Similar Subtree Search Using Extended Tree Inclusion

Tomoya Mori, Atsuhiro Takasu, *Member, IEEE*, Jesper Jansson, Jaewook Hwang, Takeyuki Tamura, and Tatsuya Akutsu, *Member, IEEE*

Abstract—This paper considers the problem of identifying all locations of subtrees in a large tree or in a large collection of trees that are similar to a specified pattern tree, where all trees are assumed to be rooted and node-labeled. The *tree edit distance* is a widely-used measure of tree (dis-)similarity, but is NP-hard to compute for unordered trees. To cope with this issue, we propose a new similarity measure which extends the concept of *unordered tree inclusion* by taking the costs of insertion and substitution operations on the pattern tree into account, and present an algorithm for computing it. Our algorithm has the same time complexity as the original one for unordered tree inclusion, i.e., it runs in $O(|T_1||T_2|)$ time, where T_1 and T_2 denote the pattern tree and the text tree, respectively, when the maximum outdegree of T_1 is bounded by a constant. Our experimental evaluation using synthetic and real datasets confirms that the proposed algorithm is fast and scalable and very useful for bibliographic matching, which is a typical entity resolution problem for tree-structured data. Furthermore, we extend our algorithm to also allow a constant number of deletion operations on T_1 while still running in $O(|T_1||T_2|)$ time.

Index Terms—Tree edit distance, tree inclusion, unordered trees, dynamic programming

1 Introduction

The comparison of tree-structured data has numerous applications in various scientific areas and data processing. For example, trees can be used to represent several kinds of biological data such as RNA secondary structures [1], vascular trees [2], glycans [3], evolutionary history [4], and cell lineage data [5], and scientists sometimes need to compare such structures. Similarly, XML data is often represented by a tree, and as a result, many studies have been done on how to compare and search such data efficiently [6], [7], [8]. In natural language processing, sentences are usually represented by parse trees and thus comparison of trees is useful for natural language information retrieval [9]. Tree-structured data matching is often applied to entity resolution [10], [11]. Certain kinds of image data are also represented by trees [12].

The similar subtree search problem studied in this paper is: given a pattern tree and a large text tree, find roots of all parts of the text tree that are similar to the pattern tree. From here on, all trees are assumed to be rooted and node-labeled. Although we focus on finding the root positions in the text tree, a mapping can be retrieved for each matched root positions only with extra cost proportional to the output size by traceback. The similar subtree search problem can be applied to various searching and matching problems. For example, recently some researchers reported that the structure of keywords is more effective for information retrieval than just a bag-of-words [9]. The structure of keywords is usually represented with a small tree structure such as a parsing tree after natural language processing. When processing a query represented with a tree (pattern tree), we need to calculate a similarity between the pattern tree and fragments of text data (text tree). Information integration is another applicable problem. Some researchers applied tree matching algorithms to record matching [10], [11], where records such as bibliographic data are represented in XML. When matching records in different schema, we need to handle heterogeneity of the schema such as different document type definitions causing the difference of element order and missing/extra elements. A record having extra elements often results in a much larger tree than typical record. For example, DBLP record contains extra elements such as citations and URLs, which are not included in standard bibliographic databases. As a result, some trees included in DBLP are more than twice as large as trees representing standard bibliographic components such as authors and titles. In this case, the tree inclusion is more suitable than just the tree matching.

Different meanings of "similar" can be employed, but here we focus on one of the most commonly used measures of similarity known as the *tree edit distance*. For *ordered* trees, there exists a polynomial-time algorithm to compute it [13]. However, in many situations, it is more appropriate to consider *unordered* trees. Although natural orderings of siblings exist in RNA secondary structures or in parse trees, it is difficult to uniquely determine the ordering for many other

- *T. Mori, J. Hwang, T. Tamura, and T. Akutsu are with the Bioinformatics Center, Institute for Chemical Research, Kyoto University, Gokasho, Uji, Kyoto 611-0011, Japan. E-mail: {tmori, hwangjw, tamura, takutsu}@kuicr.kyoto-u.ac.jp.*
- *A. Takasu is with the Content and Media Science Research Division, National Institute of Informatics, 2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo 101-8430, Japan. E-mail: takasu@nii.ac.jp.*
- *J. Jansson is with the Hakubi Project, Kyoto University, Gokasho, Uji, Kyoto 611-0011, Japan, and the Bioinformatics Center, Institute for Chemical Research, Kyoto University, Gokasho, Uji, Kyoto 611-0011, Japan. E-mail: jj@kuicr.kyoto-u.ac.jp.*

Manuscript received 26 Jan. 2015; revised 8 July 2015; accepted 9 July 2015. Date of publication 16 July 2015; date of current version 3 Nov. 2015. Recommended for acceptance by L.B. Holder. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below. Digital Object Identifier no. 10.1109/TKDE.2015.2457922

types of data. For example, although XML data is represented by an ordered tree, the order of elements can be different when matching schema of XML data in different databases [14]. Therefore, we need to handle them as unordered trees. As discussed in [6], there can be a significant gap between the ordered and unordered tree edit distances, so similar trees may be missed if we simply fix some arbitrary ordering and apply the ordered variant (see Section 3). Unfortunately, the unordered tree edit distance problem is known to be NP-hard [15].

To cope with this hardness, several approaches have been taken. The first one is the use of branch-and-bound techniques [5], [16]. This approach computes the tree edit distance exactly, but its applicability is limited to medium-size trees (i.e., trees with several tens of nodes). The second one is the use of approximate distances [6]; this approach is quite scalable, but does not yield exact distances. The third one is the use of modified or restricted types of tree edit distances that can be computed exactly in polynomial time [3], [17], [18] (see also [19] for the ordered case). Such measures are less flexible and provide less matching functionality, and thus have not been used widely. The fourth one is the use of degree constraints. Approach four alone does not help much for unordered trees [20]. An example of a method that combines approaches three and four is the *alignment of trees distance* introduced by Jiang et al. [21], which is a special case of tree edit distance that can be computed in polynomial time when the maximum outdegree of both input trees is upper-bounded by a constant. It has certain nice properties but is not a proper metric as it does not satisfy the triangle inequality. In a related line of research, Kilpeläinen and Mannila [22] considered the *tree inclusion problem* which takes as input a pattern tree and a text tree, and asks whether the text tree can be obtained from the pattern tree by applying insertion operations. They showed that the problem is NP-hard in general for unordered trees but can be solved in polynomial time when the maximum outdegree of the pattern tree is upper-bounded by a constant. On the negative side, the problem ignores the costs of insertion operations, which in many applications prohibits it from being practically useful for similar subtree search, as discussed in Section 4. To make the concept of tree inclusion more useful, we extend it so that the costs of insertion operations are incorporated and substitutions of node labels are possible. We also further extend it to allow a small number of deletion operations in the pattern tree.

In summary, the contributions of this paper are as follows. (i) We introduce the *minimum-cost unordered tree inclusion problem* (**MinCostIncl**, for short), obtained by extending unordered tree inclusion to take the costs of insertion and substitution operations into account. (ii) We give an $O(2^{2D}mn)$-time algorithm for **MinCostIncl**, where D is the maximum outdegree of a pattern tree, m is the size of a pattern tree, and n is the size of the text tree. In addition, we improve the space complexity from $O(2^D mn)$ to $O(n + 2^D m \log(n))$ when traceback is not required. (iii) We extend **MinCostIncl** so that a small number of deletion operations are allowed and present a parameterized $O((eD)^K K^{1/2} 2^{2(DK+D-K)} mn)$-time algorithm, where K is the number of allowed deletion operations and e is the base of the natural logarithm. (iv) We implement all proposed algorithms and perform computational experiments using both synthetic and real data and show the efficiency and effectiveness of the minimum-cost unordered tree inclusion algorithm. It works efficiently in practice when D is not large (i.e., $D < 8$). It is to be noted that trees of $D < 8$ cover a large class of tree structured data because binary trees appear in many fields (e.g., phylogenetic trees, parse trees) and the maximum degree of glycans is 6. We experimentally show that the minimum-cost unordered tree inclusion significantly improves the bibliographic matching accuracy compared with the ordered tree edit distance.

The paper is organized as follows. Section 2 describes some related work. Section 3 reviews the tree edit distance and unordered tree inclusion. Section 4 defines **MinCostIncl**, presents an $O(2^{2D}mn)$-time algorithm for solving it, and explains how to improve the algorithm's space complexity. Section 5 defines an extended variant of **MinCostIncl** and gives a corresponding $O((eD)^K K^{1/2} 2^{2(DK+D-K)} mn)$-time algorithm. Section 6 presents the results of our experiments using synthetic and real data. We conclude and discuss future research directions in Section 7.

2 Related Work

The *tree edit distance* is defined formally in Section 3 below. Intuitively, it asks for a minimum-cost sequence of insertion, deletion, and substitution operations that transforms one input tree (T_1) into another input tree (T_2). It has been extensively studied; see, e.g., [17] for a survey.

For the case of ordered trees, Tai's [23] classic $O(n^6)$-time tree edit distance-algorithm (where n denotes the number of nodes in the larger of the two input trees) was improved upon gradually until Demaine et al. [13] presented an $O(n^3)$-time algorithm in 2009 and proved it to be worst-case optimal among a reasonable class of algorithms. Pawlik and Augsten [8] developed a robust algorithm whose asymptotic complexity is smaller than or equal to the complexity of the best competitors for any input instance. Since even $O(n^3)$ time may be too slow for similarity search and/or join operations in XML databases, several approximate methods have been proposed. Garofalakis and Kumar [7] proposed an algorithm to embed the tree edit distance into a high-dimensional L_1-norm space with a guaranteed distortion by means of introduction of an additional 'move' operation. Yang et al. [11] proposed another method to embed the tree edit distance metric into the L_1-norm space, which provides a lower bound of the tree edit distance. Augsten et al. [10] developed a method to give an upper bound for the maximum subtree size that is used to efficiently prune irrelevant subtrees for top-k approximate subtree matching.

On the other hand, the tree edit distance problem is NP-hard for unordered trees [15]. In fact, Zhang and Jiang [20] proved that the problem is MAX SNP-hard even for binary trees, which implies that it is unlikely to admit a polynomial-time approximation scheme. The parameterized complexity has also been studied. Shasha et al. [12] developed an $O(4^{\ell_1+\ell_2}\min(\ell_1, \ell_2)mn)$-time algorithm, where ℓ_1 and ℓ_2 are the number of leaves in T_1 and T_2, respectively, and

Akutsu et al. [24] developed an $O(2^{b_1+b_2}\Delta mn)$-time algorithm, where b_1 and b_2 are the number of branching nodes in T_1 and T_2, respectively, and Δ is the maximum outdegree of T_1 and T_2. An $O(2.62^k poly(m,n))$-time algorithm was developed under the unit-cost edit operation model in [25], where k is the edit distance. Efficient exponential-time algorithms have also been developed [26].

The above algorithms for unordered trees are mostly of theoretical interest. More practical, branch-and-bound type algorithms have been also developed: Horesh et al. [5] developed an A*-type algorithm, and Mori et al. [16] a clique-based algorithm. However, the applicability of these algorithms is limited to trees consisting of up to several tens of nodes. Augsten et al. [6] developed an approximate and efficient method using windowed pq-grams, which are small subtrees of a specific shape.

As mentioned in Section 1, several alternatives to the unordered tree edit distance, including tree alignment [21] and tree inclusion [22], have been proposed. See also [3], [17], [18]. As for tree inclusion, some studies followed from [22]. Bille and Gørtz developed an improved algorithm for ordered trees [27]. Valiente developed an efficient algorithm for a constrained version of unordered trees [28]. Piernik and Morzy introduced a similar problem for ordered trees and developed an efficient algorithm [29]. However, we are interested in introducing the cost to the original definition of tree inclusion for unordered trees. If we do not allow insertions between parents and children, the tree inclusion problem corresponds to the subtree isomorphism problem (i.e., subgraph isomorphism problem for trees), which can be solved in polynomial time for unordered trees [30]. Matoušek and Thomas considered the subgraph isomorphism problem for partial k-trees, which are some extensions of unordered trees, and showed that the problem is NP-hard in general but can be solved in polynomial time if both k and the maximum vertex degree of the pattern tree are bounded by constants [30].

3 Tree Edit Distance and Tree Inclusion

This section reviews the definitions of the tree edit distance for ordered and unordered trees as well as the definition and the algorithm from [22] for unordered tree inclusion. We use the following notation. For any rooted tree T, let $r(T)$ be the root of T, $V(T)$ the set of nodes in T, and $L(T)$ the set of leaves in T. For any $v \in V(T)$, $\ell(v)$ is the label of node v, $chd(v)$ is the set of children of v, and $deg(v)$ is the outdegree of v (i.e., the number of children of v). Furthermore, $T(v)$ is the rooted subtree (connected subgraph) of T induced by v and all of its descendants. Similarly, for a set of nodes $R = \{v_1, \ldots, v_d\}$, $T(R)$ denotes the subforest (the set of rooted subtrees) of T induced by $v_1, \ldots, v_d$ and all their descendants.

3.1 Tree Edit Distance

The tree edit distance [23] is defined via *edit operations*. To simplify the presentation, we will assume that the root $r(T)$ of any tree T has an imaginary parent node $p(T)$ labeled by a unique symbol $\diamond$ that does not occur anywhere else in T and that $p(T) \notin V(T)$. Let T be a tree. An *edit operation* on T is one of the following three operations:

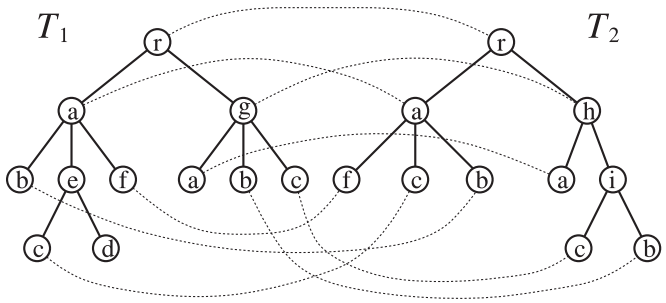


Fig. 1. An example. T_2 is obtained by deletions of the nodes labeled "d" and "e", an insertion of a node labeled "i" (nodes labeled "b" and "c" correspond to a subset of the children in the definition of insertion), and a substitution of a node labeled "g" to "h". Thus, under the unit-cost edit operation model, the tree edit distance is 4. The corresponding unordered edit distance mapping is shown by dotted curves.

(i) Deletion: remove a node $v \in V(T)$ whose parent is u, while letting the children of v become children of u (ii) Insertion (the inverse of a deletion operation): create a new node v having any label except $\diamond$ and attach it as a child of any node $u \in V(T) \cup \{p(T)\}$, while making v the parent of a (possibly empty) subset of the children of u, (iii) Substitution: change the label of a node in $V(T)$ to any label except $\diamond$. See Fig. 1 for an illustration, where imaginary nodes are not shown.

For ordered trees, all operations must preserve the left-to-right node ordering. If a deletion is done on v, the children of v are attached to v's parent at the place of v and in the same left-to-right order. An insertion of v as a child of u makes v a parent of a consecutive subsequence of the children of u.

We assign a *cost* to each edit operation by a *cost function* δ such that $\delta(a,b)$ equals the cost of substituting a node with label a to a node with label b, $\delta(a,-)$ equals the cost of deleting a node with label a, and $\delta(-,a)$ equals the cost of inserting a node with label a. As in many other studies on tree edit distance [17], we assume that $\delta(x,y)$ is a distance metric: $\delta(x,y) \geq 0$, $\delta(x,y) = 0$ if and only if $x = y$, $\delta(x,y) = \delta(y,x)$, and $\delta(x,y) + \delta(y,z) \geq \delta(x,z)$ hold for all x, y, z. The *unit-cost edit operation model* is a cost function δ satisfying $\delta(a,a) = 0$ for all labels a and $\delta(a,b) = 1$ for all labels a, b with $a \neq b$. The *edit distance* between two trees T_1, T_2 is defined as the cost of the minimum-cost sequence of edit operations that transforms T_1 to T_2.

The tree edit distance can be expressed in terms of *edit distance mappings* [17], [23] (or *mappings*, for short), which are also called *Tai mappings* [23]. For two ordered trees T_1 and T_2, $M \subseteq V(T_1) \times V(T_2)$ is *mapping* if the following conditions hold for every $(u_1, v_1), (u_2, v_2) \in M$: (i) $u_1 = u_2$ if and only if $v_1 = v_2$, (ii) u_1 is an ancestor of u_2 if and only if v_1 is an ancestor of v_2, (iii) u_1 is left of u_2 if and only if v_1 is left of v_2. The first condition states that M must be one-to-one, the second condition states that ancestor-descendant relationships must be preserved, and the third condition states that the left-to-right ordering must be preserved. For unordered trees, M is called a *mapping* if conditions (i) and (ii) above hold for every $(u_1, v_1), (u_2, v_2) \in M$. See also Fig. 1.

Now, we can relate any edit mapping M to some sequence of edit operations by letting: (i) nodes in $V(T_1)$ not

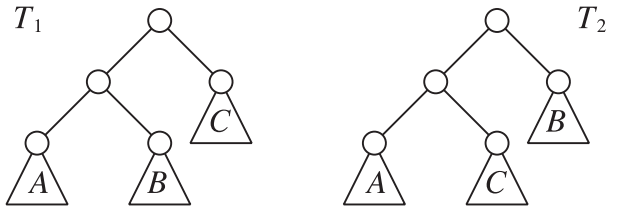


Fig. 2. Comparing the ordered and unordered tree edit distance. The ordered tree edit distance is $2x$ while the unordered tree edit distance is 2.

appearing in M correspond to nodes deleted from T_1, (ii) nodes in $V(T_2)$ not appearing in M correspond to nodes inserted into T_1, (iii) node pairs in M having different labels correspond to substitutions, and define the *cost* of M using δ in the natural manner. Then the cost of any minimum-cost edit mapping is precisely the tree edit distance [17], [23].

We end this section with a brief comparison between the ordered and unordered tree edit distance. Consider T_1 and T_2 in Fig. 2 (similar to Fig. 1 in [21]). The subtree indicated by A (resp., B and C) represents a tree consisting of x nodes all having the same label 'a' (resp., labels 'b' and 'c'), so that $|V(T_1)| = |V(T_2)| = 3x + 2$. The ordered tree distance between T_1 and T_2 is $2x$ whereas the unordered tree distance is 2, under the unit-cost edit operation model. In other words, the gap may be of size $\Omega(n)$, where n is the number of nodes. We remark that in this example, the unordered tree alignment distance [21] is not appropriate because its distance is also $2x$.

3.2 Unordered Tree Inclusion

Here we review the unordered tree inclusion problem and the algorithm of [22]. Let T_1 be the pattern tree and T_2 the text tree. In what follows, m and n denote the size (the number of nodes) of T_1 and T_2, respectively, and D denotes the maximum outdegree among all nodes in T_1. We assume that D bits can be stored in one word of the CPU because all algorithms presented here use $\Omega(2^D m)$ space and thus D bits are needed to specify each memory position.

T_1 is *included* in T_2 if T_2 can be obtained by applying a sequence of insertion operations to T_1. If T_1 is included in T_2, we write $T_1 \prec T_2$. Furthermore, this relation is extended to the case where T_2 can be obtained by applying a sequence of insertion operations to a forest (i.e., a set of rooted trees).

The unordered tree inclusion problem. Given two rooted, unordered trees T_1 and T_2, decide whether $T_1 \prec T_2$.

Note that the problem is equivalent to finding a mapping in which every node in the pattern tree is mapped to a node with the same label in the text tree. Kilpeläinen and Mannila [22] proved that the unordered tree inclusion problem is NP-hard and gave an $O(2^{2D}mn)$-time algorithm for solving it, which we refer to as UnordInclusion. See Algorithm 1 for the pseudocode. The algorithm computes a set $S(v)$ for each $v \in V(T_2)$, defined as:

$$S(v) = \{R \mid (\exists u \in V(T_1))(R \subseteq chd(u)),\ T_1(R) \prec T_2(v)\} \cup \{\{r(T_1)\} \mid T_1 \prec T_2(v)\}.$$

The $S(v)$-sets are computed in a bottom-up fashion. Clearly, $T_1 \prec T_2$ if and only if there exists some $v \in V(T_2)$ such that $\{r(T_1)\} \in S(v)$. As an example, let T_1 and T_2 be the trees in Fig. 3. Then, we have: $S(v_{11}) = \{\emptyset, \{u_4\}\}$, $S(v_{10}) = \{\emptyset\}$, $S(v_7) = \{\emptyset, \{u_3\}, \{u_4\}, \{u_5\}\}$, $S(v_5) = \{\emptyset, \{u_3\}, \{u_5\}\}$, $S(v_4) = \{\emptyset, \{u_4\}, \{u_5\}, \{u_4, u_5\}, \{u_2\}, \{u_3\}\}$, and $S(v_2) = \{\emptyset, \{u_4\}, \{u_5\}, \{u_4, u_5\}, \{u_2\}, \{u_3\}, \{u_2, u_3\}, \{u_1\}\}$.

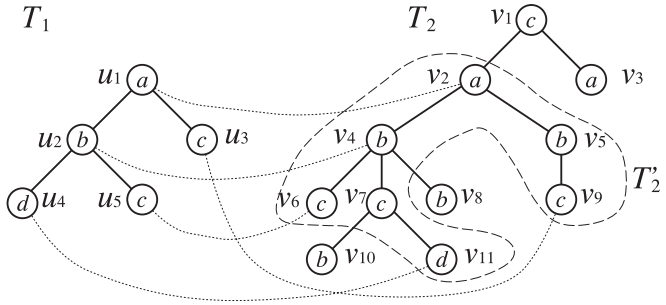


Fig. 3. An example of unordered tree inclusion.

Algorithm 1. UnordInclusion(T_1, T_2)

for all $v \in V(T_2)$ from the leaves to the root **do**
  Let $v_1, \ldots, v_d$ be the children of v;
  /* $d = 0$ if $v \in L(T_2)$ */
  $S \leftarrow \{\emptyset\}$; $S\Delta \leftarrow \emptyset$;
  for all $u \in V(T_1)$ **do**
    $S \leftarrow S \cup \{R \mid R = R_1 \cup \cdots \cup R_d,\ R_i \in S(v_i),\ R_i \subseteq chd(u)\}$;
    if $chd(u) \in S$ and $\ell(u) = \ell(v)$ **then**
      $S\Delta \leftarrow S\Delta \cup \{\{u\}\}$;
  $S(v) \leftarrow S \cup S\Delta$.

4 Minimum-Cost Unordered Tree Inclusion

At a first glance, tree inclusion may appear useful for similar subtree search. However, some drawbacks become apparent when trying to apply it to real data. For example, the costs of insertions are not accounted for. Consider the trees in Fig. 4. Intuitively, T_1^2 looks much more similar to a subtree of T_2 than T_1^1 does, even though both $T_1^1 \prec T_2$ and $T_2^2 \prec T_2$ hold. Another issue is that substitutions of node labels are not allowed in tree inclusion. To overcome these drawbacks, we introduce *costs* into the tree inclusion problem.

Define $D_0(T, T')$ as the minimum cost of transforming T into T' by insertion and substitution operations, where $D_0(T, T') = +\infty$ if there is no such transformation. Also define:

The minimum-cost unordered tree inclusion problem (MinCostIncl). Given two rooted, unordered trees T_1 and T_2,

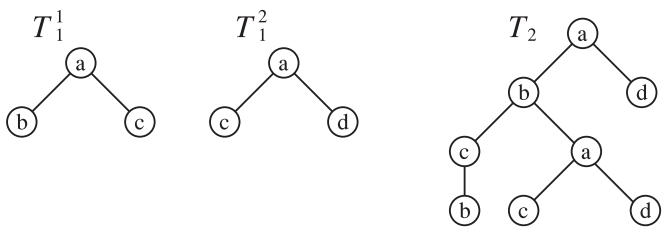


Fig. 4. Both T_1^1 and T_1^2 are included in T_2. However, T_1^2 looks like a more similar subtree since it is isomorphic to a rooted subtree of T_2.

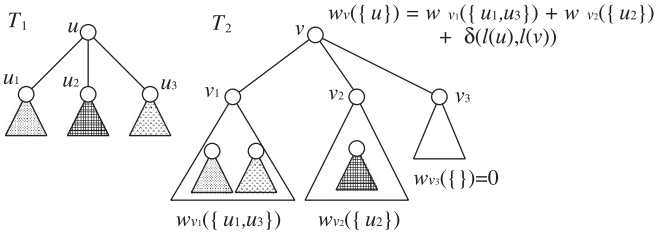


Fig. 5. Illustrating the computation of minimum-cost tree inclusion. Here, u is mapped to v.

determine the value of $D_\prec(T_1, T_2) = \min_{T_2'} D_0(T_1, T_2')$, where T_2' is taken over all connected subgraphs of T_2.

Note that T_2' corresponds to the subtree (of a large tree T_2) similar to T_1. Note also that the definition of $D_\prec$ ignores the costs of inserting irrelevant nodes into T_1. In the example in Fig. 3, the connected subgraph of T_2' giving the minimum cost consists of nodes v_2, v_4, v_5, v_6, v_7, v_9, v_{11} and the costs of inserting v_1, v_3, v_8, v_{10} are ignored. This is reasonable because the purpose is to find a relevant part of T_2 that is similar to T_1. If $D_\prec(T_1, T_2) < +\infty$, we say that T_1 is *included* in T_2 or that T_1 is *embedded* into T_2 (with cost $D_\prec(T_1, T_2)$). It should be noted that the problem **MinCostIncl** is NP-hard in general because tree inclusion corresponds to the problem of deciding whether $D_\prec(T_1, T_2) \neq +\infty$.

4.1 Main Algorithm

We assume that all insertion and substitution operations have non-negative costs because otherwise, there would be cases in which the cost between two isomorphic trees could be greater than the cost between two non-isomorphic trees.

Our algorithm is called MinCostIncl. It extends the algorithm UnordInclusion of [22] described above by also taking the costs of insertions and deletions into account. For certain subsets of nodes $R \subseteq V(T_1)$, MinCostIncl computes a score denoted by $w_v(R)$, which gives the minimum cost of embedding the forest $T_1(R)$ into $T_2(v)$. Here, embedding a forest S into a tree T means that T is obtained by insertion and substitution operations on S, where insertion of a parent of some roots of the current forest is allowed. Then, the required cost is given by: $D_\prec(T_1, T_2) = \min_{v \in V(T_2)} w_v(\{r(T_1)\})$.

The pseudocode of MinCostIncl is listed in Algorithm 2. In the algorithm, all $w_v(S)$ (resp., $W_v(S)$) are implicitly initialized as $w_v(S) \leftarrow +\infty$ (resp., $W_v(S) \leftarrow +\infty$), and $W_v(R) < +\infty$ finally gives the minimum cost of embedding $T_1(R)$ into $T_2(v) - \{v\}$.

Fig. 5 illustrates the core part of the algorithm. It corresponds to the case of $R_1 = \{u_1, u_3\}$, $R_2 = \{u_2\}$, and $R_3 = \{\}$, which means that $T_1(u_1)$ and $T_1(u_3)$ are embedded into $T_2(v_1)$, $T_1(u_2)$ is embedded into $T_2(v_2)$, and no subtree is embedded into $T_2(v_3)$. In this case, the cost of embedding $T_1(u)$ into $T_2(v)$ is given by $w_{v_1}(R_1) + w_{v_2}(R_2) + w_{v_3}(R_3) + \delta(\ell(u), \ell(v))$, where u corresponds to v. It is to be noted that $\delta(\ell(u), \ell(v))$ is computed in "then" part of (#3). We examine all partitions (i.e., all $(R_1, \ldots, R_d)$s) of the children of u and take the minimum cost one (this minimum is computed at (#1)). (#2) takes care of the case in which children of u correspond to descendants of v but u does not correspond to v. (#4) takes care of the case in which u corresponds to a descendant of v.

Algorithm 2. MinCostIncl(T_1, T_2)

for all $v \in V(T_2)$ from the leaves to the root **do**
  Let $v_1, \ldots, v_d$ be the children of v;
  $w_v(\emptyset) \leftarrow 0$;
  $W_v(\emptyset) \leftarrow 0$;
  for all $u \in V(T_1)$ from the leaves to the root **do**
    for all $R_1, \ldots, R_d$ such that $R_i \cap R_j = \emptyset$ (for all $i \neq j$), $w_{v_i}(R_i) < +\infty$, and $R_i \subseteq chd(u)$ **do**
      $R \leftarrow R_1 \cup \cdots \cup R_d$;
      $w \leftarrow w_{v_1}(R_1) + \cdots + w_{v_d}(R_d)$;
      $W_v(R) \leftarrow \min(w, W_v(R))$; (#1)
      $w_v(R) \leftarrow \min(w_v(R), w + \delta(-, \ell(v)))$; (#2)
    if $W_v(chd(u)) < +\infty$ **then** (#3)
      $w_v(\{u\}) \leftarrow W_v(chd(u)) + \delta(\ell(u), \ell(v))$;
    for all v_i such that $w_{v_i}(\{u\}) < +\infty$ **do** (#4)
      $w_v(\{u\}) \leftarrow \min(w_v(\{u\}), w_{v_i}(\{u\}) + \delta(-, \ell(v)))$.

We note that the "**for all** $R_1, \ldots, R_d$"-loop can be implemented efficiently by applying dynamic programming (DP) from the leftmost child to the rightmost child. Indeed, it is enough to replace the loop by the procedure below (MinCostInclSub).

In this procedure, R^1 and R^2 are examined from smaller sets to larger sets. Both $U_v(R)$ and $\hat{W}_v(R)$ maintain the minimum cost to embed the forest with roots R into $T_2(v)$, where $\hat{W}_v(R)$ and $U_v(R)$ maintain a temporal cost and the final cost, respectively.

Algorithm 3. MinCostInclSub

for all $R \subseteq chd(u)$ **do** $U_v(R) \leftarrow +\infty$;
$U_v(\emptyset) \leftarrow 0$;
for $i = 1$ to $|chd(v)|$ **do**
  for all $R \subseteq chd(u)$ **do** $\hat{W}_v(R) \leftarrow +\infty$;
  for all $R^1 \subseteq chd(u)$ **do**
    for all $R^2 \subseteq chd(u) \setminus R^1$ **do**
      $\hat{W}_v(R^1 \cup R^2) \leftarrow \min(\hat{W}_v(R^1 \cup R^2), U_v(R^1) + w_{v_i}(R^2))$;
  for all $R \subseteq chd(u)$ **do** $U_v(R) \leftarrow \hat{W}_v(R)$;
for all $R \subseteq chd(u)$ **do**
  $W_v(R) \leftarrow U_v(R)$;
  $w_v(R) \leftarrow \min(w_v(R), W_v(R) + \delta(-, \ell(v)))$;

Theorem 1. *$D_\prec(T_1, T_2)$ can be computed in $O(2^{2D}mn)$ time using $O(2^D mn)$ space.*

Proof. The correctness of the algorithm can be seen by observing that the following four cases are properly handled in a bottom-up manner (see also Figs. 5 and 6). (a) u corresponds to v, and u is a leaf: Since $chd(u) = \emptyset$ and $W_v(\emptyset) = 0$ hold, this case is covered by (#3). (b) u corresponds to v, and $T_1(u)$ is included in $T_2(v)$: Since $T_1(chd(u))$ can be embedded into $T_2(v) - \{v\}$, $W_v(chd(u)) < +\infty$ holds. Therefore, this case is covered also by (#3). (c) u corresponds to a descendant of v, and $T_1(u)$ is included in $T_2(v)$: Since $w_{v_i}(chd(u)) < +\infty$ holds for some $v_i \in chd(v)$, this case is covered by (#4). (d) $T_1(R')$ can be embedded into $T_2(v) - \{v\}$ where $R' \subseteq chd(u)$: This case is covered by (#1) and (#2).

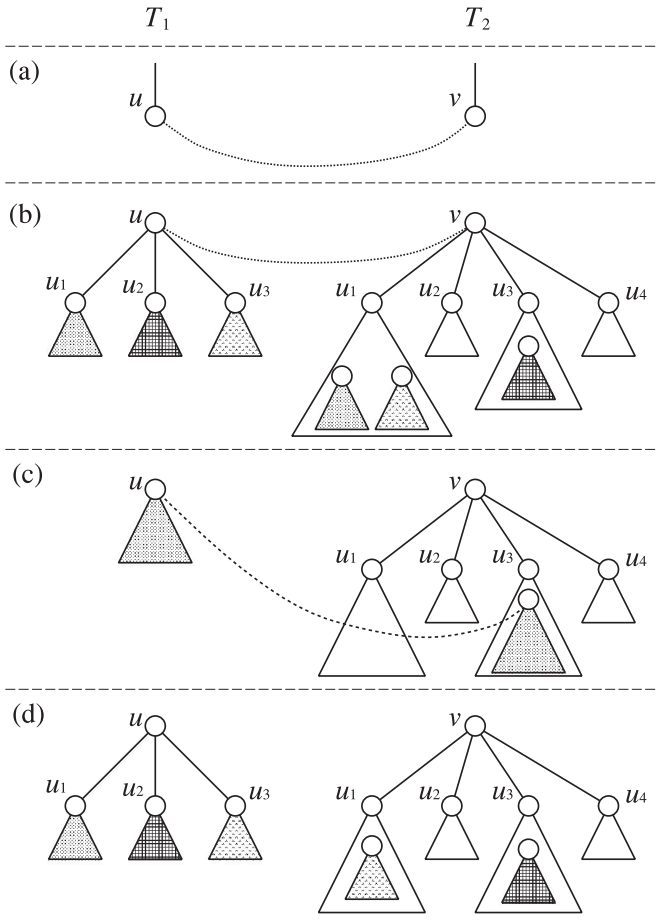


Fig. 6. The four cases in the proof of Theorem 1.

Next we analyze the time complexity. For the inner DP procedure MinCostInclSub, it is clear from the pseudocode that the innermost assignment line is executed for less than $|chd(v)| \times 2^D \times 2^D$ times. Since we assume that D bits can be represented in a word, operations such as $R^1 \cap R^2$ and $R^1 \cup R^2$ can be done in constant time, which means that the innermost assignment line can be done in constant time. Since the main loop is the most time-consuming part of MinCostInclSub. we can see that MinCostInclSub can be done in $O(|chd(v)|2^{2D})$ time. It is clear from the pseudocode of the main procedure that the total time required for MinCostInclSub is $O(2^{2D}m\sum_v |chd(v)|) = O(2^{2D}mn)$. Since repeated execution of MinCostInclSub is the most time-consuming part, the main procedure works in $O(2^{2D}mn)$ time. It is straightforward to see that the space complexity is $O(2^D mn)$. □

It is to be noted that all matched positions of T_2 can be enumerated by outputting all vs such that $\min_{v \in V(T_2)} w_v(\{r(T_1)\}) = D_{\prec}(T_1, T_2)$. Although one mapping (i.e., one embedding) can be retrieved for each matched position by using the standard traceback technique [31] (in $O(n)$ time per matched position), there may exist an exponential number of embeddings even for one matched position and thus exponential time may be required if all possible mappings should be output.

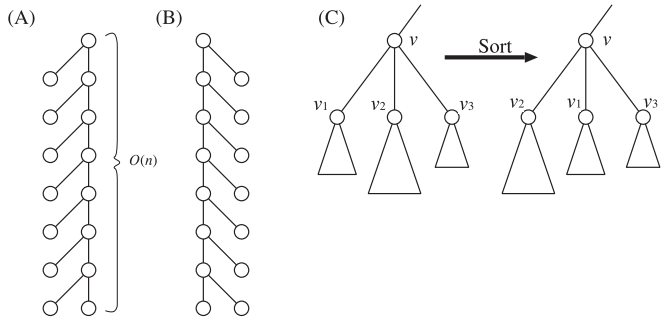


Fig. 7. Improving the space complexity. (A) needs $O(n + 2^D mn)$ space, whereas (B) needs $O(n + 2^D m)$ space. To minimize the space, it is enough to sort the children as in (C).

4.2 Improvement of the Space Complexity

It is possible to reduce the space to $O(n + 2^D m \log(n))$ in the following way. We can execute the above DP procedure in a depth-first manner for T_2. Let $r = x_0, x_1, \ldots, x_h$ be the path from the root to the current node x_h. We only need to keep $w_{x_i}(R)$s only for x_is each of which is not the leftmost child of its parent. Therefore, we need $O(n + 2^D mH)$ space, where $H = \max_{(x_0,\ldots,x_h)} |\{x_i \mid x_i \text{ is not the leftmost child}\}|$. For example, H is $O(n)$ for Fig. 7 A, whereas H is $O(1)$ for Fig. 7 B.

In order to minimize H, it is enough to sort the children $v_1, \ldots, v_k$ of each node v so that

$$|V(T_2(v_{i_1}))| \geq |V(T_2(v_{i_2}))| \geq \cdots \geq |V(T_2(v_{i_k}))|$$

holds (see Fig. 7(C)). Then, we can see that $|V(T_2(v_{i_j}))| < \frac{1}{2}|V(T_2(v))|$ holds for $j = 2, 3, \ldots, k$, which immediately shows that H is $O(\log n)$.

Theorem 2. *$D_{\prec}(T_1, T_2)$ can be computed in $O(2^{2D}mn)$ time using $O(n + 2^D m \log(n))$ space.*

The space-efficient version of the algorithm is denoted by MinCostIncl-SpS, where "SpS" means Space Saving. However, it has one practical disadvantage: traceback cannot be done. Therefore, it may be used for screening in a first step and then actual mappings between T_1 and T_2' should be obtained by running $MinCostIncl(T_1, T_2(v))$ for all v such that $w_v(\{u\})$ is less than or equal to some specified threshold value.

5 Allowing a Small Number of Deletions in the Pattern Tree

In the problem **MinCostIncl** introduced in the previous section, deletions of nodes in the pattern tree are not allowed. This may be problematic in some applications because $D_{\prec}(T_1, T_2)$ can be $+\infty$ while the tree edit distance is not infinity; e.g., if the height of T_1 is larger than the height of T_2, or if T_1 is a rooted balanced binary tree with four leaves and T_2 is a rooted caterpillar (a tree in which every node has at most one child that is an internal node) with at least four leaves. It is known that if one allows arbitrary deletions of nodes in the pattern tree, the problem becomes NP-hard even when the maximum outdegree is bounded by two [15]. However, if we limit the number K of nodes that may be deleted from the pattern

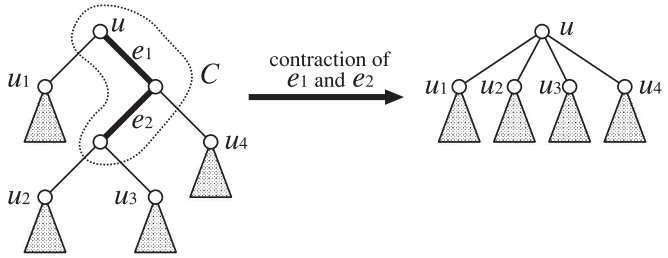


Fig. 8. A contraction of a connected subtree. In this case, $chd(u, C) = \{u_1, u_2, u_3, u_4\}$.

tree, we can still get a fixed-parameter tractable algorithm with respect to the parameters D and K, where D is the maximum outdegree of all nodes in T_1. In other words, the exponent in the time complexity depends only on D and K, and not on m or n. This is extremely useful when just a few deletions are needed. Below, we explain the details of this method.

The key observation is that the number of trees of size $K+1$, each of which is a connected subgraph of a given tree T of bounded outdegree D and includes the root of T, does not depend on the size of T, where it gives the maximum number of deletion patterns. The reason why we focus on the size of a connected subgraph is that it plays a key role in the analysis of the algorithm when allowing deletions of at most (not necessarily connected) K nodes.

Proposition 1. *The number of connected subgraphs of size $K+1$ which contain the root of a tree of maximum outdegree D is at most $\frac{2(eD)^K}{K^{3/2}}$, where e is the base of the natural logarithm ($e = 2.718\ldots$).*

Proof. According to Lemma 2.1 (a) in [32], the number is at most $\frac{1}{K+1} \cdot \binom{(K+1)D}{K}$. By the comments immediately before Lemma 2.1 in [32], this is strictly less than $\frac{e^{K+1}D^K}{(K+1)\sqrt{2\pi(K+1)}}$. Now observe that $\frac{e^{K+1}D^K}{(K+1)\sqrt{2\pi(K+1)}} = \frac{(eD)^K}{(K+1)^{3/2}} \cdot \frac{e}{\sqrt{2\pi}} < \frac{2(eD)^K}{K^{3/2}}$. □

Note that the number in Proposition 1 is lower-bounded by D^K because such a tree may contain D^K different downwards paths of length K beginning at the root. Therefore, we cannot expect any significant improvements of the above bound.

Next, we modify our algorithm MinCostIncl to allow deletions of at most K nodes from T_1. To this end, for every node u of T_1, we consider all connected subtrees of size at most $K+1$ rooted at u. For each such connected subtree, we consider the result of applying deletion operations on all nodes except u, giving the contraction of the connected subtree into the root node (see Fig. 8).

In the following, $chd(u, C)$ denotes the set of children of u after deleting $C - \{u\}$, and $w_v^k(S)$ denotes the cost to embed $T_1(S)$ to $T_2(v)$ under the condition that k nodes are deleted from $T_1(S)$. As in MinCostIncl, all $w_v^k(S)$ (resp., $W_v^k(S)$) are implicitly initialized as $w_v^k(S) \leftarrow +\infty$ (resp., $W_v^k(S) \leftarrow +\infty$), and $\min_{v \in V(T_2), k \in \{0,\ldots,K\}} w_v^k(\{r(T_1)\})$ finally gives the minimum cost of including T_1 in T_2. The pseudocode is listed in Algorithm 4.

Algorithm 4. MinCostInclWithDel(T_1, T_2, K)

for all $v \in V(T_2)$ **do** $w_v^0(\emptyset) \leftarrow 0$;
for all $u \in V(T_1)$ from the leaves to the root **do**
  for all $v \in V(T_2)$ **do**
    for $k = 0$ **to** K **do** $\hat{w}_v^k \leftarrow +\infty$;
  for all connected subgraphs C rooted at u such that $|C| \leq K+1$ **do**
    for all $v \in V(T_2)$ from the leaves to the root **do**
      for $k = 0$ **to** K **do**
        for all $R \subseteq chd(u, C)$ **do** $W_v^k(R) \leftarrow +\infty$;
      Let $v_1, \ldots, v_d$ be the children of v;
      for all $w_{v_1}^{k_1}(R_1), \ldots, w_{v_d}^{k_d}(R_d)$ such that $R_i \cap R_j = \emptyset$, $w_{v_i}^{k_i}(R_i) < +\infty$, and $R_i \subseteq chd(u, C)$ **do**
        $R \leftarrow R_1 \cup \cdots \cup R_d$;
        $w \leftarrow w_{v_1}^{k_1}(R_1) + \cdots + w_{v_d}^{k_d}(R_d)$;
        $k \leftarrow k_1 + \cdots + k_d$;
        $W_v^k(R) \leftarrow \min(w, W_v^k(R))$;
        $w_v^k(R) \leftarrow \min(w_v^k(R), w + \delta(-, \ell(v)))$;
      for $k = |C| - 1$ **to** K **do**
        $\hat{w}_v^k \leftarrow \min(\hat{w}_v^k, W_v^{k-|C|+1}(chd(u, C))$
          $+\delta(\ell(u), \ell(v)) + \sum_{x \in C - \{u\}} \delta(\ell(x), -))$;
  for all $v \in V(T_2)$ **do**
    for $k = 0$ **to** K **do**
      $w_v^k(\{u\}) \leftarrow \min(w_v^k(\{u\}), \hat{w}_v^k)$;
      for all v_i such that $w_{v_i}^k(\{u\}) < +\infty$ **do**
        $w_v^k(\{u\}) \leftarrow \min(w_v^k(\{u\}),$
          $w_{v_i}^k(\{u\}) + \delta(-, \ell(v)))$.

As in the case of MinCostIncl, we can efficiently execute the innermost '**for all**'-loop by applying DP from the leftmost child to the rightmost child. Indeed, this loop can be replaced by procedure MinCostIwdSub in Algorithm 5. Although the meaning of $W_v^k(R)$ is slightly different from that in the above procedure, we obtain the same score.

Algorithm 5. Procedure MinCostIwdSub

for $k = 0$ **to** K **do**
  for all $R \subseteq chd(u, C)$ **do** $U_v^k(R) \leftarrow +\infty$;
$U_v^0(\emptyset) \leftarrow 0$;
for $i = 1$ to $|chd(v)|$ **do**
  for all $R \subseteq chd(u, C)$ **do**
    for $k = 0$ **to** K **do** $\hat{W}_v^k(R) \leftarrow +\infty$;
  for all $R^1 \subseteq chd(u, C)$ **do**
    for all $R^2 \subseteq chd(u, C) \setminus R^1$ **do**
      for $k^1 = 0$ **to** K **do**
        for $k^2 = 0$ **to** $K - k^1$ **do**
          $k \leftarrow k^1 + k^2$;
          $\hat{W}_v^k(R^1 \cup R^2) \leftarrow \min(\hat{W}_v^k(R^1 \cup R^2),$
            $U_v^{k^1}(R^1) + w_{v_i}^{k^2}(R^2))$;
  for $k = 0$ **to** K **do**
    for all $R \subseteq chd(u, C)$ **do** $U_v^k(R) \leftarrow \hat{W}_v^k(R)$;
for $k = 0$ **to** K **do**
  for all $R \subseteq chd(u, C)$ **do**
    $W_v^k(R) \leftarrow U_v^k(R)$;
    $w_v^k(R) \leftarrow \min(w_v^k(R), W_v^k(R) + \delta(-, \ell(v)))$;

Theorem 3. *Suppose that T_1 can be embedded into T_2 with the minimum cost $w_{\min}$ under the condition that at most K nodes are deleted from T_1. Then, such an embedding can be obtained in $O((eD)^K K^{1/2} 2^{2(DK+D-K)} mn)$ time.*

Proof. The proof of the correctness of the algorithm is analogous to that for MinCostIncl although MinCostInclWithDel is more involved due to introduction of deletions of nodes from T_1. Recall that in MinCostIncl, $w_v(R)$ is calculated from $w_{v_i}(R_i)$s for children $v_1, \ldots, v_d$ of v. On the other hand, in MinCostInclWithDel, we must maintain the number of deleted nodes in pattern subtrees. Therefore, instead of $w_v(R)$ (resp., $W_v(R)$), we maintain $w_v^k(R)$ (resp., $W_v^k(R)$) for each k where k denotes the number of deleted nodes in $T_1(R)$. Note also that, instead of $R \subseteq chd(u)$, we need to consider $R \subseteq chd(u, C)$ because a set of children of a node u is given by $chd(u, C)$ in a contracted subtree (see also Fig. 8). Furthermore, since $w_v^k(R)$ is computed from $w_{v_i}^{k_i}(R_i)$s, we also need to take care so that k is obtained from k_is (i.e., the number of deleted nodes in $T_1(R)$ is the sum of the number of deleted nodes in $T_1(R_i)$s). Except these points, the structure of MinCostInclWithDel is the same as that of MinCostIncl. Since the modified points are adequately handled in MinCostInclWithDel, the correctness follows.

Here, we analyze the time complexity. First note that the size of $chd(u, C)$ is bounded by $DK + D - K$. Then we can see from the pseudocode that MinCostIwdSub can be executed in $O(K^2 2^{2(DK+D-K)} |chd(v)|)$ time. By Proposition 1 and the pseudocode of the main procedure, it is seen that the total time required for MinCostIwdSub is $O((eD)^K K^{1/2} 2^{2(DK+D-K)} m \sum_v |chd(v)|) = O((eD)^K K^{1/2} 2^{2(DK+D-K)} mn)$. This is the most time-consuming part, and the theorem follows. □

As an example, it holds that $D_{\prec}(T_1, T_2) = +\infty$ for the trees T_1 and T_2 in Fig. 8. However, if we let $K = 2$ in Theorem 3, the cost will be 2 under the unit-cost edit operation model.

We assumed above that at most K nodes are deleted in total. However, the algorithm can easily be modified so that at most K nodes are deleted from each connected subgraph C by replacing K with m except "$|C| \leq K + 1$". Then $O(m^2)$ values of k^1 and k^2 are considered in the innermost loop (instead of $O(K^2)$ values), and the time complexity increases accordingly.

Corollary 1. *Suppose that T_1 can be embedded into T_2 with the minimum cost $w_{\min}$ under the condition that the size of each of the contracted connected components is at most $K + 1$. Then, such an embedding can be obtained in $O((eD)^K 2^{2(DK+D-K)} K^{-3/2} m^3 n)$ time.*

6 Experimental Results

We performed a number of experiments to evaluate our proposed methods. To verify the theoretically derived complexities and to compare the practical performance of MinCostIncl to that of the original unordered tree inclusion algorithm in [22], we used both synthetic data and real data. To obtain synthetic data, we modified the tree generation algorithm in [11] so that the resulting tree size and the maximum outdegree can be specified. As for real data, we used glycan data taken from the KEGG database [33] and weblogs data [34]. Furthermore, to demonstrate the usefulness of MinCostIncl for searching bibliographical data, we applied it to data from ACM, DBLP, and Google Scholar. Synthetic data and glycan data were also used to assess the processing efficiency of MinCostInclWithDel.

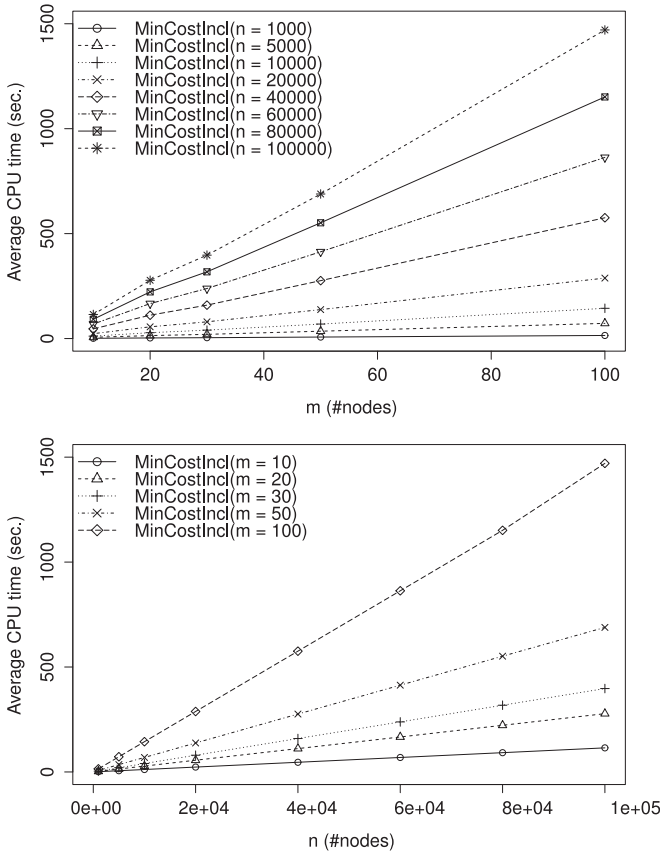


Fig. 9. Execution times of MinCostIncl for varying sizes of pattern trees (top) and text trees (bottom). Note that $m \leq n$ must hold from the definition of MinCostIncl.

All experiments were performed on a PC cluster with Intel(R) Xeon(R) CPU E5-2690 2.90 GHz and 35.87 GB memory, running on a Linux operating system. Our algorithms were implemented using the C++ language and each execution was performed as a single process (i.e., no parallel processes), where very minor simplifications were done in the implemented versions.

In this section, m, n, d, D and ℓ denote the size of the pattern tree ($|V(T_1)|$), the size of the text tree ($|V(T_2)|$), the average outdegree, the maximum outdegree, and the alphabet size ($|\Sigma|$), respectively.

6.1 Processing Efficiency

6.1.1 Results on Synthetic Data

To evaluate how the running time of MinCostIncl depends on the sizes of the input trees, we randomly generated 100 pairs of pattern and text trees and measured the average CPU time for each pair. The parameters m and n were varied and the other parameters were fixed as $(d, D, \ell) = (3, 5, 10)$ for both the pattern and text trees. The results are shown in Fig. 9. We can observe from this figure that the

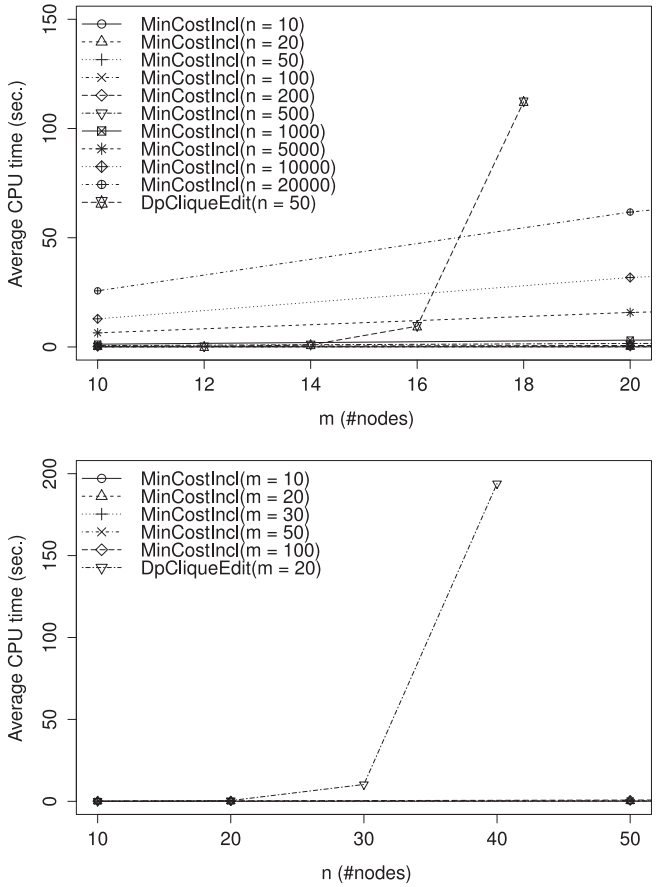


Fig. 10. Comparison between MinCostIncl and DpCliqueEdit for varying sizes of pattern trees (top) and text trees (bottom).

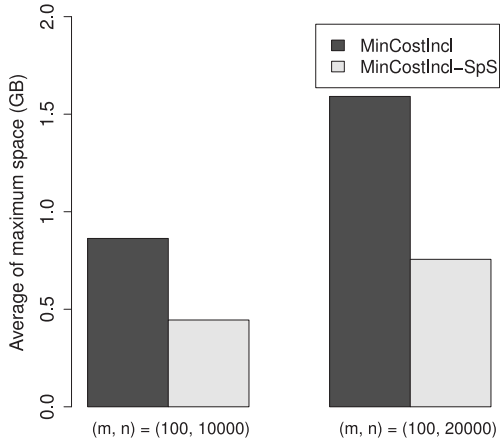


Fig. 11. The reduction in memory usage for large text data.

computation time increases linearly with both m and n, which matches the theoretical bound of $O(2^{2D}mn)$ on the running time. Note that MinCostIncl is fast for large text trees (e.g., $n = 100,000$). Although it is not fast enough for real-time applications, the performance is allowable for batch processing. Furthermore, when there are many small or medium size text trees, we may use a simple parallel processing method (i.e., searching against different text trees independently). We also examined the case of $m = 30$, $n = 10,000$ and $(d, D, \ell) = (5, 7, 10)$. In this case, the average CPU time was around 300 seconds, which is still quite fast considering the hardness of the problem. These results suggest that similar subtree search against large tree data can be done efficiently.

Next, we compared MinCostIncl to DpCliqueEdit [16] under the same settings. (DpCliqueEdit is the fastest currently available algorithm for computing the unordered tree edit distance, and is based on a combination of dynamic programming and maximum-weighted cliques.) The results are shown in Fig. 10. Here, we can see that MinCostIncl is much faster than DpCliqueEdit. Indeed, the CPU time of DpCliequeEdit rapidly increases when text tree size is about 30 as shown in Fig. 10, whereas MinCostIncl works even for text tree whose size is 100,000 with linear increase of CPU time as shown in Fig. 9. We can observe similar characteristics for pattern tree. The tree size limitation of the state-of-the-art unordered tree matching algorithm is a serious problem in practical use. Actually, the maximum tree sizes of glycan, weblog, and bibliographic data sets used in this paper is 38, 144, 12 (for pattern trees) and 54, 255, 521 (for text trees), respectively. The proposed MinCostIncl can calculate most of them in practical time. It can significantly broaden applicable data.

We also compared the average amount of memory used by MinCostIncl to its space-efficient version MinCostIncl-SpS. Here, we selected first 10 pairs of pattern and text trees from 100 generated pairs with $(m, n) = (100, 10,000)$ and $(100, 20,000)$, respectively, while the other parameters were fixed to $(d, D, \ell) = (3, 5, 10)$. The memory usage was tracked using the "ps"-command in Linux (to be precise: while true; do ps auxww | grep $<$process ID$>$; sleep 1; done). Fig. 11 shows the result. We observe that MinCostIncl-SpS uses less memory than MinCostIncl although the difference is not as significant as we had expected.

To verify the usefulness of introducing costs into the unordered tree inclusion problem, we compared our algorithm MinCostIncl and the algorithm of [22] for the original unordered tree inclusion problem. We generated a random pattern tree and searched for a similar subtree in 100 randomly generated text trees for varying m and n, while setting (d, D, ℓ) to $(3, 5, 3)$. The number of hits in each case is displayed in Table 1. Note that for each pair of pattern and text trees, tree inclusion returns either yes (hit) or no (no hit), whereas MinCostIncl returns the cost and thus a pair with a finite cost is regarded as a hit. The table shows that there was no hit according to tree inclusion when m was greater than or equal to 20, whereas there were many hits by MinCostIncl even when $m = 100$, suggesting that tree inclusion is only useful when the

TABLE 1
Comparison of Unordered Tree Inclusion and Minimum-Cost Unordered Tree Inclusion

(m, n)	#hits by tree inclusion	#hits by MinCostIncl
$(10, 50)$	8 / 100	100 / 100
$(20, 100)$	0 / 100	78 / 100
$(30, 150)$	0 / 100	97 / 100
$(50, 250)$	0 / 100	75 / 100
$(100, 500)$	0 / 100	27 / 100

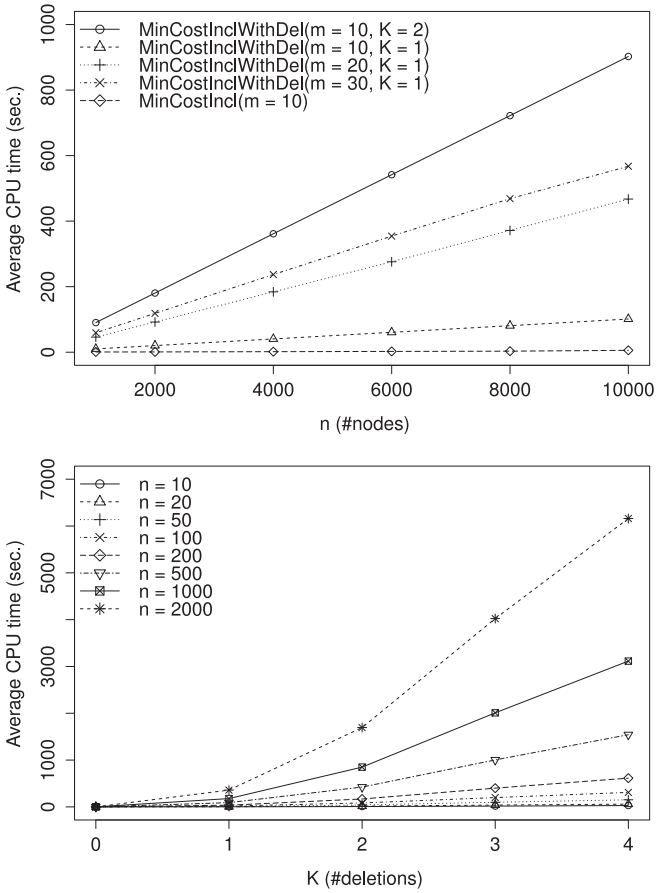


Fig. 12. Execution times for MinCostInclWithDel with varying n (top) and varying K with $m = 10$ (bottom).

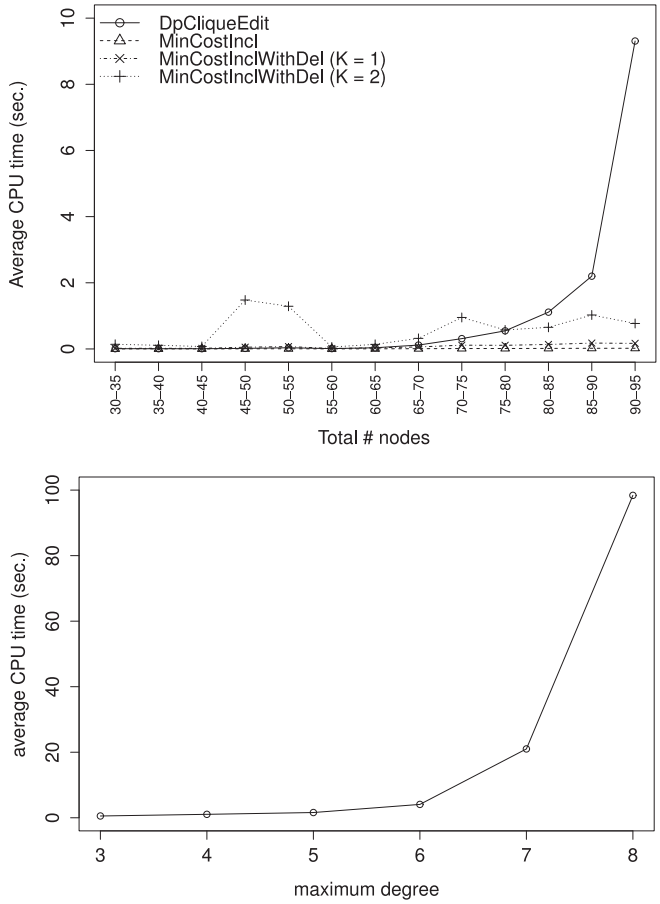


Fig. 13. Execution times for glycan data (top) and weblogs data (bottom).

pattern trees are small. The number of hits by MinCostIncl may seem too large, but by only examining hits with low computed costs (or just adjusting the threshold directly), one can expect to identify the required subtrees. The usefulness of MinCostIncl is discussed in more detail for bibliographic data in Section 6.2.

Processing efficiency of MinCostInclWithDel was also examined by using synthetic data. The results are shown in Fig. 12, where the top and bottom panels mainly show the dependencies on n and K, respectively. From these results, it is seen that MinCostInclWithDel is still fast if K is small and m is not large.

6.1.2 Results for Real Data

We evaluated the efficiency of MinCostIncl and MinCostInclWithDel for real data, glycan data in the KEGG database [33], including comparison with DpCliqueEdit [16]. As in [16], we randomly selected 100 pairs for each interval of total number of nodes of pattern and text trees and computed the average CPU time per pair. The result is shown in Fig. 13 (top). It is seen that the CPU time of DpCliqueEdit increases rapidly when the total size exceeds 70 whereas the CPU time of MinCostIncl and MinCostInclWithDel remains small. This is reasonable because the unordered tree edit distance problem is hard even for trees of outdegree two [15] while MinCostIncl works in polynomial time if the maximum outdegree is bounded by a constant, and here the maximum outdegree of glycan data is 6. MinCostInclWithDel also works in polynomial time if the maximum outdegree and K are bounded by constants. Although some nonmonotonicity is observed for MinCostInclWithDel, it may be due to fluctuation of the maximum outdegree or tradeoff between sizes of pattern and text trees.

We also evaluated the efficiency of MinCostIncl for another type of real data: weblogs data [34]. In this case, we selected trees with $50 - 500$ nodes and maximum outdegree 50, which resulted in a data set of $2,481$ trees. In this experiment, for each $D = 3, \ldots, 9$, we randomly selected 100 pairs of pattern trees with maximum outdegree D and text trees from the data set, and computed the average CPU time, where the total number of nodes of each pair was limited to less than or equal to 500. The result is shown in Fig. 13 (bottom). Here, the CPU time increases rapidly at around $D = 7$. Recall that the time complexity of MinCostIncl is $O(2^{2D}mn)$. This indicates that MinCostIncl is useful when the outdegree of the pattern tree is at most 7. Notice that this limitation is only for the pattern tree and MinCostIncl works efficiently for wide text trees.

6.2 Tree-Based Bibliographic Matching

To evaluate the usefulness of MinCostIncl in a practical setting, we applied it to bibliographic matching [35], [36], which is a typical entity resolution problem. Bibliographic matching is usually solved by measuring similarities at field-level such as the author or the article title,

TABLE 2
Summary of the Benchmark Datasets

	ACM-DBLP	Scholar-DBLP
pattern articles	2,616	2,616
DBLP articles	2,294	64,263
common articles	2,224	5,347
blocking result	4,000	20,000

TABLE 3
Bibliographic Data in Leipzig Benchmark Dataset

title	authors	venue	year
Contextualizing the information ...	M. P. Papazoglou, J. Hoppenbrouwers	SIGMOD Record	1999
⋮	⋮	⋮	⋮

which are then combining into the record-level similarity [35]. In these studies, bibliographic data is assumed to be separated into well-defined fields that are known in advance. Tree matching techniques have been applied to bibliographic matching [10], [11], where corresponding fields between bibliographic records are mapped to each other and their similarity is calculated simultaneously.

6.2.1 Data Set

We used bibliographic data included in the benchmark datasets[1] provided by the authors of [36]. It contains bibliographic data selected from DBLP, Google Scholar (Scholar, for short), and the ACM digital libraries (ACM, for short). The benchmark dataset designed for bibliographic matching between the DBLP and Scholar datasets is referred to as Scholar-DBLP, whereas the dataset between the DBLP and ACM datasets is referred to as ACM-DBLP. The ACM-DBLP dataset contains articles that are included in both its selected ACM and DBLP data. One task of bibliographic matching is to detect these *common articles*, when given the datasets of DBLP and ACM. See Table 2 for the numbers of articles, where *pattern articles* stand for the ACM articles in the ACM-DBLP dataset.

The Scholar-DBLP dataset similarly provides the articles that are included in both its selected Scholar and DBLP data (see Table 2). Note that the number of common articles is larger than pattern articles because a DBLP article is matched with multiple Scholar articles in the dataset. The bibliographic data consists of four fields, namely, "title", "authors", "venue", and "year". Authors are concatenated with punctuation into a single string as in Table 3.

We converted the bibliographic records of ACM and Scholar to trees by: (i) setting the field as a node of the tree, (ii) attaching the value as a leaf of corresponding node, and (iii) splitting the authors into separate authors, choosing at most three authors, and inserting a node "authors". (In this experiment, we omit the field "venue".) We used at most three authors for the pattern trees because MinCostIncl is less efficient when the maximum outdegree is large. Fig. 14a shows an example of the converted tree.

We downloaded the entire dataset of DBLP and used the tree-structured XML data with the insertion of a node "authors" above the list of "author"s. Fig. 14b shows an example of a text tree obtained from DBLP data.

Before applying the tree inclusion/matching, we selected candidate pairs of matched articles for ACM-DBLP and Scholar-DBLP datasets by first calculating the Jaccard coefficient of each pair of articles in each dataset, regarding an article as bag-of-words of values in the four fields, and then choosing the top-k pairs of articles. The "blocking result" in Table 2 shows the selected numbers of article pairs for each dataset.

1. http://dbs.uni-leipzig.de/de/research/projects/object_matching/fever/benchmark_datasets_for_entity_resolution

6.2.2 Tag Mapping

We first evaluated the performance of field/tag mapping of MinCostIncl. In this experiment, the tags of the pattern tree were renamed as follows:

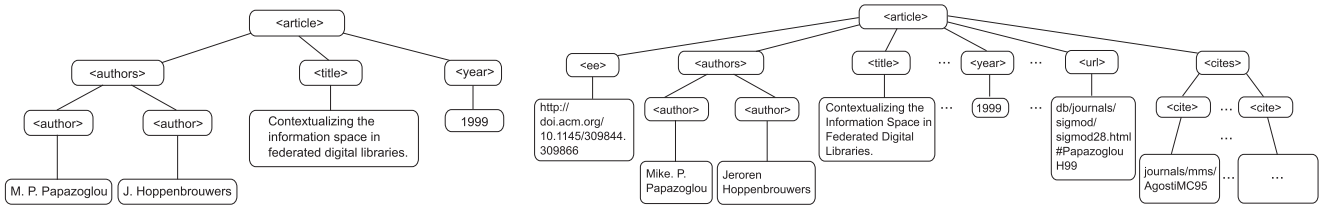
"article" → "0", "authors" → "1", "author" → "2",
"title" → "3", "year" → "4",

so that different sets of tags were used between pattern and text trees. After applying MinCostIncl to pattern and text tree pairs obtained by blocking as described in the previous section, we counted the mappings of tags between pattern and text trees. Table 4 displays a major portion of the confusion matrix for ACM-DBLP and Scholar-DBLP datasets.

Each column and row respectively stands for the tags of DBLP and ACM/Scholar data sets, and each cell shows the number of times the proposed MinCostIncl mapped the corresponding tags. The last line "others" shows the number that the ACM/Scholar tag is mapped to a tag or a leaf in the text tree that are not listed in the table. In spite of significant amount of false pairs of pattern and text trees, many tags were correctly mapped, as shown in the table. The correct tag mapping can be obtained by the maximum matching in a bipartite graph of the confusion matrix. This result indicates that MinCostIncl can find correct tag mapping based on the value similarity even when the tag names are totally different.

6.2.3 Bibliographic Matching

We compared the performance of MinCostIncl to the ordered tree edit distance when applied to bibliographic matching. In this experiment, we used the RTED tool [8] to compute the ordered tree edit distance. We also assumed that author names are compared by their last names (cost is 0 if the last names are exactly the same, otherwise it is 1), and the substitution cost for a pair of title nodes is 0 if the Jaccard coefficient is greater than or equal to 0.8, otherwise it is 1, because it was difficult to customize substitution costs of the RTED tool. All candidate pairs of pattern and text trees obtained by blocking, as described in Section 6.2.1, were ordered according to the minimum-cost of tree inclusion computed by MinCostIncl and the tree edit distance computed by RTED. The



(a) a pattern tree from ACM (b) a text tree in DBLP

Fig. 14. Pattern and text trees for bibliographic search.

average CPU time of MinCostIncl and RTED for each candidate pair is shown in Table 5. Notice that although RTED is much faster than MinCostIncl, MinCostIncl is still reasonably fast.

Figs. 15 and 16 show the receiver operating characteristic (ROC) curves for ACM-DBLP and Scholar-DBLP, respectively, where an ROC curve is a measure of the binary classification performance obtained by plotting the true positive rate (i.e., sensitivity) against the false positive rate (i.e., 1-specificity) with varying threshold. "MinCostIncl" and "RTED" respectively show the curves obtained by directly applying MinCostIncl and RTED. AUC following the label in the legend stands for the *area under the curve* (a higher AUC implies a better result). As we can see in the figures, MinCostIncl performed much better than RTED for both ACM-DBLP and Scholar-DBLP. The reason is partly because RTED calculates the deletion costs for nodes that appear only in the text trees, such as "cites" and "url" in Fig. 14b. As a result, a large text tree tends to be dissimilar to a pattern tree when the distance is measured by the tree edit distance. To avoid this affect, we removed these nodes in the text trees and recalculated the cost and tree edit distance. "MinCostIncl[Filtered]" and "RTED[Filtered]" in Figs. 15 and 16 show the ROC curves for these deleted text trees. As shown in the graphs, MinCostIncl is better than RTED even if we remove the nodes that appear only in the text trees.

In these experiments, the corresponding nodes in pattern and text trees were arranged in the same order. For example, authors are located to the left of the title. Since MinCostIncl was designed for unordered trees, it is not affected by the left-to-right ordering of nodes, whereas the order affects the result of RTED considerably. To evaluate the impact of the node order, we randomly permuted the authors in the text trees and recalculated the tree edit distance. "RTED[Random]" shows the resultant ROC curve. As shown in the figures, AUC is degraded to almost the same level for RTED.

In summary, MinCostIncl is robust against permutations in the node orderings as well as extra nodes in text trees.

TABLE 4
Confusion Matrices of Tag Mapping by MinConstInc

(a) ACM-DBLP

	0 (article)	1 (authors)	2 (author)	3 (title)	4 (year)
article	**3893**	0	0	0	0
authors	0	**3672**	0	0	0
author	0	0	**7962**	0	0
title	0	0	0	**3378**	0
year	0	0	0	0	**2987**
url	0	10	0	283	529
cites	0	95	0	44	338
ee	0	71	0	148	102
cite	0	0	199	0	0
others	0	45	126	40	37

(b) Scholar-DBLP

	0	1	2	3	4
article	**18475**	0	0	0	0
authors	0	**13519**	0	0	0
author	0	0	**27217**	0	0
title	0	0	0	**15680**	0
year	0	0	0	0	**5082**
url	0	0	0	1167	2772
cite	0	0	5768	0	0
editors	0	1798	0	0	0
editor	0	0	3365	0	0
others	0	3158	0	1261	2334

TABLE 5
Execution Times of MinCostIncl and RTED

	MinCostIncl (msec.)	RTED (msec.)
ACM-DBLP	14.1	1.9
Scholar-DBLP	10.2	1.6

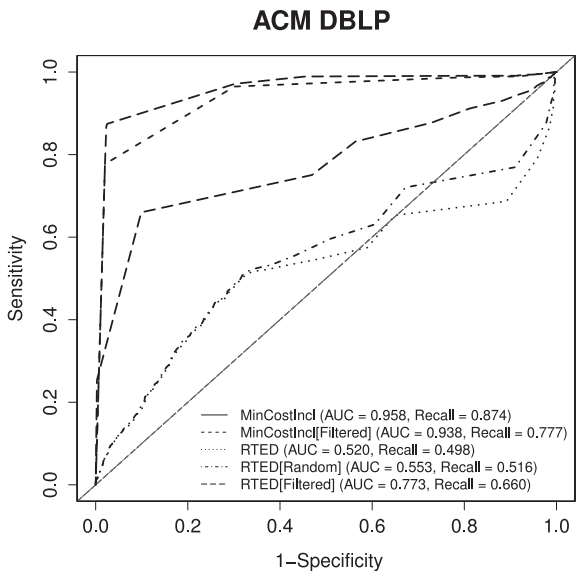


Fig. 15. Comparison of accuracy for ACM-DBLP data.

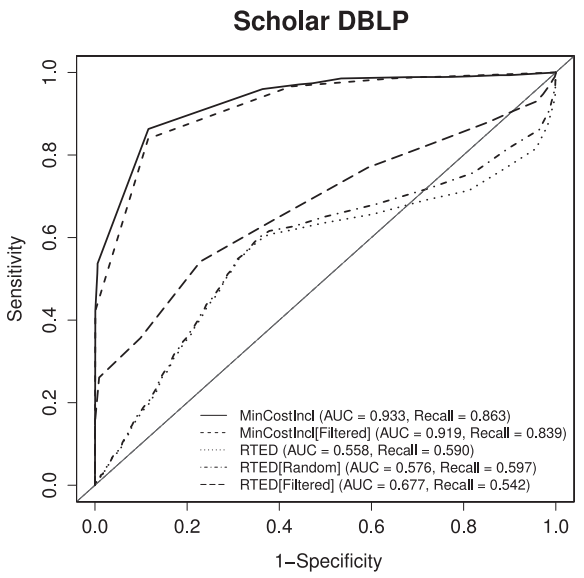


Fig. 16. Comparison of accuracy for Scholar-DBLP data.

Therefore, it is suitable for the bibliographic matching without various preprocessing steps involving adjusting the left-to-right node orderings or extra node removal.

7 Conclusion

In this paper, we have extended the concept of unordered tree inclusion to take the costs of insertions and substitutions into account. The resulting algorithm, MinCostIncl, has the same time complexity as the original algorithm of [22] for unordered tree inclusion ($O(2^{2D}mn)$). Moreover, we showed that the space complexity can be significantly reduced (from $O(2^{D}mn)$ to $O(n + 2^{D}m\log(n))$) when one only needs to report the existence of similar subtrees. It should be noted that D is the maximum outdegree of a pattern tree and there is no limitation on the degree of a text tree. We also extended MinCostIncl to allow a small number of deletions in the pattern tree, yielding an algorithm with time complexity $O((eD)^{K}K^{1/2}2^{2(DK+D-K)}mn)$, where K is the number of allowed deletion operations. Although the idea of introduction of costs into tree inclusion is simple, the modifications of the algorithm (including its space-economical version) are not trivial. Furthermore, MinCostInclWithDel and its analysis are far from straightforward. Computational experiments on large synthetic and real datasets showed that our proposed algorithm is fast, scalable, and very useful for bibliographic matching. Source codes of the implemented algorithms are available upon request.

To develop a space-efficient version of MinCostIncl that also allows traceback is left as an open problem. Another open problem is to further improve the time complexity of MinCostInclWithDel.

Acknowledgments

This research was partially supported by JSPS, Japan: Grants-in-Aid 26240034, 25730005, 24650042, and by the Collaborative Research Programs of National Institute of Informatics.

References

[1] C. Herrbach, A. Denise, and S. Dulucq, "Average complexity of the Jiang-Wang-Zhang pairwise tree alignment algorithm and of a RNA secondary structure alignment algorithm," *Theor. Comput. Sci.*, vol. 411, pp. 2423–2432, 2010.

[2] K.-C. Yu, E. L. Ritman, and W. E. Higgins, "System for the analysis and visualization of large 3D anatomical trees," *Comput. Biol. Med.*, vol. 27, pp. 1802–1830, 2007.

[3] K. F. Aoki, A. Yamaguchi, N. Ueda, T. Akutsu, H. Mamitsuka, S. Goto, and M. Kanehisa, "KCaM (KEGG Carbohydrate Matcher): A software tool for analyzing the structures of carbohydrate sugar chains," *Nucleic Acids Res.* vol. 32, pp. W267–W272, 2004.

[4] J. Felsenstein, *Inferring Phylogenies*. Sunderland, MA, USA: Sinauer Associates, 2004.

[5] Y. Horesh, R. Mehr, and R. Unger, "Designing an A* algorithm for calculating edit distance between Rooted-unordered trees," *J. Comput. Biol.*, vol. 6, pp. 1165–1176, 2006.

[6] N. Augsten, M. H. Böhlen, C. E. Dyreson, and J. Gamper, "Windowed pq-grams for approximate joins of data-centric XML," *VLDB J.*, vol. 21, pp. 463–488, 2012.

[7] M. N. Garofalakis and A. Kumar, "XML stream processing using tree-edit distance embeddings," *ACM Trans. Database Syst.*, vol. 30, pp. 279–332, 2005.

[8] M. Pawlik and N. Augsten, "RTED: A robust algorithm for the tree edit distance," *Proc. VLDB Endowment*, vol. 5, no. 4, pp. 334–345, 2012.

[9] Z. Lin, H. Wang, and S. I. McClean, "Measuring tree similarity for natural language processing based information retrieval," in *Proc. Int. Conf. Appl. Natural Language Process. Inf. Syst.*, 2010, pp. 13–23.

[10] N. Augsten, D. Barbosa, M. H. Böhlen, and T. Palpanas, "TASM: Top-k approximate subtree matching," in *Proc. Int. Conf. Data Eng.*, 2010, pp. 353–364.

[11] R. Yang, P. Kalnis, and A. K. H. Tung, "Similarity evaluation on tree-structured data," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2005, pp. 754–765.

[12] D. Shasha, J. T.-L. Wang, K. Zhang, and F. Y. Shih, "Exact and approximate algorithms for unordered tree matching," *IEEE Trans. Syst., Man, Cybern.*, vol. 24, no. 4, pp. 668–678, Apr. 1994.

[13] E. D. Demaine, S. Mozes, B. Rossman, and O. Weimann, "An optimal decomposition algorithm for tree edit distance," *ACM Trans. Algorithms*, vol. 6, p. 1, 2009.

[14] E. Rahm and P. A. Bernstein, "A survey of approaches to automatic schema matching," *VLDB J.*, vol. 10, pp. 334–350, 2001.

[15] K. Zhang, R. Statman, and D. Shasha, "On the editing distance between unordered labeled trees," *Inf. Process. Lett.*, vol. 42, pp. 133–139, 1992.

[16] T. Mori, T. Tamura, D. Fukagawa, A. Takasu, E. Tomita, and T. Akutsu, "A clique-based method using dynamic programming for computing edit distance between unordered trees," *J. Comput. Biol.*, vol. 19, pp. 1089–1104, 2012.

[17] P. Bille, "A survey on tree edit distance and related problem," *Theor. Comput. Sci.*, vol. 337, pp. 217–239, 2005.

[18] K. Zhang, "A constrained edit distance between unordered labeled trees," *Algorithmica*, vol. 15, pp. 205–222, 1996.

[19] T. Kan, S. Higuchi, and K. Hirata, "Segmental mapping and distance for rooted labeled ordered trees," *Fundam. Inf.*, vol. 132, pp. 461–483, 2014.

[20] K. Zhang and T. Jiang, "Some MAX SNP-hard results concerning unordered labeled trees," *Inf. Process. Lett.*, vol. 49, pp. 249–254, 1994.

[21] T. Jiang, L. Wang, and K. Zhang, "Alignment of trees—An alternative to tree edit," *Theor. Comput. Sci.*, vol. 143, pp. 137–148, 1995.

[22] P. Kilpeläinen and H. Mannila, "Ordered and unordered tree inclusion," *SIAM J. Comput.*, vol. 24, pp. 340–356, 1995.

[23] K.-C. Tai, "The Tree-to-tree correction problem," *J. ACM*, vol. 26, pp. 4220–4433, 1979.

[24] T. Akutsu, D. Fukagawa, M. M. Halldórsson, A. Takasu, and K. Tanaka, "Approximation and parameterized algorithms for common subtrees and edit distance between unordered trees," *Theor. Comput. Sci.*, vol. 470, pp. 10–22, 2013.

[25] T. Akutsu, D. Fukagawa, A. Takasu, and T. Tamura, "Exact algorithms for computing the tree edit distance between unordered trees," *Theor. Comput. Sci.*, vol. 412, pp. 352–364, 2011.

[26] T. Akutsu, T. Tamura, D. Fukagawa, and A. Takasu, "Efficient exponential time algorithms for edit distance between unordered trees," in *Proc. 23rd Annu. Symp. Combinatorial Pattern Matching*, 2012, pp. 360–372.
[27] P. Bille and I. Li Gørtz, "The tree inclusion problem: In optimal space and faster," *ACM Trans. Algorithm*, vol. 7, no. 38, p. 38, 2011.
[28] G. Valiente, "Constrained tree inclusion," *J. Disc. Algorithm*, vol. 3, pp. 431–447, 2004.
[29] M. Piernik and T. Morzy, "Partial tree-edit distance," Poznan Univ. Technol., Poznań, Poland, Tech. Rep. RA-10/2013, 2014.
[30] J. Matoušsek and R. Thomas, "On the complexity of finding iso- and other morphisms for partial k-trees," *Disc. Math.*, vol. 10, no. 8, pp. 343–364, 1992.
[31] R. Durbin, S. R. Eddy, A. Krogh, and G. Mitchison, *Biological Sequence Analysis—Probabilistic Models of Proteins and Nucleic Acids*. Cambridge, U.K.: Cambridge Univ. Press, 1998.
[32] C. Borgs, J. Chayes, J. Kahn, and L. Lovász, "Left and right convergence of graphs with bounded degree," *Random Struct. Algorithms*, vol. 42, pp. 1–28, 2013.
[33] M. Kanehisa, S. Goto, Y. Sato, M. Kawashima, M. Furumichi, and M. Tanabe, "Data, information, knowledge and principle: back to metabolism in KEGG," *Nucleic Acids Res.*, vol. 42, pp. D199–D205, 2014.
[34] M. J. Zaki, "Efficiently mining frequent trees in a forest: Algorithms and applications," *IEEE Trans. Knowl. Data Eng.*, vol. 17, no. 8, pp. 1021–1035, Aug. 2005.
[35] M. Bilenko, and R. J. Mooney, "Adaptive duplicate detection using learnable string similarity measures," in *Proc. ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2003, pp. 39–48.
[36] H. Köpcke, A. Thor, and E. Rahm, "Evaluation of entity resolution approaches on real-world match problems," *Proc. VLDB Endowment*, vol. 3, pp. 484–493, 2010.

Tomoya Mori received the BE and ME degrees in informatics from Doshisha University and Kyoto University in 2010 and 2012, respectively. He is currently a doctor course student at Kyoto University, Japan. His research interests are in bioinformatics and discrete algorithms.

Atsuhiro Takasu received the BE, ME, and DrEng degrees from the University of Tokyo in 1984, 1986, and 1989, respectively. He is a professor in the National Institute of Informatics, Japan. His research interests are database systems, digital libraries, and data mining. He is a member of the IEEE and ACM.

Jesper Jansson received the MSc degree in mathematics in 2002 and the PhD degree in computer science in 2003, both from Lund University, Sweden. He is currently an associate professor at Kyoto University, Japan. His research interests include graph algorithms, succinct data structures, and bioinformatics.

Jaewook Hwang received the BE degree from Ajou University, Korea, in 2012, and the ME degree in informatics from Kyoto University, Japan, in 2014. His research interests include graph algorithms.

Takeyuki Tamura received the BE, ME, and PhD degrees in informatics from Kyoto University, Japan, in 2001, 2003, and 2006, respectively. He joined Bioinformatics Center, Institute for Chemical Research, Kyoto University, as a postdoctoral fellow in April, 2006. He is an assistant professor from December 2007. His research interests are bioinformatics and the theory of combinatorial optimization for graphs and networks.

Tatsuya Akutsu received the BEng and MEng degrees in aeronautics and the DEng degree in information engineering from the University of Tokyo, 1984, 1986, and 1989, respectively. Since 2001, he has been a professor in the Bioinformatics Center, Institute for Chemical Research, Kyoto University. His research interests include bioinformatics and discrete algorithms. He is a member of the IEEE.