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) \(O(p + tDh)\) 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 \(T\), \(\sigma\) 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 \(O(p^{h+1}/\log p)\) time [10], where \(p\) is the number of vertices in \(P\) and \(h\) is the number of vertices in \(T\). On the other hand, the algorithm cannot be solved in \(O((2^{h+1})^e)\) for every \(e (0 < e < 1)\) 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) [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 \(O(n + H^2\lambda^3)\) 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 \(\sigma\) is the number of alphabets for labels in the two caterpillars [8].

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].

In this paper, we design two algorithms to solve the subcaterpillar isomorphism in (i) \(O(p + tDh)\) time and \(O(Dh)\) space and in (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

\[\lambda\] is the 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 \(\lambda\) with \(\sigma\), 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.

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 the tree 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 the tree 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 \( \Sigma \), we say that a tree is labeled over \( \Sigma \) if each vertex is assigned a symbol from \( \Sigma \). We denote the label of a vertex \( v \) by \( l(v) \), and sometimes identify a vertex with \( l(v) \). In this paper, we call a rooted labeled unordered tree over \( \Sigma \) 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, a vertex \( v \) in \( T \) and its children \( v_1, \ldots, v_i \), the postorder traversal of \( T(v) \) is obtained by first visiting \( T(v_k) \) \((1 \leq k \leq \delta)\) and then visiting \( v \). The postorder number \( \text{pv}(v) \in T \) 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 \( T \simeq S \), 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 \( P \preceq T \) 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 \( |P| = |T| \). Then, the problem of determining whether or not \( P \preceq T \) is solvable in \( O(p^{1.5}/\log p) \) 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 remaining rooted path a backbone of \( C \) and denote it by \( bb(C) \).

It is obvious that \( r(C) = r(bb(C)) \) and \( V(C) = V(bb(C)) \cup \{v(C)\} \) 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 \( P \preceq T \) for a caterpillar \( P \) and a tree \( T \). Throughout 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 \( (v_1, \ldots, v_n) \) such that \( \{v_i, v_{i+1}\} \in E(P) \) and \( v_0 = r(P) \). We denote the children of \( v_i \) by \( ch(v_i) \).

On the other hand, for a text caterpillar \( T \), we refer the vertices in \( T \) to \( w_1, \ldots, w_m \) 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 caterpillar such that \( P \preceq T \). Also let \( P' \preceq T \) be a subcaterpillar in \( T \) such that \( P \preceq P' \) and \( bb(P') = (v_1', \ldots, v_\delta') \), where \( v_\delta' = r(P') \). Then, we call the postorder number \( j \) such that \( v_j = w_j \) 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 \( \{6, 8, 9\}, \{8, 9, 18\} \) and \( \{16, 17, 18\} \).

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 \( \Sigma \) is a mapping \( S : \Sigma \to \mathbb{N} \). For a set \( V \) of vertices, we denote the multiset of labels occurring in \( V \) by \( \hat{V} \). Then, it is necessary for the subcaterpillar isomorphism to check whether or not \( \hat{ch(v_i)} \subseteq \hat{ch(w_j)} \) for \( v_i \in bb(P) \) and \( w_j \in T \). It is realized to check \( \hat{ch(v_i)}(a) \leq \hat{ch(w_j)}(a) \) 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 \( P \preceq T \). Here, the algorithm SUBCATISo output all of the matching positions if \( P \preceq T \). Then, it holds that no matching point is output if \( P \not\preceq T \).
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 */
  for i = 1 to n - 1 do match[i] ← ∅;
  for j = 2 to m do
    if h(w_{j-1}) = h(w_j) + 1 then
      /* w_j = par(w_{j-1}) */
      for i = n - 1 downto 1 do
        if match[i] ≠ ∅ then
          foreach k ∈ match[i] do
            if h(w_k) = h(w_j) + i then
              match[i] ← match[i] \ {k};
            if (l(v_i) = l(w_j) and ch(v_i+1) ⊆ ch(w_j)) then
              if i + 1 = n then output k;
            else match[i+1] ← match[i+1] ∪ {k};
        if l(v_1) = l(w_j) and ch(v_1) ⊆ ch(w_j) then
          if n = 1 then output j;
  end for
end procedure

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-statement 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] = ∅.
2. For j = 8, since match[1] ≠ ∅, set k to 6 ∈ match[1].
   Since h(w_6) = 3 = h(w_k) + 1, match[1] is changed to ∅. Since l(v_2) = a = l(w_6) and ch(v_2) = \{a, b\} ⊆ ch(w_6), 6 is stored to match[2] and then match[2] = ∅.
3. For j = 9, since match[2] ≠ ∅, set k to 6 ∈ match[2].
   Since h(w_6) = 3 = h(w_k) + 2, match[2] is changed to ∅. Since l(v_3) = a = l(w_6) and ch(v_3) = \{a, b\} = ch(w_6), 6 is output. Also since match[1] ≠ ∅, set k to 8 ∈ match[1].
   Since h(w_8) = 2 = h(w_k) + 1, match[1] is changed to ∅. Since l(v_2) = a = l(w_8) and ch(v_2) = \{a, b\} ⊆ ch(w_8), 8 is stored to match[2] and then match[2] = ∅.
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] ≠ ∅, set k to 16 ∈ match[1].
   match[1] is changed to ∅. Since l(v_2) = a = l(w_{17}) and ch(v_2) = \{a, b\} ⊆ ch(w_{17}), 16 is stored to match[2] and then match[2] = \{8, 16\}.
6. For j = 18, since match[2] ≠ ∅, 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\} ⊆ ch(w_{18}), 8 is output. Also, for k = 16, since h(w_{16}) = 2 = h(w_{18}) + 2, match[2] is changed to ∅. 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>
<thead>
<tr>
<th>Table I</th>
<th>The transition of match[i] for the algorithm SUBCATISO.</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>16</td>
<td>∅</td>
</tr>
<tr>
<td>17</td>
<td>8, 16</td>
</tr>
</tbody>
</table>

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 + tDhσ) 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_i to match[1] if l(v_i) = l(w_j) and ch(v_i) ⊆ ch(w_j) (line 14). Then, for the current j, the algorithm SUBCATISO removes the candidate k from match[i] if w_k is an ancestor of w_j (line 8) and stores k to match[i+1] if l(v_i+1) = l(w_k), ch(v_i+1) ⊆ ch(w_j) and j < 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 ≤ i ≤ 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]| ≤ D (1 ≤ i ≤ n − 1), the for-loop between lines 6 and 12 is iterated at most O(hD) times.

Since we can check ch(v_{i+1}) ⊆ ch(w_j) in O(\sigma) time, the algorithm SUBCATISO executes the for-loop is O(hD\sigma) time. Then, the for-loop is executed in O(Dh\sigma) time. Hence, the total running time of the algorithm SUBCATISO is O(p + t + tDh\sigma) = O(p + tDh) time. The total space is the space spent by the array match[i] for every i (1 ≤ i ≤ n − 1), which is bounded by O(Dh).

In order to reduce the searching time in match[i] for every i (1 ≤ i ≤ 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[j_{i-1}] without accessing to match[i] for every i (1 ≤ i ≤ n − 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 $\text{match}[i]$ and $\text{height}[j]$ for the algorithm SUBCATISO2.

<table>
<thead>
<tr>
<th>$j$</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{match}[1]$</td>
<td>6</td>
<td>8</td>
<td>0</td>
<td>16</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$\text{match}[2]$</td>
<td>0</td>
<td>6</td>
<td>8</td>
<td>16</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>output</td>
<td>6</td>
<td>8,16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\text{height}[0]$</td>
<td>0</td>
<td>0</td>
<td>8</td>
<td>8</td>
<td>16</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$\text{height}[2]$</td>
<td>0</td>
<td>6,8</td>
<td>0</td>
<td>16</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$\text{height}[3]$</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

1) For $j = 6$, by lines 3 and 13, 6 is stored to $\text{match}[1]$ and $\text{height}[3]$, and $\text{current}(6)$ is set to 1.

2) For $j = 8$, by lines 6 and 7, 6 is selected as $k \in \text{height}[3]$. Since $h(w_k) = 3 = 2 + 1 = \text{height}(6)$, 6 is deleted from $\text{match}[1]$. By line 13, 6 is stored to $\text{match}[2]$ and $\text{height}[2]$, and $\text{current}(6)$ is set to 2. By line 18, 8 is stored to $\text{match}[1]$ and $\text{height}[2]$, and $\text{current}(8)$ is set to 1.

3) For $j = 9$, by lines 6 and 7, 6 and 8 are selected as $k \in \text{height}[2]$. For $k = 6$, by line 9, $i$ is set to 2 = $\text{current}(6)$ and 6 is deleted from $\text{match}[2]$. By lines 13 and 14, 6 is output. Also, for $k = 8$, by line 9, $i$ is set to 1 = $\text{current}(8)$ and 8 is deleted from $\text{match}[1]$. By line 13, 8 is stored to $\text{match}[2]$ and $\text{height}[1]$, and $\text{current}(8)$ is set to 2.

4) For $j = 16$, by lines 3 and 13, 16 is stored to $\text{height}[2]$ and $\text{match}[1]$, and $\text{current}(16)$ is set to 1.

5) For $j = 17$, by lines 6 and 7, 16 is selected as $k \in \text{height}[2]$. By line 9, $i$ is set to 1 = $\text{current}(16)$ and 16 is deleted from $\text{match}[1]$. By line 13, 16 is stored to $\text{match}[2]$ and $\text{height}[1]$, and $\text{current}(8)$ is set to 2.

6) For $j = 18$, by lines 6 and 7, 8 and 16 are selected as $k \in \text{height}[1]$. For $k = 8$, by line 9, $i$ is set to 2 = $\text{current}(8)$ and 8 is deleted from $\text{match}[2]$. 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 $\text{current}$ and $\text{height}$ without accessing to $\text{match}[i]$ for every $i$ ($1 \leq i \leq n - 1$).

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

Hence, the algorithm SUBCATISO2 correctly accesses the array $\text{match}[i]$ 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 preprocessing time is $O(p + t)$ from the proof of Theorem 2. It is necessary to initialize $\text{current}(j)$ and $\text{height}[h]$ in $O(t)$ and $O(H)$, respectively. The foreach-loop between lines 7 and 18 is iterated in $O(D)$ time since $|\text{height}[h_{j-1}]| \leq D$, and then the foreach-loop is executed to $O(D\sigma)$ time. Since the foreach-loop between lines 4 and 22 is executed to $m + t$ time, the total running time of the algorithm SUBCATISO2 is $O(p + t + H + D\sigma) = O(p + tD\sigma)$. The total space of the algorithm SUBCATISO2 is the space spent by the arrays $\text{match}[i]$ for every $i$ ($1 \leq i \leq n - 1$) and $\text{height}[h]$ 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 $p < t$, $h < t$ and $h < H$. Then, the algorithm SubcatIso determines whether or not $P \preceq T$ in $O(t\sigma)$ time and $O(Dh)$ space. Also the algorithm SubcatIso2 determines whether or not $P \preceq T$ in $O(t\sigma)$ 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 $P_k \preceq T_{2k}$.

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>
<thead>
<tr>
<th>$P_k$</th>
<th>$T_{2k}$</th>
<th>SubcatIso</th>
<th>SubcatIso2</th>
<th>#match</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_4$</td>
<td>$T_{12}$</td>
<td>4</td>
<td>3</td>
<td>248</td>
</tr>
<tr>
<td>$P_5$</td>
<td>$T_{10}$</td>
<td>23</td>
<td>21</td>
<td>1,008</td>
</tr>
<tr>
<td>$P_6$</td>
<td>$T_{12}$</td>
<td>115</td>
<td>98</td>
<td>4,064</td>
</tr>
<tr>
<td>$P_7$</td>
<td>$T_{14}$</td>
<td>585</td>
<td>473</td>
<td>16,320</td>
</tr>
<tr>
<td>$P_8$</td>
<td>$T_{16}$</td>
<td>3,256</td>
<td>2,331</td>
<td>65,408</td>
</tr>
<tr>
<td>$P_9$</td>
<td>$T_{18}$</td>
<td>21,493</td>
<td>12,126</td>
<td>261,888</td>
</tr>
<tr>
<td>$P_{10}$</td>
<td>$T_{20}$</td>
<td>181,978</td>
<td>67,697</td>
<td>1,048,064</td>
</tr>
<tr>
<td>$P_{11}$</td>
<td>$T_{22}$</td>
<td>1,579,043</td>
<td>417,140</td>
<td>4,193,280</td>
</tr>
</tbody>
</table>

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 $P_k$ (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 $P_2$ (resp., $P_9$, $P_{10}$, $P_{11}$). Also the running time of $P_k$ (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., $P_9$, $P_{10}$, $P_{11}$).

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 dblp1\/%). 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, $\#, \theta$, $d$, and $h$ are the number of caterpillars and trees, the average number of vertices, the average degree and the average height.

<table>
<thead>
<tr>
<th></th>
<th>#caterpillars</th>
<th>#trees</th>
</tr>
</thead>
<tbody>
<tr>
<td>N-glycans</td>
<td>514</td>
<td>2,124</td>
</tr>
<tr>
<td>all-glycans</td>
<td>7,984</td>
<td>10,683</td>
</tr>
<tr>
<td>CSLOGS</td>
<td>41,592</td>
<td>59,691</td>
</tr>
<tr>
<td>dblp1/%</td>
<td>51,395</td>
<td>51,546</td>
</tr>
<tr>
<td>SwissProt</td>
<td>6,804</td>
<td>50,000</td>
</tr>
<tr>
<td>TCP-H</td>
<td>8</td>
<td>86,805</td>
</tr>
<tr>
<td>Auction</td>
<td>259</td>
<td>77</td>
</tr>
<tr>
<td>Nasa</td>
<td>33</td>
<td>23,27</td>
</tr>
<tr>
<td>Protein</td>
<td>5,150</td>
<td>262,525</td>
</tr>
<tr>
<td>University</td>
<td>26</td>
<td>7,739</td>
</tr>
</tbody>
</table>

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, ($#\text{cat}$)$\times$($#\text{tree}$).

<table>
<thead>
<tr>
<th></th>
<th>#cat</th>
<th>#tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>N-glycans</td>
<td>514</td>
<td>55,638</td>
</tr>
<tr>
<td>all-glycans</td>
<td>7,984</td>
<td>1,251,891</td>
</tr>
<tr>
<td>CSLOGS</td>
<td>41,592</td>
<td>42,296,479</td>
</tr>
<tr>
<td>dblp1/%</td>
<td>51,395</td>
<td>85,767,789</td>
</tr>
<tr>
<td>SwissProt</td>
<td>6,804</td>
<td>53,326,537</td>
</tr>
<tr>
<td>TCP-H</td>
<td>8</td>
<td>39,461</td>
</tr>
<tr>
<td>Auction</td>
<td>259</td>
<td>589</td>
</tr>
<tr>
<td>Nasa</td>
<td>33</td>
<td>21,556</td>
</tr>
<tr>
<td>Protein</td>
<td>5,150</td>
<td>81,600,905</td>
</tr>
<tr>
<td>University</td>
<td>26</td>
<td>22,52</td>
</tr>
</tbody>
</table>

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

---

\(^3\)Kyoto Encyclopedia of Genes and Genomes, http://www.kegg.jp/
\(^5\)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 \( P \leq T \).

<table>
<thead>
<tr>
<th>#cat</th>
<th>#tree</th>
<th>(#(P \leq T)) total</th>
<th>(#(P \leq T)) total ratio (%)</th>
<th>#match</th>
<th>#match total ave.</th>
</tr>
</thead>
<tbody>
<tr>
<td>N-glycans</td>
<td>514</td>
<td>2,142</td>
<td>58,128</td>
<td>5.277</td>
<td>96,729</td>
</tr>
<tr>
<td>all-glycans</td>
<td>7,894</td>
<td>10,683</td>
<td>983,163</td>
<td>1.153</td>
<td>1,625,099</td>
</tr>
<tr>
<td>CSLOGS</td>
<td>41,592</td>
<td>59,691</td>
<td>3,201,441</td>
<td>0.129</td>
<td>3,201,441</td>
</tr>
<tr>
<td>dblp</td>
<td>51,395</td>
<td>51,546</td>
<td>364,237,214</td>
<td>13.749</td>
<td>364,238,110</td>
</tr>
<tr>
<td>SwissProt</td>
<td>6,804</td>
<td>50,000</td>
<td>20,259,951</td>
<td>5.955</td>
<td>34,465,633</td>
</tr>
<tr>
<td>Protein</td>
<td>5,150</td>
<td>262,525</td>
<td>2,413,404</td>
<td>0.179</td>
<td>2,515,642</td>
</tr>
</tbody>
</table>

Table VI 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. 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 unordered subcaterpillar isomorphism like as [10].

V. CONCLUSION

In this paper, we have investigated the subcaterpillar isomorphism and designed two algorithms SUBCATISO running in \((O(p + tD\sigma)\) time and \(O(D\sigma)\) 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 number of 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 unordered subcaterpillar isomorphism like as [10].

REFERENCES