\} \,\}$.

(iv) $DL(S') = \{\, x \in DL(S) \mid x < u \,\} \cup \{\, x \in T \setminus DL(S) \mid Ch(x) \cap S = \{u\} \,\}$.

(vi) $AL(S') = \{\, x \in AL(S) \mid x > v \,\} \cup \{\, x \in T \setminus AL(S) \mid x \notin Lv(T) \,\}$.

During the enumeration, the algorithm explicitly maintains the lists $Lv(S)$, $Bd(S)$, and $EXP(S)$. Using these lists and the pointers to the maximum leaf $maxleaf(S)$ and to the minimum border node $minbord(S)$, the algorithm implicitly represents the lists $DL(S)$ and $AL(S)$.

Next, we consider the generation of children of type I from Line 10 to Line 18 in Algorithm 2, and give the bit-parallel implementation of the update procedure for bit-vectors B, L, X, and pointers p and q to them, while the lists $Lv(S)$, $Bd(S)$ and $EXP(S)$ are maintained by the algorithm. During the enumeration, we maintain B, L, and X such that $B = BP(S)$, $L[rpos(v)] = 1$ iff $v \in Lv(S)$, and $X[rpos(v)] = 1$ iff $v \in EXP(S)$ for every $v \in T$. For initialization, we set

$B = BP(\mathcal{I}_k)$, $L = RPOS(Lv(\mathcal{I}_k))$, and $X = RPOS(EXP(\mathcal{I}_k))$ in $O(k)$ time by traversing the initial k-subtree $\mathcal{I}_k$).

Definition 1 (Update for children of type I). Suppose that we generate a child k-subtree $S' = (S \setminus \{u\}) \cup \{v\}$ of type I from the parent S. Then, we update the bit-vectors B, L and X as follows, where $p = rpos(u)$ and $\ell = len(B)$:

- The right position $q = rpos(v)$ can be computed from the bit-vector X using MSB by the following code: $q \leftarrow \texttt{MSB}(X)$;
- B is updated by deleting the two bits "01" from right position p for node u, and inserting the two bits "01" for node v at right position $q-1$.
  $B \leftarrow (B \,\&\, ONE^{\ell}_{\ell-1:p+2}) \mid (B \,\&\, ONE^{\ell}_{p-1:q+1}) \gg 2$;
  $B \leftarrow B \mid (\text{"01"} \ll q) \mid (B \,\&\, ONE^{\ell}_{q:0})$;
- L is updated similarly to B. In addition, the two bits surrounding the delete position for u are overwritten with "01":
  $L \leftarrow (L \,\&\, ONE^{\ell}_{\ell-1:p+3}) \mid (L \,\&\, ONE^{\ell}_{p-1:q+1}) \ll 2 \mid (\text{"01"} \ll q)$
  $\mid (L \,\&\, ONE^{\ell}_{q-1:0}) \mid \{ (\text{"01"} \ll p+1)$ **if** $Ch(pa(u)) = \{u\}$ $\}$;
- X is updated similarly to B. In addition, the two bits surrounding the delete position for u are overwritten with "01":
  $X \leftarrow (X \,\&\, ONE^{\ell}_{q-1:0}) \mid \{ (\text{"1"} \ll q)$ **if** v has a child $\}$
  $\mid (\text{"1"} \ll q-1)$ **if** ($\exists$younger sibling r of v) $r \notin S$;
  Moreover, after the for-loop at line 17, we update X by deleting the extension point at the highest position one by one:
  $X \leftarrow X \,\&\, (\sim(\text{"1"} \ll (\texttt{MSB}(X) - 1)))$ **if** v has no younger sibling in T;
- We proceed the pointer $p = rpos(u)$ at line 18 by: $p \leftarrow \texttt{MSB}(L \,\&\, ONE^{\ell}_{1:p-1})$;

Next, the code from Line 19 to Line 24 generates the children of type II updating the bit-vectors B, L, and X.

Definition 2 (Update for children of type II). Suppose that we generate a child k-subtree $S' = (S \setminus \{u\}) \cup \{v\}$ of type II from the parent S. Then, we update the bit-vectors B, L and X as follows:

- The right position q can be computed $q \leftarrow \texttt{MSB}(X)$;
- B is updated by deleting the most left bit of B and the most right bit of B, and inserting the two bits "01" for node v position q.
  $B \leftarrow (B \,\&\, ONE^{2k}_{2k-2:q+1}) \ll 1 \mid (\text{"01"} \ll q-1) \mid (B \,\&\, ONE^{2k}_{q:1}) \gg 1$;
- L is updated similarly to B. In addition, the right position bit of the inserting node v, is overwritten with "0".
  $L \leftarrow (L \,\&\, ONE^{2k}_{2k-2:q+1}) \ll 1 \mid (\text{"01"} \ll q-1) \mid (L \,\&\, ONE^{2k}_{q-1:1}) \gg 1$;
- X is updated by overwriting bits in the left of q with "0". In addition, two bits are overwritten with "1" if corresponding nodes satisfy some conditions.
  $X \leftarrow (X \,\&\, ONE^{2k}_{q-1:1}) \gg 1 \mid \{ (\text{"1"} \ll q-1)$ **if** v has a child; $\}$
  $\mid (\text{"1"} \ll q-2)$ **if** ($\exists$younger sibling r of v) $r \notin S$;

In the small tree case that $2k \leq w$, it follows from Lemma 5 that the above procedure correctly updates the data structure in constant time per iteration using $O(1)$ words. In the large tree case that $k = O(2^w)$, a similar discussion to Lemma 4 shows that the procedure also run in constant time using $O(k/w)$ words. Therefore, we have the following theorem.

Theorem 2 (Compressed enumeration of exact k-subtrees). *Let T be an input tree and k be a positive integer. The algorithm* EnumExact *in Algorithm 2 enumerates all compressed representations of the exact k-subtrees appearing in T in $O(1)$ time per compressed representation, generated on the bit-vector $B \in \{0,1\}^{2k}$, using $O(k/w)$ words of space in addition to the space for an enumeration algorithm.*

From the above theorem, we obtained a constant-delay algorithm for compressed enumeration for exact k-subtrees, which improves on the $O(k)$-delay algorithm in Sec. 3 by a factor of $O(k)$.

5 Experiments

In the experiments, we compared the running time of the algorithms in Sec. 3 and Sec. 4 on artificial and real datasets. We implemented in C++ the algorithms EnumAtMost in Sec.3 and EnumExact in Sec.4, denoted by Atmost(α) and Exact(α), respectively, where α indicates the types of algorithms as follows:

- "Enum" enumerates subtrees without printing them.
- "Naive" is the original algorithm that first enumerates a subtree and then computes its bit signature.
- "Fast" is the modified algorithm that directly enumerates the bit signature of a subtree with bit-parallel signature maintenance.

The algorithms were complied by g++ 4.2.1 and were run on a PC (CPU Intel® Xeon(R) 3.6GHz, 34GB RAM) operating on Ubuntu OS 13.04.

Comparison of algorithms on real data: As input, we use a phylogenetic tree of influenza virus of $n = 4240$ nodes, which was constructed from virus data in NCBI Influenza Virus Resource[4] by neighbor-joining method. In Fig.2, we show the running time of algorithms for computing the feature vector $\phi(T)$ of an input tree T varying the size of subtrees for $k = 15$ to 19. From this figure, for each of Exact and Atmost, the fast version (Fast) was faster than the naive version (Naive). For example, the speedup ratio for $k = 19$ were 36% for Exact, and 22% for Atmost. It depends on the type of update α which is faster between Exact and Atmost. In the case of Exact, the overhead of computing bit signatures over enumeration only (Enum) are 6 times for Fast and 9.5 times for Naive.

Comparison of algorithms on artificial data: We used a artificial tree with size $n = 35$, which has depth one and consists of a root node and 34 leaves.

[4] http://www.ncbi.nlm.nih.gov/genomes/FLU/

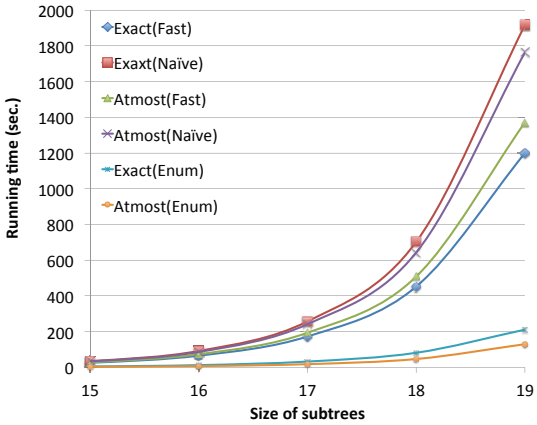


Fig. 2. The running time against the subtree size k on the real phylogenetic tree with $n = 4240$ nodes.

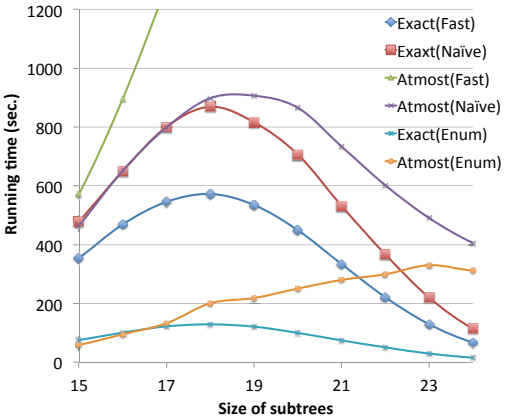


Fig. 3. The running time against the subtree size k on the artificial tree with $n = 35$ nodes and depth one.

Fig.3 shows the result of experiment. The fastest algorithm was Exact(Fast), which was 34% faster than Exact(Naive) for $k = 18$.

Computing the gram matrix of a set of trees: To evaluate the usefulness of our algorithms in the context of tree mining [8,12], we applied Exact(Fast) to similarly matrix computation [10,11]. We computed $M = 70,532$ dependency trees, one tree per one Japanese sentence, in total size $1,140,098$ nodes and 3.8 MB from a Japanese newspaper corpus[5] in 44.9 MB by CaboCha.[6] Their average and standard deviation sizes are 16.16 and 12.65 (nodes). Applying Exact(Fast) to this dataset with $k = 4$, we computed the $M \times M$-similarly matrix for 4-subtrees using cosine-tree distance in $1,276.12$ seconds, where only 0.1% (1.61 seconds) of the time was spent for computing feature vectors and 99.9% for matrix computation.

Summary of experimental results: Overall, the proposed method (Fast) achieved around 22% to 30% speedup over the naive method (Naive). From the last experiment, the proposed method seems to have reasonable performance for data mining from middle size datasets.

6 Conclusion

In this paper, we studied the tree similarity based on bounded-sized ordered subtrees using fast compressed k-subtree enumeration. We presented two speed up techniques for bit signature generation based on bit-parallel approach. It is an interesting future problem to extend this work for various types of subtrees will be another future research problem including unordered subtrees.

[5] http://www.ndk.co.jp/yomiuri/e_yomiuri/e_index.html
[6] http://code.google.com/p/cabocha/

Acknowledgements.

The authors would like thank anonymous reviewers for their comments which improved the correctness and the presentation of this paper very much, and thank Shin-ichi Nakano, Kunihiko Sadakane, and Tetsuji Kuboyama for helpful discussions and comments. This research was partly supported by MEXT Grant-in-Aid for Scientific Research (A), 24240021, FY2012—2015, and Grant-in-Aid for JSPS Fellows (25·1149).

References

1. D. Arroyuelo, R. Cánovas, G. Navarro, and K. Sadakane. Succinct trees in practice. In *Proc. ALENEX 2010*, pages 84–97, 2010.
2. T. Asai, K. Abe, S. Kawasoe, H. Arimura, H. Sakamoto, and S. Arikawa. Efficient substructure discovery from large semi-structured data. In *Proc. SDM'02*, 2002.
3. H. Chim and X. Deng. A new suffix tree similarity measure for document clustering. In *Proc. WWW '07*, pages 121–130, 2007.
4. M. Collins and N. Duffy. Convolution kernels for natural language. In *Proc. of Advances in Neural Information Processing Systems*, NIPS, pages 625–632, 2001.
5. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 2nd edition, 2001.
6. L. A. Goldberg. Polynomial space polynomial delay algorithms for listing families of graphs. In *Proc. ACM STOC'93*, pages 218–225. ACM, 1993.
7. J. Jansson, K. Sadakane, and W.-K. Sung. CRAM: Compressed random access memory. In *Proc. 39th ICALP, LNCS 6756*, pages 510–521, 2012.
8. H. Kashima and T. Koyanagi. Kernels for semi-structured data. In *Proc. 19th ICML*, ICML '02, pages 291–298. Morgan Kaufmann Publishers Inc., 2002.
9. D. Kimura, T. Kuboyama, T. Shibuya, and H. Kashima. A subpath kernel for rooted unordered trees. In *Proc. PAKDD'11, LNCS 6634*, pages 62–74, 2011.
10. T. Kuboyama, K. Hirata, and K. Aoki-Kinoshita. An efficient unordered tree kernel and its application to glycan classification. In *Proc. PAKDD'08*, pages 184–195, 2008.
11. T. Kuboyama, K. Hirata, H. Kashima, K. F. Aoki-Kinoshita, and H. Yasuda. A spectrum tree kernel. *Information and Media Technologies*, 22(2):292–299, 2007.
12. T. Kudo, E. Maeda, and Y. Matsumoto. An application of boosting to graph classification. In *Proc. NIPS'04*, 2004.
13. P. Lakkaraju, S. Gauch, and M. Speretta. Document similarity based on concept tree distance. In *Proc. 19th ACM HT'08*, pages 127–132, 2008.
14. C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to information retrieval*. Cambridge University Press, 2008.
15. S. Nakano. Efficient generation of plane trees. *IPL*, 84(3):167–172, 2002.
16. K. Tsuda and T. Kudo. Clustering graphs by weighted substructure mining. In *Proc. 23rd ICML*, pages 953–960. ACM, 2006.
17. K. Wasa, Y. Kaneta, T. Uno, and H. Arimura. Constant time enumeration of bounded-size subtrees in trees and its application. In *Proc. COCOON'12, LNCS 7434*, pages 347–359, 2012.
18. D. Xin, J. Han, X. Yan, and H. Cheng. Mining compressed frequent-pattern sets. In *Proc. VLDB'05*, pages 709–720, 2005.
19. M. J. Zaki. Efficiently mining frequent trees in a forest. In *Proc. KDD 2002*, pages 71–80, 2002.