Proceedings of the of the 17th Conference on Computer Science and Intelligence Systems pp. 351–356
DOI: 10.15439/2022F113
ISSN 2300-5963 ACSIS, Vol. 30

Subcaterpillar Isomorphism: Subtree Isomorphism Restricted Pattern Trees To Caterpillars

Tomoya Miyazaki
Graduate School of Computer Science and Systems Engineering
Kyushu Institute of Technology
Kawazu 680-4, Iizuka 820-8502, Japan
Email: miyazaki.tomoya481@mail.kyutech.jp

Kouichi Hirata
Department of Artificial Intelligence
Kyushu Institute of Technology
Kawazu 680-4, Iizuka 820-8502, Japan
Email: hirata@ai.kyutech.ac.jp

***Abstract*—In this paper, we investigate a *subcaterpillar isomorphism* that is a problem, for a rooted labeled caterpillar P and a rooted labeled tree T, of determining whether or not there exists a subtree in T which is isomorphic to P. Then, we design two algorithms to solve the subcaterpillar isomorphism for a caterpillar P and a tree T in (i) $O(p+tDh\sigma)$ time and $O(Dh)$ space and in (ii) $O(p+tD\sigma)$ time and $O(D(h+H))$ space, respectively. Here, p is the number of vertices in P, t is the number of vertices in T, h is the height of P, H is the height of T, σ is the number of alphabets for labels and D is the degree of T. Furthermore, we give experimental results of the two algorithms for artificial data and real data.**

I. Introduction

THE PATTERN matching for tree-structured data such as HTML and XML documents for web mining or DNA and glycan data for bioinformatics is one of the fundamental tasks for information retrieval or query processing. As such pattern matching for *rooted labeled unordered trees* (a *tree*, for short), a *subtree isomorphism* is the problem of determining, for a *pattern tree* P and a *text tree* T, whether or not there exists a subtree of T which is isomorphic to P. It is known that the subtree isomorphism can be solved in $O(p^{1.5}t/\log p)$ time [10], where p is the number of vertices in P and t is the number of vertices in T. On the other hand, it cannot be solved in $O(t^{2-\varepsilon})$ time for every ε $(0<\varepsilon<1)$ under SETH [1].

In this paper, we focus on *subcaterpillar isomorphism* that is a subtree isomorphism when P is a *rooted labeled caterpillar* (a *caterpillar*, for short) (*cf.*, [3]). The caterpillar is an unordered tree transformed to a rooted path after removing all the leaves in it. The caterpillar provides the structural restriction of the tractability of computing the *edit distance* [8] and *inclusion problem* [7] for unordered trees.

It is known that the problem of computing the edit distance between unordered trees is MAX SNP-hard [11]. This statement also holds even if two trees are binary, the maximum height is at most 3 or the cost function is the unit cost function [2], [4]. On the other hand, we can compute the edit distance between caterpillars in $O(n+H^2\sigma^3)$ time in the general cost function and $O(n+H^2\sigma)$ time under the unit cost function, where n is the total number of vertices of the two caterpillars, H is the maximum height of the two caterpillars and σ is the number of alphabets for labels in the two caterpillars [8][1].

It is known that the inclusion problem of determining whether or not a text tree T achieves to a pattern tree P by deleting vertices in T is NP-complete [6]. This statement also holds even if P is a caterpillar [6]. On the other hand, if both P and T are caterpillars, then we can solve the inclusion problem in $O(p+t+(h+H)\sigma)$ time, where h is the height of P and H is the height of T [7][2].

In this paper, we design two algorithms to solve the subcaterpillar isomorphism in (i) $O(p+tDh\sigma)$ time and $O(Dh)$ space and (ii) $O(p+tD\sigma)$ time and $O(D(h+H))$ space, respectively. Here, D is the degree of T. Since there may exist many matching positions that match P in T when P is much smaller than T, the above algorithms also output all of such positions. Hence, under the assumption that $p<t$, $h \ll t$ and $h<H$, the algorithm (i) runs in $O(tD\sigma)$ time and $O(Dh)$ space and the algorithm (ii) runs in $O(tD\sigma)$ time and $O(DH)$ space.

Note that both algorithms do not use the maximum cardinality matching algorithm for bipartite graphs [5], which is essential for the subtree isomorphism algorithm [10]. Also we cannot apply the proof of the SETH-hardness in [1] when a pattern tree P is a caterpillar.

Furthermore, by implementing the algorithms (i) and (ii), we give experimental results of the two algorithms for artificial data and real data. Then, we confirm that, whereas the algorithm (ii) is faster than the algorithm (i) as same as the theoretical results for artificial data of which number of matching positions is large, the algorithm (i) is faster than the algorithm (ii) for real data.

II. Preliminaries

A *tree* is a connected graph without cycles. For a tree $T=(V,E)$, we denote V and E by $V(T)$ and $E(T)$. We sometimes

[1]The time complexity represented in [8] is $O(H^2\lambda^3)$ time and $O(H^2\lambda)$ time, where λ is the maximum number of leaves in the two caterpillars. Since $O(\lambda^3)$ and $O(\lambda)$ in them are corresponding to the time complexity of computing the multiset edit distances under the general and the unit cost functions (*cf.* [9]), we can replace λ with σ, by storing the labels occurring in the leaves. Also, in order to compare the time complexity of this paper, we add $O(n)$ as the initialization of the algorithm, containing the above storing.

[2]The time complexity represented in [7] is $O((h+H)\sigma)$ time. In order to compare the time complexity of this paper, we add $O(p+t)$ as the initialization of the algorithm.

denote $v \in V(T)$ by $v \in T$. A *rooted tree* is a tree with one vertex r chosen as its *root*, which we denote by $r(T)$.

For each vertex v in a rooted tree with the root r, let $UP_r(v)$ be the unique path from v to r. The *parent* of $v(\neq r)$, which we denote by $par(v)$, is its adjacent vertex on $UP_r(v)$ and the *ancestors* of $v(\neq r)$ are the vertices on $UP_r(v) \setminus \{v\}$. We denote $u < v$ if v is an ancestor of u, and we denote $u \leq v$ if either $u < v$ or $u = v$. The parent and the ancestors of the root r are undefined. We say that u is a *child* of v if v is the parent of u, and u is a *descendant* of v if v is an ancestor of u. We denote the set of all children of v by $ch(v)$. Two vertices with the same parent are called *siblings*. A *leaf* is a vertex having no children and we denote the set of all the leaves in T by $lv(T)$. We call a vertex that is not a leaf an *internal vertex*.

For a rooted tree $T = (V, E)$ and a vertex $v \in T$, the *complete subtree of* T *at* v, denoted by $T(v)$, is a rooted tree $S = (V', E')$ such that $r(S) = v$, $V' = \{w \in V \mid w \leq v\}$ and $E' = \{(u, w) \in E \mid u, w \in V'\}$.

The *height* $h(v)$ of a vertex v is defined as $|UP_r(v)| - 1$ and the *height* $h(T)$ of T is the maximum height for every vertex $v \in T$. The *degree* $d(v)$ of a vertex v is the number of the children of v, and the *degree* $d(T)$ of T is the maximum degree for every vertex in T.

We say that a rooted tree is *ordered* if a left-to-right order among siblings is given; *Unordered* otherwise. For a fixed finite alphabet Σ, we say that a tree is *labeled* over Σ if each vertex is assigned a symbol from Σ. We denote the label of a vertex v by $l(v)$, and sometimes identify v with $l(v)$. In this paper, we call a rooted labeled unordered tree over Σ a *tree*, simply.

In this paper, we often represent a rooted labeled unordered tree as a rooted labeled *ordered* tree under a fixed order of siblings. Then, for a rooted labeled ordered tree T, a vertex v in T and its children $v_1, \dots, v_i$, the *postorder traversal* of $T(v)$ is obtained by first visiting $T(v_k)$ $(1 \leq k \leq i)$ and then visiting v. The *postorder number* of $v \in T$ is the number of vertices preceding v in the postorder traversal of T.

Definition 1: Let T and S be trees.

1) We say that T is a *subtree* of S, denoted by $T \preceq S$, if T is a tree such that $V(T) \subseteq V(S)$ and $E(T) = \{(v, w) \in E(S) \mid v, w \in V(T)\}$.
2) We say that T and S are *isomorphic*, denoted by $T \simeq S$, if $T \preceq S$ and $S \preceq T$.
3) We say that T is a *subtree isomorphism* of S, denoted by $T \trianglelefteq S$, if there exists a tree $S' \preceq S$ such that $T \simeq S'$.

In this paper, we deal with a *subtree isomorphism problem* of P for T whether or not $P \trianglelefteq T$ for trees P and T. We call P a *pattern tree* and T a *text tree*. Then, the following theorem holds.

Theorem 1 (Shamir & Tsur [10]): Let P and T be trees where $p = |P|$ and $t = |T|$. Then, the problem of determining whether or not $P \trianglelefteq T$ is solvable in $O(p^{1.5}t/\log p)$ time.

As the restricted form of trees, we introduce a *rooted labeled caterpillar* (a *caterpillar*, for short) as follows.

Definition 2: We say that a tree is a *caterpillar* (*cf.* [3]) if it is transformed to a rooted path after removing all the leaves in it. For a caterpillar C, we call the remained rooted path a *backbone* of C and denote it by $bb(C)$.

It is obvious that $r(C) = r(bb(C))$ and $V(C) = V(bb(C)) \cup lv(C)$ for a caterpillar C, that is, every vertex in a caterpillar is either a leaf or an element of the backbone.

III. Algorithms for Subcaterpillar Isomorphism

In this section, we focus on a *subcaterpillar isomorphism* that is a subtree isomorphism when P is a caterpillar. In other words, we focus on the problem of whether or not $P \trianglelefteq T$ for a caterpillar P and a tree T. Throughout of this section, we refer $p = |P|$, $t = |T|$, $h = h(P)$, $H = h(T)$, $D = d(T)$ and $\sigma = |\Sigma|$.

For a pattern caterpillar P, we refer the backbone of P to a sequence $\langle v_1, \dots, v_n \rangle$ such that $(v_i, v_{i+1}) \in E(P)$ and $v_n = r(P)$. We denote the children of v_i by $ch(v_i)$.

On the other hand, for a text tree T, we refer the vertices in T to $w_1, \dots, w_m$ in postorder traversal. We denote the height of w_j by $h(w_j)$ and the set of children of w_j by $ch(w_j)$.

Let P be a pattern caterpillar and T a text tree such that $P \trianglelefteq T$. Also let $P' \preceq T$ be a subcaterpillar in T such that $P \simeq P'$ and $bb(P') = \langle v'_1, \dots, v'_n \rangle$, where $v'_n = r(P')$. Then, we call the postorder number j such that $v'_1 = w_j$ in T a *matching position* of P in T.

Example 1: Consider a pattern caterpillar P and a text tree T in Figure 1. Here, the number assigned to every vertex in T denotes the postorder number. Also v_i denotes the backbone. Then, $\{6, 8, 16\}$ is the set of all the matching positions of P in T. The corresponding backbones to P in T are $\langle 6, 8, 9 \rangle$, $\langle 8, 9, 18 \rangle$ and $\langle 16, 17, 18 \rangle$.

Fig. 1. A pattern caterpillar P and a text tree T in Example 1.

To design the algorithm to determine subcaterpillar isomorphism, we use a *multiset* of labels in order to compare two sets of vertices. A *multiset* on Σ is a mapping $S : \Sigma \to \mathbf{N}$. For a set V of vertices, we denote the multiset of labels occurring in V by $\widetilde{V}$. Then, it is necessary for the subcaterpillar isomorphism to check whether or not $\widetilde{ch(v_i)} \subseteq \widetilde{ch(w_j)}$ for $v_i \in bb(P)$ and $w_j \in T$. It is realized to check $\left(\widetilde{ch(v_i)}\right)(a) \leq \left(\widetilde{ch(w_j)}\right)(a)$ for every $a \in \Sigma$ in $O(\sigma)$ time (*cf.* [9]).

Then, we design the algorithm SubcatIso in Algorithm 1 to determine whether or not $P \trianglelefteq T$. Here, the algorithm SubcatIso output all of the matching positions if $P \trianglelefteq T$. Then, it holds that no matching point is output if $P \ntrianglelefteq T$.

procedure SUBCATISO(P, T)
/* P : caterpillar such that $bb(P) = \langle v_1, \ldots, v_n \rangle$ */
/* T : tree consisting of vertices $w_1, \ldots, w_m$ in postorder traversal */
1 **for** $i = 1$ **to** $n - 1$ **do** $match[i] \leftarrow \emptyset$;
2 **for** $j = 2$ **to** m **do**
3 **if** $h(w_{j-1}) = h(w_j) + 1$ **then**
/* $w_j = par(w_{j-1})$ */
4 **for** $i = n - 1$ **downto** 1 **do**
5 **if** $match[i] \neq \emptyset$ **then**
6 **foreach** $k \in match[i]$ **do**
7 **if** $h(w_k) = h(w_j) + i$ **then**
/* $w_j = par(w_k)$ */
8 $match[i] \leftarrow match[i] \setminus \{k\}$;
9 **if** $l(v_{i+1}) = l(w_j)$ **and** $ch(v_{i+1}) \subseteq ch(w_j)$ **then**
10 **if** $i + 1 = n$ **then** **output** k;
11 **else**
12 $match[i + 1] \leftarrow match[i + 1] \cup \{k\}$;
13 **if** $l(v_1) = l(w_j)$ **and** $ch(v_1) \subseteq ch(w_j)$ **then**
14 **if** $n = 1$ **then output** j;
15 **else** $match[1] \leftarrow match[1] \cup \{j\}$;

Algorithm 1: SUBCATISO.

Example 2: We apply the algorithm SUBCATISO to the pattern caterpillar P and the text tree T in Example 1 in Figure 1. The if-sentence in line 3 works just internal vertices.

1) For $j = 6$, since $l(v_1) = a = l(w_6)$ and $ch(v_1) = \{a, b, c\} = ch(w_6)$, 6 is stored to $match[1]$ and then $match[1] = \{6\}$.
2) For $j = 8$, since $match[1] \neq \emptyset$, set k to $6 \in match[1]$. Since $h(w_6) = 3 = h(w_8) + 1$, $match[1]$ is changed to $\emptyset$. Since $l(v_2) = a = l(w_8)$ and $ch(v_2) = \{a, b\} \subseteq ch(w_8)$, 6 is stored to $match[2]$ and then $match[2] = \{6\}$. Also since $ch(v_1) = \{a, b, c\} = ch(w_8)$, 8 is stored to $match[1]$ and then $match[1] = \{8\}$.
3) For $j = 9$, since $match[2] \neq \emptyset$, set k to $6 \in match[2]$. Since $h(w_6) = 3 = h(w_9) + 2$, $match[2]$ is changed to $\emptyset$. Since $l(v_3) = a = l(w_9)$ and $ch(v_3) = \{a, b\} = ch(w_9)$, 6 is output. Also since $match[1] \neq \emptyset$, set k to $8 \in match[1]$. Since $h(w_8) = 2 = h(w_9) + 1$, $match[1]$ is changed to $\emptyset$. Since $l(v_2) = a = l(w_9)$ and $ch(v_2) = \{a, b\} \subseteq ch(w_8)$, 8 is stored to $match[2]$ and then $match[2] = \{8\}$.
4) For $j = 16$, since $ch(v_1) = \{a, b, c\} = ch(w_{16})$, 16 is stored to $match[1]$ and then $match[1] = \{16\}$.
5) For $j = 17$, since $match[1] \neq \emptyset$, set k to $16 \in match[1]$. $match[1]$ is changed to $\emptyset$. Since $l(v_2) = a = l(w_{17})$ and $ch(v_2) = \{a, b\} \subseteq ch(w_{17})$, 16 is stored to $match[2]$ and then $match[2] = \{8, 16\}$.
6) For $j = 18$, since $match[2] \neq \emptyset$, set k to 8 and 16. For $k = 8$, since $h(w_8) = 2 = h(w_{18}) + 2$, $match[2]$ is changed to $\{16\}$. Since $l(v_3) = a = l(w_{18})$ and $ch(v_3) = \{a, b\} \subseteq ch(w_{18})$, 8 is output. Also, for $k = 16$, since $h(w_{16}) = 2 = h(w_{18}) + 2$, $match[2]$ is changed to $\emptyset$. As the same reason, 16 is output.

Hence, the set of all the matching positions of P in T is $\{6, 8, 16\}$. As summarising, Table I illustrates the transition of $match[i]$ for the algorithm SUBCATISO.

TABLE I
THE TRANSITION OF $match[i]$ FOR THE ALGORITHM SUBCATISO.

	$j = 6$	$j = 8$	$j = 9$	$j = 16$	$j = 17$	$j = 18$
$match[1]$	6	8	$\emptyset$	16	$\emptyset$	$\emptyset$
$match[2]$	$\emptyset$	6	8	8	$8, 16$	$\emptyset$
$output$			6			$8, 16$

Theorem 2: Let P be a caterpillar and T a tree. Then, the algorithm SUBCATISO correctly outputs all of the matching positions of P in T in $O(p + tDh\sigma)$ time and $O(Dh)$ space.

Proof: First, we show the correctness of the algorithm SUBCATISO. The matching point of P in T is the internal vertices of T. Then, the algorithm SUBCATISO first stores the candidate j of the matching point corresponding to v_1 to $match[1]$ if $l(v_1) = l(w_j)$ and $ch(v_1) \subseteq ch(w_j)$ (line 14). Then, for the current j, the algorithm SUBCATISO removes the candidate k from $match[i]$ if w_j is an ancestor of w_k (line 8) and stores k to $match[i + 1]$ if $l(v_{i+1}) = l(w_j)$, $ch(v_{i+1}) \subseteq ch(w_j)$ and $i < n - 1$ (line 12). If $i = n - 1$, then the algorithm SUBCATISO outputs k (line 10).

Hence, every output k at line 10 satisfies that $l(v_i) = l(par^{i-1}(w_k))$ and $ch(v_i) = ch(par^{i-1}(w_k))$ for every i ($1 \leq i \leq n$), where $par^0(v) = v$ and $par^{i+1}(v) = par(par^i(v))$. As a result, the algorithm SUBCATISO outputs all of the matching points of P in T.

Next, consider the complexity of the algorithm SUBCATISO. As prepossessing, it is necessary to store $ch(v_i)$ for v_i in P and $ch(w_j)$ for internal vertex w_j in T in $O(p)$ time and $O(t)$ time, respectively. Also it is necessary to initialize $match$ in $O(h)$ time. For the for-loop between lines 2 and 12 in the algorithm SUBCATISO, the line 3 works at just internal vertices in T. Since $n = h$ and $|match[i]| \leq D$ ($1 \leq i \leq n - 1$), the foreach-loop between lines 6 and 12 is iterated at most $O(hD)$ times. Since we can check $ch(v_{i+1}) \subseteq ch(w_j)$ in $O(\sigma)$ time, the algorithm SUBCATISO executes the foreach-loop is $O(hD\sigma)$ time. Then, the for-loop is executed in $O(tDh\sigma)$ time. Hence, the total running time of the algorithm SUBCATISO is $O(p + t + h + tDh\sigma) = O(p + tDh\sigma)$ time. The total space is the space spent by the array $match[i]$ for every i ($1 \leq i \leq n - 1$), which is bounded by $O(Dh)$. ■

In order to reduce the searching time in $match[i]$ for every i ($1 \leq i \leq n - 1$) of the algorithm SUBCATISO, we design another algorithm SUBCATISO2 in Algorithm 2.

The main difference between the algorithms SUBCATISO and SUBCATISO2 is that the index i accessed to the array $match$ is determined by $height[h_{j-1}]$ without accessing to $match[i]$ for every i ($1 \leq i \leq n - 1$).

procedure SUBCATISO2(P, T)
/* P : caterpillar such that $bb(P) = \langle v_1, \dots, v_n \rangle$ */
/* T : tree consisting of vertices $w_1, \dots, w_m$ in postorder traversal */
1 **for** $i = 1$ **to** n **do** $match[i] \leftarrow \emptyset$;
2 **for** $j = 1$ **to** m **do** $current(j) \leftarrow 0$;
3 **for** $h = 1$ **to** $h(T) - 1$ **do** $height[h] \leftarrow \emptyset$;
4 **for** $j = 2$ **to** m **do**
5 $h_j \leftarrow h(w_j)$; $h_{j-1} \leftarrow h(w_{j-1})$;
6 **if** $h_{j-1} = h_j + 1$ **then**
/* $w_j = par(w_{j-1})$ */
7 **foreach** $k \in height[h_{j-1}]$ **do**
8 $height[h_{j-1}] \leftarrow height[h_{j-1}] \setminus \{k\}$;
9 **if** $h(w_k) = h_j + current(k)$ **then**
/* $w_j = par(w_k)$ */
10 $i \leftarrow current(k)$;
11 $match[i] \leftarrow match[i] \setminus \{k\}$;
12 $current(k) \leftarrow 0$;
13 **if** $l(v_{i+1}) = l(w_j)$ **and** $\widetilde{ch(v_{i+1})} \subseteq \widetilde{ch(w_j)}$ **then**
14 **if** $i + 1 = n$ **then output** k;
15 **else**
16 $match[i+1] \leftarrow match[i+1] \cup \{k\}$;
17 $height[h_j] \leftarrow height[h_j] \cup \{k\}$;
18 $current(k) \leftarrow i + 1$;
19 **if** $l(v_1) = l(w_j)$ **and** $\widetilde{ch(v_1)} \subseteq \widetilde{ch(w_j)}$ **then**
20 **if** $n = 1$ **then output** j;
21 **else**
22 $match[1] \leftarrow match[1] \cup \{j\}$;
23 $height[h_j] \leftarrow height[h_j] \cup \{j\}$;
24 $current(j) \leftarrow 1$;

Algorithm 2: SUBCATISO2.

Example 3: We apply the algorithm SUBCATISO2 to the pattern caterpillar P and the text tree T in Example 1 in Figure 1. Then, Table II illustrates the transitions of $match[i]$ and $height[j]$ for the algorithm SUBCATISO2.

TABLE II
THE TRANSITIONS OF $match[i]$ AND $height[j]$ FOR THE ALGORITHM SUBCATISO2.

	$j = 6$	$j = 8$	$j = 9$	$j = 16$	$j = 17$	$j = 18$
$match[1]$	6	8	$\emptyset$	16	$\emptyset$	$\emptyset$
$match[2]$	$\emptyset$	6	8	8	8, 16	$\emptyset$
$output$			6			8, 16
$height[1]$	$\emptyset$	$\emptyset$	8	8	8, 16	$\emptyset$
$height[2]$	$\emptyset$	6, 8	$\emptyset$	16	$\emptyset$	$\emptyset$
$height[3]$	6	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

1) For $j = 6$, by lines 3 and 13, 6 is stored to $match[1]$ and $height[3]$, and $current(6)$ is set to 1.
2) For $j = 8$, by lines 6 and 7, 6 is selected as $k \in height[3]$. Since $h(w_6) = 3 = 2 + 1 = h_8 + current(6)$, 6 is deleted from $match[1]$. By line 13, 6 is stored to $match[2]$ and $height[2]$, and $current(6)$ is set to 2. By line 18, 8 is stored to $match[1]$ and $height[2]$, and $current(8)$ is set to 1.
3) For $j = 9$, by lines 6 and 7, 6 and 8 are selected as $k \in height[2]$. For $k = 6$, by line 9, i is set to $2 = current(6)$ and 6 is deleted from $match[2]$. By lines 13 and 14, 6 is output. Also, for $k = 8$, by line 9, i is set to $1 = current(8)$ and 8 is deleted from $match[1]$. By line 13, 8 is stored to $match[2]$ and $height[1]$, and $current(8)$ is set to 2.
4) For $j = 16$, by lines 3 and 13, 16 is stored to $height[2]$ and $match[1]$, and $current(16)$ is set to 1.
5) For $j = 17$, by lines 6 and 7, 16 is selected as $k \in height[2]$. By line 9, i is set to $1 = current(16)$ and 16 is deleted from $match[1]$. By line 13, 16 is stored to $match[2]$ and $height[1]$, and $current(8)$ is set to 2.
6) For $j = 18$, by lines 6 and 7, 8 and 16 are selected as $k \in height[1]$. For $k = 8$, by line 9, i is set to $2 = current(8)$ and 8 is deleted from $match[2]$. By lines 13 and 14, 8 is output. Also, for $k = 16$, by the same reason, 16 is output.

Theorem 3: Let P be a caterpillar and T a tree. Then, the algorithm SUBCATISO2 correctly outputs all of the matching positions of P in T in $O(p + tD\sigma)$ time and $O(D(h + H))$ space.

Proof: The difference between the algorithms SUBCATISO and SUBCATISO2 is the usage of $current$ and $height$ without accessing to $match[i]$ for every i ($1 \leq i \leq n - 1$).

For the selected $k \in height[h_{j-1}]$ at line 7, $current(k)$ is the already processed index i as v_i ($1 \leq i \leq n - 1$). Then, if w_k satisfies the condition at line 13, then $current(k)$ is updated to $i + 1$ at line 17.

On the other hand, for the current j in the for-loop at line 4 and for $k \in height[h_{j-1}]$ at line 7, k is deleted from $height[h_{j-1}]$ at line 8. If $h(w_k) = h_j + current(k)$ at line 9, then h_j is the corresponding height to v_i such that $i = current(k)$. Then, k determines the index i at line 10 to access the array $match[i]$. Furthermore, if v_{i+1} satisfies the condition at line 13, then k is stored to $match[i + 1]$ and $height[h_j]$, and $current(k)$ is updated to $i + 1$.

Hence, the algorithm SUBCATISO2 correctly accesses the array $match[i]$ for every i ($1 \leq i \leq n - 1$). Then, by Theorem 2, the algorithm SUBCATISO2 is correct.

Next, consider the complexity of the algorithm SUBCATISO2. The prepossessing time is $O(p + t)$ from the proof of Theorem 2. It is necessary to initialize $current(j)$ and $height[h]$ in $O(t)$ and $O(H)$, respectively. The foreach-loop between lines 7 and 18 is iterated in $O(D)$ time since $|height[h_{j-1}]| \leq D$, and then the foreach-loop is executed to $O(D\sigma)$ time. Since the for-loop between lines 4 and 22 is executed to $m = t$ time, the total running time of the algorithm SUBCATISO2 is $O(p+t+t+H+tD\sigma) = O(p+tD\sigma)$. The total space of the algorithm SUBCATISO2 is the space spent by the arrays $match[i]$ for every i ($1 \leq i \leq n - 1$) and $height[h]$ for every h ($1 \leq h \leq h(T) - 1$), which is bounded by $O(D(h + H))$. ∎

Theorems 2 and 3 imply the following corollary.

Corollary 1: Let P be a caterpillar and T a tree such that $p < t$, $h \ll t$ and $h < H$. Then, the algorithm SUBCATISO determines whether or not $P \trianglelefteq T$ in $O(tD\sigma)$ time and $O(Dh)$ space. Also the algorithm SUBCATISO2 determines whether or not $P \trianglelefteq T$ in $O(tD\sigma)$ time and $O(DH)$ space.

IV. EXPERIMENTAL RESULTS

In this section, we give experimental results of the algorithms SUBCATISO and SUBCATISO2 for both the artificial data and the real data. Here, the computer environment is that OS is Ubuntu 18.04.4, CPU is Intel Xeon E5-1650 v3 (3.50GHz) and RAM is 3.8GB.

A. Artificial data

First, in order to investigate the efficiency of the algorithm SUBCATISO2, we adopt a *binary caterpillar P_k with height k and the unique label*, which is a caterpillar such that every internal vertex has just two children, and a *complete binary tree T_{2k} with height $2k$ and the unique label*, which is a tree such that every internal vertex has just two children and the height of every leaf is just $2k$. It is obvious that $P_k \trianglelefteq T_{2k}$.

Note that the algorithm SUBCATISO2 is more efficient than the algorithm SUBCATISO when the number of the matching points of P in T are large. Then, Table III illustrates the running time of the algorithms SUBCATISO and SUBCATISO2 for P_k and T_{2k} and the number (#match) of matching points of P_k for T_{2k} for $4 \leq k \leq 11$.

TABLE III
THE RUNNING TIME (MSEC.) OF THE ALGORITHMS SUBCATISO AND SUBCATISO2 FOR P_k AND T_{2k} AND THE NUMBER (#MATCH) OF MATCHING POINTS OF P_k FOR T_{2k} FOR $4 \leq k \leq 11$.

P_k	T_{2k}	SUBCATISO	SUBCATISO2	#match
P_4	T_8	4	3	248
P_5	T_{10}	23	21	1,008
P_6	T_{12}	115	98	4,064
P_7	T_{14}	585	473	16,320
P_8	T_{16}	3,256	2,331	65,408
P_9	T_{18}	21,493	12,126	261,888
P_{10}	T_{20}	181,978	67,697	1,048,064
P_{11}	T_{22}	1,579,043	417,140	4,193,280

Table III shows that the algorithm SUBCATISO2 is faster than the algorithm SUBCATISO for P_k and T_{2k} when k is larger.

The number of the matching points of P_{k+1} is about 4 times of those of P_k. On the other hand, the running time of P_8 (*resp.*, P_9, P_{10}, P_{11}) by the algorithm SUBCATISO is about 5.5 times (*resp.*, about 6.5 times, about 8.5 times, about 8.7 times) of that of P_7 (*resp.*, P_8, P_9, P_{10}). Also the running time of P_8 (*resp.*, P_9, P_{10}, P_{11}) by the algorithm SUBCATISO2 is about 5 times (*resp.*, about 5.2 times, about 5.6 times, about 6.2 times) of that of P_7 (*resp.*, P_8, P_9, P_{10}).

B. Real data

Next, we give experimental results for caterpillars and trees in real data. We deal with data for N-glycans and all-glycans from KEGG[3], CSLOGS[4], dblp[5] and TPC-H, Auction, Nasa, Protein and University from UW XML Repository[6]. In particular, we deal with the largest 51,546 trees (1%) in dblp (refer to $\text{dblp}_{1\%}$). As pattern caterpillars, we deal with non-isomorphic caterpillars in TPC-H, caterpillars obtained by deleting the root in Auction and non-isomorphic caterpillars obtained by deleting the root in Nasa, Protein, and University. Note that we use all the trees as text trees in TPC-H, Auction, Nasa, Protein and University.

Table IV illustrates the information of such caterpillars and trees. Here, #, n, d and h are the number of caterpillars and trees, the average number of vertices, the average degree and the average height.

TABLE IV
THE INFORMATION OF CATERPILLARS AND TREES.

	caterpillars				trees			
	#	n	d	h	#	n	d	h
N-glycans	514	6.40	1.84	4.22	2,124	11.06	2.07	5.38
all-glycans	7,984	4.74	1.49	3.02	10,683	6.38	1.65	3.59
CSLOGS	41,592	5.84	3.05	2.20	59,691	12.93	4.48	3.42
$\text{dblp}_{1\%}$	51,395	21.29	20.21	1.04	51,546	21.29	20.18	1.04
SwissProt	6,804	35.10	24.96	2.00	50,000	59.54	31.33	2.76
TCP-H	8	8.63	7.63	1.00	86,805	14.46	13.46	1.00
Auction	259	4.29	3.00	0.71	37	31.00	12.00	3.00
Nasa	33	7.27	5.15	1.64	2,435	195.74	21.53	5.76
Protein	5,150	4,97	3.63	1.16	262,525	81.15	23.27	4.99
University	26	1.35	0.35	0.19	6,739	22.52	11.75	2.31

Then, Table V illustrates the total and average running time (msec.) of the algorithms SUBCATISO and SUBCATISO2 applying to data in Table IV by regarding caterpillars as pattern caterpillars and trees as text trees. Here, #cat denotes the number of pattern caterpillars and #tree denotes the number of text trees. Also the average running time is obtained by dividing the total running time by the total number of pairs, that is, (#cat)×(#tree).

TABLE V
THE TOTAL AND AVERAGE RUNNING TIME (MSEC.) OF THE ALGORITHMS SUBCATISO AND SUBCATISO2 APPLYING TO DATA IN TABLE IV.

			SUBCATISO		SUBCATISO2	
	#cat	#tree	total	ave.	total	ave.
N-glycans	514	2,142	53,969	0.0490	55,638	0.0505
all-glycans	7,894	10,683	1,353,490	0.0159	1,521,891	0.0178
CSLOGS	41,592	59,691	35,681,928	0.0144	42,296,479	0.0170
$\text{dblp}_{1\%}$	51,395	51,546	82,881,047	0.0313	85,767,789	0.0324
SwissProt	6,804	50,000	52,694,341	0.1549	53,326,537	0.1568
TCP-H	8	86,805	37,488	0.0540	39,461	0.0568
Auction	259	37	566	0.0591	589	0.0615
Nasa	33	2,435	21,462	0.2671	21,556	0.2683
Protein	5,150	262,525	78,939,972	0.0584	81,660,905	0.0604
University	26	6,739	1,348	0.0077	1,401	0.0080

Furthermore, Table VI illustrates the number (#($P \trianglelefteq T$)) of pairs such that $P \trianglelefteq T$ and its ratio in the total number of

[3]Kyoto Encyclopedia of Genes and Genomes, http://www.kegg.jp/
[4]http://www.cs.rpi.edu/~zaki/www-new/pmwiki.php/Software/Software
[5]http://dblp.uni-trier.de/
[6]http://aiweb.cs.washington.edu/research/projects/xmltk/xmldata/www/repository.html

pairs, and the total and average number (#match) of matching points when $P \trianglelefteq T$.

TABLE VI
THE NUMBER (#($P \trianglelefteq T$)) OF PAIRS SUCH THAT $P \trianglelefteq T$ AND ITS RATIO, AND THE TOTAL AND AVERAGE NUMBER (#MATCH) OF MATCHING POINTS WHEN $P \trianglelefteq T$.

	#cat	#tree	#($P \trianglelefteq T$) total	ratio (%)	#match total	ave.
N-glycans	514	2,142	58,128	5.277	96,729	1.664
all-glycans	7,894	10,683	983,163	1.153	1,625,099	1.653
CSLOGS	41,592	59,691	3,201,441	0.129	3,201,441	1.000
$\text{dblp}_{1\%}$	51,395	51,546	364,237,724	13.749	364,238,110	1.000
SwissProt	6,804	50,000	20,259,951	5.955	34,465,633	1.701
TCP-H	8	86,805	86,805	12.500	86,805	1.000
Auction	259	37	5,476	57.143	5,476	1.000
Nasa	33	2,435	17,850	22.214	42,687	2.391
Protein	5,150	262,525	2,413,404	0.179	2,515,642	1.042
University	26	6,739	9.260	5.285	9,260	1.000

In contrast to Table III, Table V shows that the algorithm SUBCATISO is faster than the algorithm SUBCATISO2 for real data like as Corollary 1. One of the reasons is that the number of the matching points for real data is much smaller than that for artificial data. In fact, Table VI shows that the average number of matching points for real data when $P \trianglelefteq T$ is less than 2.

Table V also shows that the average running time of both algorithms for Nasa is largest and that for SwissProt is next largest for all the data. The reason is that both the average number of vertices of text trees in Table IV and the average number of matching points in Table VI are larger than other data.

Table VI shows that, for CSLOGS, TCP-H, Auction and University, the number of the matching point is always exactly one when $P \trianglelefteq T$. Then, for the other data as N-glycans, all-glycans, $\text{dblp}_{1\%}$, SwissProt, Nasa and Protein, Table VII illustrates the histograms of the number of matching points and the maximum value (max) of matching points when $P \trianglelefteq T$.

TABLE VII
THE HISTOGRAMS FOR THE NUMBER OF MATCHING POINTS AND THE MAXIMUM VALUE (MAX) OF MATCHING POINTS FOR N-GLYCANS, ALL-GLYCANS, $\text{DBLP}_{1\%}$, SWISSPROT, NASA AND PROTEIN WHEN $P \trianglelefteq T$.

#match	N-glycans	all-glycans	$\text{dblp}_{1\%}$	SwissProt	Nasa	Protein
1	29,909	640,541	364,237,418	13,589,834	10,209	2,344,390
2	20,869	195,221	253	3,828,308	2,692	51,684
3	4,432	72,597	40	1,422,806	3,558	9,862
4	2,804	39,496	6	598,676	590	3,882
5	114	16,172	0	295,015	265	1,394
6	0	9,799	7	161,599	150	1,160
7	0	4,297	0	100,317	99	425
8	0	1,998	0	64,968	59	298
9	0	1,211	0	47,901	48	97
≥ 10	0	1,858	0	150,527	180	212
max	5	21	6	79	1,178	32

Table VII shows that the number of cases whose matching points are more than 1 for SwissProt is largest and that for all-glycans is next largest for all the data. On the other hand, the maximum value of matching points for Nasa is extremely largest and that for SwissProt is next largest for all the data.

V. CONCLUSION

In this paper, we have investigated the *subcaterpillar isomorphism* and designed two algorithms SUBCATISO running in $O(p + tDh\sigma)$ time and $O(Dh)$ space and SUBCATISO2 running in $O(p+tD\sigma)$ time and $O(D(h+H))$ space, where $p = |P|$, $t = |T|$, $h = h(P)$, $H = h(T)$, $D = d(T)$ and $\sigma = |\Sigma|$. Also we give experimental results for artificial data and real data.

Then, as same as Theorems 2 and 3, we have confirmed that the algorithm SUBCATISO2 is faster than the algorithm SUBCATISO for artificial data whose number of the matching points of P in T are large. On the other hand, we have confirmed that the algorithm SUBCATISO is faster than the algorithm SUBCATISO2 for real data. One of the reason is that the running time of using the array $height[h]$ in the algorithm SUBCATISO2 cannot be absorbed like as Corollary 1 when the number of the matching points is not large.

The reason why we cannot apply the SETH-hardness to subcaterpillar isomorphism is that a caterpillar has a unique backbone. Then, it is a future work to extend a caterpillar to a tree with the bounded number of backbones, in order to avoid to the SETH-hardness of subtree isomorphism [1]. Also it is a future work to extend the algorithms in this paper to *unrooted* subcaterpillar isomorphism like as [10].

REFERENCES

[1] A. Abboud, A. Backurs, T. D. Hansen, V. v. Williams, O. Zamir: *Subtree isomorphism revisited*, ACM Trans. Algo. **14**, 27 (2018). https://doi.org/10.1145/3093239.

[2] T. Akutsu, D. Fukagawa, M. M. Halldórsson, A. Takasu, K. Tanaka: *Approximation and parameterized algorithms for common subtrees and edit distance between unordered trees*, Theoret. Comput. Sci. **470**, 10–22 (2013). https://doi.org/10.1016/j.tcs.2012.11.017.

[3] J. A. Gallian: *A dynamic survey of graph labeling*, Electorn. J. Combin., DS6 (2018).

[4] K. Hirata, Y. Yamamoto, T. Kuboyama: *Improved MAX SNP-hard results for finding an edit distance between unordered trees*, Proc. CPM'11, LNCS **6661**, 402–415 (2011). https://doi.org/10.1007/978-3-642-21458-5_34.

[5] J. E. Hopcroft, R. M. Karp: *An $n^{5/2}$ algorithm for maximum matching in bipartite graphs*, SIAM J. Comput. **2**, 225–231 (1973). https://doi.org/10.1137/10.1137/0202019.

[6] P. Kilpeläinen, H. Mannila: *Ordered and unordered tree inclusion*, SIAM J. Comput. **24**, 340–356 (1995). https://doi.org/10.1137/S0097539791218202.

[7] T. Miyazaki, M. Hagihara, K. Hirata: *Caterpillar inclusion: Inclusion problem for rooted labeled caterpillars*, Proc. ICPRAM'22, 280–287 (2022). https://doi.org/10.5220/0010826300003122.

[8] K. Muraka, T. Yoshino, K. Hirata: *Computing edit distance between rooted labeled caterpillars*, Proc. FedCSIS'18, 245–252 (2018). http://dx.doi.org/10.15439/2018F179.

[9] K. Muraka, T. Yoshino, K. Hirata: *Vertical and horizontal distance to approximate edit distance for rooted labeled caterpillars*, Proc. ICPRAM'19, 590–597 (2019). https://dx.doi.org/10.5220/0007387205900597.

[10] R. Shamir, D. Tsur: *Faster subtree isomorphism*, J. Algo. **33**, 267–280 (1999). https://doi.org/10.1006/jagm.1999.1044.

[11] K. Zhang, T. Jiang: *Some MAX SNP-hard results concerning unordered labeled trees*, Inform. Process. Lett. **49**, 249–254 (1994). https://doi.org/10.1016/0020-0190(94)90062-0.