To appear in: Journal of Discrete Algorithms, Special issue from the 21st International Workshop on Combinatorial Algorithms, 2011. doi:10.1016/j.jda.2011.12.018

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

Yusaku Kaneta[a,*], Hiroki Arimura[a], Rajeev Raman[b]

[a] *Graduate School of Information Science and Technology, Hokkaido University, N14, W9, Sapporo 060-0814, Japan*
[b] *Department of Computer Science, University of Leicester, University Road, Leicester, LE1 7RH, United Kingdom*

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-to-one matchings that preserve node labels and parent-child relationship. This problem is closely related to the 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, and we assume that $w \geq \log n$. We also discuss a modification of our algorithm for the unordered tree homeomorphism problem, which corresponds to the tree pattern matching problem for XPath queries with descendant axis only.

Keywords: Tree matching, Bit-parallel algorithms, Unordered trees, XML queries, String matching

1. Introduction

1.1. Background

Tree matching is a fundamental problem in computer science, and it has a wide range of applications in XML/Web databases, schema validation, information extraction, document analysis, image processing, and semi-structured data processing. In particular, the tree matching and inclusion problems have attracted much attention and have been extensively studied [8, 10, 18, 19, 30].

In the *unordered tree matching problem*, where an input is a pair of a small pattern tree P and a larger text tree T on some alphabet of labels, and the task is finding a one-to-one mapping that maps nodes of P to nodes of T preserving their node labels and the parent-child relationship [18], while the *unordered tree inclusion problem* is defined similarly except that a mapping preserves the ancestor-descendant relationship instead.

Although the formal one-to-one requirement for matching in both of the above problems seems useful in general, there are some cases in the real application that the requirement for one-to-one matching is too restricted to achieve flexible information retrieval. Thus, it is sometimes appropriate to relax this requirement, and there have been a number of researches on flexible tree pattern matching problems including tree matching with mismatches, approximate tree matching, and tree regular expression matching [5, 7, 24, 29].

1.2. Main results

In this paper, we study a non-standard version of the unordered tree matching and inclusion problems, called the unordered pseudo-tree matching problem [36] and the unordered tree homeomorphism problem [14], respectively, where a matching can be many-to-one so that it may map different pattern nodes into the same text node (See Fig. 1).

☆An earlier version of this paper was presented in the 21st International Workshop on Combinatorial Algorithms (IWOCA'10), London, UK, July 26–28, 2010.

*Corresponding author.
Email addresses: y-kaneta@ist.hokudai.ac.jp (Yusaku Kaneta), arim@ist.hokudai.ac.jp (Hiroki Arimura), r.raman@leicester.ac.uk (Rajeev Raman)

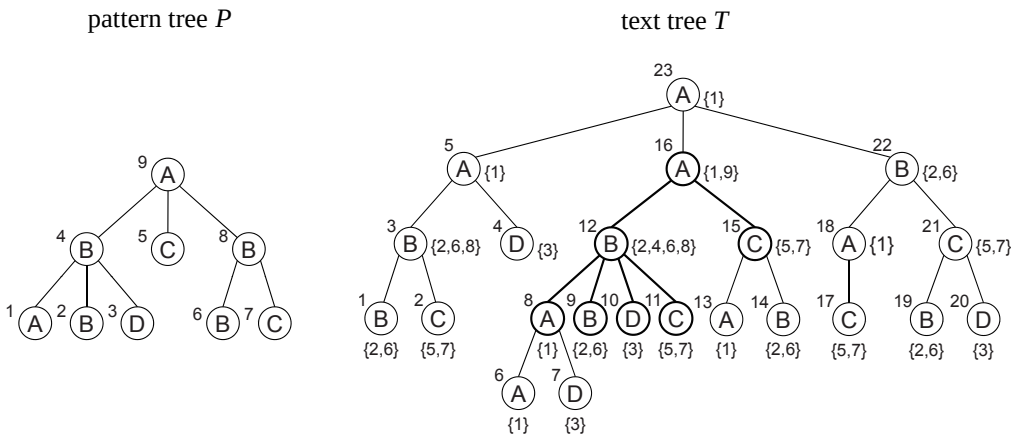

Figure 1: A pattern tree P and a text tree T whose nodes are numbered in the post-order. The tree that consists of thick nodes and edges in T is an occurrence of P in T w.r.t unordered pseudo-tree matching (UPTM). To each node v in T, we attach the embedding set $Emb^{P,T}(v)$ for v, which is defined in Section 3.

In Section 3, we study the unordered pseudo-tree matching problem (UPTM) that is the problem of, given a pattern tree P and a text tree T as input, finding a many-to-one mapping that maps nodes of P to nodes of T preserving their node labels and the parent-child relationship. As main results, we present an efficient algorithm that solves the unordered pseudo-tree matching problem on RAM with integer addition in the following complexities (Theorem 3):

- 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.
- 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 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, and we assume that $w \geq \log n$. Compared with the previous work for UPTM [36], our algorithm improves on the $O(nm^3/w)$ time and $O(nm^2/w)$ space complexities of [36] by factor of $O(m^2/\log w)$ time and $O(nm/h)$ space. Moreover, our algorithm works *online* using only a stack of $O(hm)$ space instead of $O(nm)$ space. This means that it can perform matching by scanning an input stream for tree data, such as XML documents, once from left to right, where an input stream consists of a sequence of balanced open and closed parentheses on alphabet $\Sigma \cup \{\,\overline{a} \mid a \in \Sigma\,\}$ as in XML databases [14, 29, 32]. Since the text size n is much larger than the height h in practice and even unbounded in some applications, such an algorithm will be useful for processing massive tree data.

A key to our algorithm is a data structure for the small pattern case, where $m \leq w$, based on bit-parallel computation of set operations, including *tree aggregation operation* that checks the branching of internal nodes. In Section 4, we improve the complexity of the tree aggregation operation from $O(m)$ time and space to $O(\log m)$ time and space developing bit-assignment technique based on separator trees. Combining this result to dynamic programming tree matching algorithms and the module decomposition technique of [25], we have claimed results for both UPTM and UTH problems.

In Section 5, we also discuss a simpler algorithm for tree aggregation using the bit-parallel version [3] of the monotone routing technique [22]. The resulting algorithm for UPTM runs in the same time and space complexities as above on RAM with integer addition and shift operations. We note that both algorithms use only AC^0 instructions, including Boolean operations, shift, and integer addition, without using multiplication so that the computation cost do not heavily depend on the size of a computer word. Our algorithms obtain a speedup by careful use of RAM operations, and not by table lookup. As a result, the speedup remains valid regardless of the relationship between w and n, m (we only assume that $w \geq \log n$, not that $w = \Theta(\log n)$). In our results, the RAM operations can be replaced by table lookup on k bits at a time, giving a speed of $O(nm/k)$, but increasing the space usage by an additive $O(2^k)$ term. From a theoretical perspective, this means that $k = O(\log h + \log m) = O(\log n)$, whereas the bit-parallel approach works for all $w \geq \log n$. From a practical perspective, the superiority of bit-parallel operations over table lookup, particularly for modern 64-bit processors, has been shown by [31].

In Section 6, we consider the unordered tree homeomorphism problem, UTH, which is the problem of finding a many-to-one mapping that maps nodes of a pattern tree P to nodes of a text tree T preserving their node labels and the ancestor-descendant relationship. We show that the UTH problem is solvable in the same time and space complexities as above (Theorem 4). Compared with the previous $O(nm^2)$ time and $O(n^2)$ space algorithm in [14], our algorithm is at least faster by a factor of w by the use of bit-parallelism.

In Section 7, we consider the relationship to fragments of XPath language for tree structured data. We show that the UPTM and UTH problems exactly correspond to the matching problem for the classes of *conjunctive Core XPath queries with child axis only* and *descendant axis only*, respectively, under a semantics with root occurrence. As a corollary, we show that the evaluation problems for the above fragments of Core XPath queries can be efficiently solvable using our algorithms for UPTM and UTH.

Since the tree matching and inclusion problems under many-to-one mappings have less constraint than the traditional ones, one might think that these problems are less interesting from the view of combinatorial pattern matching. However, this is not necessarily true; The complexities of many generalizations of these problems under many-to-one mappings have been still open when we allow additional features in structured queries, such as, reverse axes, positions, counting, Boolean operations, and transitive closures [5, 8, 13, 14]. Hence, we hope that these results become steps towards development of efficient query mechanism for data intensive applications.

1.3. Related work

The tree matching and inclusion problems have been extensively studied in computer science. In what follows, we denote by ℓ_T, h_T, b_T, r_T, w are the number of leaves, the maximum depth, the maximum branching of T, the maximum number of the same label on a path in T, and the bit-length of a computer word.

In the ordered tree matching problem (OTM, for short), which has been widely studied, a matching is required to be one-to-one and to preserve node labels, parent-child relationship, and previous-following sibling relationship [18]. The classical work by Hoffmann and O'Donnell [15] presented a tree matching algorithm for ranked trees that solves the OTM problem in $O(nm)$ time and $O(h_T m)$ space. Kosaraju [21] have broken first time the $O(nm)$ time barrier for OTM by presenting an algorithm that runs in $O(nm^{0.75}\mathit{polylog}(m))$ time combining tree convolution and partition techniques. Tsuji, Ishino, and Takeda [29] presented a bit-parallel algorithm for OTM that runs in $O(nm/w)$ time and $O(h_T m/w)$ space on RAM with word length w by extending the SHIFT-AND algorithm [4, 35].

In the ordered tree inclusion problem (OTI, for short), a matching is also required to be one-to-one, and to preserve node labels, ancestor-descendant relationship, and previous-following sibling relationship [18]. Kilpeläinen and Mannila [19] studied this problem and presented $O(nm)$ time and space algorithm. The paper [19] also showed that the decision version of the unordered tree inclusion problem (UTI, for short) is NP-complete. Chen [10] devised a more efficient algorithm that solves the OTI problem in $O(n\ell_P)$ time and $O(\ell_P \min\{h_T, \ell_T\})$ space, whose time complexity is still $O(nm)$ in the worst case. Bille and Gørtz [8] presented the first algorithm that has time and space complexities strictly less than $O(nm)$ for OTI. Their algorithm achieves $O(n+m)$ space and its running time is proportional to the minimum of $O(n\ell_P)$, $O(\ell_P \ell_T \log\log n + n)$, and $O(nm/\log n + n\log n)$.

We note that mappings in the OTI problem must satisfy stricter constraints than ones in the UTH problem. Actually, we can easily see that any mapping in OTI is also a proper mapping in UTH. Thus, one may think of the possibility of obvious reduction from, say, the UTH problem to the OTI problem so that we can solve the UTH problem by, e.g., the algorithm in [8]. However, the above observation does not immediately mean that we can reduce the matching problem for UTH to that for OTI. Since the technique of [8] heavily relies on the order constraint to make sequential processing, it is not easy to apply their method to the UTH problem. Hence, it is still an open problem to extend the technique of [8] for UTH to devise time and space optimal matching algorithms. Kilpeläinen [18] and Chapter 5 of Bille [7] are good surveys of tree inclusion problems.

Tree matching problems allowing many-to-one matching have been studied in the area of query languages over combinatorial structures such as strings, trees, and graphs [11, 23, 27] as well as in document engineering and Web systems [1, 5, 9]. The unordered pseudo-tree matching problem (UPTM) is a many-to-one and unordered version of OTM. Gottlob, Koch, and Pichler [11] first showed that a large subclass of XPath queries, called *Core XPath queries* (CXP, for short), can be efficiently evaluated in $O(nm)$ time and space using bottom-up computation over a text tree. From the results of Section 7, we see that their problem essentially includes tree matching problems in UPTM and UTH [14]. For the UPTM problem, Yamamoto and Takenouchi [36] presented an efficient bit-parallel algorithm for

the problem that runs in $O(n r_P \ell_P h_P / w) = O(nm^3/w)$ time and $O(n \ell_P h_P / w) = O(nm^2/w)$ space in the worst case.

The unordered tree homeomorphism problem (UTH) is a many-to-one and unordered version of OTI. From the result of Gottlob, Koch, and Pichler [11] mentioned above, we can see that the UTH problem can be solved in $O(nm)$ time and space. Olteanu *et al.* [28] presented a more space-efficient algorithm that runs in $O(nm h_T)$ time and $O(m h_T + n)$ space for the evaluation problem of CXP queries including UTH. Götz, Koch, and Martens [14] studied the UTH in detail, and presented an efficient matching algorithm for UTH that runs in $O(nm h_P) = O(nm^2)$ time and $O(h_T b_T) = O(n^2)$ space.

Related to tree matching problems allowing many-to-one matching, a number of query languages have been designed in the framework of formal logic and automata on labeled trees including XPath [11, 14], XML Schema [23], Regular Tree Grammars [24], and tree queries [27] to name a few. See [5, 12, 13] for survey of recent results. Some of them use tree matching under many-to-one matching as their core mechanisms [13, 24, 27]. Although they are closely related to our problem, their results cannot be directly applied to our problems at the present.

1.4. Outline of this paper

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 implementation of tree aggregation using separator trees. Section 5 gives another simpler implementation of tree aggregation using monotone routing. Section 6 gives an extension to UTH. Section 7 discuss the relationship among fragments of XPath queries, UPTM, and UTH. In Section 8, we conclude.

2. Preliminaries

2.1. Basic definitions

In this subsection, we give basic definitions and notations on our unordered tree matching problems according to [14, 18, 36]. For a set S, we denote by $|S|$ the cardinality of S. Let $\mathbf{N}_+ = \{1, 2, \ldots\}$. We define the *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 $I(S) = [\min S .. \max S] \subseteq \mathbf{N}_+$. 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 $R \subseteq S^2$ on a set S, we denote by R^+ and $R^* \subseteq S^2$ the *transitive closure* and the *transitive reflexive closure* of R, respectively.

2.2. 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 T be an unordered tree of m nodes whose labels are drawn from Σ. We denote by $V(T) = \{1, \ldots, m\}$ the *node set*, by $E(T) \subseteq V(T) \times V(T)$ the *edge set*, and by $root(T) \in V(T)$ the *root* of T. For each node $v \in V(T)$, $label_T(v) \in \Sigma$ denotes the *label* of v in T, and $T(v)$ denotes the *subtree* of T rooted at v.

If $(v, w) \in E(T)$ then we say that v is the *parent* of w, and w is a *child* of v. We define the ancestor relation $\preceq_T \subseteq V(T) \times V(T)$ for T as follows. If there exists some downward path from v to w with length greater than or equal to zero, i.e., $(v, w) \in E(T)^*$, then we say that v is an *ancestor* of w, w is a *descendant* of v, and write $v \preceq_T w$ (or $w \succeq_T v$). If $v \preceq_T w$ and $v \neq w$, equivalently, $(v, w) \in E(T)^+$, then we say that v is a *proper ancestor* of w, and w is a *proper descendant* of v, and write $v \prec_T w$ (or $w \succ_T v$). If both of $v \not\preceq_T w$ and $w \not\preceq_T v$ hold, then v and w are *incomparable* each other.

For unordered tree T, we define the *size* of T, denoted by $|T|$, to be the number of nodes in T, and the *height* of T, denoted by $height(T)$, to be the maximum length of the paths from the root to the leaves. We denote the sets of all leaves and all internal nodes in T, respectively, by $leaves(T)$ and $internal(T)$. Clearly, $V(T) = internal(T) \cup leaves(T)$. Let v be any node in T. The *arity* of v, denoted by $arity(v) \geq 0$, is the number of the children of v. For convenience, we sometimes assume that there is a certain order among the siblings in T as an internal representation regarding T as an *ordered trees*. For every $1 \leq i \leq arity(v)$, we denote the i-th child of node v by $v[i]$. We denote the *set of the children* of v by $children_T(v) = \{ v[i] \mid i = 1, \ldots, arity(v) \}$. If it is clear from context, we omit the subscript T from $label_T(v)$, $\preceq_T$, and $\succeq_T$ etc in what follows.

2.3. Unordered tree matching problem

In this subsection, we introduce the unordered pseudo-tree matching and unordered tree homeomorphism problems. For other variations of tree matching problems, readers are invited to consult the literatures [8, 14, 18, 19, 30, 36]. Let P be an unordered tree of size m, called a *pattern tree*, and T be an unordered tree of size n, called a *text tree*. We refer to each node of P as *pattern node*, denoted by $x, y, \ldots$, and to each node of T as *text node*, denoted by $v, w, \ldots$, which are possibly subscripted.

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

(E0) ϕ preserves the *node labels*. That is, for any node $x \in V(P)$, $label_P(x) = label_T(\phi(x))$ holds.

(E1) ϕ preserves the *parent-child relationship*. That is, for any 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 any node $x, y \in V(P)$, $(x, y) \in E(P) \Rightarrow (\phi(x), \phi(y)) \in E(T)^+$ holds. This is equivalent to the implication $x \prec_P y \Rightarrow \phi(x) \prec_T \phi(y)$.

In what follows, we refer to any mapping ϕ from $V(P)$ to $V(T)$ satisfying (E0) as *matching*. An *unordered pseudo-tree matching* (UPTM) [36] is a matching ϕ with (E0) and (E1), i.e., a many-to-one version of unordered tree matching [18]. An *unordered tree homeomorphism* (UTH) [14] is a matching ϕ with (E0) and (E1'), i.e., a many-to-one version of unordered tree inclusion [19]. 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. We sometimes refer to $UPTM$ and UTH as related classes of matchings.

Let $\mathcal{F}$ be a class of matchings from $V(P)$ to $V(T)$. Then, a pattern tree P *maps to* a node $v \in V(T)$ in T w.r.t. $\mathcal{F}$ if $\phi(root(P)) = v$ for some $\phi \in \mathcal{F}$. Then, the node v is called an *root occurrence* (*occurrence*, for short) of P in T w.r.t. $\mathcal{F}$. In general, the *tree pattern matching problem w.r.t.* $\mathcal{F}$ is the problem of, given a pattern tree P and a text tree T, finding all occurrences of P in T w.r.t. $\mathcal{F}$. According to this terminology, the *unordered pseudo-tree matching problem* (UPTM) and the *unordered tree homeomorphism problem* (UTH) are tree matching problems related to classes $UPTM$ and UTH of matchings, respectively.

2.4. Model of computation

As the model of computation, we assume a *unit-cost RAM* with w-bit words [2]. The word length satisfies that $w \geq \log n$ so that each word can hold a letter in an input of size n. Our algorithms studied in this paper fully employ bit-parallelism technique taking advantage of parallelism inside a computer word. A *bitmask* of bit-length $1 \leq L \leq w$ is a bit vector $X = b_L \cdots b_1 \in \{0, 1\}^L$, where the most significant bit (MSB) b_L and the least significant bit (LSB) b_1 come to the left and the right ends, respectively. We assume a standard instruction set with its C-like syntax including *bitwise* AND "&", *bitwise* OR "|", *bitwise* NOT "~", *left shift* "$\ll$", *right shift* "$\gg$", *integer addition* "+", and *integer multiplication* "$*$". We refer to a RAM with Boolean operations only as a *Boolean RAM*, denoted by *RAM*(), and for a subset $S \subseteq \{\gg, +, *, \ldots\}$, to the Boolean RAM with additional instructions in S as RAM with S, denoted by $RAM(S)$, where "$\gg$" denotes both of left and right shifts with arbitrary shift length, and "+" denotes integer addition. The space complexity is measured in the number of words.

3. Faster Bit-parallel Algorithm for Unordered Pseudo-Tree Matching

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

3.1. Decomposition formula and a bottom-up algorithm

In Fig. 2, we show a dynamic programming algorithm MatchUPTM for the unordered pseudo-tree matching problem. Our algorithm computes, for every text node v in T, the set $Emb^{P,T}(v)$ of integers in $V(P) = [1..m]$, called the

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

procedure VisitUPTM(v: a text node, P: a pattern tree of size m, T: a text tree of size n):
Output: the embedding set $Emb^{P,T}(v)$ of v;
2: $S \leftarrow$ Constant($\emptyset$); {See Definition 2}
3: **for** $i = 1, \ldots arity(v)$ **do**
4: $S \leftarrow$ Union(S, VisitUPTM($v[i], P, T$));
5: $R \leftarrow$ Constant($V(P)$);
6: $R \leftarrow \mathsf{LabelMatch}_P(R, label_T(v))$; {See Definition 2}
7: $R \leftarrow \mathsf{TreeAggr}_P(R, S)$; {See Definition 2}
8: **if** Member($R, root(P)$) **then** {See Definition 2}
9: **print** "*A match is found at a node v.*";
10: **return** R;

Figure 2: An algorithm for the unordered pseudo-tree matching problem.

embedding set of v, defined by:

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

where $P(x)$ is the subtree of P rooted at pattern node $x \in V(P)$. Clearly, for every pattern node $x \in V(P)$, $x \in Emb^{P,T}(v)$ if and only if the corresponding subtree $P(x)$ has an occurrence at the current text node v by some UPTM ϕ. By definition, we see that P matches T at node v iff $root(P) \in Emb^{P,T}(v)$. Now, we have the next lemma, which is crucial to the correctness of our algorithm in this section.

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) $children_P(x) \subseteq \bigcup_{1 \le j \le arity(v)} Emb^{P,T}(v[j])$.

Proof. "Only if" direction: Suppose that $x \in Emb^{P,T}(v)$ holds. Then, there exists some mapping ϕ that maps $P(x)$ to the root of $T(v)$ satisfying (E0) and (E1) of Definition 1. From (E0), we have (i) $label_P(x) = label_T(v)$. From (E1), we have some mapping $h : [1..arity(x)] \to [1..arity(v)]$ satisfying the following: for every child $x[i]$ of x, $1 \le i \le arity(x)$, the subtree $P(x[i])$ maps to the subtree $T(v[h(i)])$ via some UPTM ϕ_i. Actually, we know that ϕ_i is the restriction ϕ_i of the original ϕ to $P(x[i])$. From this, we have $x[i] \in Emb^{P,T}(v[h(i)])$, and the claim follows. "If" direction: Suppose that conditions (i) and (ii) hold. From (ii), for every $1 \le i \le arity(x)$, there exists some $1 \le h(i) \le arity(v)$ such that $x[i] \in Emb^{P,T}(v[h(i)])$. By definition of $Emb^{P,T}(\cdot)$, we know that for every i, there exists some UPTM ϕ_i such that the subtree $P(x[i])$ maps to the subtree $T(v[h(i)])$ via ϕ_i. Since all destination nodes $v_1 = v[h(1)], \ldots, v_{arity(x)} = v[h(arity(x))]$ are incomparable w.r.t. the ancestor relation $\preceq_T$, the domains of $\phi_1, \ldots, \phi_{arity(x)}$ are mutually disjoint. Therefore, the mapping $\phi = \{(x, v)\} \cup \bigcup_{1 \le i \le arity(x)} \phi_i$ is well-defined. Moreover, we can show from condition (i) $label_P(x) = label_T(v)$ and the above discussion that ϕ is a UPTM that maps $P(x)$ to $T(v)$. Hence, we have $x \in Emb^{P,T}(v)$, and the result follows. □

Based on the recurrence of Lemma 1 above, we show in Fig. 2 a bottom-up procedure VisitUPTM that computes $Emb^{P,T}(v)$ by using the post-order traversal of T. To describe the procedure, we need the following operations on subsets of $V(P)$, where the appropriate encoding will be given later.

Definition 2 (set manipulation operations). We define operations Constant, Union, Member, $\mathsf{LabelMatch}_P$ (label matching), and $\mathsf{TreeAggr}_P$ (tree aggregation) on subsets of $V(P)$ as follows, where $R, S \subseteq V(P)$ and $x \in V(P)$ are appropriately encoded, and $\alpha \in \Sigma$:

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

The procedure VisitUPTM computes the embedding set $Emb^{P,T}(v)$ of v as follows. At each text node v, the procedure maintains a *candidate set* R and a *child set* S, where $R, S \subseteq V(P)$. First, the procedure recursively computes the embedding sets $Emb^{P,T}(v[i])$ of the children $v[i]$ of v for every $1 \leq i \leq arity(v)$, and merges them into the child set S from Line 2 to 4. Next, after setting the candidate set $R = V(P)$ at Line 5, the procedure applies the set manipulation operations $\mathsf{LabelMatch}_P$ and $\mathsf{TreeAggr}_P$ to filter R at Line 6 and Line 7, respectively, by checking conditions (i) and (ii) of Lemma 1. Finally, VisitUPTM obtains R as the embedding set $Emb^{P,T}(v)$ of v. Later, the above set manipulation operations will be implemented in a bit-parallel manner in Section 3.2, Section 4, and Section 5.

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

Lemma 2. *Let P be any pattern tree and T be any text tree. For the unordered pseudo-tree matching problem, the algorithm* MatchUPTM *in Fig. 2 correctly finds all occurrences of P in T. Moreover, the algorithm can be implemented to run in $O(nm)$ 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 be implemented to run in streaming setting using a stack of embedding sets with length $O(h)$ using $O(hm)$ space, where T is given as an input stream consisting of a sequence of balanced open and closed parentheses on alphabet $\Sigma \cup \{ \overline{a} \mid a \in \Sigma \}$ as in XML databases [14, 29, 32]. The details are left to the readers.

3.2. Outline of bit-parallel implementation

In the following subsections, we give the bit-parallel version of the algorithm MatchUPTM, called BP-MatchUPTM, that runs 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 is the word length. Let us fix a pattern tree P of size m on a finite alphabet Σ. In what follows, we assume that $|\Sigma| = O(1)$.

Position mapping. In the bit-parallel implementation of MatchUPTM, we introduce a data structure for representing a subset S of the universe $V(P)$ of size m that efficiently supports the collection of the set manipulation operations in Definition 2 in Section 3.1. A *position mapping* is any one-to-one mapping $pos : V(P) \to [1..m]$ that assigns to each node $x \in V(P)$ a bit-position $pos(x) \in [1..m]$. We define $pos(S) = \{ pos(x) \mid x \in S \}$ for a set $S \subseteq V(P)$, and $NUM(U) \in \{0,1\}^m$ to be the bitmask representing U for a subset $U \subseteq [1..m]$. Now, we represent any subsets S of $V(P)$ by bitmasks $X \in \{0,1\}^m$ of length m as m-bit integers from 0 to $2^m - 1$ as $NUM(pos(S))$. At this moment, we leave pos unspecified and will give the appropriate definition for pos later in the next subsection.

Basic set manipulation operations. Among the set manipulation operations in Definition 2, the following operations are easy to implement.

Lemma 3. *Let $R \subseteq V(P)$ be any node set, and $X, Y \in \{0,1\}^m$ be any bitmasks of bit-length m. Then, the following codes correctly implement the operations. Moreover, all operations are executed in $O(1)$ time if $m \leq w$.*

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

Label matching operation. The label matching operation $\mathsf{LabelMatch}_P$ can be implemented using a set of character masks as in the SHIFT-AND algorithm for exact string matching [4, 26, 35] or in Move operation for regular expression matching [6].

Lemma 4. *Suppose that if $m \leq w$. Then, the operation* $\mathsf{LabelMatch}_P$ *can be correctly implemented by the following codes, where* $\{\, \mathtt{LABEL}[\alpha] \in \{0,1\}^m \mid \alpha \in \Sigma \,\}$ *is a set of bitmasks for a pattern tree P. Moreover, the operation can be executed in $O(1)$ time and $O(|\Sigma|)$ space using $O(|\Sigma| + m)$ preprocessing.*

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

Later, bit-parallel implementations of the operation $\mathsf{TreeAggr}_P$ will be given in Section 4 and Section 5. In Section 4, we will show the following theorem using an algorithm based on separator tree technique.

Theorem 1. *Suppose that $m \leq w$. Then, the operation* $\mathsf{TreeAggr}_P$ *can be implemented to run in $O(\log m)$ time using $O(\log m)$ space and $O(m \log m)$ preprocessing on RAM$(+)$.*

In Section 5, we also show the following theorem using an algorithm based on monotone routing technique. Though the second algorithm requires the left and right shifts with arbitrary large width, it is much simpler than the first one.

Theorem 2. *Suppose that $m \leq w$. Then, the operation* $\mathsf{TreeAggr}_P$ *can be implemented to run in $O(\log m)$ time using $O(\log m)$ space and $O(m \log m)$ preprocessing on RAM$(+, \gg)$.*

3.3. Complexity analysis

Combining the algorithm MatchUPTM of Fig. 2 in Section 3.1, the bit-parallel implementations of the set manipulation operations in Section 3.2, and the implementation of the tree aggregation operation based on separator trees in Section 4, we now have a bit-parallel version of the algorithm, called BP-MatchUPTM for UPTM. By applying the module decomposition technique of Myers [25] and Bille [6], we have the main theorem of this paper below:

Theorem 3 (complexity of the unordered pseudo-tree matching problem). *The algorithm* BP-MatchUPTM *solves the unordered pseudo-tree matching problem (UPTM) on RAM$(+)$ 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 a pattern tree P and a text tree T, h is the height of T, and w is the word length.

Proof. In the small pattern case that $m \leq w$, combining Lemma 3, Lemma 4, and Theorem 1, we have that our bit-parallel algorithm runs in $O(\log m)$ time per text node using $O(h + \log m)$ space and $O(m \log m)$ preprocessing. Thus, the claim holds. In the large pattern case that $m > w$, as in Myers [25] and Bille [6], we partition a pattern tree P into a collection of $\lceil m/w \rceil$ small subtrees, called *modules* of size $O(w)$ so that each module fits into one word. Then, we simulate each module in $O(\log w)$ time per text node using $O(h + \log w)$ space and $O(w \log w)$ preprocessing using the method in the small pattern case. For the whole computation, the algorithm traverses the modules in the post-order of the original pattern tree P by simulating each module one by one and propagating the result sequentially to the parent module. Hence, the theorem follows. □

In the above theorem, we can replace the $O(\log w)$-term in the time, preprocessing, and space complexities with $\min\{\log w, height(P)\}$. Thus, we know that the algorithm based on separator trees Section 4 is particularly fast for those pattern trees that have small height. On the otherhand, using the implementation of the tree aggregation operation based on monotone routing in Section 5, we also obtain the algorithm for UPTM in the same time, preprocessing, and space complexities on *RAM*$(+, \gg)$.

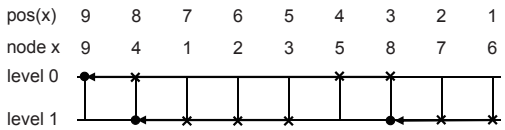

Figure 3: An example of a pair $\langle pos, level \rangle$ of position and level mappings for the pattern tree P in Fig. 1, which is both monotone and non-overlapping.

4. Bit-parallel Implementation of Tree Aggregation using Separator Trees

In this section, we show an efficient bit-parallel implementation of the tree aggregation operation $\mathsf{TreeAggr}_P$. Given any subsets $R, S \subseteq V(P)$, which are encoded as bitmasks via *pos*, the tree aggregation operation returns the set $\{ x \in R \mid children_P(x) \subseteq S \}$ of elements in R satisfying condition (ii) of Lemma 1. In other words, this means gathering the values of the children $x[1], \ldots, x[arity(x)]$ of node x, taking their logical AND, and setting it to the value of their parent x in parallel.

4.1. Monotone and non-overlap mappings

In the followings, we give some technical definitions necessary to define the operation. Let P be a pattern tree of size m. Its node set $V(P)$ is partitioned into two subsets, the set $internal(P)$ of its internal nodes and the set $leaves(P)$ of its leaves.

Our algorithm computes a pair $\theta = \langle pos, level \rangle$ of mappings, where $pos : V(P) \to [1..m]$ is a position mapping that assigns to each node x a bit-position $pos(x)$, introduced in the previous section, and $level : internal(P) \to \{0, \ldots, K\}$ is a *level mapping* that assigns to each internal node x in P a non-negative integer $level(x)$ for some $K \geq 0$, which is called the *maximum level* of the level mapping. We define the *level* of a pair $\theta = \langle pos, level \rangle$ of mappings to be the maximum level of the level mapping.

Let x be any internal node in P. The *branching component* of x in P is the set $\{x\} \cup children_P(x) \subseteq V(P)$ that consists of the node x itself and its children. The *interval* for x is defined by the interval $interval_P(x, pos) = I(pos(\{x\}) \cup pos(children_P(x)))$, which is the minimum interval containing the branching component of x.

Definition 3. Let $\theta = \langle pos, level \rangle$ be a pair of mappings defined above.

- The pair θ is said to be *monotone* if it preserves the ancestor-descendant relationship, i.e., for any nodes $x, y \in V(P)$, if x is a proper ancestor of y then $pos(x) > pos(y)$.
- The pair θ is said to be *non-overlapping* if for any nodes $x, y \in internal(P)$, $level(x) = level(y)$ implies $interval_P(x, pos) \cap interval_P(y, pos) = \emptyset$.

Any pattern tree P of size m always has an obvious monotone and non-overlapping pair $\langle pos, level \rangle$ of mappings of $height(P) = O(m)$, where *pos* is induced in the post-order traversal of P, and *level* is defined such that a node of depth k has level k.

Example 1. In Fig. 3, we show an example of a pair $\langle pos, level \rangle$ of mappings for the pattern tree P of size $m = 9$ in Fig. 1, where *pos* maps nodes in $V(P)$ to bit-positions $[1..9]$. The level mapping *level* is non-overlapping and maps the internal nodes 4, 8, and 9 to at most two levels $\{0, 1\}$, while the obvious non-overlapping level mapping requires three levels.

4.2. Basic idea

Now, we explain a basic idea of our bit-parallel implementation of the tree aggregation operation. Suppose that we are given an input bitmask, and that we perform the tree aggregation operation for computing the bit for an internal node $x \in V(P)$, called the *parent bit*, from the bits for its children, called the *child bits*. To do this, it is sufficient to check if

all the child bits are 1. To implement this operation efficiently, we use a bit-parallel technique with carry propagation of integer addition, which is used in the Extended SHIFT-AND algorithm by Navarro and Raffinot [26].

First of all, let us consider a special case that an input bitmask contains only one branching component, i.e., the parent bit for x and its child bits. Suppose that the position mapping pos is monotone, that is, the bit-position for x is located higher than that for any of its children. Given an input bitmask, we set the parent bit for x to 0, and leave all the child bits unchanged. Then, we set all bits b_i ($1 \leq i \leq m$) in the input bitmask as follows: if b_i is the bit for either x or one of its children, or if i is outside $interval_P(x, pos)$, then we leave b_i unchanged; otherwise, b_i is either a bit between x and one of x's child, or a bit between a consecutive pair of x's children, and then we set b_i to 1. Now, we can compute the conjunction of all the child bits by adding 1 at the lowest child bit-position. Since the bits other than the parent and child bits are filled with 1's as described above, the application of integer addition causes carry propagation from the lowest bits to the parent bit, and the parent bit gets 1 if and only if all the child bits are 1 in the input bitmask.

Next, consider the case that a bitmask contains more than one branching components. We need a pair $\theta = \langle pos, level \rangle$ of mappings which are both monotone and non-overlapping, which is introduced in Section 4.1. We partition the internal nodes of P according to the values of mapping $level$ from $k = 0$ to K, where K is the level of θ. For each $k = 0$ to K, we separately apply the above method for computing tree aggregation of the internal nodes in level k. If θ is monotone and non-overlapping, this procedure correctly implements the tree aggregation operation in the general case.

4.3. Detail of bit-parallel implementation

Now, we explain how to implement the tree aggregation operation in bit-parallel manner. First, we explain the preprocessing phase of the algorithm, which computes a set of bitmasks from a pattern tree P. For every level $k = 0, \ldots, K$, we denote by $internal(P, k)$ the set of all internal nodes $x \in internal(P)$ with level k.

Definition 4 (Preprocess). In the preprocessing phase, the bitmasks are computed as follows. The *leaf mask* `LEAF` is the bitmask of bit-length m that sets 1 at the bit-position for every leaf in P, that is,

$$\texttt{LEAF} = |_{x \in leaves(P)}\, NUM(pos(\{x\})).$$

For every level $k = 0, \ldots, K$, we define the following bitmasks. The *parent mask* `PARENT`$[k]$ is the bitmask of bit-length m that sets 1 at the bit-position for every internal node x with $level(x) = k$, that is,

$$\texttt{PARENT}[k] = |_{x \in internal(P,k)}\, NUM(pos(\{x\}))).$$

The *child mask* `CHILD`$[k]$ is the bitmask of bit-length m that sets 1s at the bit-positions for the children of every internal node x with $level(x) = k$, that is,

$$\texttt{CHILD}[k] = |_{x \in internal(P,k)}\, NUM(pos(children_P(x))).$$

The *padding mask* `PADDING`$[k]$ is the bitmask of bit-length m that sets 1 at every bit-position which is neither the bit for the parent x nor the bit for its child, that is,

$$\texttt{PADDING}[k] = |_{x \in internal(P,k)}\, NUM(interval_P(x, pos) \setminus pos(\{x\} \cup children_P(x))).$$

The *lowest mask* `LOWEST`$[k]$ is the bitmask of bit-length m that sets 1 at the lowest bit-position for the interval for an internal node x with $level(x) = k$, that is,

$$\texttt{LOWEST}[k] = |_{x \in internal(P,k)}\, NUM(\{\min(interval_P(x, pos))\}).$$

Given a monotone and non-overlapping pair $\theta = \langle pos, level \rangle$ of mappings for an input pattern P of size m, all the bitmasks above can be computed in $O(mK)$ time and space, where K is the maximum level of the mapping $level$. In Fig. 4, we show a bit-parallel algorithm $\mathsf{TreeAggr}_P$ that implements the runtime phase of the tree aggregation operation for P, which is designed based on the idea in Section 4.2.

procedure $\mathsf{TreeAggr}_P$(X: a bitmask representing a candidate set R; Y: a bitmask representing a child set S):
Note: See Section 4 for the definition of bitmasks `LEAF`, `PARENT`[·], `CHILD`[·], `PADDING`[·], and `LOWEST`[·];
Output: the bitmask $Z = NUM(pos(\{\ x \in R \mid children_P(x) \subseteq S\ \}))$;

1: **for** every level $k = 0, \ldots K$ **do**
2: $A[k] \leftarrow Y$ & `CHILD`[k];
3: $B[k] \leftarrow A[k]$ | `PADDING`[k];
4: $C[k] \leftarrow B[k]$ + `LOWEST`[k];
5: $Z[k] \leftarrow C[k]$ & `PARENT`[k];
6: $Z \leftarrow Z[0] \mid \cdots \mid Z[K]$;
7: $Z \leftarrow X$ & (Z | `LEAF`);
8: **return** Z;

Figure 4: A bit-parallel implementation of the tree aggregation operation based on a monotone and non-overlapping pair $\theta = \langle pos, level \rangle$ of position and level mappings.

Lemma 5 (Runtime complexity of the tree aggregation). *Let P be a pattern tree of size m. Given a monotone and non-overlapping pair $\theta = \langle pos, level \rangle$ of mappings for P, the procedure* $\mathsf{TreeAggr}_P$ *of Fig. 4 correctly implements the tree aggregation operation for P. Moreover, it runs in $O(K)$ time if $m \leq w$, where K is the maximum level of mapping level.*

If we use an obvious monotone and non-overlapping pair $\langle pos, level \rangle$ of mappings with level $K = O(m)$ based on the height of a pattern tree, then we have only an algorithm that solves UPTM in $O(nm^2/w)$ time using $O(hm/w+m^2/w)$ additional space and $O(m^2)$ preprocessing in the large pattern case, where we do not apply module decomposition. Even if we apply module decomposition, we only get an algorithm with $O(nm)$ time and $O(hm)$ space complexities, which are same as that of the dynamic programming algorithm in Section 3.1. In the next subsection, however, we will show an efficient algorithm that computes a monotone and non-overlapping pair of $O(\log m)$ level in $O(m \log m)$ time and space, which results in the algorithm for UPTM in $O(nm \log(w)/w)$ time and $O(hm/w + m \log(w)/w)$ space.

4.4. Computing monotone and non-overlapping mappings using separator trees

In this subsection, we show how to find a monotone and non-overlapping pair $\theta = \langle pos, level \rangle$ of mappings whose level is $O(\log m)$ using a data structure called a *separator tree* [16]. Using this algorithm as a subprocedure, we can solve the UPTM problem in $O(nm \log(w)/w)$ time using $O(hm/w + m \log(w)/w)$ additional space and $O(m \log(w))$ preprocessing.

The following lemma plays an important role in computing the monotone and non-overlapping pair $\theta = \langle pos, level \rangle$ of mappings.

Lemma 6 (Jordan [16]). *Let Q be any binary tree. Then, there exists a node x in Q such that $|Q(x)| \leq (2/3)|Q|$ and $|Q(\overline{x})| \leq (2/3)|Q|$, where $Q(x)$ is the subtree of Q rooted at x and $Q(\overline{x})$ is the tree obtained by pruning $Q(x)$ from Q.*

Iteratively applying the above lemma to a given rooted tree Q of size n, we can build a *separator tree*, denoted by $\mathcal{S}(Q)$, for Q, which is an almost balanced, rooted tree of size $2n$, whose height is bounded from above by $O(\log n)$. If it is clear from context, in what follows, we may write $\mathcal{S}$ for $\mathcal{S}(Q)$ by omitting the input Q. Given a binary tree Q, we can build a separator tree $\mathcal{S} = \mathcal{S}(Q)$ for Q having height $O(\log n)$ in $O(n \log n)$ time using the procedure $\mathsf{BuildSeparatorTree}$ in Fig. 5. During computation of $\mathcal{S}$, we also associate the information $S(v), N(v)$, and $H(v)$ to each node v in $\mathcal{S}$, where v is the node created for an argument Q' in the current iteration of the procedure, $S(v) = V(Q') \subseteq V(Q)$ is the node set of a subtree of Q, $N(v) \in S(v)$ is called the *splitting node* for Q', and $H(v)$ is kept undefined.

Now, we are ready for describing our algorithm that computes a monotone and non-overlapping pair $\langle pos, level \rangle$ of mappings. Given an input pattern tree P, the algorithm consists of the following steps:

(1) To each non-root node x in P, we associate the *parent pointer* $pa(x)$ to the parent of x.

procedure BuildSeparatorTree(Q: a binary tree):
Output: a separator tree $\mathcal{S}(Q)$ for Q;
1: Create a new node v;
2: **if** Q is a singleton node x **then begin** $S(v) = \{x\}$ and $N(v) = \perp$;
3: **else**
4: Find a node x in Q that splits Q into two subgraphs $Q(x)$ and $Q(\overline{x})$ of almost equal sizes;
5: v_L = BuildSeparatorTree($Q(\overline{x})$);
6: v_R = BuildSeparatorTree($Q(x)$);
7: $S(v) = V(Q)$ and $N(v) = x$;
8: Set v_L and v_R to the left child and the right child of v, respectively;
9: $H(v)$ is kept undefined;
10: **return** v;

Figure 5: A recursive procedure for constructing a separator tree for a binary tree.

(2) Next, we transform P into one of its binary versions Q by inserting newly created nodes to P in a standard way. Then, we extend the parent pointer for newly created nodes $y \notin V(Q) \setminus V(P)$ such that $pa(y)$ is the nearest ancestor originally contained in P.
(3) We build a separator tree $\mathcal{S} = \mathcal{S}(Q)$ from Q by using the procedure BuildSeparatorTree above, where the associated information $S(v)$ and $N(v)$ are calculated for each node v in $\mathcal{S}$.
(4) For each internal node v in $\mathcal{S}$, we define the label $H(v) = pa(N(v))$ to be the original node in P that is pointed by the parent pointer of the splitting node $N(v)$. We call $H(v)$ the *target node* at v. From (1) of Lemma 7, any node v has the unique ancestor node $\hat{v} = \min_{\preceq_{\mathcal{S}}}[v]_H$, where $[v]_H$ is the set of nodes defined by $[v]_H = \{ w \in V(\mathcal{S}) \mid H(v) = H(w) \}$. Then, we mark all of such highest ancestors $\hat{v}$ in $\mathcal{S}$. We also mark all the leaves in $\mathcal{S}$ whose label $S(v)$ consists of an original node in P. Compute the *contracted separator tree* $\mathcal{CS} = \mathcal{CS}(Q)$ by removing all unmarked nodes by contracting the incident edges and by attaching the related children to the parent.
(5) In the resulting tree $\mathcal{CS}$, we observe that the labels $S(\ell)$ of all leaves ℓ have one-to-one correspondence to $V(P)$ and the labels $H(v)$ of all internal nodes v in $\mathcal{CS}$ also have one-to-one correspondence to $internal(P)$ in the original pattern tree P.
   (a) From the leaves of $\mathcal{CS}$, we compute the mapping pos as follows: we scan the leaves from right to left; if the i-th leaf ℓ_i from the rightend has label $S(\ell) = \{x\}$ then we set $pos(x) = i$, where $1 \le i \le m$.
   (b) From the internal nodes of $\mathcal{CS}$, we compute the mapping $level$ as follows: we visit all internal nodes from the top to the bottom; If an internal node v has label $H(v) = x$ and depth k, then we set $level(x) = k$, where $0 \le k \le O(\log m)$.
(6) Return a pair $\langle pos, level \rangle$ of mappings.

For example, we show a pattern tree P, a binary version Q of P, a separator tree $\mathcal{S}(Q)$, and a contracted separator tree $\mathcal{CS}(Q)$ in (a), (b), (c), and (d) of Fig. 6. In Fig. 3, we also show the bit-assignment for nodes in P by a pair $\langle pos, level \rangle$ of position and level mappings obtained from the contracted separator tree $\mathcal{CS}(Q)$ in (d) of Fig. 6.

In the above algorithm, the contraction phase at step (4) with $H(\cdot)$ and $pa(\cdot)$ is for saving the number of bits in a bitmask, and do not affect the correctness and the asymptotic time complexity.

To show the correctness, we introduce some technical definitions and a lemma below. Let x be any node in Q. We define the set $D(x) = \{ y \in V(Q) \mid pa(y) = x \} \subseteq V(Q)$ and $C(x) = D(x) \cup \{x\}$. The set $D(x)$ corresponds to the union of the set $L_P(x)$ of the nodes which originally was the branching component $\{x\} \cup children_P(x)$ in P and the set $L_Q(x)$ of the nodes which are newly introduced in Q to "binarize" the component (For example, see (b) and (c) of Fig. 6). We say that a node v is *higher than* a node w in $\mathcal{S}$ if $v \preceq_{\mathcal{S}} w$ holds.

Lemma 7. *Let $\mathcal{S} = \mathcal{S}(Q)$ be the separator tree for Q. For any node x in P, the following properties hold:*

(1) *There exists the unique highest node v in $\mathcal{S}(Q)$ such that $H(v) = x$, which we denote by $Locus(x)$.*

(a) original pattern tree P (b) binary version Q of P

(c) separator tree $\mathcal{S}(Q)$ (d) contracted separator tree $C\mathcal{S}(Q)$

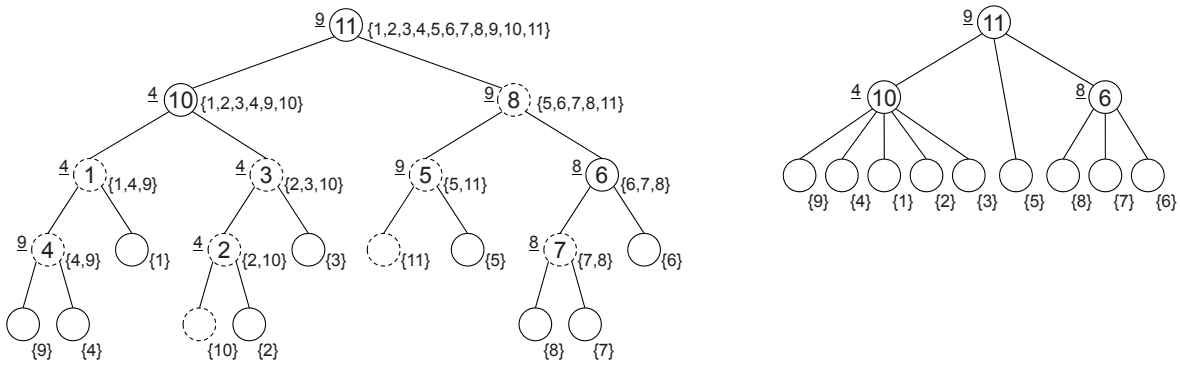

Figure 6: (a) An original pattern tree P with post-order numbering. (b) A binary version Q of P, where white and shadowed circles indicate the original nodes in P and newly created nodes not in P, respectively, and each dotted upward arrow indicates the function *pa*. (c) A separator tree $\mathcal{S} = \mathcal{S}(Q)$, where dotted circles will be removed at Step (4) of the algorithm. We attach $H(v)$ and $S(v)$ to the top-left of each internal node v and the bottom-right of each node $v \in V(\mathcal{S})$, respectively. (d) A contracted separator tree $C\mathcal{S} = C\mathcal{S}(Q)$. We attach $H(v)$ and $S(\ell)$ to the top-left of each internal node $v \in \mathit{internal}(C\mathcal{S})$ and to the bottom-right of each leaf $\ell \in \mathit{leaves}(C\mathcal{S})$, respectively.

(2) *All the members of $C(x) = D(x) \cup \{x\}$ are associated to some leaves below $v = \mathit{Locus}(x)$ in $\mathcal{S}$.*

Proof. (1) We search for the claimed highest node $\mathit{Locus}(x)$ by traversing $\mathcal{S}$ downward from the root to a leaf. We repeat the search while the current node v satisfies the invariant $C(x) \subseteq S(v)$. Initially, at $\mathit{root}(\mathcal{S})$, $C(x) \subseteq S(\mathit{root}(\mathcal{S}))$ clearly holds. If $C(x) \subseteq S(v)$ holds at some node v, then we see that $S(v) = V(Q')$ for some Q'. Then, $z = N(v) \in Q'$ splits $V(Q')$ into $V(Q'(z))$ and $V(Q'(\overline{z}))$, and thus $C(x)$ to $C_R(x) = C(x) \cap V(Q'(z))$ and $C_L(x) = C(x) \cap V(Q'(\overline{z}))$. (Case 1) If $z \in D(x)$, then v is the highest node with $H(v) = x$, denoted by $\mathit{Locus}(x)$, that we are looking for. (Case 2) If $z \notin D(x)$, then one of $C_R(x)$ and $C_L(x)$ completely contains $C(x)$ because $C_R(x)$ and $C_L(x)$ are mutually disjoint. Then, we go down to the associated child w of v such that $C(x) \subseteq S(w)$, and the proof continues. From these cases, the above claim follows. Since the choice in Case 2 is deterministic, so is the selection along the path from the root to $\mathit{Locus}(x)$. This completes the proof. (2) From the property (1) we see that $C(x) \subseteq S(v)$ at the locus $v = \mathit{Locus}(x)$. Suppose that we are going downward from $v = \mathit{Locus}(x)$ to leaves by keeping track of the subsets $S(w)$ containing x at each descendant w of v. At each node w, $S(w)$ is split to disjoint subsets, and we follow exactly one of the subsets. Repeating this process, we eventually reach some leaf to which x is associated. This completes the proof. □

Lemma 8. *If $\theta = \langle pos, level \rangle$ be the pair of mappings computed by the above algorithm, then θ is monotone and non-overlapping for $V(P)$.*

Proof. *Monotonicity*: By the construction of $\mathcal{S}(Q)$, for any pattern node x and its ancestor y, if $x = N(v)$ at a node v in $\mathcal{S}(Q')$, then x falls into the right subtree $\mathcal{S}(Q'(x))$ and y falls into the left subtree $\mathcal{S}(Q'(\overline{x}))$. Since the bit-positions are numbered from right to left on leaves, it follows that $pos(y) > pos(x)$ is ensured. *Non-overlappingness*: Suppose that two nodes x_1 and x_2 in P are assigned the same level, i.e., $level(x_1) = level(x_2)$. From the construction, $v_i = \mathit{Locus}(x_i)$

holds for every $i = 1, 2$. From (2) of Lemma 7, all elements of $C(x_i)$ are assigned to some leaves below v_i for every $i = 1, 2$. On the other hand, if $level(x_1) = level(x_2)$ then v_1 and v_2 are incomparable internal nodes in $\mathcal{S}(Q)$ w.r.t. $\preceq_{\mathcal{S}(Q)}$, and thus the sets of their leaves do not intersect each other. Hence, the result is proved. □

From Lemma 5 and Lemma 8, we have the main result of this section.

Theorem 1. Suppose that $m \leq w$. Then, the operation $\mathsf{TreeAggr}_P$ can be implemented to run in $O(\log m)$ time using $O(\log m)$ space and $O(m \log m)$ preprocessing on $RAM(+)$.

Proof. From Lemma 8, a monotone and non-overlapping pair $\theta = \langle pos, level \rangle$ of mappings for a pattern tree P is computable in $O(m \log m)$ time and $O(\log m)$ space, where m is the size of P. Then, the result follows from Lemma 5. □

4.5. Example execution of the tree aggregation algorithm

Below, we show an example execution of the bit-parallel implementation of the tree aggregation operation based on separator trees. Consider the pattern tree P of size $m = 9$ and the text tree T of size $n = 23$ in Fig. 1. Suppose that we visit the current text node $v = 16$ labeled with $\mathtt{A} \in \Sigma$. After the label matching operation at the text node 16, we have the input bitmasks $X = 101000000$ and $Y = 010101111$ for candidate set R and child set S, respectively, where $R = \mathsf{LabelMatch}_P([1..m], \mathtt{A}) = \{1, 9\}$ and $S = Emb^{P,T}(12) \cup Emb^{P,T}(15) = \{2, 4, 5, 6, 7, 8\}$. We assume a monotone and non-overlapping pair $\langle pos, level \rangle$ of mappings in Fig. 3, where level mapping $level$ maps nodes in P to two levels $\{0, 1\}$. In what follows, we show computation at level $k = 0$ that includes the branching component for an internal node $x = 9$ in P. We call by the *parent bit* the bit for node 9 and by the *child bits* the bits for node 4, 5, and 8.

In the preprocessing phase, we compute the bitmasks LEAF, PARENT[k], CHILD[k], PADDING[k], and LOWEST[k] for level $k \in \{0, 1\}$. We show the bitmasks used in computation at level $k = 0$, where the circled number 9 and the boxed numbers 4, 5, and 8 represent the parent and children of the branching component for node 9, respectively:

pattern node x	⑨	4	1	2	3	5	8	7	6
bit-position $pos(x)$	9	8	7	6	5	4	3	2	1
LEAF	0	0	1	1	1	1	0	1	1
PARENT[0]	1	0	0	0	0	0	0	0	0
CHILD[0]	0	1	0	0	0	1	1	0	0
PADDING[0]	0	0	1	1	1	0	0	0	0
LOWEST[0]	0	0	0	0	0	0	1	0	0

In the runtime phase, we perform the codes in Fig. 4. At Line 2, we first leave the child bits in original input Y by AND-ing Y with bitmask CHILD[0]:

Y	0	1	0	1	0	1	1	1	1
& CHILD[0]	0	1	0	0	0	1	1	0	0
A[0]	0	1	0	0	0	1	1	0	0

Then, at Line 3, we fill with 1's all the bits for the interval $I_9 = interval_P(9, pos) = [3..9]$ for node 9 other than the parent and child bits by OR-ing the result A[0] of Line 2 with bitmask PADDING[0]:

A[0]	0	1	0	0	0	1	1	0	0
| PADDING[0]	0	0	1	1	1	0	0	0	0
B[0]	0	1	1	1	1	1	1	0	0

At Line 4, we check if all the child bits are 1 in original input Y, and set the parent bit to 1 if the test is true, 0 otherwise. This is done using carry propagation caused by integer addition of the result B[0] of Line 3 and bitmask LOWEST[0]. In this example, we can observe that the parent bit is set to 1 since all the child bits are 1 in Y:

B[0]	0	1	1	1	1	1	1	0	0
+ LOWEST[0]	0	0	0	0	0	0	1	0	0
C[0]	1	0	0	0	0	0	0	0	0

At Line 5, we leave the parent bit by AND-ing the result $C[0]$ of Line 4 with bitmask PARENT[0], and obtain the result $Z[0] = 10000000$ for level $k = 0$:

$C[0]$	1	0	0	0	0	0	0	0	0
& PARENT[0]	1	0	0	0	0	0	0	0	0
$Z[0]$	1	0	0	0	0	0	0	0	0

Similarly, we can obtain the result $Z[1] = 000000100$ for level $k = 1$. At Line 6, we gather $Z[0]$ and $Z[1]$ to Z by OR-ing them, and we have $Z = 100000100$ that represents $\{8, 9\} \subseteq V(P)$. At Line 7, we set each bit for leaves 1, 2, 3, 5, 6, and 7 to 1 by OR-ing the result Z of Line 6 with bitmask LEAF, and filter out the incorrect candidates by AND-ing X with $Z \mid$ LEAF. We finally have the desired result $Z = 101000000$, and find an occurrence of P in T at text node 16 w.r.t unordered pseudo-tree matching since Z contains $root(P) = 9$.

5. Simpler Implementation of Tree Aggregation using Monotone Routing

In this section, we show another algorithm for the tree aggregation operation using monotone routing technique that runs in $O(m \log(w)/w)$ time and space on $RAM(+, \gg)$ for a bitmask of bit-length m.

We consider the small pattern case such that $m \leq w$. Let P be a pattern tree of size m. We number the nodes of P in the *level-order*, where the nodes of P are partitioned into levels from the root to the leaves, and then numbered consecutively from left to right at each level. Then, we define the position mapping $pos : V(P) \to [1..m]$ from the nodes of P to the interval $[1..m]$ such that $pos(x) = m - i + 1$, where i is the numbering for a node x in P obtained by the level-order. Moreover, we define the *parent ordinal* of an internal node $x \in internal(P)$, denoted by $ord(x)$, to be the number of the internal nodes preceding to x in the level-order plus one. This mapping pos computed from a level-order numbering satisfies the monotone property that an internal node as a parent is numbered greater than all of its children, and ensures all the children of a node to be numbered consecutively.

Let v be the current node in T. Then, we denote by Z and $Y \in \{0, 1\}^m$, respectively, the bitmasks for the embedding set $Emb^{P,T}(v)$ for v and the child set $S = \bigcup_{w \in children(v)} Emb^{P,T}(w)$, the union of the sets $Emb^{P,T}(w)$ for all children w of v. The i-th bit of Z is 1 if there is an embedding of the subtree rooted at x with $pos(x) = i$ such that x is mapped to v and 0 otherwise. Now, we compute the bitmask Z for the embedding set $Emb^{P,T}(v)$ at node v from the bitmask Y for the child set. The algorithm is described as follows.

(1) The algorithm is given the bitmask Y for the *child set* $S = \bigcup_{w \in children(v)} Emb^{P,T}(w)$, which the union of $Emb^{P,T}(w)$ for all children w of v.

(2) Divide Y into segments that represent groups of sibling nodes. For an internal node x in P, we define the set of bit-positions for the segment for x by $seg(x) = pos(children_P(x)) \subseteq [1..m]$. We number the segment $seg(x)$ by its parent ordinal $ord(x)$. For each segment, we have to compute the logical AND of all bits in that segment. We can do this by adding 1 to the least significant bit in that segment, namely $\min(seg(x))$, and looking to see if there is a carry out of the segment. To guard against segments interfering with each other, we compute this logical AND for all even-numbered segments in one step, and then for all odd-numbered segments in another step. When computing the logical AND for all even-numbered segments, we zero out all odd-numbered segments, and vice versa; this prevents carry propagation. It is easy to take the above result and rearrange it so that the result of the logical AND is at the leftmost position of a segment, namely $\max(seg(x))$, this is just a single shift. Clearly this takes O(1) time.

The above idea is implemented in bit-parallel manner as follows. Let $i \in \{0, 1\}$ denotes the current step, where $i = 0$ and $i = 1$ indicate the steps for even and odd segments, respectively. In the preprocessing, we build the bitmasks SEGMENTi, LOWESTi, and HIGHESTi defined as follows:

$$\texttt{SEGMENT}i = NUM(\{\ j \mid j \in seg(x), x \in internal(P), ord(x) = i \bmod 2\ \}).$$
$$\texttt{LOWEST}i = NUM(\{\ j \mid j = \min(seg(x)), x \in internal(P), ord(x) = i \bmod 2\ \}).$$
$$\texttt{HIGHEST}i = NUM(\{\ j \mid j = \max(seg(x)), x \in internal(P), ord(x) = i \bmod 2\ \}).$$

In the runtime, we then perform the following code:

$$\texttt{AGGR}i \leftarrow (((Y \ \&\ \texttt{SEGMENT}i) + \texttt{LOWEST}i) \gg 1)\ \&\ \texttt{HIGHEST}i;$$

(3) After OR-ing AGGR0 and AGGR1, we have a word where an *interesting bit* is associated with each node i such that i is the leftmost child of its parent (representing the logical AND of its siblings), and all other bits are 0. If i's parent is at position j, then we now want to move this bit to position j, and this must be done in parallel for all interesting bits. This is, however, a *monotone routing problem*, and Section 3.4.3 of Leighton [22] gives a simple two-phase algorithm for doing this routing on the hypercube routing networks. Andersson *et al.* [3] employ this algorithm to move a collection of bits in a monotonic way. In the algorithm, the first phase, called *packing*, is to group all the interesting bits together consecutively at the start of a word, and then the second phase, called *spreading*, is to route all the interesting bits to their final destinations.

At packing phase, we first compute the binary expansion $bin(dist(x)) = b_{K-1} \cdots b_0 \in \{0,1\}^K$ of the move distance for each segment $seg(x)$ by $dist(x) = (m - ord(x) + 1) - \max(seg(x))$, where the index i of bit b_i ranges from 0 to $K-1$. For every k from 0 to $K-1$, we move the interesting bit leftward by 2^k bits using left shift "$\ll$" if $b_k = 1$ and keep it unchanged otherwise by changing k from 0 to $K-1$. This can be implemented as follows. In the preprocessing, we define the following bitmask for every k from 0 to $K-1$:

$$\texttt{PACK}[k] = NUM(\{\ j \mid j = pos\_pack(x,k), x \in internal(P)\ \}),$$

where for every k, the position $pos\_pack(x,k)$ is defined by: $pos\_pack(x,0) = \max(seg(x))$ and $pos\_pack(x,k) = pos\_pack(x,k-1) + b_{k-1}2^{k-1}$ for $0 < k \leq K-1$. In the runtime, we execute the following code for every k from 0 to $K-1$:

$$Z \leftarrow ((Z\ \&\ \texttt{PACK}[k]) \ll 2^k) \mid (Z\ \&\ \sim \texttt{PACK}[k]);$$

This code correctly packs the interesting bit x leftward from $\max(seg(x))$ to $m - ord(x) + 1$ in $O(\log m)$ time using $O(\log m)$ bitmasks.

At spreading phase, we move the interesting bit for x rightward from the current position $m - ord(x) + 1$ to the final destination $pos(x)$ by using the binary expansion $bin(dist(x)) = b_{K-1} \cdots b_0 \in \{0,1\}^K$ of the move distance $dist(x) = (m - ord(x) + 1) - pos(x)$ in a similar way as above. In the preprocessing, the bitmask for every k from $K-1$ to 0 is give by:

$$\texttt{SPREAD}[k] = NUM(\{\ j \mid j = pos\_spread(x,k), x \in internal(P)\ \}),$$

where for every $k = K-1, \ldots, 0$, the position $pos\_spread(x,k)$ is defined by: $pos\_spread(x,K-1) = m - ord(x) + 1$ and $pos\_spread(x,k) = pos\_spread(x,k-1) + b_k 2^k$ for $0 < k \leq K-1$. In the runtime, we execute the following code for every k from $K-1$ to 0:

$$Z \leftarrow ((Z\ \&\ \texttt{SPREAD}[k]) \gg 2^k) \mid (Z\ \&\ \sim \texttt{SPREAD}[k]);$$

As shown in Lemma 6.4 of [3], the above code correctly performs the monotone routing from the source bits to the destination bits in $O(\log m)$ time using $O(\log m)$ bitmasks. Although the lemma from [3] cannot be used directly, we can show that the conflicts between any pair of bits never occur during the steps from the discussion in Chapter 3.4 of [22].

(4) Finally, the bits are AND-ed with the bitmask X for the condition (E0) and returned as the answer:

$$Z \leftarrow X\ \&\ (Z \mid \texttt{LEAF});$$

In Fig. 7, we show the bit-parallel algorithm for the tree aggregation operation based on monotone routing. Combining the above arguments, we have the main result of this section.

Theorem 2. Suppose that $m \leq w$. Then, the operation $\mathsf{TreeAggr}_P$ can be implemented to run in $O(\log m)$ time using $O(\log m)$ space and $O(m \log m)$ preprocessing on $RAM(+, \gg)$.

procedure TreeAggr$_P$(X: a bitmask representing a candidate set R, Y: a bitmask representing a child set S):
Note: See Section 5 for the definition of bitmasks SEGMENTi, LOWESTi, HIGHESTi, PACK[$\cdot$], and SPREAD[$\cdot$] for every $i \in \{0, 1\}$;
Output: the set $Z = NUM(\{\ j \in [1..m] \mid x \in R, children_P(x) \subseteq S, j = pos(x)\ \})$;
1: AGGR0 $\leftarrow$ (((Y & SEGMENT0) + LOWEST0) $\gg$ 1) & HIGHEST0;
2: AGGR1 $\leftarrow$ (((Y & SEGMENT1) + LOWEST1) $\gg$ 1) & HIGHEST1;
3: $Z \leftarrow$ AGGR0 | AGGR1;
4: **for** every level $k = 0, \ldots, \lceil \log m \rceil - 1$ **do**
5: $\quad Z \leftarrow ((Z$ & PACK[k]) $\ll 2^k) \mid (Z$ & $\sim$ PACK[k]);
6: **for** every level $k = \lceil \log m \rceil - 1, \ldots, 0$ **do**
7: $\quad Z \leftarrow ((Z$ & SPREAD[k]) $\gg 2^k) \mid (Z$ & $\sim$ SPREAD[k]);
8: $Z \leftarrow X$ & (Z | LEAF);
9: **return** Z;

Figure 7: A bit-parallel implementation of the tree aggregation operation based on monotone routing.

6. Extension to 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 $DescEmb^{P,T}(v)$, called the *descendant embedding set* and the auxiliary set $SubEmb^{P,T}(v)$ are defined by:

$$DescEmb^{P,T}(v) = \{\ x \in [1..m] \mid (\exists \phi)\ \phi \in UTH(P(x), T) \wedge \phi(x) = v\ \}. \tag{2}$$
$$SubEmb^{P,T}(v) = \{\ x \in [1..m] \mid (\exists \phi)\ \phi \in UTH(P(x), T) \wedge (\exists w)\ w \geq_P v \wedge \phi(x) = w\ \}.$$

Lemma 9. *For any pattern tree P and text tree T, we have the following properties:*

(1) *For every $x \in V(P)$ and $v \in V(T)$, $x \in DescEmb^{P,T}(v)$ if and only if (i) $label_P(x) = label_T(v)$, and (ii) $children_P(x) \subseteq \bigcup_{1 \leq j \leq arity(v)} SubEmb^{P,T}(v[j])$.*
(2) *For any $v \in V(T)$, $SubEmb^{P,T}(v) = DescEmb^{P,T}(v) \cup \bigcup_{1 \leq j \leq arity(v)} SubEmb^{P,T}(v)$.*

Proof. Part (1): The proof proceeds similarly to the case for UPTM in Lemma 1 as follows. If $x \in DescEmb^{P,T}(v)$ then (i) is obvious. Moreover, it follows from the definition of UPH that for every $1 \leq i \leq arity(x)$, there exists some $h(i)$ and some descendant w_i of $v[h(i)]$ such that $x[i] \in DescEmb^{P,T}(v)$. Since the existence of such w_i is equivalent to that $x[i] \in SubEmb^{P,T}(b)$ by definition, the condition (ii) immediately follows. By a similar argument, the converse is also proved. Part (2): For any v and w, it is obvious that w is a descendant of v if and only if either $w = v$ or w is a proper descendant of v. Therefore, $x \in DescEmb^{P,T}(v)$ if and only if either $x \in SubEmb^{P,T}(v)$, or $(\exists w)$ $w \geq_P v \wedge x \in DescEmb^{P,T}(v)$. Hence, the claim immediately follows from the definition. □

From the above decomposition lemma, we can develop an efficient bit-parallel algorithm, called BP-MatchUTH, for the unordered tree homeomorphism problem (UTH) based on a dynamic programming algorithm similar to MatchUPTM in Section 3 and the bit-parallel implementation of set manipulation operations including LabelMatch$_P$ and TreeAggr$_P$. We can obtain an algorithm MatchUTH for UTH from the algorithm MatchUPTM by replacing Line 10 of the recursive subprocedure VisitUPTM with the following line:

10: **return** Union(R, S); $\{R \cup S = SubEmb^{P,T}(v)\}$

Then, we have the following theorem. The proof is similar to that of Theorem 3 and is omitted.

Theorem 4 (complexity of the unordered tree homeomorphism problem). *A modified version of the 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 P and text tree T, h is the height of T, and w is the word length.

7. Relationship to a fragment of XPath queries

In this section, we show the correspondence of tree patterns in our UPTM and UTH problems to a fragment of XPath query language [5, 32]. *Core XPath* (CXP, for short), introduced by Gottlob and Koch [12], is the logical core of XPath language, whose query expressions consist of node tests, child and nextsibling axes and their inverse and the transitive closures, and Boolean predicates over expressions [11]. Below, we will give the syntax and the semantics of a fragment of Core XPath queries, called conjunctive Core XPath queries, following the framework of [11, 12]. We note that our definitions for the syntax and semantics are essentially the restriction of the original version in [11] to although they look slightly different.

Axes relations are binary relations R over text nodes in T, where $R \subseteq V(T) \times V(T)$. The *child* and *descendant* axes are binary relations *child* and *desc* defined as follows: for every nodes v and w, $\langle v, w\rangle \in child$ ($\langle v, w\rangle \in desc$, resp.) iff w is a child (descendant, resp.) of v. Let *axis* be either *child* or *desc*. A *conjunctive Core XPath query* (CCXP, for short) with $axis \in \{child, desc\}$ is a Core XPath query that has *axis* as only axis relation and consists of constant labels as node test, logical AND as its connectives, and has no *axis* at the root position. We denote by $CCXP[axis]$ the class of all CCXP queries with *axis*. Formally, we define the language of $CCXP[axis]$ by its abstract EBNF syntax as follows:

$$\begin{array}{lcl} exp & := & \alpha \mid \alpha \text{ '[' } pred \text{ ']'} \\ pred & := & axis \text{ '::' } exp \mid pred \text{ '\&' } pred \end{array}$$

where *exp* is a query expression, *pred* is a predicate, $axis \in \{child, desc\}$, and $\alpha \in \Sigma$ is a label. Note that a CCXP query is an expression with tree-shape. Since CCXP has no axis at the root and no concatenation operator "/" for paths to define the target node, their semantics is naturally based on the root occurrences. For example, the CCXP query corresponding to the pattern tree P in Fig. 1 is given by:

$$Q_1 = A[\, child::B[\, child::A \,\&\, child::B \,\&\, child::D\,] \,\&\, child::C \,\&\, child::B[\, child::B \,\&\, child::C\,]\,].$$

The semantics of a CCXP expression *exp* is given by the set $\mathcal{E}[pred]$ of nodes at which the root of the expression matches. Formally, the semantics of queries is defined as follows:

$$\begin{array}{rcl} \mathcal{E}[\,\alpha\,] & = & \{\, v \mid label_T(v) = \alpha \,\} \\ \mathcal{E}[\,\alpha[p]\,] & = & \{\, v \mid label_T(v) = \alpha \,\wedge\, v \in \mathcal{E}[p] \,\} \\ \mathcal{E}[\,axis::e\,] & = & \{\, v \mid \exists w, \langle v, w\rangle \in axis \,\wedge\, w \in \mathcal{E}[e] \,\} \\ \mathcal{E}[\,p_1 \,\&\, p_2\,] & = & \mathcal{E}[\,p_1\,] \,\cap\, \mathcal{E}[\,p_2\,] \end{array}$$

where e is an expression, p is a predicate, v and w range over text nodes in T. The *evaluation problem* for $CCXP[axis]$ is the problem of computing the elements of $\mathcal{E}[\pi]$ for a query π and a text tree T.

We transform a query $\pi \in CCXP[axis]$ to the pattern tree $\psi(\pi)$ as: (a) If $e = \alpha$ then $\psi(e) = \alpha$; (b) If $e = \alpha[\,\&_{i=1}^{n} axis_i::e_i\,]$ for some $n \geq 1$ then $\psi(e)$ is the pattern tree P such that for $root(P) = v$, (i) $label_P(v) = \alpha$, and (ii) the i-th immediate subtree of v is given by $P(v[i]) = \psi(e_i)$ for every $i = 1, \ldots, arity(v)$. The converse transformation can be given similarly. The following lemma shows the equivalence of our tree matching problems and the corresponding fragments of conjunctive Core XPath queries.

Lemma 10. *Let T be any text tree on Σ, $axis \in \{child, desc\}$, $\pi \in CCXP[axis]$ be any conjunctive CXP query with axis as only axis, and $P_\pi = \psi(\pi)$.*

(1) *If axis = child, then for any $v \in V(T)$, $v \in \mathcal{E}[\pi]$ if and only if v is an root occurrence of P_π in T w.r.t. UPTM.*
(2) *If axis = desc, then for any $v \in V(T)$, $v \in \mathcal{E}[\pi]$ if and only if v is an root occurrence of P_π in T w.r.t. UTH.*

From the above lemma, we have the following corollary of Theorem 3 and Theorem 4.

Corollary 5. *The evaluation problems for the classes of conjunctive Core XPath queries with child axis only and descendant axis only can be solvable in the same time, space, and preprocessing complexities as in Theorem 3 and Theorem 4, respectively, where the size m of a query π is measured by the number of labels in π.*

8. Conclusion

In this paper, we considered the unordered pseudo-tree matching problem and the unordered tree homeomorphism problem. As main results, we presented efficient bit-parallel algorithms for both problems that run 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 integer addition, where m is the number of nodes in a pattern tree, n and h are the number of nodes and the height of a text tree, and w is the word length.

The tree matching for UPTM and UTH correspond to evaluation of fragments of XPath queries with child axis only and with descendant axis only [14], respectively. As future work, applications of our algorithms to tree matching for practical subclasses of XPath queries are interesting problems. XPath with mixture of child axis, descendant axis, and arbitrary Boolean expressions at predicates are some examples of such subclasses. In particular, there is an extension of XPath and XML Schema for ordered trees that allows regular expressions over subtrees [23, 24, 27]. To this direction, there is efficient regular expression matching using Boolean operations and integer addition [6, 17, 26]. Therefore, it is another interesting problem to combine such regular expression matching and tree matching algorithms for XPath evaluation.

The tree aggregation operation based on a monotone and non-overlapping pair of position and level mappings satisfies the condition that each bit of the output word depends only on bits at and to the right of each input word. Interestingly, it was shown by Warren Jr. [33] that this condition is necessary and sufficient to determine whether a given function f mapping words to words can be implemented with a sequence of Boolean operations, additions, and subtractions. See also [20, 34] for details. Unfortunately, the length of such a sequence of operations can be exponential in the word length w in general. Therefore, it would be an interesting problem to identify a subclass of such functions with left-to-right dependency that are efficiently realizable in bit-parallel manner by extending the approach in this paper.

Acknowledgements

The authors thank anonymous reviewers for their comments which improved the correctness and the presentation of this paper very much. The authors thank Takeaki Uno for helpful discussions about monotone and non-overlapping mappings, and also would like to thank Tatsuya Akutsu, Kouichi Hirata, Akira Ishino, Takuya Kida, Shin-ichi Minato, Yoshio Okamoto, Hiroshi Sakamoto, Shinichi Shimozono, Kilho Shin, Masayuki Takeda, Koji Tsuda, Akihiro Yamamoto, and Thomas Zeugmann for their fruitful discussions and valuable comments on the earlier version of this paper and related topics. This research was partly supported by MEXT Grant-in-Aid for Scientific Research (A), 20240014, FY2008–2011, and MEXT/JSPS Global COE Program, FY2007–2011.

References

[1] S. Abiteboul, P. Buneman, D. Suciu, Data on the Web, Morgan Kaufmann, 2000.
[2] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.
[3] A. Andersson, T. Hagerup, S. Nilsson, R. Raman. Sorting in Linear Time?, J. Computer System Sciences 57 (1998) 74–93.
[4] R. A. Baeza-Yates, G. H. Gonnet, A new approach to text searching, Commun. ACM 35 (10) (1992) 74–82.
[5] M. Benedikt, C. Koch, XPath leashed, ACM Comput. Surv. 41(1) (2008) 1–54.
[6] P. Bille, New algorithms for regular expression matching, in: Proceedings of the 33rd International Colloquium on Automata, Languages and Programming (ICALP'06), in: Lecture Notes in Computer Science, vol. 4051, Springer-Verlag, 2006, pp. 643–654.
[7] P. Bille, A survey on tree edit distance and related problems, Theoretical Computer Science, 337(1–3) (2005) 217–239.
[8] P. Bille, I. L. Gørtz, The tree inclusion problem: In optimal space and faster, ACM Transactions on Algorithms, 7(3), (2011) 1–47.
[9] N. Bruno, D. Srivastava, N. Koudas, Holistic twig joins: Optimal XML pattern matching, in: Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data (SIGMOD'02), ACM, 2002, pp. 310–321.
[10] W. Chen, More efficient algorithm for ordered tree inclusion, J. Algorithms 26 (2) (1998) 370-385.

[11] G. Gottlob, C. Koch, R. Pichler, Efficient algorithms for processing XPath queries, ACM Trans. Database Syst. 30 (2) (2005) 444–491.
[12] G. Gottlob, C. Koch, Monadic queries over tree-structured data, in: Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS'02). IEEE, 2002, pp. 189–202.
[13] G. Gottlob, C. Koch, K. U. Schulz, Conjunctive queries over trees, J. ACM 53(2) (2006) 238–272.
[14] M. Götz, C. Koch, W. Martens, Efficient algorithms for descendant-only tree pattern queries, Inf. Syst. 34 (7) (2009) 602–623.
[15] C. M. Hoffmann, M. J. O'Donnell, Pattern matching in trees, J. ACM 29(1) (1982) 68–95.
[16] C. Jordan, Sur les assemblages de lignes, Journal für die Reine und Angewandte Mathematik 70 (1869) 185–190.
[17] Y. Kaneta, S. Minato, H. Arimura, Fast bit-parallel matching for network and regular expressions, in: Proceedings of the 17th International Symposium on String Processing and Information Retrieval (SPIRE'10), in: Lecture Notes in Computer Science, vol. 6393, Springer-Verlag, 2010, pp. 372–384.
[18] P. Kilpeläinen, Tree matching problems with applications to structured text databases, Ph.D Thesis, Report A-1992-6, Department of Computer Science, University of Helsinki, November 1992.
[19] P. Kilpeläinen, H. Mannila, Ordered and unordered tree inclusion, SIAM J. Comput. 24 (2) (1995) 340–356.
[20] D. E. Knuth, The Art of Computer Programming, Volume 4, Fascicle 1: Bitwise Tricks & Techniques; Binary Decision Diagrams, Addison-Wesley, 2009.
[21] R. Kosaraju, Efficient tree pattern matching, in: Proceedings of the 30th Annual IEEE Symposium on Foundations of Computer Science (FOCS'89), IEEE, 1989, pp. 178–183.
[22] F. T. Leighton, An Introduction to Parallel Algorithms and Architectures, Morgan Kaufmann, 1992.
[23] W. Martens, F. Neven, T. Schwentick, G. J. Bex, Expressiveness and complexity of XML schema, ACM Trans. Database Syst., 31 (3) (2006) 770–813.
[24] M. Murata, D. Lee, M. Mani, K. Kawaguchi, Taxonomy of XML schema languages using formal language theory, ACM Trans. Internet Technology, 5 (4) (2005) 660–704.
[25] E. W. Myers, A four Russian algorithm for regular expression pattern matching, J. ACM 39 (2) (1992) 430–448.
[26] G. Navarro, M. Raffinot, Flexible Pattern Matching in Strings: Practical On-Line Search Algorithms for Texts and Biological Sequences, Cambridge University Press, 2002.
[27] F. Neven, T. Schwentick, Expressive and efficient pattern languages for tree-structured data, in: Proceedings of the 19th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS'00), 2000, pp.145–156.
[28] D. Olteanu, T. Furche, F. Bry, Evaluating complex queries against xml streams with polynomial combined complexity, in: Proceedings of the British National Conference on Databases (BNCOD'04), in: Lecture Notes in Computer Science, vol. 3112, Springer-Verlag, 2004, pp. 31–44.
[29] H. Tsuji, A. Ishino, M. Takeda, A bit-parallel tree matching algorithm for patterns with horizontal VLDC's, in: Proceedings of the 12th International Conference on String Processing and Information Retrieval (SPIRE'05), in: Lecture Notes in Computer Science, vol. 3772, Springer-Verlag, 2005, pp. 388–398.
[30] G. Valiente, Constrained tree inclusion, J. Discrete Algorithms 3 (2–4) (2005) 431–447.
[31] S. Vigna, Broadword implementation of rank/select queries, in Processings of the 7th International Workshop on Experimental Algorithms (WEA'08), in: Lecture Notes in Computer Science, vol. 5038, Springer-Verlag, 2008, pp. 154–168.
[32] W3C, Extensive Markup Language (XML) 1.1 (Second Edition), W3C Recommendation, `http://www.w3.org/TR/REC-xml`, 2006.
[33] H. S. Warren Jr., Functions realizable with word-parallel logical and two's-complement addition instructions, Commun. ACM 20 (6) (1977) 439–441.
[34] H. S. Warren Jr., Hacker's Delight, Addison-Wesley, 2003.
[35] S. Wu, U. Manber, Fast text searching: allowing errors, Commun. ACM 35 (10) (1992) 83–91.
[36] H. Yamamoto, D. Takenouchi, Bit-parallel tree pattern matching algorithms for unordered labeled trees, in: Proceedings of the 11th International Symposium on Algorithms and Data Structures (WADS'09), in: Lecture Notes in Computer Science, vol. 5664, Springer-Verlag, 2009, pp. 554–565.