𝕋ℂ𝕊-𝕋ℝ-𝔸-**10-43**

𝕋ℂ𝕊 Technical Report

Faster Bit-Parallel Algorithms for Unordered Pseudo-Tree Matching and Tree Homeomorphism

by

Yusaku Kaneta and Hiroki Arimura

Division of Computer Science

Report Series A

May 7, 2010

Hokkaido University
Graduate School of
Information Science and Technology

Email: arim@ist.hokudai.ac.jp

Phone: +81-011-706-7680
Fax: +81-011-706-7680

Faster Bit-Parallel Algorithms for Unordered Pseudo-Tree Matching and Tree Homeomorphism

Yusaku Kaneta and Hiroki Arimura

Hokkaido University, N14, W9, Sapporo 060-0814, Japan
{y-kaneta,arim}@ist.hokudai.ac.jp

Abstract. In this paper, we consider the unordered pseudo-tree matching problem, which is a problem of, given two unordered labeled trees P and T, finding all occurrences of P in T via such many-one embeddings that preserve node labels and parent-child relationship. This problem is closely related to tree pattern matching problem for XPath queries with child axis only. If $m > w$, we present an efficient algorithm that solves the problem in $O(nm \log w/w)$ time using $O(hm/w + m \log w/w)$ space and $O(m \log w)$ preprocessing on a unit-cost arithmetic RAM model with addition, where m is the number of nodes in P, n is the number of nodes in T, h is the height of T, and w is the word length. We also discuss a modification of our algorithm for the unordered tree homeomorphism problem, which corresponds to a tree pattern matching problem for XPath queries with descendant axis only.

1 Introduction

Tree matching is a fundamental problem in computer science, and it has a wide range of applications in XML/Web database, schema validation, information extraction, document analysis, image processing, and semi-structured data processing. In particular, tree matching and tree inclusion problems have attracted much attention and have been extensively studied [2, 6, 7, 12]. In this paper, we study non-standard version of the unordered tree matching and inclusion problems, called the unordered pseudo-tree matching problem (UPTM) [14] and the unordered tree homeomorphism problem (UTH) [4], respectively, where embedding mappings can be many-one.

As main results, we present an efficient algorithm that solves UPTM problem with the following complexities (Theorem 1):

- $O(nm \log w/w)$ time using $O(hm/w + m \log w/w)$ space and $O(m \log w)$ preprocessing if $m > w$. (the large pattern case)
- $O(n \log m)$ time using $O(h+\log m)$ space and $O(m \log m)$ preprocessing if $m \leq w$. (the small pattern case)

where m and n are the sizes of pattern tree P and text tree T, h is the height of T, and $w = \Theta(\log n)$ is the word length of RAM. We also show that UTH problem is solvable in the same time and space complexities as above (Theorem 2).

A key of our algorithm is a data structure for the small pattern case ($m \geq w$) based on bit-parallel computation of set operations, including *tree aggregation* that checks the branching of internal nodes. Developing bit-assignment technique based on separator trees, we improve the time complexity of the tree aggregation from $O(m)$ time and space to $O(\log m)$ time and space . Combining this result to dynamic programming tree matching algorithms and a module decomposition technique of [8], we have claimed results for both UPTM and UTH.

For the UPTM, our $O(nm \log w/w)$ time and $O(hm \log w/w)$ space algorithm improves the complexity of the previous $O(n{\cdot}r{\cdot}\mathit{leaves}(P)\mathit{depth}(P)/w) = O(nm^3/w)$ time and $O(n \cdot \mathit{leaves}(P)\mathit{depth}(P)/w) = O(nm^2/w)$ space algorithm[1] by Yamamoto and Takenouchi [14] in the worst case. For the UTH, our algorithm is one of the first bit-parallel algorithm for the problem and slightly faster than the previous $O(nm{\cdot}\mathit{depth}(P))$ time and $O(\mathit{depth}(T){\cdot}\mathit{branch}(T))$ space algorithm[1] by Gotz, Koch, and Martens [4]. These results for UPTM and UTH correspond to evaluation of fragments of Core XPath queries consisting with child axis only and with descendant axis only [4], respectively.

Tree matching problems with many-one embeddings have been studied in the area of FO and MSO logics over combinatorial structures such as strings, trees, and graphs as well as in database and Web systems [4]. These problems have less constraints than the other tree matching problems, but this does not necessarily mean that many-one matching problems are easiest among them. Hence, we hope that these result becomes steps towards development of efficient query mechanism for such data intensive applications.

Organization of this paper is as follows. Section 2 prepares definitions and notations. Section 3 shows a fast bit-parallel algorithm for UPTM. Section 4 gives an extension to UTH. In Section 5, we conclude.

2 Preliminaries

In this section, we give basic definitions and notation on our unordered tree matching problems according to [4, 6, 14]. For a set S, we denote by $|S|$ the cardinality of S. Let $\mathbf{N}_+ = \{1, 2, \ldots\}$. We define an *interval from i to j* by $[i..j] = \{i, i+1, \ldots, j\} \subseteq \mathbf{N}_+$, where $i \leq j$. We define the *smallest interval* including set $S \subseteq \mathbf{N}_+$ by $\mathit{Int}(S) = [\min S, \max S] \subseteq \mathbf{N}_+$. For an array $A = A[1] \cdots A[n]$ and $i \leq j$, we define $A[i..j] = A[i] \cdots A[j]$. For a binary relation $R \subseteq A^2$ on a set A, we denote by $R^+ \subseteq A^2$ the *transitive closure* of R.

2.1 Unordered trees

Let $\Sigma = \{a, b, a_1, a_2, \ldots\}$ be a finite alphabet of labels. In this paper, we will mainly consider *unordered trees*, which are the labeled, rooted trees, where the ordering among their siblings is irrelevant.

Let P be an unordered tree of m nodes whose labels are drawn from Σ. We denote by $V(P)$ the node set, by $E(P)$ the edge set, and by $\mathit{root}(P)$ the root of P. For each node x, $\mathit{label}_P(x) \in \Sigma$ denotes the label of x in P and $P(x)$ denotes the subtree of P rooted at x.

If $(x, y) \in E(P)$ then we say that x is a *parent* of y and y is a *child* of x. If there exists some downward path from x to y, i.e., $(x, y) \in E(P)^*$, then we say that x is an *ancestor* of y and y is a *descendant* of x and write $x \preceq y$. If $x \preceq y$ and $x \neq y$ then we say that x is a *proper ancestor* of y and y is a *proper descendant* of x. If both of $x \not\preceq y$ and $y \not\preceq x$ hold then x and y are *incomparable* each other and write

[1] In the results, $\mathit{leaves}(P)$, $\mathit{depth}(P)$, and $\mathit{branch}(P)$ is the number of leaves, the maximum depth, and the maximum branching in a tree P. The parameter $r = O(m)$ is the maximum number of the same label on paths in P.

$x \sharp y$. For nodes x and y in P, if $x \sharp y$ and x precedes y in the preorder travesal of P, then we say that x *precedes* y in P (or, x *is to the left of* y) and write $x \lhd y$. If $x \sharp y$ then either $x \lhd y$ or $y \lhd x$ holds.

For unordered tree P, we denote by $|P|$ and by $height(P)$ the number of nodes in P and the height of P. We denote the sets of all leaves and all internal nodes in P, respectively, by $internal(P)$ and $leaves(P)$. Clearly, $V(P) = internal(P) \uplus leaves(P)$. Let x be any node in P. The *arity* of x, denote by $\alpha(x) \geq 0$, is the number of children of x. For every $1 \leq i \leq \alpha(x)$, we denote the i-th child of node x by $x[i]$, and the list of the children of x by $children(x) = x[1] \cdots x[\alpha(x)]$.

2.2 Unordered tree matching problem

Let $P = P[1..m]$ be an unordered tree of size m, called a *pattern tree*, and $T = P[1..n]$ be an unordered tree of size n, called a *text tree*. In this subsection, we introduce the unordered pseudo-tree matching and unordered tree homeomorphism problems. For other variations of tree matching problems as in [2, 4, 6, 7, 12, 14], please consult Appendix A.1.

Definition 1 (conditions for tree matching and inclusion). For any (possibly many-one) mapping $\phi : V(P) \to V(T)$, we define the following conditions:

(E0) ϕ preserves node labels. That is, for every node $x \in V(P)$, $label_P(x) = label_T(\phi(x))$ holds.
(E1) ϕ preserves the parent-child relationship. That is, for every node $x, y \in V(P)$, $(x, y) \in E_P \Rightarrow (\phi(x), \phi(y)) \in E_T$ holds.
(E1') ϕ preserves the ancestor-descendant relationship. That is, for every node $x, y \in V(P)$, $(x, y) \in E_P \Rightarrow \phi(x) \prec \phi(y)$ holds.

Let $\mathcal{F}$ be a class of mappings. Then, a pattern P *maps to* a node $v \in V(T)$ in T w.r.t. class $\mathcal{F}$ if $\phi(root(P)) = v$ for some $\phi \in \mathcal{F}$. Then, the node v is called an *occurrence* of P in T w.r.t. class $\mathcal{F}$. Then, the tree pattern matching problem w.r.t. $\mathcal{F}$ ($\mathcal{F}$-matching problem) is the problem of, given a pattern tree P and a text tree T, finding all occurrences of P in T w.r.t. class $\mathcal{F}$.

An embedding from P to T is a possibly many-one mapping $\phi : V(P) \to V(T)$ with (E0). A *unordered pseudo-tree matching* (UPTM) [14] is a many-one version of unordered tree matching, i.e., an embedding ϕ with (E0) and (E1). A *unordered tree homeomorphism* (UTH) [4] is a many-one version of unordered tree inclusion, i.e., an embedding ϕ with (E0) and (E1'). We denote by $UPTM(P, T)$ and $UTH(P, T)$ the sets of all pseudo-tree matching and all tree homeomorphism from P to T. The *unordered pseudo-tree matching problem* (UPTM) and the *unordered tree homeomorphism problem* (UTH) are tree matching problem related to the above classes of mappings.

3 Faster Bit-parallel Algorithm for Unordered Pseudo Tree Matching

In this section, present efficient algorithm BP-MatchUPTM based on bit-parallel pattern matching method for the pseudo-tree matching problem. Let $P = P[1..m]$ be a pattern tree of size m and $T = T[1..n]$ be a text tree of size n.

```
algorithm MatchUPTM($P[1..m]$: a pattern tree, $T[1..n]$: a text tree):
Global Variables: $P$ and $T$;
Output: all occurrences of $P$ in $T$ w.r.t. unordered pseudo-tree matching (UPTM);
 1: VisitUPTM($root(T)$);

procedure VisitUPTM($v$: text node)
Return Value: $R = Emb^{P,T}(v)$;
 2: $S \leftarrow$ Constant($\emptyset$); {See Definition 2}
 3: for $i = 1, \dots \alpha(v)$ do
 4:    $S \leftarrow$ Union($S$, VisitUPTM($v[i]$));
 5: $R \leftarrow$ Constant($[1..m]$);
 6: $R \leftarrow$ LabelMatch$_P$($R, label_T(v)$); {See Definition 2}
 7: $R \leftarrow$ TreeAggr$_P$($R, S$); {See Definition 2}
 8: if Member($R, root(P)$) then {See Definition 2}
 9:    output "A match is found at a node $v$.";
10: return $R$;
```

Fig. 1. An algorithm for the unordered pseudo-tree matching problem.

3.1 Decomposition formula and a bottom-up algorithm

In Fig. 1, we show an algorithm MatchUPTM for Unordered Pseudo Tree Matching. Our matching algorithm computes, for every text node v in T, the set $Emb^{P,T}(v)$ of integers in $V(P) = [1..m]$, called the *embedding set*, defined by:

$$Emb^{P,T}(v) = \{\, x \in [1..m] \mid (\exists \phi)\ \phi \in UPTM(P(x), T) \land \phi(x) = v \,\}. \qquad (1)$$

Clearly, For every $x \in [1..m]$, $x \in Emb^{P,T}(v)$ if and only if the corresponding subtree $P(x)$ has an occurrence at the current node v. By definition, we see that P matches T at node v iff $root(P) \in Emb(P, v)$. Now, we have the next lemma, which is crucial to the correctness of our algorithm.

Lemma 1 (decomposition formula for UPTM). *For every $x \in V(P)$ and $v \in V(T)$, $x \in Emb^{P,T}(v)$ if and only if*

(i) $label_P(x) = label_T(v)$, and
(ii) $children(x) \subseteq \bigcup_{1 \le j \le \alpha(v)} Emb^{P,T}(v[j])$.

From Lemma 1 above, we show in Fig. 1 a bottom-up procedure VisitUPTM to compute $Emb^{P,T}(v)$ by using the post-order traversal of T. To describe the procedure VisitUPTM, we need the following operators.

Definition 2 (set manipulation operators). We define operators Constant, Union, Member, LabelMatch (label matching), and TreeAggr (tree aggregation) on subsets of $[1..m]$ as follows, where $R, S \subseteq [1..m]$, $x \in [1..m]$, and $\alpha \in \Sigma$:

- Constant(S) $\equiv$ S. This operation returns the set S itself.
- Union$_P$(S, R) $\equiv$ $S \cup R$. this returns the set-union of R and $S \subseteq [1..m]$.
- Member$_P$(R, x) $\equiv$ $S \cup R$. Given a set R and an element x, this operation returns "*yes*" if $x \in R$ and "*no*" otherwise.
- LabelMatch$_P$(R, α) $\equiv$ $\{\, k \in R \mid label_P(k) = \alpha \,\}$. Given any set R and label α, this operation returns the set elements in R satisfying (i) of Lemma 1.
- TreeAggr$_P$(R, S) $\equiv$ $\{\, k \in R \mid children_P(k) \subseteq S \,\}$. Given any sets R, S, this operation returns the set of elements in R satisfying (ii) of Lemma 1.

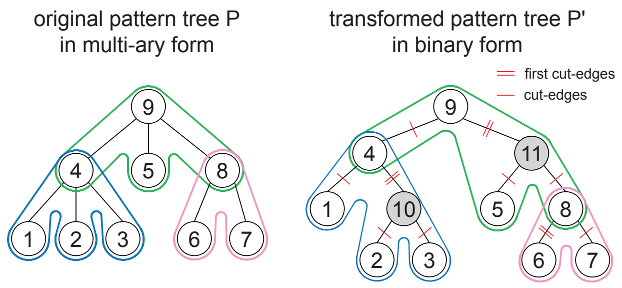

Fig. 2. An original pattern tree P and its binarization P', where white and shadowed circles indicate original (real) nodes and dummy (virtual) nodes, respectively. The number in each circle indicates the node id.

In the procedure VisitUPTM, we use the last two operators LabelMatch and TreeAggr to check (i) and (ii) of Lemma 1. Later, the above set operations will be implemented in bit-parallel manner in Sec. 3.2.

By representing sets R and $S \subseteq [1..m]$ in lists of integers, it is easy to see that these operators can be implemented to run in $O(m)$ time and space. Then, we have the following lemma.

Lemma 2. *For the unordered pseudo-tree matching problem, For every pattern tree P and a text tree T, the algorithm* MatchUPTM *in Fig. 1 correctly finds all occurrences of P in T. Moreover, the algorithm can be implemented to run in $O(mn)$ time and $O(hm)$ additional space, where m is the size of P, n and h are the size and the height of T, respectively.*

The algorithm MatchUPTM can run in streaming setting using a stack of length $O(hm)$, where T is given as an input stream consisting of a sequence of balanced open and close parentheses on alphabet $\Sigma \cup \{\, \bar{a} \mid a \in \Sigma \,\}$ as in XML databases [4, 10, 11].

3.2 Bit-parallel implementation: overview

In the following subsections, we give the bit-parallel version of the algorithm MatchUPTM, called BP-MatchUPTM, that runs in in $O(nm \log w/w)$ time and $O(hm/w + m \log w/w)$ space, where m is the size of pattern tree P, n and h are the size and the height of text tree T, and $w = \Theta(\log n)$ is the word length of underlying computer. Let us fix a pattern $P = P[1..m]$ of size m on a finite alphabet Σ. In what follows, we assume that $|\Sigma| = O(1)$.

In the bit-parallel implementation of MatchUPTM, we introduce a data structure $\mathcal{A}$ for representing a subset S of the universe $[1..m]$ that efficiently supports the collection of set manipulation operators in Definition 2 in Sec. 3.1.

In $\mathcal{A}$, we represent any subsets of $V(P)$ by bitmasks $X \in \{0,1\}^m$ with length m as m-bit integers from 0 to $2^m - 1$. To do this, we need an assignment $Bit : V(P) \to [1..m]$ of the unique bit-position $Bit(x)$ in the interval $[1..m]$ to each node x in P.

Basic set operations. Once the assignment Bit is given, for any node set $S \subseteq V(P)$, we extend this Bit by $BIT(S) = \{\, Bit(x) \mid x \in S \,\} \subseteq [1..m]$. At this moment, we leave Bit undefined and the appropriate definition for Bit will be given later in the next subsection. For any subset $X \subseteq [1..m]$, we define $NUM(X) \in \{0,1\}^m$ to be

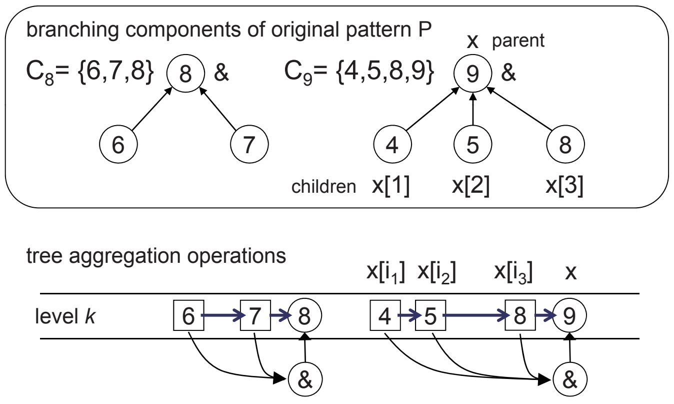

Fig. 3. Branching components of a pattern tree P and the corresponding tree aggregation operation in bit-parallel computation.

the bitmask for X. Among the set operators in Definition 2, the following operators are easy to implement.

Lemma 3. *Let $S, R \subseteq V(P)$ be any sets, and $X, Y \in \{0.1\}^m$ be the corresponding bitmasks, respectively. Then, the following codes correctly implements the operators. Moreover, all operations are executed in $O(1)$ time if $m \le w$.*

- *Preprocess:* $\mathsf{Constant}(S) \equiv NUM(BIT(S))$;
- *Runtime:* $\mathsf{Union}_P(X, Y) \equiv (X \mid Y)$;
- *Runtime:* $\mathsf{Member}_P(X, x) \equiv$ **if** $(X \cap BIT(\{x\})) > 0$ **then** 1 **else** 0;

Label matching operation. The label matching operation can be implemented using a set of character masks as in SHIFT-AND method for exact match [1, 13, 9] or Move operation for regular expression match [3]

Lemma 4. *The operator* LabelMatch *can be correctly implemented by the following codes, where $\{\, LAB[\alpha] \in \{0,1\}^m \mid \alpha \in \Sigma \,\}$ is a set of bitmasks for P. Moreover the operation can be executed in $O(1)$ time if $m \le w$.*

- *Preprocess: For each $\alpha \in \Sigma$, $LAB[\alpha] = |_{x \in V(P), label_P(x) = \alpha} NUM(Bit(x))$;*
- *Runtime:* $\mathsf{LabelMatch}_P(X, \alpha) \equiv (X \,\&\, LAB[\alpha])$;

Tree aggregation operation. Remaining task is to show how to efficiently implement TreeAggr operation in bit-parallel computation. For each node x in P, we define the *branching component* for x in P by the connected component $C_x = \{x\} \cup children(x)$ of P consisting of parent x and its children. If no confusion arises, we identify C_x and the induced depth-one subtree $P(C_x)$ rooted at x, called a *branching tree*. We denote by $\mathcal{C}_P = \{C_x \mid x \in internal(P)\}$ the set of all branching components of P.

For example, pattern P of Fig. 2 has three branching components $C_4 = \{1, 2, 3, 4\}$, $C_8 = \{4, 5, 8, 9\}$, and $C_9 = \{6, 7, 8\}$. The upper half of Fig. 3 shows C_8 and C_9 with their branching trees.

Then, the tree aggregation operation means gathering the values of children of x and then copying their conjunction to the value of parent x (See Fig. 3). We want to compute tree aggregation simultaneously for all internal node x in P. To do this,

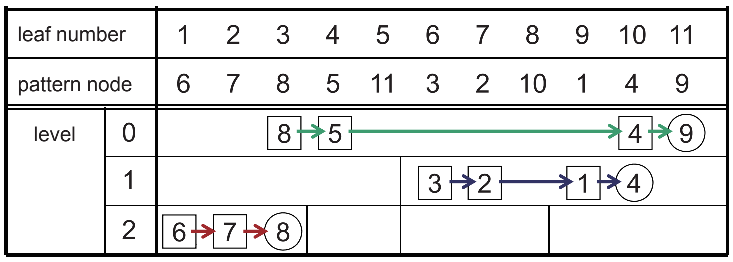

Fig. 4. Tree aggregation on interval $[1..m]$ based on a monotone bit-assignment Bit.

we require bit assignment $Bit : V(P) \to [1..m]$ and a decomposition of $\mathcal{C}$ with some properties described below.

First, to implement the tree aggregation operation in correct and efficient way, we require the assignment Bit to have the following properties:

Definition 3 (monotone bit assignment). A bit assignment mapping $Bit : V(P) \to [1..m]$ is said to be *monotone* w.r.t. the ancestor relation $\preceq$ for P if for any $x, y \in V(P)$, $(x \succ y) \Rightarrow (Bit(x) < Bit(y))$ holds.

See Fig. 4, for an example of monotone bit-assignment. Next, we give a decomposition of $\mathcal{C}_P$ as follows. For any component $C_x \in \mathcal{C}_P$, we denote by $I_x = Int(BIT(C_x))$ the *smallest interval* in $[1..m]$ containing all bit positions of C_x. Then, two components C_x and C_y $(x \neq y)$ *overlap* if $I_x \cap I_y \neq \emptyset$. A subset $\mathcal{D} \subseteq \mathcal{C}_P$ is said to be *overlap-free* if there are no pairs of components in $\mathcal{D}$ that overlap each other.

Definition 4 (overlap-free decomposition). A partition $\mathcal{C}_P = \mathcal{C}[1] + \cdots + \mathcal{C}[K]$ for some $k \geq 1$ is said to be an *overlap-free decomposition* of $\mathcal{C}_P$ w.r.t. Bit if for every $k = 1, \ldots, K$, the k-th subset $\mathcal{C}[k]$ is overlap-free w.r.t. Bit. Then, K is called the *height* and $\mathcal{C}[k]$ is called the *k-th layer* of the partition.

Suppose that there exists some monotone bit assignment Bit and some overlap-free decomposition $\mathcal{C}_P = \mathcal{C}[1] + \cdots + \mathcal{C}[K]$ of $\mathcal{C}_P$ for some Bit. Then, tree aggregation is implemented in bit-parallel way as follows.

Definition 5 (Preprocess). We first precompute the following bitmasks:

– $LEAF = |_{x \in leaves(P)} NUM(BIT(\{x\}))$.
– For every level $k = 1, \ldots, K$, and for each C_x in $\mathcal{C}[k]$, we define
  • $DST[k][x] = NUM(BIT(\{x\})))$. The position of parent x.
  • $SRC[k][x] = NUM(BIT(children(x)))$. The positions of $children(x)$.
  • $INT[k][x] = NUM(Int(BIT(C_x)))$. The interval for C_x.
  • $SEED[k][x] = NUM(\min BIT(C_x))$. The "seed" position.
– For every level $k = 1, \ldots, K$, and for each $Mask \in \{DST, SRC, INT, SEED\}$,
  • $Mask[k] = |_{C_x \in \mathcal{C}[k]} Mask[k][x]$.

Lemma 5 (Runtime). *Suppose that $\mathcal{C}_P = \mathcal{C}[1] + \cdots + \mathcal{C}[K]$ of $\mathcal{C}_P$ is an overlap-free decomposition with height $K \geq 1$ for a monotone bit assignment Bit w.r.t. $\preceq$. Then, the code in Fig. 5 correctly implements the tree aggregation operator for the component C_x. Moreover, this procedure runs in $O(K)$ time if $m \leq w$.*

Therefore, the remaining thing is how to find a good overlap-free decomposition $\mathcal{C}_P$ with small height as well as monotone bit-assignment Bit. We discuss this issue in the next subsection.

```
procedure TreeAggr_P(X, Y):
1: $Z \leftarrow \mathsf{Constant}(\emptyset)$;
2: For every level $k = 1, \ldots, K$ do:
3:     $BLK \leftarrow (Y \mathbin{\&} SRC[k]) \mid (INT[k] \mathbin{\&} (\sim (SRC[k] \mid DST[k])))$;
4:     $Z \leftarrow Z \mid ((BLK + SEED[k]) \mathbin{\&} DST[k])$;
5: $Z \leftarrow X \mathbin{\&} (Z \mid LEAF)$;
6: return $Z$;
```

Fig. 5. An algorithm for implementing TreeAggr operator

3.3 Construction of a monotone bit-assignment and an overlap-free decomposition based on separator trees

In this subsection, we show how to find both a monotone bit-assignment Bit and an overlap-free decomposition $\mathcal{C}_P$ with height $O(\log m)$. For this purpose, we use a data structure called a *separator tree*.

Binarization of P. Let P be a pattern tree of m nodes. We note that P is a multi-ary tree whose internal node may have arity $\alpha(x) > 2$. First, before constructing separator tree composition, we apply a standard transformation, called *binarization* to P for obtaining a binary version P' of P. The binarization transforms each branching component $C_x = \{x\} \cup \{x[1], \cdots, x[\alpha]\}$ with the root x and $\alpha(x)$ children into a new component C'_x a binary subtree with the same root and the same number of children by inserting $\alpha(x) - 2$ dummy internal nodes of out-degree two. In general, the resulting binary tree P' has size at most $2m$. In what follows, let $m' = O(m)$ be the size of P'.

Fig. 2 shows an example of the binarization P' of the original pattern P, where the component $C_4 = \{1, 2, 3, 4\}$ with root 4 is transformed into $C'_4 = \{1, 2, 3, 4, 10\}$ with the same root 4.

Construction of a separator tree for P. Secondly, we build a separator tree $\mathcal{M}$ from a binarization P' of pattern tree. A separator tree is a binary tree obtained from P' by iteratively removing edges in $E(P')$. The following well-known lemma is sufficient for our purpose:

Lemma 6 (Jordan [5]). *Let S be a binary tree. Then, there exists a node in S such that $|S(v)| \leq (2/3)|S|$ and $|S(\bar{v})| \leq (2/3)|S|$, where $S(v)$ is the subtree of S rooted at v and $S(\bar{v})$ is the tree obtained by pruning $S(v)$ from S.*

Suppose that each node w of $\mathcal{M}$ has the fields U_w for a subset of $V(P')$, and e_w for a cut-edge. Applying the above theorem recursively, we construct a separator tree $\mathcal{M}$ from P' as follows.

- We start with a new node as the root of $\mathcal{M}$. We associate $V(P')$ to the root by setting $U_{root(\mathcal{M})} = V(P')$. We visit $root(\mathcal{M})$, and repeat the following process.
- Suppose that $|U_w| = 1$. Then, the associated node set U_w is a singleton $\{x\}$, and the current node w is a leaf. We stop the process.
- Otherwise, $|U_w| > 1$. Then, we find an edge $e = (x, y) \in E(P')$ according to Lemma 6 so that removal of e splits the associated component U_w into two subcomponents U^1_w and U^0_w of almost equal sizes no more than $(2/3)|U_w|$, where U^1_w is the subcomponent containing $root(P')$ and U^0_w is the other subcomponent. Record the *cut-edge* e for w as e_w. Create the left and the right children, w_L and

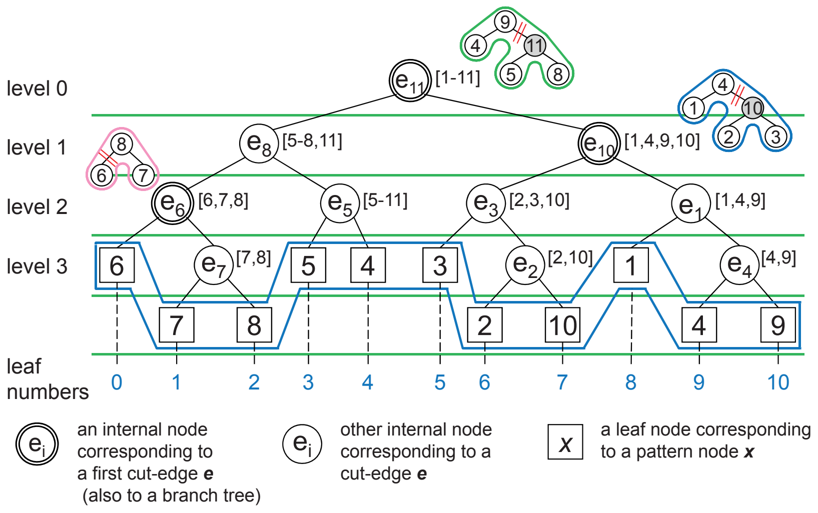
level 0
level 1
level 2
level 3
leaf numbers
e11 [1-11]
e8 [5-8,11]
e10 [1,4,9,10]
e6 [6,7,8]
e5 [5-11]
e3 [2,3,10]
e1 [1,4,9]
e7 [7,8]
e2 [2,10]
e4 [4,9]
0 1 2 3 4 5 6 7 8 9 10
an internal node corresponding to a first cut-edge e (also to a branch tree)
other internal node corresponding to a cut-edge e
a leaf node corresponding to a pattern node x

Fig. 6. A separator tree $\mathcal{M}$ for the binarization P' of a pattern tree. Circles and squares indicate internal nodes and leaf nodes in $\mathcal{M}$, respectively. At each node w, the associated set of numbers in a pair of brackets indicates the connected component associated to w. Each edge (x, y) is denoted as e_y indexed by its lower end point y.

w_R, and associate the component U_w^1 to w_R and U_w^0 to w_L. Then, recursively visit both of w_L and w_R.

Lemma 6 ensures that the height of $\mathcal{M}$ is $O(\log m)$ and the construction requires $O(m \log m)$ time. Furthermore, we can observe that (1) there exists a one-one correspondence between $internal(\mathcal{M})$ and $E(P')$, and (2) there exists a one-one correspondence between $leaves(\mathcal{M})$ and $V(P')$.

Now, we compute a bit assignment $Bit : V(P') \to [1..m]$ as follows. We order $leaves(\mathcal{M})$ from left to right. Then, we number all leaves in $leaves(\mathcal{M})$, which are original (real) nodes in P, from left to right consecutively from 1 to m (not m'). We just skip and unnumber dummy (virtual) nodes included in binarization. Then, we define bit-assignment $Bit : V(P) \to [1..m]$ such that for each node x in P, if x is the i-th leaf in this listing over $leaves(\mathcal{M})$, then $Bit(x) = i$. For the proof of the next lemma, see Appendix A.2.

Lemma 7. *The bit assignment Bit constructed above from $\mathcal{M}$ is monotone w.r.t. the ancestor relation $\preceq$ for P.*

By using Bit based on leaf numbering of $\mathcal{M}$, we associate an interval I_w to each node w in $\mathcal{M}$ by $I_w = Int(BIT(U_w))$. Then, we give a technical lemma.

Lemma 8. *For any nodes u, w in $\mathcal{M}$, the following properties hold:*

(1) $BIT(U_w) \subseteq I_w$.
(2) If $u \succeq w$ *then* $I_u \subseteq I_w$.
(3) If $u \sharp w$ *then* $I_u \cap I_w = \emptyset$.

Finally, we construct a overlap-free decomposition for $\mathcal{C}_P$ as follows. We traverse the separator tree $\mathcal{M}$ from the root to leaves top-down. Initially, we visit $root$ and U_{root} contains the whole $\mathcal{C}_P$. Assume that we are going down and visiting a node w

in $\mathcal{M}$. Let $C_x \in \mathcal{C}_P$ be the unique component in $\mathcal{C}_P$ that contains e_w in the induced tree $P(C_x)$. Then, there are two cases. If this happens at the first time with C_x, that is, e_w is the *first cut-edge* for C_x on the path from the root to w, then we mark the current node w and associate to w the C_x by $Comp(w) = C_x$. Otherwise, this is at least second time cut or so. Then, we skip w and continue the traversal for descendants. After the traversal, we perform breadth-first search for level $k = 0$ to $K = depth(\mathcal{M})$. Then, we construct a decomposition $\mathcal{C}_P = \mathcal{C}[1] + \cdots + \mathcal{C}[K]$ such that $\mathcal{C}[k] = \{\, C_x \,|\, C_x = Comp(w), w \text{ is a marked internal node in } \mathcal{M} \,\}$ for each $k = 0, \ldots, K$. For the proof of the next lemma, see Appendix A.3.

Lemma 9. *An overlap-free decomposition $\mathcal{C}_P = \mathcal{C}[1] + \cdots + \mathcal{C}[K]$ of $\mathcal{C}_P$ w.r.t. Bit can be computed by the above procedure in $O(m \log m)$ time and $O(\log m)$ space.*

3.4 Complexity analysis

Combining the algorithm MatchUPTM of Fig. 1 in Sec. 3.1 and the bit-parallel implementation of the set manipulation operations in Sec. 3.2, and results in Sec. 3.3, we now have a modified version of the algorithm, called BP-MatchUPTM for the UPTM problem.

Lemma 10. *The operation* $\mathsf{TreeAggr}_P$ *can be implemented to run $O(\log m)$ time using $O(m \log m)$ preprocessing and $O(\log m)$ space if $m \le w$.*

Proof. The claim follows from Lemma 7, Lemma 9, and Lemma 5. □

By applying the module decomposition technique of Myers [8] and Bille [3], we have the main theorem of this paper below:

Theorem 1. *The algorithm* BP-UPTreeMatch *solves the unordered pseudo-tree matching problem with the following complexities:*

- *In the large pattern case ($m > w$): $O(nm \log w / w)$ time using $O(hm/w + m \log w / w)$ space and $O(m \log w)$ preprocessing.*
- *In the small pattern case ($m \le w$): $O(n \log m)$ time using $O(h + \log m)$ space and $O(m \log m)$ preprocessing.*

where m and n are the sizes of pattern tree and text tree, h is the height of T, and $w = \Theta(\log n)$ is the length of computer word.

4 Extension for unordered tree homeomorphism

In this section, we give a modified algorithm for the unordered tree homeomorphism problem (UTH). Let v be any node in T. Then, the set $Desc\text{-}Emb^{P,T}(v)$, called the *descendant embedding set* and the auxiliary set $Sub\text{-}Emb^{P,T}(v)$ are defined by:

$$Desc\text{-}Emb^{P,T}(v) = \{\, x \in [1..m] \,|\, (\exists \phi \in UTH(P(x), T)\ \phi(x) = v \,\}. \qquad (2)$$
$$Sub\text{-}Emb^{P,T}(v) = \{\, x \in [1..m] \,|\, (\exists \phi \in UTH(P(x), T)(\exists w \succeq v)\ \phi(x) = w \,\}.$$

Lemma 11. *For any P and T, we have the following properties:*

(1) For every $x \in V(P)$ and $v \in V(T)$, $x \in \mathit{Desc\text{-}Emb}^{P,T}(v)$ if and only if (i) $\mathit{label}_P(x) = \mathit{label}_T(v)$, and (ii) $\mathit{children}(x) \subseteq \bigcup_{1 \le j \le \alpha(v)} \mathit{Sub\text{-}Emb}^{P,T}(v[j])$.
(2) For any v in T, $\mathit{Sub\text{-}Emb}^{P,T}(v) = \mathit{Desc\text{-}Emb}^{P,T}(v) \cup \bigcup_{1 \le j \le \alpha(v)} \mathit{Sub\text{-}Emb}^{P,T}(v)$.

From the above decomposition lemma, we can develop a bit-parallel algorithm equipped with the bit-parallel implementation of operators including TreeAggr as shown in Sec. 3. For details, please consult Appendix A.5.

Theorem 2 (complexity of the unordered tree homeomorphism problem). *A modified version of algorithm,* BP-MatchUTH*, solves the unordered tree homeomorphism problem (UTH) with the following complexities:*

– *In the large pattern case ($m > w$): $O(nm \log w/w)$ time using $O(hm/w + m \log w/w)$ additional space and $O(m \log w)$ preprocessing.*
– *In the small pattern case ($m \le w$): $O(n \log m)$ time using $O(h + \log m)$ additional space and $O(m \log m)$ preprocessing.*

where m and n are the sizes of pattern tree and text tree, and $w = \Theta(\log n)$ is the length of computer word.

5 Conclusion

In this paper, we consider the unordered pseudo-tree matching problem and the unordered tree homeomorphism problem. As results, we present efficient algorithms for both problems that runs in $O(nm \log w/w)$ time using $O(hm/w + m \log w/w)$ space and $O(m \log w)$ preprocessing with $m > w$ on a unit-cost arithmetic RAM model with addition. As future work, application to tree pattern matching for practical subclasses of XPath and XQuery queries are interesting problems.

References

1. R. Baeza-Yates and G. H. Gonnet, A new approach to text searching, *CACM*, 35(10), 74–82, 1992.
2. P. Bille, I. L. Gortz, The tree inclusion problem: in optimal space and faster, In *Proc. ICALP'05*, 66–77, 2005.
3. P. Bille, New algorithms for regular expression matching, In *Proc. ICALP'06*, 643–654, 2006.
4. M. Gotz, C. Koch, W. Martens, Efficient algorithms for descendant-only tree pattern queries, *Inf. Syst.*, 34(7), 602–623, 2009.
5. C. Jordan, Sur les assemblages de lignes, *Journal für reine und angewandte Mathematik*, 70, 185–190, 1869.
6. P. Kilpelainen, Tree Matching Problems with Applications to Structured Text Databases, *Ph.D thesis*, Report A-1992-6, DCS, University of Helsinki, 1992.
7. P. Kilpelainen, H. Mannila, Ordered and unordered tree inclusion, *SIAM J. Computing*, 24(2), 340-356, 1995.
8. E. W. Myers, A four-russian algorithm for regular expression pattern matching, *JACM*, 39(2), 430–448, 1992.
9. G. Navarro, and M. Raffinot, *Flexible pattern matching in strings: practical on-line search algorithms for texts and biological sequences*, Cambridge, 2002.
10. H. Tsuji, A. Ishino, M. Takeda, A bit-parallel tree matching algorithm for patterns with horizontal VLDC's, In *Proc. SPIRE 2005*, 388–398, 2005.
11. W3C, Extensive Markup Language (XML) 1.0 (Second Edition), *W3C Recommendation*, 06 October 2000. `http://www.w3.org/TR/REC-xml`
12. G. Valiente, Constrained tree inclusion, *Journal of Discrete Algorithms*, 3(2/4), 431–447, 2005.
13. S. Wu, U. Manber, Fast text searching: allowing errors, *CACM*, 35(10), 83–91, 1992.
14. H. Yamamoto and D. Takenouchi, Bit-parallel tree pattern matching algorithms for unordered labeled trees, In *Proc. WADS'09*, 554-565, 2009.

A Appendix

A.1 A taxonomy of tree matching problems

There are many variations of tree matching problems, as in [2, 4, 7, 12, 14] (See also for survey [6]). Below, we summarize some of them. Let $P = P[1..m]$ be an unordered tree of size m, called a *pattern tree*, and $T = P[1..n]$ be an unordered tree of size n, called a *text tree*.

Definition 6 (conditions for tree matching and inclusion). For any (possibly many-one) mapping $\phi : V(P) \to V(T)$, we define conditions (E0)–(E3) as follows:

(E0) ϕ preserves node labels. That is, for every node $x \in V(P)$, $label_P(x) = label_T(\phi(x))$ holds.
(E1) ϕ preserves the parent-child relationship. That is, for every node $x, y \in V(P)$, $(x, y) \in E_P \Rightarrow (\phi(x), \phi(y)) \in E_T$ holds.
(E1') ϕ preserves the ancestor-descendant relationship. That is, for every node $x, y \in V(P)$, $(x, y) \in E_P \Rightarrow \phi(x) \prec \phi(y)$ holds.
(E2) ϕ preserves the precedence relationship. That is, for every node $x, y \in V(P)$ with $x \sharp y$, $x \lhd y \Rightarrow \phi(x) \lhd \phi(y)$ holds.
(E3) ϕ is a one-one mapping. That is, for every node $x, y \in V(P)$, $x \neq y$ implies $\phi(x) \neq \phi(y)$ holds.

Most variations of tree pattern matching problems are described in terms of embeddings [2, 4, 7, 12, 14] as follows.

– A mapping ϕ is said to be an *embedding* from P to T if it satisfies condition (E0) above.
– An embedding ϕ is a *tree matching* if ϕ satisfies (E1), and a *tree inclusion* if ϕ satisfies (E1') instead of (E1).
– ϕ is *ordered* if it satisfies (E2) and *unordered* otherwise.
– An embedding ϕ is *one-one* if ϕ satisfies (E3), and *many-one* otherwise. We often omit the term *one-one* if it is clear from context.

By combination of the above conditions, we can describe a number of tree matching problems. For example, the ordered tree matching [7, 10] is a mapping with (E0), (E1), (E2), and (E3), and the ordered tree inclusion [2, 7] satisfies (E0), (E1'), (E2), and (E3). the unordered tree matching [7] is a mapping with (E0), (E1), and (E3), and the unordered tree inclusion [2, 7] satisfies (E0), (E1'), and (E3). Tree matching problems with many-one embeddings have been studied in the area of first-order and second-order logics over combinatorial structures such as strings, trees, cycles, chains and graphs [4].

Definition 7 (unordered psuedo-tree matching [4]). A *unordered pseudo-tree matching* from P to T is a possibly many-one, unordered, tree matching $\phi : V(P) \to V(T)$, that is ϕ satisfies conditions (E0) and (E1). We denote by $UPTM(P, T)$ the set of all pseudo-tree matching from P to T

Definition 8 (unordered tree homeomorphism). A *unordered tree homeomorphism* from P to T is a possibly many-one, unordered, tree inclusion $\phi : V(P) \to V(T)$, that is ϕ satisfies conditions (E0) and (E1'). We denote by $UTH(P, T)$ the set of all tree homeomorphism from P to T.

We say that pattern P *maps to* a node $v \in V(T)$ in T w.r.t. $UPTM$ (w.r.t. UTH, resp.) if $\phi(root(P)) = v$ for some $\phi \in UPTM(P, T)$ ($\phi \in UTH(P, T)$, resp.). Then, the node v is called an *occurrence* of P in T.

A.2 Proof for Lemma 7

Lemma 7. The bit assignment Bit constructed above from $\mathcal{M}$ is monotone w.r.t. the ancestor relation $\preceq$ for P.

Proof. Suppose that $x \succeq y$ and both of them are included in some component U in $\mathcal{M}$. Then, there exists an upward path π from x to y, and eventually, some edge in π becomes a cut-edge at some node w in $\mathcal{M}$. This split U_w into U_w^0 and U_w^1 such that $x \in U_w^0$ and $y \in U_w^1$ since the latter locates the upper part. We see that U_w^0 precedes U_w^1 in interval $[1..m]$ by Bit. Thus, $x \succeq y$ implies $Bit(x) < Bit(y)$. □

A.3 Proof for Lemma 9

Lemma 9. An overlap-free decomposition $\mathcal{C}_P = \mathcal{C}[1] + \cdots + \mathcal{C}[K]$ of $\mathcal{C}_P$ w.r.t. Bit can be computed by the above procedure in $O(m \log m)$ time and $O(\log m)$ space.

Proof. First, we can show that $C_x = Comp(w) \subseteq U_w$ because initially any component C_x is included in $I_{root} = [1..m]$, and C_x is registered as $Comp(w)$ whenever it is splitted at the first time at some marked node w. Therefore, if $I_x = Int(BIT(C_x))$ and $I_w = Int(BIT(U_w))$, then we have $I_x \subseteq I_w$ holds for every marked node w in $\mathcal{M}$. On the other hand, let $C_x, C_{x'}$ be any mutually distinct components associated to some nodes w and w', respectively, in $\mathcal{M}$. If both of C_x and C_y belong to $\mathcal{C}[k]$ for some $k \geq 0$. then w and w' also have the same depth k, and thus, $w \sharp w'$ holds. From (3) of Lemma 8, it follows that $I_u \cap I_w$. Hence, the result follows. The claimed complexities are derived as follows. It is not hard to see that the time complexity is $O(m \log m)$ time for constructing a separator tree $\mathcal{M}$ by recursive fasion using a stack. For the space complexity, we use a stack of $O(\log m)$ depth to keep nodes of a path from the root to the current node, and also use $O(\log m)$ bitmasks for keeping the necessary information. Therefore, the space complexity of $O(\log m)$ follows. □

A.4 Proof for Theorem 1

[Note: This section is under development. Hiroki Arimura, 2010/05/05]

Suppose a data structure $\mathcal{A}$ for set manipulation in Definition 2 with mask size $0 \leq x \leq w$. In this paper, we assume that $|\Sigma| = O(1)$ Below, we appreviate Label-Match, TreeAggr, and Mask by LM, TA, and MK, respectively. For $\tau \in \{LM, TA\}$, let $T_\tau(x), S_\tau(x)$, and $P_\tau(x)$ be the time, space, and preprocessing of operation τ with mask of size x associated to $\mathcal{A}$. Let $S_{\mathrm{MK}}(x)$ be the space required to store mask of size x. Clearly, $S_{\mathrm{MK}}(x) = \lceil m/w \rceil$ space in words.

General complexity. From the construction of the algorithm MatchUPTM in Fig. 1 and Lemma 2, we have the next claim.

Lemma 12. *The algorithm* MatchUPTM *can be implemented to run in the following complexities:*

– $T = O(n \cdot \{T_{\mathrm{LM}}(m) + T_{\mathrm{TA}}(m) + T_{\mathrm{Other}}(m)\})$ *time,*

– $S = O(h \cdot S_{\mathrm{MK}}(m) + \{S_{\mathrm{LM}}(m) + S_{\mathrm{TA}}(m) + S_{\mathrm{Other}}(m)\})$ *additional space, and*
– $P = O(P_{\mathrm{LM}}(m) + P_{\mathrm{TA}}(m) + P_{\mathrm{Other}}(m))$ *preprocessing,*

where m is the size of P, n and h are the size and the height of T, respectively.

The small pattern case. First, we consider the small pattern case such that $m \leq w$. By putting $x = m$ and substituting Lemma 3, Lemma 4, and Lemma 5 for Lemma 12, we have the following result.

Lemma 13. *In the small pattern case ($m \leq w$), the unordered pseudo-tree matching problem is solvable in*

– $O(n \log m)$ *time using*
– $O(h + \log m)$ *space and*
– $O(m \log m)$ *preprocessing.*

The large pattern case. In the large pattern case, we assume that $m > w$. For the Constant, Union, Member operations, we have the following.

Lemma 14. *In the large pattern case with $m > w$, we have the following.*

Claim 1: The Constant, Union, Member operations can be implemented to run in

– $T_{\mathrm{Other}}(m) = O(\lceil m/w \rceil)$ *time.*
– $S_{\mathrm{Other}}(m) = O(\lceil m/w \rceil)$ *space.*
– $P_{\mathrm{Other}}(m) = O(m)$ *preprocessing.*

Claim 2: The LabelMatch (LM) operation can be implemented to run in

– $T_{\mathrm{LM}}(m) = O(\lceil m/w \rceil)$ *time.*
– $S_{\mathrm{LM}}(m) = O(|\Sigma| \cdot \lceil m/w \rceil) = O(\lceil m/w \rceil)$ *space.*
– $P_{\mathrm{LM}}(m) = O(|\Sigma| \cdot m) = O(m)$ *prerocessing.*

Claim 3: The TreeAggre (TA) operation can be implemented to run in

– $T_{\mathrm{TA}}(m) = O(\lceil m/w \rceil \log m)$ *time.*
– $S_{\mathrm{TA}}(m) = O(\lceil m/w \rceil \log m)$ *space.*
– $P_{\mathrm{TA}}(m) = O(m \log w)$ *preprocessing.*

Proof. We use the module decomposition technique in Myers [8]. By Jordan's theorem [5] iteratively, we can split the original pattern tree P into $h = O(m/w)$ small subtrees of size at most w. Let $\mathcal{P} = \{Q_1, \ldots, Q_h\}$ be the set of resulting small subtrees, called components of P. For each component $Q \in \mathcal{P}$, $root(Q)$ corresponds to a cut-edge $(y, x) \in E(P)$. For each component $Q \in \mathcal{P}$, we assign a module M_Q to Q. Let $parent(Q) = Q_y$ be the component in $\mathcal{P}$ that contains y as its leaf and called the parent component of Q. Since the node x is missing in $parent(Q)$, we recover this by adding x to the module for $parent(Q)$ so that every internal node in M_Q has both of left and the right child. For each node x in P, we call the position of x as a parent the *real position* and the other as a child the *dammy position* in the modules. Basically, we apply each operation Opr to each module Q separately. Then, we copy the value of $root(Q)$ at the child position to the real position of the node in $parent(Q)$ module by module. This takes at most $h = O(\lceil m/w \rceil)$ time.

For instance, we analyze the complexity of the tree aggregation operation as follows. From Lemma 5, we apply the tree aggregation operation to each module separately in $T_{TA}(w) = O(\log w)$ time, $S_{TA}(w) = O(\log w)$ space, and $P_{TA}(w) = O(w \log w)$ preprocessing with module size $m \leq w$ as in small pattern case. By summing up the complexities for all modules, we finally obtain $T_{TA}(m) = O(\lceil m/w \rceil \log w) = O(m \log w/w)$ time, $S_{TA}(m) = O(\lceil m/w \rceil \log w) = O(m \log w/w)$ space, and $P_{TA}(m) = O(\lceil m/w \rceil w \log w) = O(m \log w)$ preprocessing. Similarly, analysis of other operations follows from Lemma 3 and Lemma 4. Hence, the result is proved. □

By substituting the complexities of Lemma 14 for Lemma 12, we have the following result.

Lemma 15. *In the large pattern case ($m > w$), the unordered pseudo-tree matching problem is solvable in*

- $O(nm \log w/w)$ *time using*
- $O(hm/w + m \log w/w)$ *space and*
- $O(m \log w)$ *preprocessing.*

Combining above arguments in this subsection, we have the main result of this paper.

Theorem 1. The algorithm BP-UPTreeMatch solves the unordered pseudo-tree matching problem with the following complexities:

- In the large pattern case ($m > w$): $O(nm \log w/w)$ time using $O(hm/w + m \log w/w)$ space and $O(m \log w)$ preprocessing.
- In the small pattern case ($m \leq w$): $O(n \log m)$ time using $O(h + \log m)$ space and $O(m \log m)$ preprocessing.

where m and n are the sizes of pattern tree and text tree, h is the height of T, and $w = \Theta(\log n)$ is the length of computer word. ◇

A.5 Bit-parallel algorithm for unordered tree homeomorphism

In this section, we show detailed definitions and lemmas for Sec. 4. In the unordered tree homeomorphism problem (UTH), we employ two sets $\mathit{Desc\text{-}Emb}^{P,T}(v)$ and $\mathit{Sub\text{-}Emb}^{P,T}(v) \subseteq V(P) = [1..m]$ for every node v in T defined as follows.

Let v be any node in T. Then, the set $\mathit{Desc\text{-}Emb}^{P,T}(v)$, called the *descendant embedding set* and the auxiliary set $\mathit{Sub\text{-}Emb}^{P,T}(v)$ are defined by:

$$\mathit{Desc\text{-}Emb}^{P,T}(v) = \{\, x \in [1..m] \mid (\exists \phi \in \mathit{UTH}(P(x), T)\ \phi(x) = v \,\}. \quad (3)$$
$$\mathit{Sub\text{-}Emb}^{P,T}(v) = \{\, x \in [1..m] \mid (\exists \phi \in \mathit{UTH}(P(x), T)(\exists w \succeq v)\ \phi(x) = w \,\}.$$

Lemma 11 in Sec. 4 gives a decomposition formula for UTH as in one for UPTM in Sec. 3.1.

We can obtain an algorithm MatchUTH for the unordered tree homeomorphism problem (UTH) from the algorithm MatchUPTM by replacing Line 10 of the recursive subprocedure VisitUPTM with the following line:

```
algorithm MatchUTH($P[1..m]$: a pattern tree, $T[1..n]$: a text tree):
Global Variables: $P$ and $T$;
Output: all occurrences of $P$ in $T$ w.r.t. unordered tree homeomorphism (UTH);
 1: VisitUTH($root(T)$);

procedure VisitUTH($v$: text node)
Return Value: $R \cup S = Sub\text{-}Emb^{P,T}(v)$;
 2: $S \leftarrow$ Constant($\emptyset$); {See Definition 2}
 3: for $i = 1, \dots \alpha(v)$ do
 4:     $S \leftarrow$ Union($S$, VisitUTH($v[i]$));
 5: $R \leftarrow$ Constant($[1..m]$);
 6: $R \leftarrow$ LabelMatch$_P$($R$, $label_T(v)$); {See Definition 2}
 7: $R \leftarrow$ TreeAggr$_P$($R$, $S$); {See Definition 2}
 8: if Member($R$, $root(P)$) then {See Definition 2}
 9:     output "A match is found at a node v.";
10: return Union($R$, $S$); {$R \cup S = Sub\text{-}Emb^{P,T}(v)$}
```

Fig. 7. An algorithm for the unordered tree homeomorphism problem (UTH).

10: **return** $R \cup S$; $\{R \cup S = Sub\text{-}Emb^{P,T}(v)\}$

In Fig. 7, we show an algorithm MatchUTH for for the unordered tree homeomorphism problem (UTH) and its recursive subprocedure VisitUTH.

From Lemma 11, the next lemma immediately follows, we see that the obtained algorithm correctly solves the unordered tree homeomorphism problem in $O(mn)$ time and $O(hm)$ additional space. Furthermore, by similar arguments in the previous section, we can develop a bit-parallel version, called BP-MatchUTH, of the above algorithm MatchUTH equipped with the bit-parallel implementation of two operators TreeAggr and LabelMatch.

Theorem 2. (complexity of the unordered tree homeomorphism problem) A modified version of algorithm, BP-MatchUTH, solves the unordered tree homeomorphism problem (UTH) with the following complexities:

– In the large pattern case ($m > w$): $O(nm \log w/w)$ time using $O(hm/w + m \log w/w)$ additional space and $O(m \log w)$ preprocessing.
– In the small pattern case ($m \leq w$): $O(n \log m)$ time using $O(h + \log m)$ additional space and $O(m \log m)$ preprocessing.

where m and n are the sizes of pattern tree and text tree, and $w = \Theta(\log n)$ is the length of computer word.

Proof. From the construction of the MatchUTH, it is not hard to see that the algorithm has the same complexity to MatchUPTM as given in Lemma 12. Hence, the result immediately follows from Lemma 14. □