Methods for Analyzing Tree-Structured Data and their Applications to Computational Biology

木構造データの解析手法と その計算生物学への応用

Tomoya Mori

森　智弥

Abstract

Computational analysis of large-scale data requires efficient data structures for fast processing. Therefore, various data structures have been developed and applied to real data. Among them, *tree* is used in many scientific fields due to its simplicity and expressive power. For example, XML data, sentences in natural language processing, and certain kinds of image data are often represented by trees. On the other hand, in the computational biology, important molecules in living cells such as RNAs and glycans are also represented as tree-structured data. In order to analyze such tree-structured data, it is important to compute the similarity and to search similar substructures between them.

As for the computation of the similarity between the tree-structured data, *tree edit distance* is the most widely used measure and it is computed by the minimum cost sequence of *edit operations* (*deletions*, *insertions*, and *substitutions*). For the ordered cases, an $O(n^3)$ time algorithm has been developed and this bound is shown to be optimal under some computation strategy. However, the tree edit distance problem for unordered trees is NP-hard. Therefore, in order to cope with this hardness, various methods have been proposed. Among them, a clique-based method is fast and practical. The algorithm transforms the tree edit distance problem into the maximum vertex weighted clique problem and applies an efficient clique solver to it. However, the clique-based method is not fast when large trees are input. In this thesis, we propose an improved clique-based method by introducing a dynamic programming (DP) scheme and some heuristic techniques. Furthermore, we prove that the heuristics do not violate the optimality of the solutions. By means of computational experiments for Weblogs data and glycan data, the proposed method is shown to be much faster than the previous method.

On the other hand, for similar subtree searching, a variant of the tree edit distance called *tree inclusion* can be applied. The tree inclusion problem is to decide whether the pattern tree is included in another tree (i.e., *text tree*) or not. Although the problem is also NP-hard for the unordered cases, it can be solved in polynomial time when the maximum outdegree of an input pattern tree is bounded by a constant. However, it is not so useful for real data since the concept of costs has not been introduced as well as the fact that the substitution operations are

not allowed. In this thesis, we extend the unordered tree inclusion by introducing a cost of inclusion and taking the substitution operations into account. Then, we also develop a fast and scalable algorithm for computing the cost. Furthermore, we present two variants of the algorithm, one is a space saving algorithm and the other is an extended version in which a small number of deletion operations are allowed. From experimental results, it is shown that the proposed algorithm significantly improves the bibliographic matching accuracy compared with the ordered tree edit distance. In addition, for glycan data, we experimentally show that the computation of the extended tree inclusion can be done more efficiently than that of the tree edit distance.

論文概要

大規模なデータを素早く処理するためには，それに適した効率の良いデータ構造を用いる必要がある．そのため，これまでに様々なデータ構造が提案され，そして応用されてきたが，その中でも “木” と呼ばれるデータ構造はその単純さと表現力の高さから広く用いられている．例えば，マークアップ言語の一つである XML や，自然言語処理で用いられる構文木，さらには，ある特定の画像データが木を用いて表現される．一方で計算生物学の分野では，細胞内の重要な分子である RNA や糖鎖なども同様に木構造のデータとして用いられることが多い．木構造データの解析にはデータ間の類似性に着目した比較が行われるが，その中でも，木構造データ間の類似度計算と類似部分構造の探索は特に重要である．

木構造データ間の類似度計算には木の編集距離と呼ばれる計量が最も広く用いられており，それは頂点に対する削除，挿入，置換からなる編集操作列の最小コストとして計算される．順序木に対しては，$O(n^3)$ 時間で計算可能なアルゴリズムが開発されており，さらに，ある枠組みのもとではこれが下界であることも示されている．しかしながら，無順序木間の編集距離計算は NP 困難であることが知られている．そのため，この計算困難性に対処しようと様々な手法が提案されてきた．その中でも，クリークに基づく手法は高速かつ実用的であることが知られている．クリークに基づく手法は木の編集距離問題を最大重みクリーク問題に帰着させ，最大重みクリークを計算する高効率な専用のアルゴリズムを適用することで高速に解を得る．しかしながら，入力される木のサイズが大きい場合には計算が高速でないことが報告されていた．このように一般的な場合に対する高速かつ実用的なアルゴリズムは未だ開発されていない．そこで本研究では，クリークに基づく手法に動的計画法とヒューリスティクスを導入することで改良を行う．また，導入されたヒューリスティクスが解の最適性を損わないことも証明する．さらに，本手法を Web のアクセスログデータと糖鎖データに適用させた計算機実験では，提案手法が既存手法よりも高速に無順序木間の編集距離を計算可能であることを示す．

一方で，類似部分構造の探索には tree inclusion と呼ばれる計量が適用可能である．ここで，tree inclusion とは一方の木がもう一方の木に含まれるかどうかを判定する指標である．無順序木間における tree inclusion 問題は木の編集距離問題と同様に NP 困難であることが知られているが，パターンとなる木の最大次数がある定数で抑えられている場合は多項式時間で計算が可能であることが示されている．しかしながら，tree inclusion には頂点に付与されているラベルの置換が許されていな

いことに加えてコストが定義されていないことなどから，類似度に関する定量的な判断ができず，あまり実用的ではないという問題があった．そこで本研究では，これらの問題に対処するために，パターンの木に対する置換操作を許し，さらにコストの概念を新たに定義することで無順序木間の tree inclusion を拡張する．また，そのコストを計算する高速かつスケーラブルなアルゴリズムを提案する．さらに，そのアルゴリズムの領域計算量を削減したアルゴリズムとパターンの木から少数の削除操作を許したアルゴリズムも同時に提案する．文献マッチングに関する計算機実験では，提案アルゴリズムは順序木間の編集距離を用いた既存手法に比べて高精度であることを示す．さらに糖鎖を用いた実験では，拡張型の tree inclusion は無順序木間の編集距離計算よりも高速に実行可能であることを示す．

Acknowledgments

I would like to express my gratitude to my thesis advisor Professor Tatsuya Akutsu for all his kind support and variable advice. Without his guidance, this thesis would not be possible. I would also like to show my appreciation to the other referees of this thesis, Professor Akihiro Yamamoto and Professor Yasuo Okabe, for their valuable discussion and comments.

I am deeply grateful to Assistant Professor Morihiro Hayashida for his technical support and warm encouragement. I owe it him to have variable experience in study. I also would like to thank to Assistant Professor Takeyuki Tamura and Associate Professor Jesper Jansson for their helpful advice and discussions. Their advice improved the quality of this thesis.

I also appreciate Assistant Professor Daiji Fukagawa at Doshisha University, Professor Atsuhiro Takasu at National Institute of Informatics, and Professor Etsuji Tomita at University of Electro-Communications for their help to this study.

I want to thank to Dr. Takanori Hasegawa, Jaewook Hwang, and the other members of Akutsu Laboratory.

I would like to express the deepest appreciation to Tsuyuha Suzuki. I would never have finished this thesis without her daily support.

Finally, I would like to express my deep gratitude to my family and friends for all their support.

Publication Notes

Chapter 3 is based on the paper [41], which is publised in *Journal of Computational Biology*[1].

Chapter 4 is based on the paper [40], which is in press (DOI: 10.1109/TKDE.2015.2457922) in *IEEE Transactions on Knowledge and Data Engineering*[2].

[1]Reprinted with permission from JOURNAL OF COMPUTATIONAL BIOLOGY 19/10, 2012, published by Mary Ann Liebert, Inc., New Rochelle, NY.

[2]

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

Computational analysis of large-scale data requires appropriate data structures for efficient processing, so that various kinds of data structures have been developed and applied to real data in many scientific fields. *Tree* is one of the most widely used data structures due to its simplicity and expressive power. For example HTML and XML are represented by trees in order to deal with a large number of electronic documents efficiently. Another example is sentences in natural language processing, which are called syntax trees and they enable us to analyze languages systematically. In the field of data mining, decision trees constructed in the process of learning are used for obtaining knowledge from a large data set. Furthermore, important biological molecules such as RNAs and glycans are also represented by trees. Therefore, comparison methods for tree-structured data are required in many scientific fields. On the other hand, recent progress of graph theory enables us to decompose complex network data into the tree-structured data for efficient analysis, which also leads to an increase in the requirement.

1.1.1 Comparison of Tree-Structured Data

In order to analyze such tree-structured data, it is important to compare them, so that various similarity measures between the tree-structured data have been developed. Among them, the *tree edit distance*, which is extended from the well known string edit distance in informatics, is one of the most widely used measures for comparison of tree-structured data. The tree edit distance is computed by the minimum cost sequence of *edit operations*, where an edit operation is either a *deletion* of a node, an *insersion* of a node, or a *substitution* of a label of a node. For the ordered case (i.e., a left-to-right order among sibling nodes is given), an $O(n^3)$ time algorithm has been developed and this bound is shown to be optimal under some

computation strategy [16]. However, for the unordered cases, the tree edit distance problem is shown to be NP-hard [61]. In order to cope with this NP-hardness, a fixed parameter algorithm [5] and A^*-algorithm [24] have been developed. Although they are useful under some constraints, they are not practical for general cases. To deal with the general cases of the unordered tree edit distance problem, Fukagawa et al. proposed a practical algorithms [19], in which the tree edit distance problem is transformed into the *maximum clique* problem in graph theory, but this algorithm is also not fast enough when large trees are given. Furthermore, Kondo et al. proposed an *integer programming* (IP) based algorithm [32]. Although the IP-based algorithm is much faster than the existing methods (including the method proposed in this thesis) when the maximum outdegrees of input trees are large, it is not so effective for the cases of the small maximum outdegrees. In a related line of research, for the purpose of searching similar subtree structures, a variant of the tree edit distance termed *tree inclusion* has been proposed by Kilpelänian and Mannila [31]. Given two trees, a *pattern tree* and a *text tree*, tree inclusion determines whether the text tree can be obtained from the pattern tree by using only insertion operations. Although the tree inclusion problem for unordered trees is also proved to be NP-hard, it can be solved in polynomial time if the maximum outdegree of the pattern tree is upper-bounded by a constant [14]. However, for the tree inclusion problem, the concept of costs has not been introduced. Therefore, there is no practical method for the tree inclusion problem. In this thesis, to cope with these difficulties, we propose fast and efficient algorithms for the tree edit distance problem and the tree inclusion problem, respectively. As for the tree edit distance problem, we improve the clique-based method proposed in [19] by introducing a *dynamic programming* (DP) approach and heuristic techniques (Chapter 3). As for the tree inclusion problem, we extend the tree inclusion so that substitution operations to the pattern tree are allowed and define a cost of inclusion to make the concept of tree inclusion more useful, and then we present a fast and scalable algorithm for computing it (Chapter 4).

1.1.2 Applications to Tree-Stractured Biological Data

The tree-structured data are also found in the field of computational biology. A comprehensive understanding of biological systems in living cells is one of the main goals of molecular biology. However, in spite of progress of experimental techniques, it is still challenging since biological systems are highly complex. In the field of molecular biology, it has been aimed to catalog all genes in living things from the viewpoint that systems mainly consist of functions of proteins, which are produced according to genomic information. Actually, whole-genome sequences of several thousands of organisms have already been determined. However, not all functions of each protein have been revealed. Thus, various studies such as three-dimensional structural

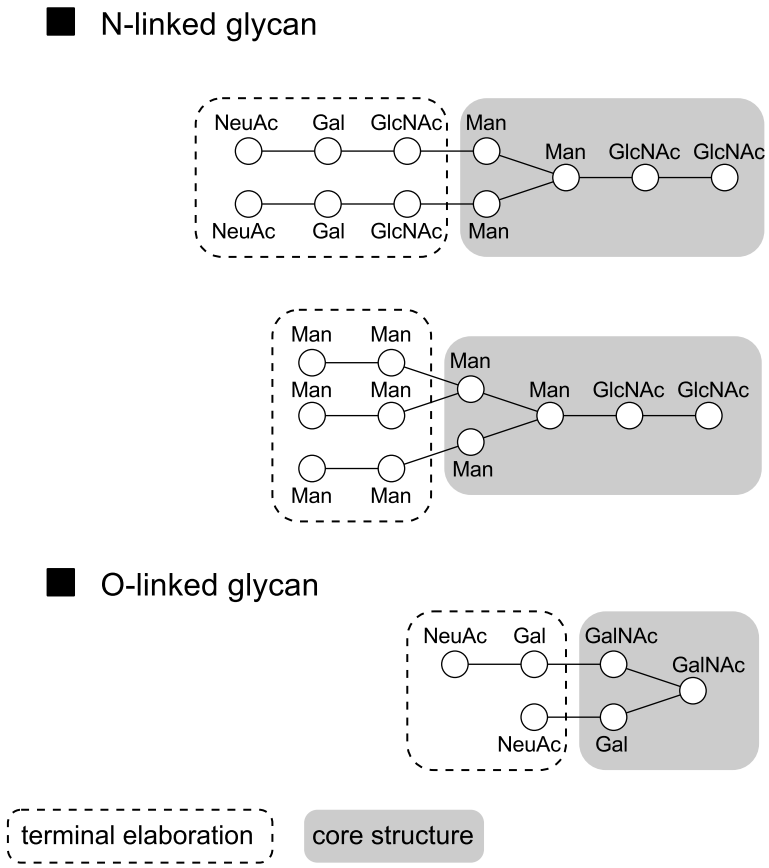


Figure 1.1: Example of two types of glycan: N-linked glycans and O-linked glycan. N-linked glycans attach to an asparagine residue. On the other hand, O-linked glycans attach to a serene or a threonine residue. For each glycan, the core structure is shaded and the terminal elaboration is surrounded by dashed line.

analysis have been done for the purpose of elucidating functions of proteins. On the other hand, it has been shown by recent studies that non-coding RNAs (ncRNAs), which do not encode proteins, control gene expressions, and glycans (i.e., sugar chains) also modify proteins and regulate their functions. Therefore, it appears that analysis of not only proteins but also molecules such as ncRNAs and glycans becomes more important to understand biological systems. In the field of computational biology, such important molecules in living cells are often represented as tree-structured data, and they are analyzed by applying computational methods. However, the computational costs of dealing with such biological data are often very high due to their size as well as their complexity. Especially, glycans are regarded as unordered tree-structured data, so that their comparison requires high compu-

tational cost in spite of a strong need for understanding of relationships between their structures and functions.

In this thesis, we focus on comparison of glycans. Although to relate the structures of glycans to their functions is a main goal of glycobiology, it is not easy to do that partially because our knowledge on the conformations (three-dimensional structure) of glycans is limited and there exist the cases that different proteins are modified by common set of glycans and vice versa (i.e., common proteins are modified by different glycans) [49]. Thus, only comparison of glycans is not enough to analyze relationships between their structures and functions. However, we believe that computing the similarity and searching similar substructures between glycans are good start points for achieving the goal. Applications of our proposed methods to glycan data are as follows. In Chapter 3, we apply the improved clique-based method to glycans and show that the improved method enables us to compute the tree edit distance of every glycan pair in a database in a short time. It is to be noted that there exist some glycan pairs whose distance cannot be computed in an hour by the previous method. In Chapter 4, we apply the extended tree inclusion to similar subtree searching of glycan data and compute the cost of inclusion. Each glycan consists of the *core structure* and the *terminal elaborations* (Figure 1.1), and their differences lead to the differences of functions. For example, glycans often protect their binding protein from aqueous solvent and proteases, which is caused by the core structures. In such cases, differences between their terminal elaborations are ignored. On the other hand, there is a case such that modification of terminal elaborations is recognized by the mannose receptor, which leads to regulation of hormone activity. Thus, we often need to focus on only the substructures of each glycan. Therefore, the extended tree inclusion is expected to be effective for analyzing glycan data.

1.2 Summary of Contributions

In this thesis, we deal with two research topics in order to develop efficient and effective methods for analyzing the tree-structured data. Contributions of this thesis are summarized as follows.

In Chapter 2, we introduce notation of trees and measures for comparing the tree-structured data. In addition, we briefly review major algorithms for computing them.

In Chapter 3, we propose an improved clique-based method (**DpCliqueEdit**) for the tree edit distance problem for unordered trees. The improved method is obtained by introducing a dynamic programming scheme to a previous clique-based method proposed in [19]. Although some heuristic techniques are also introduced to **DpCliqueEdit** in order to further reduce the computation time, **DpCliqueEdit**

can compute the tree edit distance without violating the optimality of the solution. We also show the correctness of the optimality as a theorem. In computational experiments performed to evaluate the effectiveness, we apply **DpCliqueEdit** to Weblogs data and glycan data, and then we show that **DpCliqueEdit** is much faster than the previous clique-based method, especially for trees with many leaves or many isomorphic subtrees.

In Chapter 4, 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. For the problem, 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 a 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. Furthermore, 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. All proposed algorithms are implemented and computational experiments are performed using both synthetic and real data in order to show the efficiency and effectiveness. The algorithm for **MinCostIncl** is also applied to bibliographic matching, which is a typical entity resolution problem for tree-structured data, in order to show its usefulness for another data appearing in informatics. For glycan data, we also experimentally show that the computation of the extended tree inclusion can be done more efficiently than that of the tree edit distance.

1.3 Organization of Thesis

The rest of this thesis is organized as follows. Chapter 2 describes notation of trees, measures for the similarity between trees, and algorithms for computing them. Chapter 3 reviews the unordered tree edit distance problem and proposes an improved clique-based method using DP scheme for the problem. Chapter 4 defines the minimum-cost unordered tree inclusion problem and gives an algorithm for computing the cost of inclusion. We conclude and discuss future research directions in Chapter 5.

Chapter 2

Measures for Comparison of Tree-Structured Data

As mentioned in Chapter 1, comparison of tree-structured data has numerous applications in various scientific areas and data processing. For example, 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 [12, 20, 45]. Similarly, tree-structured data matching is often applied to entity resolution [10, 55]. Furthermore, trees can be used to represent several kinds of biological data such as RNA secondary structures [23], vascular trees [56], glycans [8], phylogenetic trees [18], and cell lineage data [24], and scientists sometimes need to compare such structures. Consequently, various metrics have been developed and applied to analysis of these tree-structured data. In this chapter, first we review the *tree edit distance*, which is one of the most widely used measures for comparison of the tree-structured data. After that, we describe some variants of it as related work.

2.1 Notation

For any rooted tree T, let $r(T)$ be the root of T, $V(T)$ the set of nodes in T, and $E(T)$ the set of edges 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, $deg(v)$ is the outdegree of v (i.e., the number of children of v), and $depth(v)$ is the depth of v (i.e., the length of the path from the root node to v). The height of a tree T is defined as the depth of the deepest node in T and is denoted by $height(T)$. Furthermore, $T(v)$ is the rooted subtree (connected subgraph) of T induced by v and all of its descendants $des(v)$ (not including v). 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. In this chapter, n denotes the number of nodes in a larger input tree, that is, $n = \max\{|V(T_1)|, |V(T_2)|\})$ where

T_1 and T_2 are input trees. 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 $p(T) \notin V(T)$.

2.2 Tree Edit Distance

2.2.1 General Ordered and Unordered Tree Edit Distance

Before presenting the *tree edit distance*, we introduce *edit operations* and *edit distance mapping* for rooted and labeled trees [14, 61].

An *edit operation on a tree* T is either a *deletion*, an *insertion*, or a *substitution*, each of which is defined by (see also Figure. 2.1):

- **Deletion**: Remove a node $v \in V(T)$ whose parent is u, while letting the children of v become children of u.
- **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.
- **Substitution**: Change the label of a node in $V(T)$ to any label except $\diamond$.

For each edit operation, the *cost* is defined as follows:

- $\delta(a, -)$: cost of deleting a node with label a.
- $\delta(-, a)$: cost of inserting a node with label a,
- $\delta(a, b)$: cost of substituting a node with label a to label b,

The *edit distance* $dist(T_1, T_2)$ between two trees T_1 and T_2 is the cost of the minimum cost sequence of edit operations that transforms T_1 to T_2, where we adopt the following standard assumption so that $dist(T_1, T_2)$ becomes a distance metric [14, 61]:

- $\delta(x, y) \geq 0$,
- $\delta(x, y) = 0$ if and only if $x = y$,
- $\delta(x, y) = \delta(y, x)$,
- $\delta(x, z) \leq \delta(x, y) + \delta(y, z)$

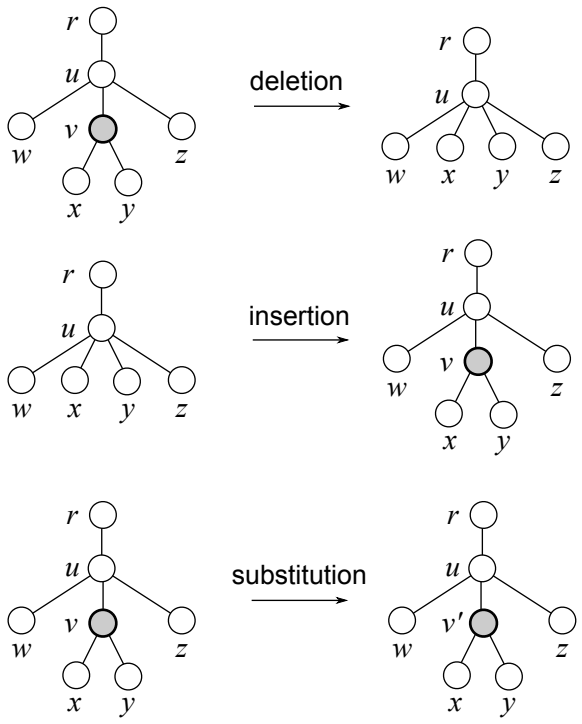


Figure 2.1: Edit operations on a tree. The top, middle, and bottom indicate a deletion, an insertion, and a substitution operation, respectively.

for all x, y, z.

The tree edit distance can be expressed in terms of *edit distance mappings* [14, 48] (or *mappings*, for short), which are also called *Tai mappings* [48] (Figure 2.2). 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) $u_1 \in des(u_2)$ if and only if $v_1 \in des(v_2)$,

(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 Figure 2.3.

Let I_1 and I_2 be the sets of nodes in $V(T_1)$ and $V(T_2)$ not appearing in M, respectively. Then, the following equality holds [14, 61]:

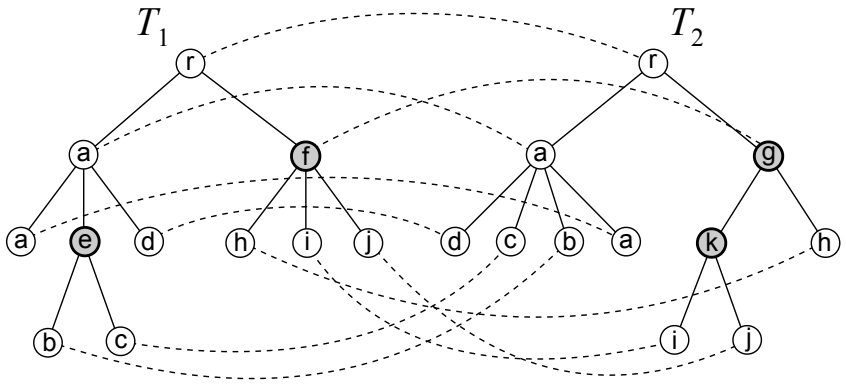


Figure 2.2: Example of tree edit operations and edit distance mapping for unordered trees. T_2 is obtained from T_1 by deletion of node labeled **e**, insertion of node labeled **k**, and substitution of node labeled **f** with node labeled **g**, where a tree can contain nodes with the same label. The corresponding mapping M is shown by broken curves.

$$dist(T_1, T_2) \;=\; \min_{M}\Big\{ \sum_{u \in I_1} \delta(\ell(u), -) + \sum_{v \in I_2} \delta(-, \ell(v)) + \sum_{(u,v) \in M} \delta(\ell(u), \ell(v)) \Big\}.$$

Here we introduce a *score function* $f(u, v)$ for $(u, v) \in V(T_1) \times V(T_2)$ defined by

$$f(u, v) = \delta(\ell(u), -) + \delta(-, \ell(v)) - \delta(\ell(u), \ell(v)). \tag{2.1}$$

Then, we can see that $f(u, v) = f(v, u) \geq 0$ holds. It should also be noted that under the unit-cost edit operation model (i.e., $\delta(a, b) = 1$ for all $a \neq b$), $f(u, v) = 2$ holds if $\ell(u) = \ell(v)$, and $f(u, v) = 1$ holds otherwise. Let $score(M)$ be the score of a mapping M defined by

$$score(M) = \sum_{(u,v) \in M} f(u, v).$$

Let M_{OPT} be the mapping with the maximum score. Then, the following property

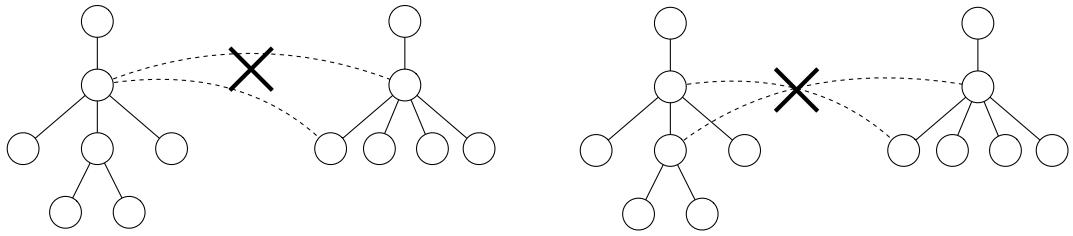


Figure 2.3: Example of the condition of the edit distance mapping. The left and right figures correspond to conditions (i) and (ii) for the edit distance mapping, respectively, where (i) stands for one-to-one condition, and (ii) stands for descendant condition.

holds [5]:

$$
\begin{aligned}
dist(T_1, T_2) &= \min_M \Big\{ \sum_{u \in I_1} \delta(\ell(u), -) + \sum_{v \in I_2} \delta(-, \ell(v)) + \sum_{(u,v) \in M} \delta(\ell(u), \ell(v)) \Big\} \\
&= \min_M \Big\{ \sum_{u \in V(T_1)} \delta(\ell(u), -) + \sum_{v \in V(T_2)} \delta(-, \ell(v)) \\
&\qquad + \sum_{(u,v) \in M} (\delta(\ell(u), \ell(v)) - \delta(\ell(u), -) - \delta(-, \ell(v))) \Big\} \\
&= \sum_{u \in V(T_1)} \delta(\ell(u), -) + \sum_{v \in V(T_2)} \delta(-, \ell(v)) \\
&\qquad - \max_M \Big\{ \sum_{(u,v) \in M} (\delta(\ell(u), -) + \delta(-, \ell(v)) - \delta(\ell(u), \ell(v))) \Big\} \\
&= \sum_{u \in V(T_1)} \delta(\ell(u), -) + \sum_{v \in V(T_2)} \delta(-, \ell(v)) - score(M_{OPT}), \qquad (2.2)
\end{aligned}
$$

assuming that the root of T_1 corresponds to the root of T_2 in M_{OPT}, where this assumption can be removed if we add dummy nodes as new roots. It is to be noted that the first and second terms in the right hand side of the last equality are invariant with a mapping. Therefore, this equality means that the tree edit distance can be obtained by computing a mapping with the maximum score.

We show a brief comparison between the ordered and unordered tree edit distance. Consider T_1 and T_2 in Figure 2.4 (similar to Fig. 1 in [25]). 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

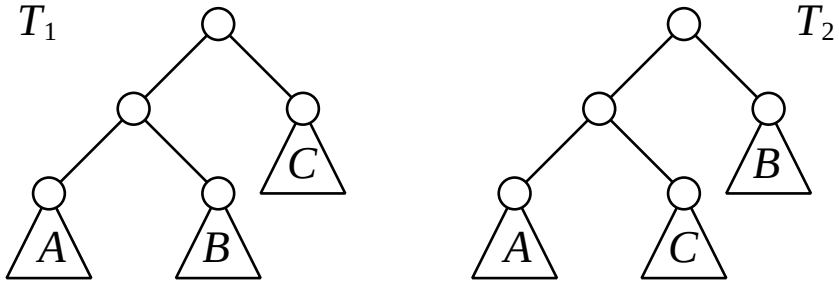


Figure 2.4: Comparing the ordered and unordered tree edit distance. The ordered tree edit distance is $2x$ while the unordered tree edit distance is 2.

size $\Omega(n)$, where n is the number of nodes. We remark that in this example, the unordered tree alignment [25] (described in Section 2.3) is not appropriate because its cost is also $2x$.

Although the ordered tree edit distance problem can be solved in polynomial time, the general unordered tree edit distance problem is NP-hard [61]. Furthermore, Zhang and Jiang proved that the problem is MAX SNP-hard [60]. In order to cope with this hardness, various algorithms for computing the tree edit distance between unordered trees have been developed. In the rest of this subsection, we briefly review the existing algorithms.

Dynamic Programming Algorithm

The tree edit distance problem can be solved efficiently using a dynamic programming (DP) scheme. Let F and $r(F)$ be a forest (i.e., connected/disconnected subgraph) of a tree T and a multi set of the roots of trees in F. $T - v$, $F - v$, and $F - T(v)$ denote the tree obtained by deleting v from T, the forest obtained by deleting v from F, and the forest obtained by deleting $T(v)$ from F, respectively. The cost $D(F_1, F_2)$ between the forests F_1 and F_2 is defined by the following DP procedure [5] (see also Figure 2.5).

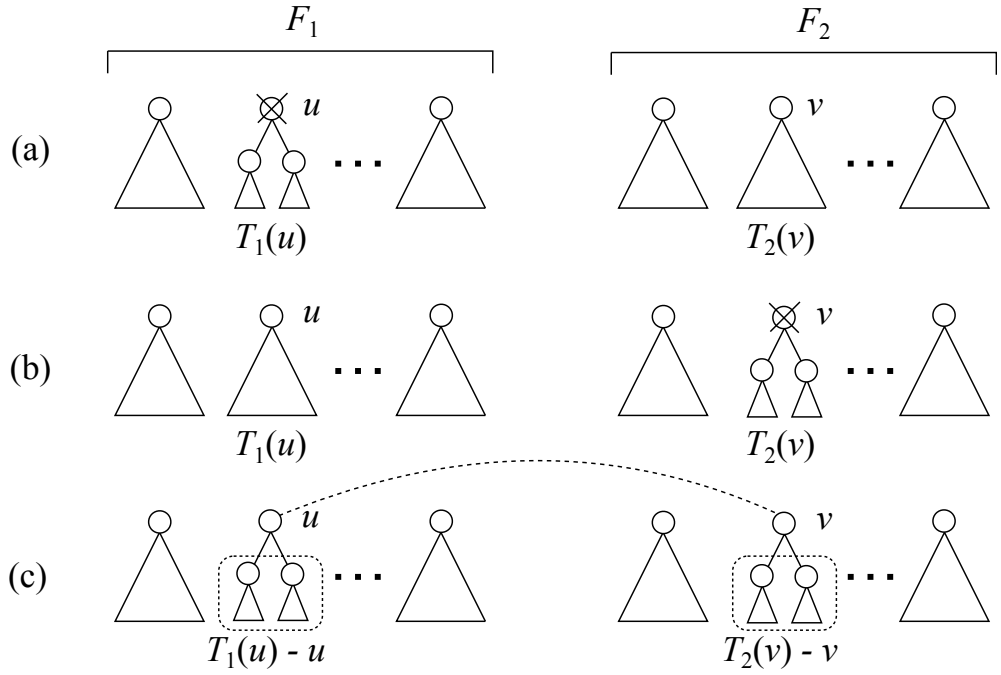


Figure 2.5: Dynamic programming procedure for the edit distance between unordered trees. (a), (b), and (c) denote the top, middle, and bottom of Eq. (2.3), respectively.

$$
\begin{aligned}
D(F_1, -) &= \sum_{u \in V(F_1)} \delta(\ell(u), -), \\
D(-, F_2) &= \sum_{v \in V(F_2)} \delta(-, \ell(v)), \\
D(F_1, F_2) &= \min \begin{cases}
\min_{u \in r(F_1)} \{D(F_1 - u, F_2) + \delta(\ell(u), -)\}, \\
\min_{v \in r(F_2)} \{D(F_1, F_2 - v) + \delta(-, \ell(v))\}, \\
\min_{(u,v) \in r(F_1) \times r(F_2)} \{D(F_1 - T_1(u), F_2 - T_2(v)) \\
\qquad + D(T_1(u) - u, T_2(v) - v) \\
\qquad + \delta(\ell(u), \ell(v))\}.
\end{cases}
\end{aligned} \tag{2.3}
$$

We can obtain $D(T_1, T_2)$ by applying this procedure, where $dist(T_1, T_2) = D(T_1, T_2)$ holds [14]. Although this DP algorithm works efficiently, it does not run in polynomial time since the number of forests appearing in this procedure is exponential. The time complexity of this algorithm has been proved to be $O(|T_1||T_2| + L_1!\, 3^{L_1}(L_1^3 + D_2)^2)|T_2|)$ time [61], where L_1 is the number of leaves of T_1 and D_2 is the maximum outdegree of T_2.

A^*-algorithm

Horesh et al. applied a search technique called A* to the tree edit distance between *unlabeled* unordered trees [24], and the resulting algorithm is often called *A^*-algorithm*.

A* is often used for the shortest path problem, which is to find shortest paths in a given edge-weighted undirected graph [54]. The concept of A* is as follows: Starting from the source (start) node in a given graph, A* explores and evolves a search tree efficiently in accordance with the costs of paths, where each cost is computed by summing up the actual cost (i.e., distance) from the source to a current searching point and an estimated cost from a current searching point to the target (goal) node.

On the basis of the concept, A^*-algorithm constructs a search tree from input trees by scanning the all possible pairs of one subtree to the other subtree, and then it gives an optimal set of pairs between input trees by exploring the search tree. Although the set of the pairs represents the edit operations needed to transform one tree into the other tree, it does not include substitutions since input trees are unlabeled trees. The cost of each set of pairs is computed by two functions as with A*. The first function calculates the number of internal nodes which must be deleted or inserted to transform one tree into the other tree. The second function computes a lower bound which estimates how many nodes will have to be deleted or inserted for making two subtrees isomorphic. The estimation is done by taking the maximum value of the following three heuristic functions h_1, h_2, and h_3. Each function returns the difference of size between the subtrees of T_1 and T_2, the maximum difference of the number of nodes with outdegree k between the subtrees, and the summation of differences of the number of nodes with outdegree k divided by 3 and rounded up, respectively. Although this algorithm includes such heuristics, it is proved to be an exact algorithm [24]. The results of computational experiments performed in [24] show that A^*-algorithm works efficiently for comparison of moderate size unlabeled trees under the unit-cost edit operation model. However, it is unclear whether it can be efficiently applied to labeled trees or the general unordered edit distance cases.

Fixed-Parameter Algorithm

Akutsu et al. proposed a *fixed-parameter algorithm* [5], which determines whether $dist(T_1, T_2) \leq k$ or not in $O(2.62^k \cdot poly(n))$ time, where k is an integer and n is the maximum size of input trees (i.e., $\max\{|T_1|, |T_2|\}$). The fixed-parameter algorithm employs DP scheme and decides whether or not $dist(T_1(u), T_2(v)) \leq k$ for all $(u, v, k) \in V(T_1) \times V(T_2) \times \{0, 1, 2, ..., k\}$ in a bottom up manner, where u must be corresponded to v. The algorithm maintains $T_1(u)$ and $T_2(v)$ obtained by

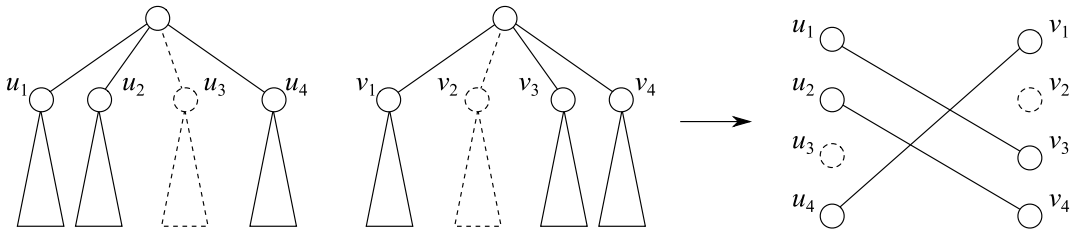


Figure 2.6: Example of the fixed-parameter algorithm. $T_1(u_3)$ and $T_2(v_2)$ are deleted when they are isomorphic, and then the residual subtrees are matched.

deleting some of child nodes of u and v from the largest subtree recursively, and the deleted nodes are stored in sets W_1 and W_2, respectively. Furthermore, it also deletes $T_1(u_i)$ and $T_2(v_j)$ if they are isomorphic where u_i and v_j are children of u and v, respectively. It is to be noted that deletions of isomorphic trees do not violate the optimality of the distance [5]. The algorithm repeats this procedure while there exist children which are not in $W_1 \cup W_2$ and $|W_1| \leq k$ and $|W_2| \leq k$ holds. Finally, the algorithm computes an optimal mapping between the residual subtrees by solving the *minimum weight bipartite matching* problem (see also Figure. 2.6). After slight modifications on sets W_1 and W_2, it is proved that the fixed-parameter algorithm works in $O(2.62^k \cdot poly(n))$ time. However, it is not useful for comparison of non-similar trees (i.e., k is not small).

Clique-Based Algorithm

Fukagawa et al. developed a clique-based algorithm, which transforms the tree edit distance problem into the *maximum vertex weighted clique problem* [19] as follows: first, the algorithm create a vertex weighted graph from T_1 and T_2 in which each vertex denotes a mapping between nodes $u \in V(T_1)$ and $v \in V(T_2)$ and whose weight is computed by the score function $f(u, v)$ (Eq. (2.1)), where edges are created between the nodes that satisfy the condition of the edit distance mapping. Second, the algorithm computes the maximum vertex weighted clique from the graph using the fastest clique solver called MWCQ. Finally, the clique is output as an optimal mapping between T_1 and T_2 since there is a one-to-one correspondence between the set of cliques and the set of mappings. Although this algorithm is practical since it can be applied to the general labeled unordered trees and works under the general cost model, it is not fast for large input trees. The details of this algorithm are described in Chapter 3.

Integer Linear Programming-Based Algorithm

Another practical method has been proposed by Kondo et al. [32]. The algorithm is based on an integer linear programing (ILP), which transforms the tree edit distance problem into the integer linear problem. In this algorithm, the cost of a mapping M between T_1 and T_2 is defined as follows:

$$cost(M) = \sum_{u \in I_1} \delta(\ell(u), -) + \sum_{v \in I_2} \delta(-, \ell(v)) + \sum_{(u,v) \in M} \delta(\ell(u), \ell(v)),$$

where I_1 and I_2 are the sets of nodes in $V(T_1)$ and $V(T_2)$ not appearing in M, respectively. From the cost function, an objective function for computing $dist(T_1, T_2)$ is designed as

$$\begin{aligned} cost(M) \;=\; & \sum_{(u,v) \in V(T_1) \times V(T_2)} \{\delta(\ell(u), \ell(v)) - \delta(\ell(u), -) - \delta(-, \ell(v))\} m_{u,v} \\ & + \sum_{u \in V(T_1)} \delta(\ell(u), -) + \sum_{v \in V(T_2)} \delta(-, \ell(v)), \end{aligned}$$

where $m_{u,v} \in \{0, 1\}$ and $m_{u,v} = 1$ if and only if $(u, v) \in M$ for $u \in V(T_1)$ and $v \in V(T_2)$. Furthermore, constraints are introduced from the conditions of the mapping: the one-to-one mapping condition and the preserving descendants condition. In summary, the ILP formulation for the tree edit distance problem is defined as

$$\begin{aligned} \text{minimize} \quad & \sum_{(u,v) \in V(T_1) \times V(T_2)} \{\delta(\ell(u), \ell(v)) - \delta(\ell(u), -) - \delta(-, \ell(v))\} m_{u,v} \\ & + \sum_{u \in V(T_1)} \delta(\ell(u), -) + \sum_{v \in V(T_2)} \delta(-, \ell(v)) \\ \text{subject to} \quad & m_{u,v} \in \{0, 1\} \quad (\text{for all } (u, v) \in V(T_1) \times V(T_2)) \\ & \sum_{v \in V(T_2)} m_{u,v} \leq 1 \quad (\text{for all } u \in V(T_1)) \\ & \sum_{u \in V(T_1)} m_{u,v} \leq 1 \quad (\text{for all } v \in V(T_2)) \\ & m_{u_1,v_1} + m_{u_2,v_2} \leq 1 \\ & \quad (\text{for all } (u_1, v_2), (u_2, v_1) \in V(T_1) \times V(T_2) \text{ s.t. } u_1 \prec u_2 \not\Leftrightarrow v_1 \prec v_2), \end{aligned}$$

where $x \prec y$ means x is an descendant of y. The second and third constraints mean the one-to-one constraint, and the fourth constraint means the preserving descendant constraints. The ILP-based algorithm is much faster than the existing methods (including the method proposed in this thesis) when the maximum outdegrees of T_1 and T_2 are large. However it is not so effective for the cases of the small maximum outdegrees.

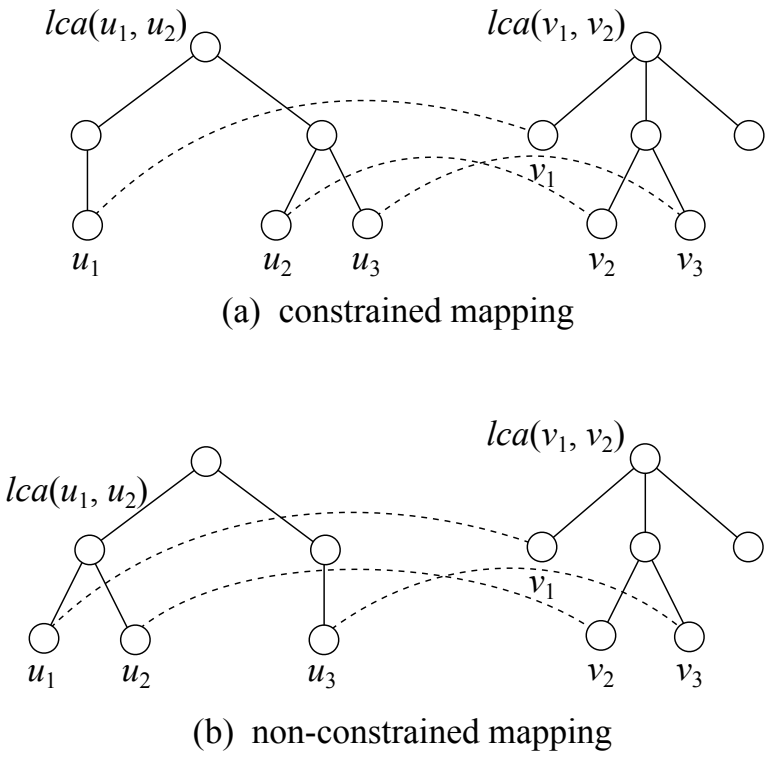


Figure 2.7: Example of constrained mapping.

2.2.2 Constrained Edit Distance

In order to reduce the computational cost, Zhang introduced the *constrained mapping* and the *constrained edit distance* [58, 59], which is a restricted version of the tree edit distance.

Before presenting them, we introduce *least common ancestor*. For a tree T and a pair of nodes $(v_1, v_2) \in V(T) \times V(T)$, a *common ancestor* of (v_1, v_2) is a node $v_3 \in V(T)$ such that $v_3 \in anc(v_1) \wedge anc(v_2)$, where $anc(v)$ denotes ancestors of v (not including v). The *least common ancestor* of (v_1, v_2) is a common ancestor x of (v_1, v_2) such that $y \in anc(x)$ holds for any common ancestor y $(\neq x)$ of (v_1, v_2) and denoted by $lca(v_1, v_2)$ (see Figure 2.7).

A mapping M is called a constrained mapping M_c if an edit distance mapping M satisfies the following condition: for all (u_1, v_1), (u_2, v_2), $(u_3, v_3) \in M$ (see also Figure 2.7),

$$lca(u_1, u_2) \in anc(u_3) \Leftrightarrow lca(v_1, v_2) \in anc(v_3).$$

For example, Figure 2.7(a) is constrained since $lca(u_1, u_2) \in anc(u_3)$ and $lca(v_1, v_2) \in anc(v_3)$. On the other hand, Figure 2.7(b) is not constrained since $lca(u_1, u_2) \notin$

$anc(u_3)$ but $lca(v_1, v_2) \in anc(v_3)$. Under this constraint, the edit distance problem can be computed more efficiently.

In a related line of work, Richter proposed the *structure-respecting mapping* [47]. A mapping M is called a structure-respecting mapping M_s if an edit distance mapping M satisfies the following condition: for all (u_1, v_1), (u_2, v_2), $(u_3, v_3) \in M$ such that none of u_1, u_2, and u_3 is an ancestor of the others,

$$lca(u_1, u_2) = lca(u_1, u_3) \Leftrightarrow lca(v_1, v_2) = lca(v_1, v_3).$$

Though the constrained mapping and the structure-respecting mapping are independently proposed, these two mapping are proved to be equivalent in [34].

Zhang's Algorithm

Zhang et al. developed DP-based algorithms for ordered and unordered constrained edit distance problem [58, 59]. For the ordered cases, the computation for the restricted mapping is related to the string edit distance problem. Therefore, this problem can be solved efficiently using a practical string edit distance algorithm and its time complexity is bounded by $O(|T_1||T_2|)$. On the other hand, the unordered constrained edit distance problem can be reduced into the minimum weight bipartite matching problem. Zhang employed an efficient algorithm for the minimum cost maximum flow and proved that the time complexity of the algorithm is $O(|T_1||T_2|(D_1 + D_2)\log(D_1 + D_2))$, where D_i is the maximum outdegree of T_i.

2.2.3 Less-Constrained Edit Distance

Lu et al. proposed the *less-constrained edit distance* [37], which relaxes the condition of the constrained edit distance mapping. A mapping M is called a *less-constrained mapping* M_l if a tree edit distance mapping M satisfies the following condition (see also Figure 2.8): for all $(u_1, v_1), (u_2, v_2), (u_3, v_3) \in M$ such that none of u_1, u_2, and u_3 is an ancestor of the others,

$$\begin{aligned} &depth(lca(u_1, u_2)) \geq depth(lca(u_1, u_3)) \wedge lca(u_1, u_3) = lca(u_2, u_3) \\ \Leftrightarrow\ &depth(lca(v_1, v_2)) \geq depth(lca(v_1, v_3)) \wedge lca(v_1, v_3) = lca(v_2, v_3). \end{aligned}$$

For example, the mapping in Figure 2.8 is not constrained since $lca(u_1, u_2) \neq lca(u_1, u_3)$ but $lca(v_1, v_2) = lca(v_1, v_3)$. However, it is less-constrained because $depth(lca(u_1, u_2)) \geq depth(lca(u_1, u_3)) \wedge lca(u_1, u_3) = lca(u_2, u_3)$ and $depth(lca(v_1, v_2)) \geq depth(lca(v_1, v_3)) \wedge lca(v_1, v_3) = lca(v_2, v_3)$.

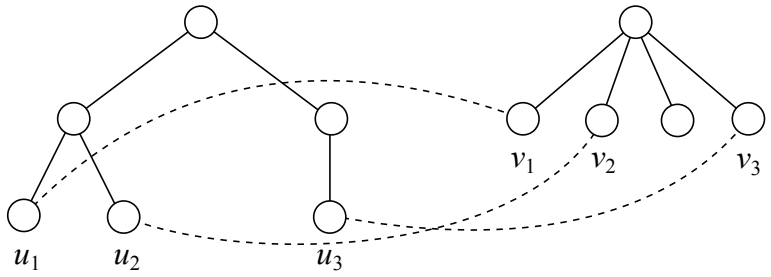


Figure 2.8: Not constrained but less-constrained mapping.

Lu's Algorithm

Lu et al. presented an algorithm for computing the less-constrained edit distance for the ordered cases [37]. The algorithm runs in $O(|T_1||T_2|D_1^3D_2^3(D_1 + D_2))$ time and space. For the unordered cases, this problem is proved to be NP-complete. Furthermore, the problem is proved to be MAX SNP-hard, which means that there exists no polynomial time approximation scheme unless P=NP [37].

2.2.4 Approximation of Tree Edit Distance

The unordered tree edit distance problem requires high computational cost. Motivated by the computational hardness, various approximation techniques have been developed instead of computing the exact tree edit distance between two trees [35]. For the ordered cases, string-based approximation algorithm [1, 4, 21] and a set-based approximation algorithm [55] are well-known. On the other hand, for the unordered cases, Kailing et al. have proposed another set-based algorithm, in which three kinds of histogram are introduced [26]. After that, Akutsu et al. have presented $O(n/\log n)$- and $O(h)$-approximation algorithms [2], where h is the maximum height of two input trees. As another approximation algorithm, pq-gram for ordered trees has been extended to evaluate the similarity between unordered trees [11]. Furthermore, Tatikonda et al. have also proposed an algorithm which employs the Jaccard Coefficient to approximate the tree edit distance. However, the last two approaches have no guaranteed approximation ratio. Therefore, we review only the histogram-based approach, $O(n/\log n)$-approximation algorithm, and $O(h)$-approximation algorithm in the rest of this subsection.

Histogram-Based Approach

Three kinds of histograms are developed in [26]: leaf distance histogram, degree histogram, and label histogram. Recall that $height(T)$, $deg(v)$, and $dist(T_1, T_2)$

denote the height of a tree T, the outdegree (i.e., the number of children) of $v \in V(T)$, and the tree edit distance between T_1 and T_2, respectively.

For each node $v \in V(T)$, the leaf distance $d_l(v)$ is the maximum length of a path from v to any leaf in $T(v)$. In addition, the leaf distance histogram $h_l(T)$ is defined as a vector of size $k = 1 + height(T)$, where $h_l(T)[i] = |\{v \mid v \in T,\ d_l(v) = i\ (0 \leq i \leq k)\}|$. Using these definitions, the following theorem is obtained [26]:

Theorem 2.2.1. *Let T_1 and T_2 be rooted trees. For any T_1 and T_2, the L_1-distance of the leaf distance histogram gives the following lower bound of the tree edit distance between T_1 and T_2 under the unit-cost edit operation model:*

$$L_1(h_l(T_1), h_l(T_2)) \leq dist(T_1, T_2).$$

Similarly, the degree histogram $h_d(T)$ is defined as a vector of size $k = 1 + D$, where $h_d(T)[i] = |\{v \mid v \in T,\ deg(v) = i\ (0 \leq i \leq k)\}|$ and D is the maximum outdegree of T, and then the following theorem is also obtained [26]:

Theorem 2.2.2. *Let T_1 and T_2 be rooted trees. For any T_1 and T_2, the L_1-distance of the degree histogram gives the following lower bound of the tree edit distance between T_1 and T_2 under the unit-cost edit operation model:*

$$\frac{L_1(h_d(T_1), h_d(T_2))}{3} \leq dist(T_1, T_2).$$

Let $S_{lab}(T)$ be a set of distinct labels appearing in a tree T. The label histogram $h_{lab}(T)$ is defined as a vector of size $k = |S_{lab}(T)|$ where $h_{lab}[i] = |\{v \mid v \in T,\ \ell(v) = S_{lab}(T)[i]\ (1 \leq i \leq k)\}|$. The theorem on the label histogram is as follows [26]:

Theorem 2.2.3. *Let T_1 and T_2 be rooted trees. For any T_1 and T_2, the L_1-distance of the label histogram gives the following lower bound of the tree edit distance between T_1 and T_2 under the unit-cost edit operation model:*

$$\frac{L_1(h_{lab}(T_1), h_{lab}(T_2))}{2} \leq dist(T_1, T_2).$$

Finally, we end this subsection with giving an example of the above estimations in Figure 2.9. In Figure 2.9(i), $L_1(h_l(T_1), h_l(T_2)) = |5 - 5| + |1 - 0| + |0 - 1| = 1$, so that $dist(T_1, T_2)$ is at least 1. Similarly, $L_1(h_d(T_1), h_d(T_2))/3 = 1$ and $L_1(h_{lab}(T_1), h_{lab}(T_2))/2 = 1$ give 1 and 3.5 as lower bounds of $dist(T_1, T_2)$, respectively. We can obtain tighter lower bound than each lower bound by taking the maximum value of the three results. It should be noted that Theorem 2.2.2 is introduced to A^*-algorithm (described in Section 2.2.1) and it corresponds to the heuristic function h_3 in the algorithm.

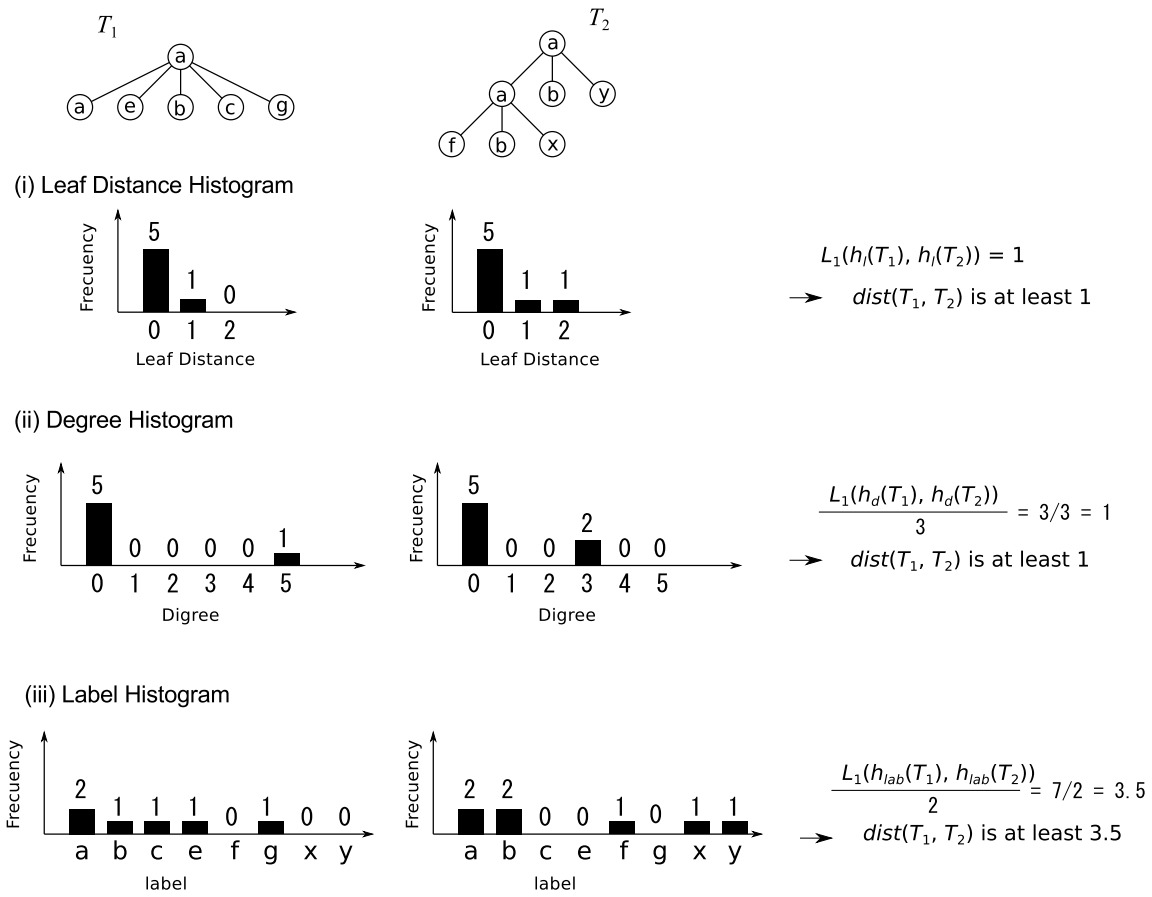


Figure 2.9: Histgram-based estimations for a lower bound of the tree edit distance. T_1 and T_2 are input trees. (i), (ii), and (iii) represent the leaf distance histogram, the degree histogram, and the label histogram of the input trees, respectively.

$O(n/\log n)$-approximation Algorithm

On the basis of the fixed-parameter algorithm (presented in Section 2.2.1), Akutsu et al. developed an approximation algorithm which gives the following theorem [2]:

Theorem 2.2.4. *Let T_1 and T_2 be rooted unordered trees. The tree edit distance between T_1 and T_2 can be approximated within an $O(n/\log n)$-factor under the unit-cost edit operation model, where $n = \max\{|V(T_1)|, |V(T_2)|\}$.*

The factor $O(n/\log n)$ is derived by focusing on whether or not $dist(T_1, T_2) \leq K$ holds, where $K = c \log n$ (c is an arbitrary constant). If $dist(T_1, T_2) \leq K$, the fixed-parameter algorithm can compute the exact distance. Otherwise, we obtain the approximate distance by deleting all $u \in V(T_1)$ (except for $r(T_1)$) from T_1,

substituting the label $\ell(r(T_1))$ into $\ell(r(T_2))$ (if necessary), and inserting all $v \in V(T_2)$ (except for $r(T_2)$) to T_1. More details of the proof are given in [2].

$O(h)$-approximation Algorithm

In addition to the $O(n/\log n)$-approximation algorithm, Akutsu et al. considered bounded height trees and developed an $O(h)$-approximation algorithm [2]. For a pair of trees t and T, $\phi_t(T)$ is defined as the number of $T(v)'s$ isomorphic to t (i.e., $\phi_t(T) = |\{v \in V(T) \mid T(v) \approx t\}|$, where $T_1 \approx T_2$ means that T_1 and T_2 are isomorphic). The feature vector $\phi(T) = (\phi_t(T))_{t \in \mathcal{S}_n}$ is also defined, where $\mathcal{S}_n$ is the set of all possible trees of size at most n. It should be noted that $\phi(T)$ can be represented in polynomial size in spite of its actual size is exponential to n since we only need to keep the non-zero elements and the number of non-zero elements is at most n for each input tree. Then, the following lemmas are obtained [2], where $L_1(\phi(T_1), \phi(T_2))$ denotes the L_1-distance between $\phi(T_1)$ and $\phi(T_2)$:

Lemma 2.2.4.1. $L_1(\phi(T_1), \phi(T_2)) \leq (2h + 2) \cdot dist(T_1, T_2)$.

Lemma 2.2.4.2. $dist(T_1, T_2) \leq L_1(\phi(T_1), \phi(T_2))$.

Finally, the main result is obtained from these lemmas [2].

Theorem 2.2.5. *Let T_1 and T_2 be unordered trees. The tree edit distance between T_1 and T_2 can be approximated within a factor of $2h + 2$ under the unit-cost edit operation model, where $h = \max\{height(T_1), height(T_2)\}$.*

2.2.5 Approximation of Largest Common Subtree

Before the approximation algorithms mentioned in Section 2.2.4 have been proposed, there exists no approximation algorithm with a guaranteed approximation ratio for the general unordered cases. Instead, some approximation algorithms for the *largest common subtree* (LCST) have been developed in [22]. After that, the approximation ratios have been improved in [2, 3], respectively.

LCST is defined through the unordered edit distance mapping (unordered mapping, for short). Recall that $M \subseteq V(T_1) \times V(T_2)$ is an unordered mapping if the following conditions are satisfied for any two pairs (u_1, v_1), $(u_2, v_2) \in M$:

- $u_1 = u_2$ if and only if $v_1 = v_2$,
- $u_1 \in des(u_2)$ if and only if $v_1 \in des(v_2)$.

If $\ell(u) = \ell(v)$ holds for any pairs of labels $(\ell(u), \ell(v))$, the subtrees obtained by deleting nodes not appearing in M from T (T is either T_1 or T_2) are isomorphic. Such a resulting tree is called a *common subtree* between T_1 and T_2. Moreover,

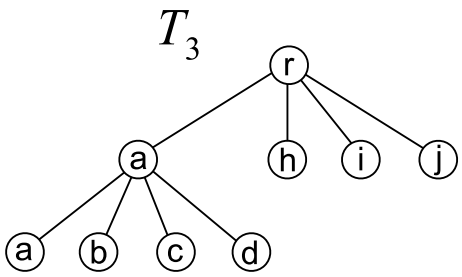


Figure 2.10: Example of largest common subtree (LCST). T_3 is the LCST between T_1 and T_2 in Figure 2.2.

a common subtree with the maximum score is called the LCST, where a score of common subtree is $\sum_{(u,v)\in M} f(u,v)$ (refer to Section 2.2.1). For example, T_3 in Figure 2.10 is the LCST between T_1 and T_2 in Figure 2.2. For the unordered LCST problem, $2h$- and $\log^2 n$-approximation algorithms have been proposed in [22], where h is the maximum height of input trees. These approximation ratios have been improved to $1.5h$ [3] and $\log n$ [2], respectively. We briefly review these algorithms in the rest of this subsection.

$1.5h$-approximation Algorithm

Before presenting the $1.5h$-approximation algorithm, we review a $2h$-approximation algorithm proposed in [22].

The procedure of $2h$-approximation algorithm is as follows: Let T_1 and T_2 be rooted unordered trees, where $height(T_1) \leq height(T_2)$ without loss of generality. For each depth d, a leaf node v of T_2 is matched to $u \in V(T_1)$ of depth d greedily and repeatedly. When v is matched to u, v and its ancestors are deleted from T_2. In order to analyze the approximation ratio, we consider the case such that the height of T_1 equals to 1. Let I be the set of nodes of T_1 at depth d. For each node v, when v is matched to u of depth d, at most two nodes in I can be removed from an optimal mapping: u and a node matched to an ancestor of v. From this observation, we can see that the approximation ratio is 2 if the height of T_1 is 1 under the unit-cost edit operation model. Using the partitioning technique employed in [22], T_1 can be partitioned into $height(T_1)$ zero-height forests. Then, applying Proposition 6 in [22], the approximation ratio $2h$ for the general case is obtained, where $h = height(T_1)$.

Akutsu et al. has improved the approximation ratio to $1.5h$ [3]. At first, they consider a special case of the LCST problem in which the maximum height of T_1 and T_2 are 1 and h, respectively, and each node in T_2 at depth greater than 1 has at most one child node. The special case is denoted by $LCST_2(1,h)$ for short. The algorithm $ApxLCST_2(T_1,T_2)$ for $LCST_2(1,h)$ is proposed as follows [3]:

Procedure $ApxLCST_2(T_1, T_2)$
$T^0 \leftarrow \{r(T_2)\}$;
Let $v_1, \ldots, v_m$ be nodes with depth 1 in T_2;
for $i = 1$ to m **do**
  if $|OPT(T_1, T^{i-1} + T_2(v_i) - v_i)| \geq |OPT(T_1, T^{i-1})| + 2$
  then $T^i \leftarrow T^{i-1} + T_2(v_i) - v_i$;
  else $T^i \leftarrow T^{i-1} + linear(T_2(v_i))$;
Compute $OPT(T_1, T^m)$,

where $T_1 + T_2$ denotes the tree obtained by making $r(T_2)$ to be a child of $r(T_1)$, $T - v$ denotes the tree obtained by deleting v from T, $OPT(T_1, T_2)$ denotes an arbitrary optimal mapping between T_1 and T_2, and $linear(T)$ denotes a linear chain obtained by ordering distinct labels appearing in T, respectively. In this procedure, $ApxLCST_2(T_1, T_2)$ checks whether there exist at least two more nodes in $T(v_i)$ being able to be matched to some leaves of T_1 or not for each node v_i at depth 1 in T_2 repeatedly. If such nodes are founded, v_i is deleted from T_2. Otherwise, $T(v_i)$ is transformed into the corresponding linear chain. Finally, $ApxLCST_2(T_1, T_2)$ computes an optimal mapping between T_1 and a resulting tree (i.e., T^m in the above procedure) by solving the corresponding maximum weighted bipartite matching problem.

Akutsu et al. has extended $ApxLCST_2(T_1, T_2)$ to the more general case in which the maximum height of input trees is bounded by a constant h, and then the approximation ratio of the extended algorithm has been proved to be $1.5h$ (refer to [3] for the details of the proof).

$\log n$-approximation algorithm

In a related line of research, a $\log^2 n$-approximation algorithm also has proposed proposed in [22]. The algorithm employs the heavy chain decomposition technique for decomposing input trees into a set of $\log n$ forests of linear chains, and then finds an optimal mapping between two forests using weighted bipartite matching, where a weight of a pair of linear chains is the size of LCST between the pair. From this procedure, the approximation ratio is derived to be $\log^2 n$ [22].

After that, the approximation ratio has been improved to $\log n$ [2]. The approximation algorithm is obtained using a reduction to a special case of the maximum independent set problem in graphs. In the reduction, a class of trees, termed *spiders*, is considered where the outdegree of each non-root node of a spider is at most 2. Then, an approximation algorithm for computing a maximum common subtree between spiders is given. Finally, it is shown that any tree must contain a large spider and the algorithm is applied to the general case. From the reduction, the following theorem for the general common subtree problem is obtained [2]:

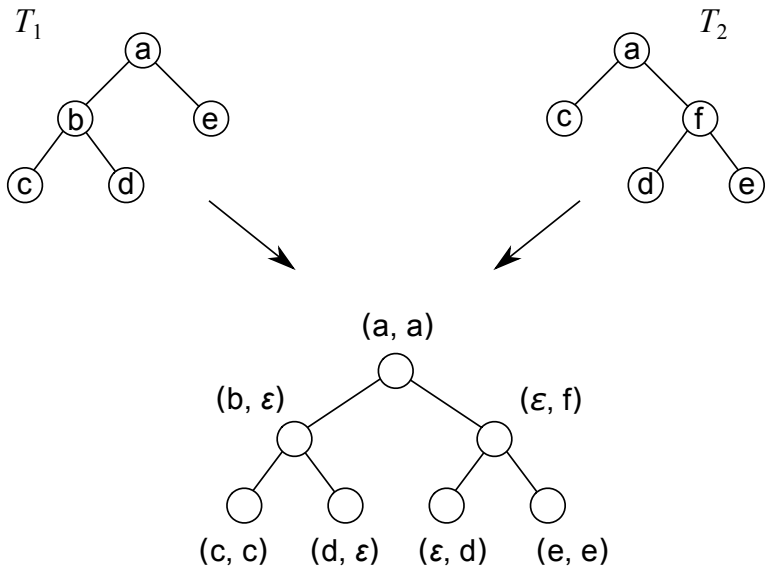


Figure 2.11: Tree alignment

Theorem 2.2.6. *There is a* $t \cdot \log(b_{OPT}+1)$*-approximation algorithm for the largest common subtree problem of* t *trees, where* b_{OPT} *is the number of branching nodes (i.e., nodes having two or more children) in an optimal common subtree.*

It should be noted that b_{OPT} is bounded by n [2]. Therefore, the approximation ratio of this algorithm is $\log n$.

2.3 Tree Alignment

The tree edit distance problem for the unordered cases is NP-hard, so that other distance measures have been proposed in order to reduce the computational cost. Among them, *tree alignment* is well-known [25]. Let T_1 and T_2 be rooted trees. An *alignment* A between T_1 and T_2 is computed by inserting nodes labeled ϵ into T_1 and T_2 so that the resulting trees are isomorphic without including label information, and then overlaying the trees (see Figure 2.11). Each node $v \in V(A)$ has a label (x, y) and its cost is defined as $\delta(x, y)$, and then the cost of A is defined as the sum of the costs of all nodes. An alignment between T_1 and T_2 with the minimum cost is called an *optimal tree alignment*. Let $dist_\alpha(T_1, T_2)$ be the cost of an optimal tree alignment. The tree alignment problem can be regarded as a restricted case of the tree edit distance problem, in which all insertions must be done before any deletions. Thus, the following inequality holds:

$$dist(T_1, T_2) \leq dist_\alpha(T_1, T_2).$$

Figure 2.11 gives an example that the above inequality holds. Let $\delta(x, y) = 1$ for all pairs of label x and y including ϵ. In the case, the alignment A is optimal and the cost is 4 since two insertion operations are required in T_1 and T_2, respectively, so that the total number of insertion operations is 4. On the other hand, deleting a node labeled "b" from T_1 and inserting a node labeled "f" to T_1 as a parent of nodes labeled "d" and "e" are required for transforming T_1 and T_2 and this is the minimum cost sequence under the unit-cost edit operation model. Therefore, the tree edit distance is 2. It should be noted that the tree alignment does not satisfy the conditions of the distance metric.

Jiang's Algorithm

Jiang et al. have proposed a DP-based algorithm for computing an optimal tree alignment [25]. For the ordered cases, the algorithm works in $O(|T_1||T_2|\cdot (D_1 + D_2^2)$ time, where D_i is the maximum outdegree of T_i. On the other hand, the unordered tree alignment problem is NP-hard [25]. However, if the maximum outdegrees of T_1 and T_2 are bounded by a constant, the problem can be solved in $O(|T_1||T_2|)$ time. It should be noted that the unordered tree edit distance problem is MAX SNP-hard even when the trees have bounded outdegrees [60]. Therefore, an optimal tree alignment might be useful instead of the tree edit distance.

2.4 Tree Inclusion

Kilpeläinen et al. considered *tree inclusion*, which is another variant of the tree edit distance [30, 31]. Let T_1 and T_2 be rooted trees. T_1 is *included* in T_2 if T_2 can be obtained only by inserting nodes to T_1 (see also Figure 2.12). The tree inclusion problem is defined as to determine whether one tree is included in another tree or not. In tree inclusion, an injective function $f : V(T_1) \to V(T_2)$ is called an *embedding* if the following conditions hold for every u and v in $V(T_1)$:

- $\ell(u) = \ell(f(u))$, (label preservation condition)
- $u \in des(v)$ if and only if $f(u) \in des(f(v))$, (descendant condition)
- u is the left of v if and only if $f(u)$ is the left of $f(v)$, (sibling condition)

where the sibling condition is not required for the unordered cases.

For the unordered cases, the tree inclusion problem is NP-hard. However, it can be solved in $O(|T_1||T_2|)$ time if the maximum outdegree of T_1 is bounded by a constant [31].

In Chapter 4, we deal with the unordered tree inclusion problem, where more details are given.

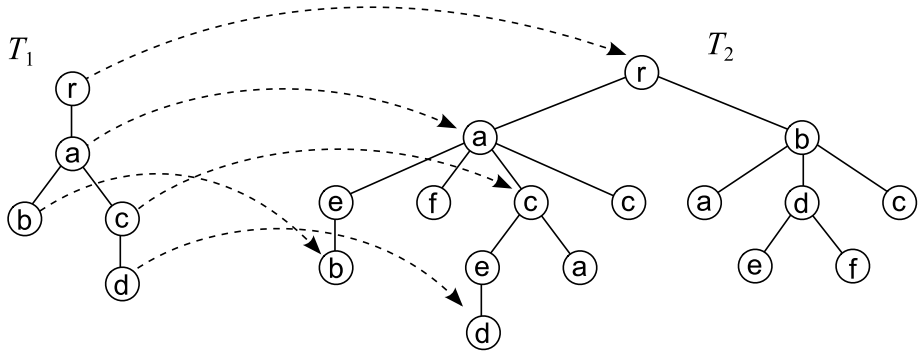


Figure 2.12: Tree inclusion. T_1 is included in T_2. The embedding between T_1 and T_2 is indicated by dashed curves.

Chapter 3

Clique-Based Method Using Dynamic Programming for Computing Edit Distance between Unordered Trees

This chapter presents an improved clique-based method for the tree edit distance problem for unordered trees. The improved method is obtained by introducing a dynamic programming scheme and heuristic techniques to the previous clique-based method. In order to evaluate the efficiency of the improved method, we apply the method to comparison of real tree structured data: Weblogs data and glycan structures.

3.1 Background

As mentioned in Chapter 2, various techniques have been developed and applied to analysis of the tree-structured data, such as XML data, RNA secondary structures, and glycans. Among them, comparison of tree-structured data is important because it can be used for searching for similar objects. The *tree edit distance* is one of the most widely used measures for comparison of tree-structured data [14]. In this measure, the distance between two trees is measured by the minimum cost sequence of edit operations that transforms one tree into another tree where an edit operation is either a *deletion* of a node, an *insertion* of a node, or a *substitution* of a label of a node. For the tree edit distance problem for ordered trees, Tai developed an $O(n^6)$ time algorithm [48], where n is the number of nodes in a larger input tree. After several improvements, Demaine et al. developed an $O(n^3)$ time algorithm and showed that this bound is optimal under some computation strategy [16].

The tree edit distance between ordered trees is useful if the ordering among children has important meanings. However, it is preferable to regard input trees as unordered trees in some applications [8, 24]. Unfortunately, Zhang et al. proved that the tree edit distance problem for unordered trees is NP-hard [61]. In order to cope with this hardness, Akutsu et al. developed a fixed parameter algorithm which works in $O(2.62^k \cdot poly(n))$ time [5], where k is the maximum allowed edit distance. Although their algorithm might be useful for comparison of very similar trees (i.e., where k is small), it is not useful for comparison of non-similar trees. Horesh et al. developed an A^*-algorithm [24]. Although their algorithm works efficiently for comparison of moderate size unlabeled trees under the unit cost distance (i.e., the cost of each edit operation is 1), it is unclear whether it can be efficiently applied to labeled trees or general cost cases. Kondo et al. proposed a practical integer programming (IP) based algorithm [32]. Although the IP-based algorithm is much faster than the existing methods (including our method proposed in this chapter) when the maximum degrees of input trees are large, it is not so effective for the cases of the small maximum degrees.

Fukagawa et al. recently proposed a practical method for computing the tree edit distance between unordered trees [19] using algorithms for computing the *maximum clique* [52, 50]. In this method, an instance of the tree edit distance is directly transformed into an instance of the *maximum vertex weighted clique* problem and then an existing clique solver [42] is applied. Although similar reductions have been proposed for variants of the tree edit distance problem [46, 53] and other problems [44], to the best of our knowledge, it was the first method that exactly solves the proper tree edit distance problem for unordered trees using the maximum clique. The method was applied to comparison and search of similar glycan structures and shown to be efficient for moderate size tree structures [19]. However, it was not fast enough if large glycan or tree structures were given.

Therefore, in the preliminary version of this study [6], we improved the method of [19] and developed a *dynamic programming*-based (DP) algorithm that repeatedly solves instances of the maximum vertex weighted clique problem as sub-problems. Due to this improvement, sparser graphs are generated and thus maximum clique instances can be solved more efficiently in many cases. Although multiple clique instances must be solved in the improved method, it is expected that speed up due to sparsity is more beneficial if input trees are large. Furthermore, by utilizing the feature of DP, we introduced heuristic techniques which do not violate the optimality of the solution. When it was applied to comparison of large glycan structures, our improved method showed speed-up in most cases. However, there still exist cases for which it takes long CPU time. In particular, it takes very long CPU time if there exist many leaves. In such a case, constructed graphs would contain many vertices and edges and thus a clique algorithm does not work efficiently.

In this chapter, we augment this DP-based approach by introducing new heuris-

tic techniques to further reduce the computation time without violating the optimality of the solution, especially for trees with many leaves or many isomorphic subtrees. Furthermore, in order to utilize maximum clique algorithms in place of maximum vertex weighted clique algorithms, we develop a new clique-based method for computing the unordered tree edit distance in which the maximum vertex weighted clique problem is transformed into the maximum clique problem.

We compare the improved clique-based method and the maximum clique-based method with the previous maximum vertex weighted clique-based method [19] using Weblogs data CSLOGS opened to the public[1] [57] and glycan data obtained from the KEGG database [28]. The results suggest that the improved clique-based method is much faster than the maximum clique-based method and the previous clique-based method [19] in most cases of comparison of large tree-structured data. In particular, when there exist many leaves or isomorphic subtrees, our improved method shows significant speed-up.

3.2 Method

3.2.1 Maximum Vertex Weighted Clique

Let $G = (V, E)$ be an undirected graph. A subgraph $G' = (V', E')$ of $G = (V, E)$ is called a *clique* if it is a complete subgraph (i.e., $\{\{v_i, v_j\} \mid v_i, v_j \in V',\ v_i \neq v_j\} = E'$). The *maximum clique* problem is to find a clique with the maximum number of vertices in a given undirected graph $G = (V, E)$. Although the maximum clique problem is NP-hard, several practical algorithms have been developed [52, 50]. In this chapter, we use a variant of the maximum clique problem called *the maximum vertex weighted clique problem*. In this variant, each vertex v has a weight $w(v)$ and the problem is to find a clique $G' = (V', E')$ which maximizes $\sum_{v \in V'} w(v)$ (see also Figure 3.1).

3.2.2 Algorithm MWCQ and MCS

Nakamura and Tomita developed a practically efficient algorithm called MWCQ for the maximum vertex weighted clique problem [42]. After preliminary experiments on maximum vertex weighted clique algorithms [42], we employ MWCQ as a solver for the maximum vertex weighted clique problem. Here, we briefly review MWCQ.

The underlining algorithm of MWCQ is a very simple and fast branch-and-bound depth-first-search algorithm MCQ for finding a maximum clique of an unweighted graph [51, 50]. MCQ employs greedy approximate coloring to obtain an upper bound of the size of a maximum clique. The size of a maximum clique in an

[1] http://www.cs.rpi.edu/~zaki/software/

Figure 3.1: Example of the maximum clique and the maximum vertex weighted clique. The size of the maximum clique of the left graph is four, while the weight of the maximum vertex weighted clique of the right graph is 10.

unweighted graph is bounded above by the number of approximate color classes (the total number of disjoint sets of independent set). This relation contributes to an effective bounding condition.

For a vertex weighted graph, the maximum weight of a clique in a graph is bounded above by the summation of the maximum weight in each approximate color class (independent set). Then we have a simple algorithm MWCQ for finding a maximum vertex weighted clique by introducing the above new bounding condition into MCQ instead of the previous one together with appropriate ordering of vertices as in MCQ.

Furthermore, Tomita et al. proposed a new branch-and-bound algorithm MCS for the maximum clique problem [52]. In MCS, new approximate coloring is introduced along with other new techniques, which makes MCS much faster than MCQ for most instances. In order to utilize MCS, to be shown below, we develop a method which does not use a maximum vertex weighted clique algorithm but instead uses a maximum clique algorithm.

3.2.3 Previous Method

Before presenting our improved clique-based method, we briefly review the previous clique-based method [19] (see also Figure 3.2), which is referred to as **CliqueEdit** in this chapter.

CliqueEdit is based on a simple reduction from the tree edit distance problem for unordered trees to the maximum vertex weighted clique problem. Based on Eq. (2.2), for calculating the tree edit distance, it is enough to find a mapping M maximizing $\sum_{(u,v)\in M} f(u,v)$. In order to find such a mapping, an undirected graph

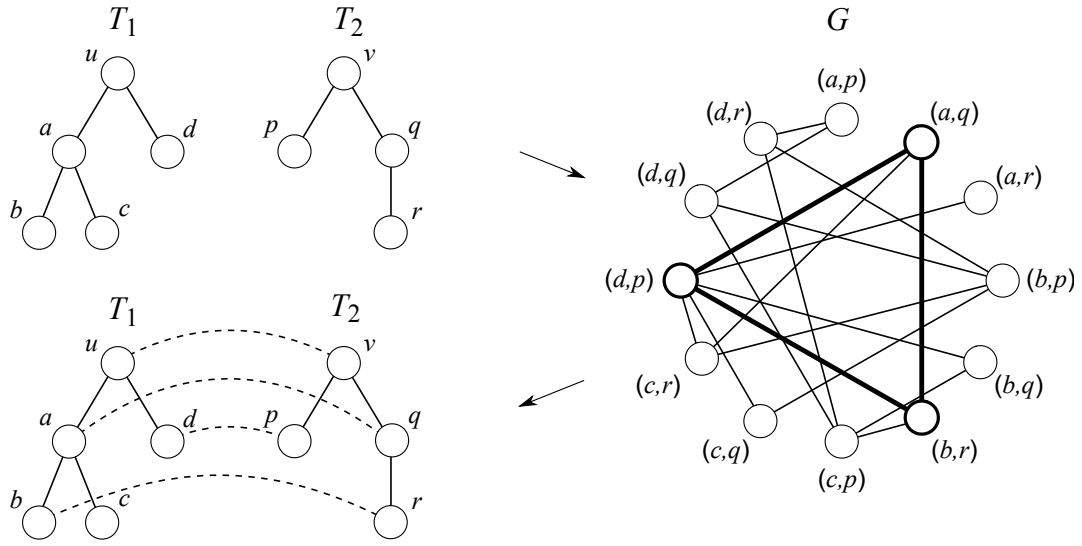


Figure 3.2: Example of a reduction in the previous method (CliqueEdit). A maximum clique is shown by bold lines in graph G, and the corresponding mapping is shown by broken lines in bottom left side.

$G = (V, E)$ is constructed from two input trees T_1 and T_2 by

$$\begin{aligned} V &= \{\ (u,v) \mid u \in V(T_1),\ u \neq r(T_1),\ v \in V(T_2),\ v \neq r(T_2)\ \}, \\ E &= \{\ \{(u_1,v_1),(u_2,v_2)\} \mid u_1 \neq u_2,\ v_1 \neq v_2, \\ &\qquad u_1 \in des(u_2) \text{ iff } v_1 \in des(v_2), \\ &\qquad u_2 \in des(u_1) \text{ iff } v_2 \in des(v_1)\ \}. \end{aligned}$$

Then, we can see that there is a one-to-one correspondence between the set of cliques and the set of mappings (i.e., (u, v) in a clique corresponds to (u, v) in a mapping M). By assigning a weight $w(x) = f(u, v)$ to each vertex $x = (u, v) \in V$, an optimal mapping M_{OPT} corresponds to a maximum vertex weighted clique. Therefore, the tree edit distance problem can be solved by computing a maximum vertex weighted clique.

3.2.4 Reduction from the Maximum Vertex Weighted Clique Problem to the Maximum Clique Problem

In order to utilize MCS [52] instead of MWCQ [42], we develop a simple method that transforms the maximum vertex weighted clique problem into the maximum clique problem.

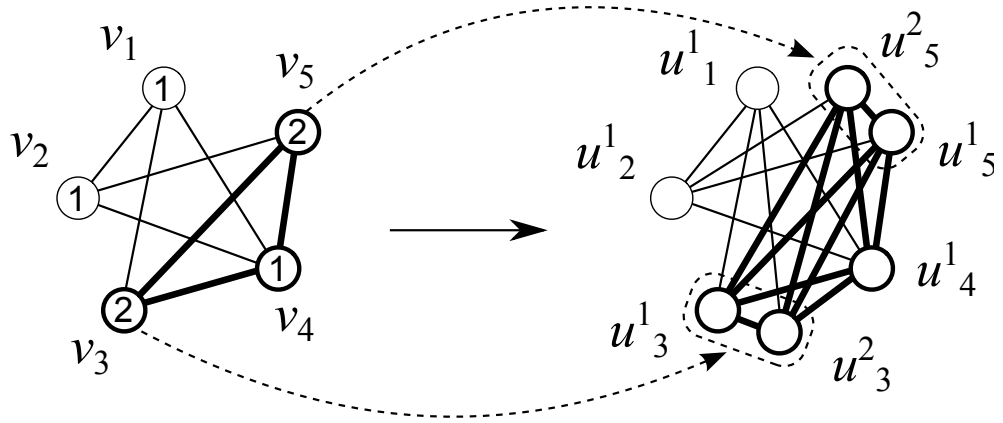


Figure 3.3: Example of transformation of a vertex weighted graph into an unweighted graph. v_3 and v_5 are duplicated.

Let $G = (V, E)$ be a weighted graph such that $V = \{v_1, \ldots, v_n\}$ and each vertex v_i has a weight $w(v_i)$. From $G = (V, E)$, we construct an unweighted graph $\hat{G} = (\hat{V}, \hat{E})$ by (see also Figure 3.3)

$$\hat{V} = \{v_i^j \mid v_i \in V,\ j = 1, 2, \ldots, w(v_i)\}, \tag{3.1}$$
$$\hat{E} = \{\{v_i^j, v_k^\ell\} \mid \{v_i, v_k\} \in E \vee (\ i = k \wedge j \neq \ell\ )\}. \tag{3.2}$$

Proposition 3.2.1. *The weight of the maximum vertex weighted clique of $G = (V, E)$ is equal to the size of the maximum clique of $\hat{G} = (\hat{V}, \hat{E})$.*

Proof. Suppose that there exists a clique $G_c = (V_c, E_c)$ with the weight W and the size m in G, where

$$V_c = \{v_{i_1}, \ldots, v_{i_m}\},$$
$$E_c = \binom{V_c}{2}.$$

In this case, there also exists a clique $\hat{G}_c = (\hat{V}_c, \hat{E}_c)$ in $\hat{G}$, where

$$\hat{V}_c = \{v_{i_k}^j \mid v_{i_k} \in V_c,\ j = 1, 2, \ldots, w(v_{i_k})\},$$
$$\hat{E}_c = \{\{v_{i_h}^j, v_{i_k}^\ell\} \mid \{v_{i_h}, v_{i_k}\} \in E_c \vee (i_h = i_k \wedge j \neq \ell)\}.$$

Since $|\hat{V}_c| = W$ and $\hat{E}_c = \binom{\hat{V}_c}{2}$, $\hat{G}_c$ is a clique with the size W. Hence, if there exists a clique with the weight W in G, there exists a clique with the size W in $\hat{G}$, so

that if G has the maximum vertex weighted clique with the weight W, $\hat{G}$ has the maximum clique with the size W.

Conversely, we assume that there exists a clique $\hat{G}_c = (\hat{V}_c, \hat{E}_c)$ with the size W in $\hat{G}$, where

$$\begin{aligned} \hat{V}_c &= \{v_{i_1}^{j_1}, \ldots, v_{i_W}^{j_W}\}, \\ \hat{E}_c &= \binom{\hat{V}_c}{2}. \end{aligned}$$

Here, $v_{i_k}^{j_k}$ is a copy of v_{i_k} and $i_h = i_k$ can hold. From the way to construct $\hat{G}$ (see Eq. (3.1) and (3.2)), there also exists at least one clique $\hat{G}'_c = (\hat{V}'_c, \hat{E}'_c)$ with the size $W'(\geq W)$ in $\hat{G}$, where

$$\begin{aligned} \hat{V}'_c &= \{v_{i_k}^j \mid v_{i_k}^{j_k} \in \hat{V}_c,\ j = 1, \ldots, w(v_{i_k})\}, \\ \hat{E}'_c &= \binom{\hat{V}'_c}{2}. \end{aligned}$$

In this case, there also exists a clique $G_c = (V_c, E_c)$ in G, where

$$\begin{aligned} V_c &= \{v_{i_k} \mid v_{i_k}^{j_k} \in \hat{V}_c\}, \\ E_c &= \{\{v_{i_h}, v_{i_k}\} \mid \{v_{i_h}^j, v_{i_k}^\ell\} \in \hat{E}'_c \wedge i_h \neq i_k\}. \end{aligned}$$

Since the weight of G_c is equal to W' and $E_c = \binom{V_c}{2}$, G_c is a clique with the weight W'. Hence, if there exists a clique with the size W in $\hat{G}$, there exists a clique with the size $W'(\geq W)$ in $\hat{G}$, so that there exist a clique with the weight W' in G. Thus, if $\hat{G}$ has the maximum clique with the size W, G has the maximum vertex weighted clique with the weight W. Therefore, the weight of the maximum vertex weighted clique of G is equal to the size of the maximum clique of $\hat{G}$. □

The method of combining this transformation with CliqueEdit is called **Uw-CliqueEdit**. Since a vertex with weight w is transformed into w vertices, this method can only be applied to graphs with small integer vertex weights. However, if we consider the unit cost edit distance, each vertex in $G = (V, E)$ has weights 1 or 2. Therefore, this method can be applied to computation of the unit cost tree edit distance.

3.2.5 Improved Method

In order to improve CliqueEdit, we combine a *dynamic programming* (DP) approach employed in [5] with the clique-based approach. We call the resulting method **DpCliqueEdit**.

Let $(u, v) \in V(T_1) \times V(T_2)$. We define $W[u, v]$ be the score of an optimal mapping between $T_1(u)$ and $T_2(v)$ where the root of $T_1(u)$ need not correspond to the root of $T_2(v)$. We compute $W[u, v]$ in a bottom up way (i.e., from leaves to roots) using DP. Suppose that $W[u', v']$ are already computed for all $(u', v') \in des(u) \times des(v)$. Then, we construct an undirected vertex weighted graph $G_{(u,v)} = (V_{(u,v)}, E_{(u,v)})$ by

$$\begin{aligned} V_{(u,v)} &= \{\ (u_1, v_1) \mid u_1 \in des(u),\ v_1 \in des(v)\ \}, \\ E_{(u,v)} &= \{\ \{(u_1, v_1), (u_2, v_2)\} \mid u_1 \neq u_2,\ v_1 \neq v_2, \\ &\qquad u_1 \notin des(u_2),\ u_2 \notin des(u_1), \\ &\qquad v_1 \notin des(v_2),\ v_2 \notin des(v_1)\ \}, \\ w((u_1, v_1)) &= W[u_1, v_1]. \end{aligned}$$

Let W_{max} be the weight of the maximum vertex weighted clique for $G_{(u,v)}$. Then, we calculate $W[u, v]$ by[2]

$$W[u, v] = \max \begin{cases} \max_{v' \in des(v)} W[u, v'], \\ \max_{u' \in des(u)} W[u', v], \\ W_{max} + f(u, v), \end{cases}$$

where $W[u, v]$ is initialized by

$$W[u, v] = \begin{cases} \max_{v' \in \{v\} \cup des(v)} f(u, v') \text{ if } u \text{ is a leaf}, \\ \max_{u' \in \{u\} \cup des(u)} f(u', v) \text{ if } v \text{ is a leaf}. \end{cases}$$

Different from the reduction in CliqueEdit, edges are not created in DpCliqueEdit if there is a descendant-ancestor relation between u_1 and u_2 (or between v_1 and v_2, see also Figure 3.4). Therefore, it is expected that graphs constructed in DpCliqueEdit are much sparser than those in CliqueEdit though DpCliqueEdit must solve many clique instances. Since sparseness of the graph greatly affects the efficiency of clique finding, it is also expected that DpCliqueEdit is faster than CliqueEdit if non-small trees are given. It is to be noted that transformation to maximum clique cannot be applied to this case because $W[u, v]$ might take a large value even for the unit cost case.

3.2.6 Heuristics

In addition to the use of dynamic programming, we introduce some heuristic techniques to reduce the computation time without violating the optimality of the solution.

[2]A slight modification is required if u or v is a root because roots cannot be deleted or inserted.

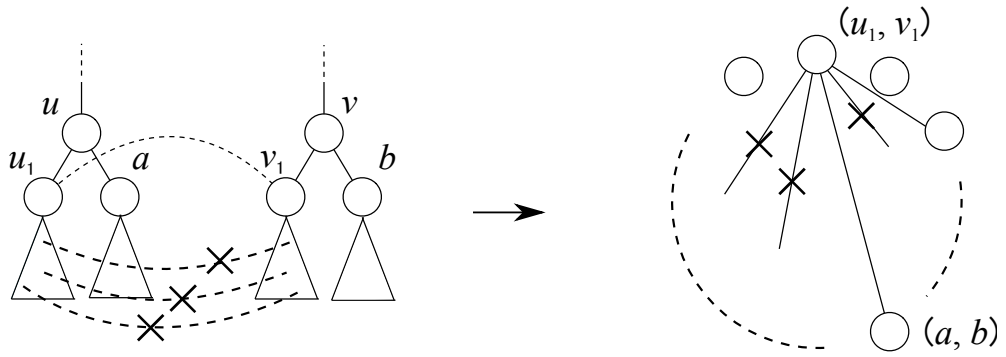


Figure 3.4: Difference between the reductions in CliqueEdit and DpCliqueEdit. In computation of $W[u, v]$ in DpCliqueEdit, vertex (u_1, v_1) in $G_{(u,v)}$ is not connected to any one of vertices corresponding to pairs of descendants of u_1 and v_1.

An important observation is that

$$W[u_1, v_1] \geq W[u_2, v_1] \tag{3.3}$$

always holds if u_2 is a descendant of u_1. Based on it, we introduce the following two heuristic techniques.

Heuristic technique (1) Each of u and v has only one child.

In this case, we need not construct $G_{(u,v)}$. Instead, we can compute $W[u, v]$ simply by taking the maximum of

$$W[u, v_1],\ W[u_1, v],\ W[u_1, v_1] + f(u, v),$$

where u_1 and v_1 are the children of u and v, respectively (see also Figure 3.5).

Proposition 3.2.2. *If each of $u \in T_1$ and $v \in T_2$ has only one child, $W[u, v]$ can be computed by $W[u, v] = \max\{W[u, v_1], W[u_1, v], W[u_1, v_1] + f(u, v)\}$, where u_1 and v_1 are the children of u and v, respectively.*

Proof. We consider the following three cases.

(i) In the case that u (resp. v) is deleted and v (resp. u) is not deleted, from the assumption and Eq. (3.3), since $W[u_1, v] \geq W[u'_1, v]$ for all $u'_1 \in des(u_1)$ (resp. $W[u, v_1] \geq W[u, v'_1]$ for all $v'_1 \in des(v_1)$), $W[u, v] = W[u_1, v]$ (resp. $W[u, v] = W[u, v_1]$) holds.

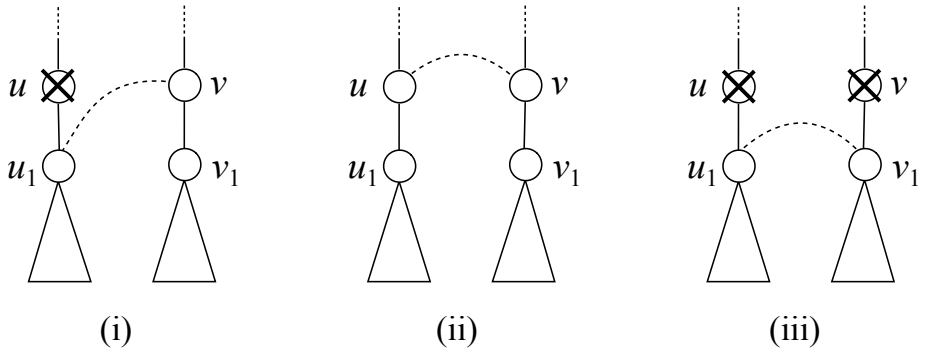


Figure 3.5: Heuristic technique (1). (i), (ii), and (iii) denote the case (i) $\sim$ (iii) in the proof of the Proposition 3.2.2.

(ii) In the case that u corresponds to v, since each of u and v has only one child, $W_{max} = W[u_1, v_1]$, where W_{max} is the weight of the maximum vertex weight clique for $G_{(u,v)}$. Hence, $W[u, v] = W_{max} + f(u, v) = W[u_1, v_1] + f(u, v)$.

(iii) In the case that both u and v are deleted, $W[u, v] = W[u_1, v_1]$ apparently. Now, the score of this case is smaller than or equal to that of case (ii), because $f(u, v) \geq 0$.

Therefore, we can calculate $W[u, v]$ by

$$W[u,v] \;=\; \max \begin{cases} W[u, v_1], \\ W[u_1, v], \\ W[u_1, v_1] + f(u, v). \end{cases}$$

□

Heuristic technique (2) $u_2 \in des(u)$ (resp. $v_2 \in des(v)$) does not have a sibling.

In this case, we need not generate a vertex (u_2, v') for any v' (resp. (u', v_2) for any u') in the construction of $G_{(u,v)}$ because any mapping between $T_1(u_2)$ and $T_2(v)$ can be included in some mapping between $T_1(u_1)$ and $T_2(v)$ where u_1 is the parent of u_2 (see also Figure 3.6).

Proposition 3.2.3. *Suppose that $u \in T_1$ and $v \in T_2$. If $u_2 \in des(u)$ (resp. $v_2 \in des(v)$) does not have a sibling, $W[u, v]$ can be computed without generating a vertex (u_2, v') for any v' (resp. (u', v_2) for any u') in the construction of $G_{(u,v)}$, where $v' \in des(v)$ (resp. $u' \in des(u)$).*

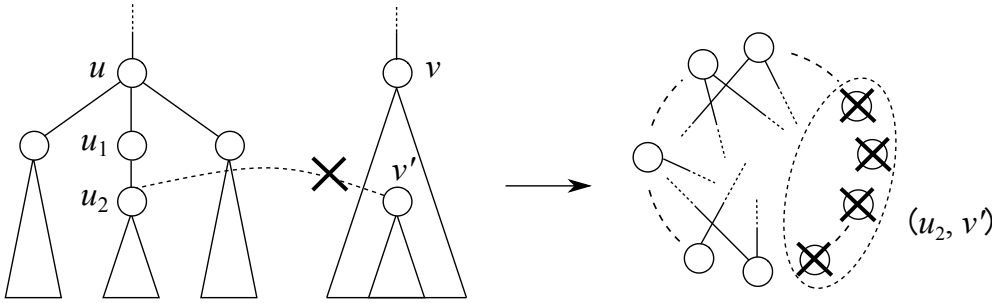


Figure 3.6: Heuristic technique (2).

Proof. Consider a mapping between $T_1(u)$ and $T_2(v)$. A weighted graph $G_{(u,v)}$ is constructed from $T_1(u)$ and $T_2(v)$. Besides, let $G_{c_2} = (V_{c_2}, E_{c_2})$ be a clique including $(u_2, v') \in V_{(u,v)}(G_{(u,v)})$. Here, because any mapping between $T_1(u_2)$ and $T_2(v)$ can be included in some mapping between $T_1(u_1)$ and $T_2(v)$, there also exists a clique $G_{c_1} = (V_{c_1}, E_{c_1})$ including $(u_1, v') \in V_{(u,v)}(G_{(u,v)})$, where u_1 is the parent of u_2, and $V_{c_2}(G_{c_2}) \setminus \{(u_2, v')\} = V_{c_1}(G_{c_1}) \setminus \{(u_1, v')\}$. Furthermore, $W[u_1, v'] \geq W[u_2, v']$. Let $w(G_{c_1})$ and $w(G_{c_2})$ be the weights of the G_{c_1} and G_{c_2}, respectively. Then, the following inequality holds:

$$\begin{aligned}
w(G_{c_1}) &= \sum_{(x,y)\in V_{c_1}} W[x,y] \\
&= \sum_{(x,y)\in V_{c_1}\setminus\{(u_1,v')\}} W[x,y] + W[u_1,v'] \\
&\geq \sum_{(x,y)\in V_{c_2}\setminus\{(u_2,v')\}} W[x,y] + W[u_2,v'] \\
&= \sum_{(x,y)\in V_{c_2}} W[x,y] \\
&= w(G_{c_2}).
\end{aligned}$$

Hence, the score of G_{c_2} is smaller than or equal to that of G_{c_1}. Therefore, we need not generate a vertex (u_2, v') for any v' in the construction of $G_{(u,v)}$. □

Although DpCliqueEdit with heuristic techniques (1) and (2) is much faster than CliqueEdit [6] in most cases, it takes very long CPU time in some cases, especially if there exist many leaves. In such a case, constructed graphs would contain many vertices and edges and thus a clique algorithm does not work efficiently. In order to cope with such difficult cases, we introduce other heuristic techniques as follows. The efficiency of MWCQ is much affected by the number of edges in $G_{(u,v)}$. Due to

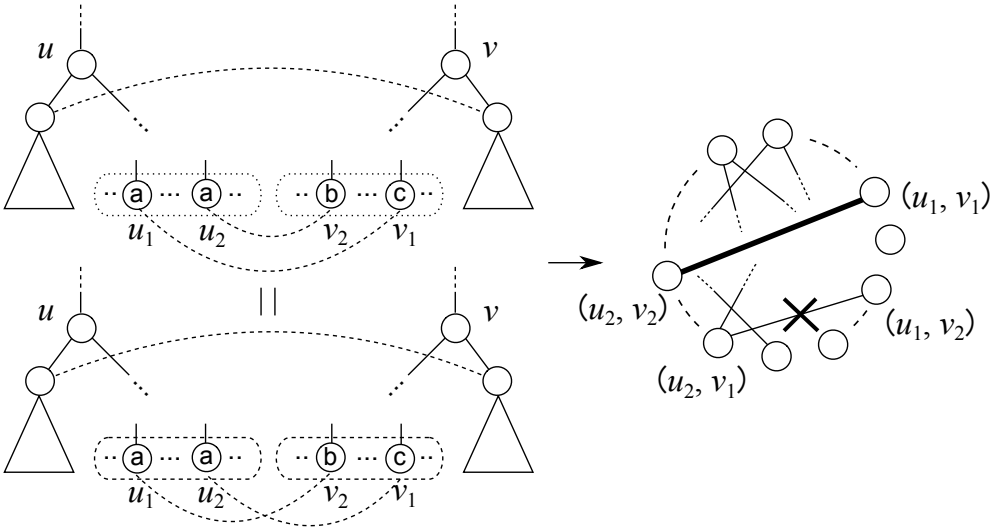


Figure 3.7: Example of heuristic technique (3). u_1, u_2, v_1, and v_2 are leaves, and $\ell(u_1) = \ell(u_2)$. In this case, we need not create $\{(u_1, v_2), (u_2, v_1)\}$.

the definition of $E_{(u,v)}$ described in Section 3.2.5, if u_1, v_1, u_2, v_2 are leaves, there is always an edge between (u_1, v_1) and (u_2, v_2). Therefore, if there are many leaves in T_1 and T_2, $G_{(u,v)}$ has many edges and then MWCQ takes much longer computation time. Since more than a half of nodes are leaves even in binary trees, some heuristic techniques handling leaves are necessary for further speed up.

Heuristic technique (3) u_1, u_2, v_1, and v_2 ($u_1 \neq u_2$, $v_1 \neq v_2$) are leaves, and $\ell(u_1) = \ell(u_2)$ or $\ell(v_1) = \ell(v_2)$.

In this case, we need not create an edge $\{(u_1, v_2), (u_2, v_1)\}$ for any u_1, u_2, v_1, and v_2 in the construction of $G_{(u,v)}$ because the score of a mapping including two pairs (u_1, v_2), (u_2, v_1) is equal to that of a mapping including two pairs (u_1, v_1), (u_2, v_2). Therefore, we have only to create an edge $\{(u_1, v_1), (u_2, v_2)\}$ without creating an edge $\{(u_1, v_2), (u_2, v_1)\}$ (see also Figure 3.7).

Proposition 3.2.4. *Suppose that $u_1, u_2 \in T_1$ and $v_1, v_2 \in T_2$. If u_1, u_2, v_1, and v_2 ($u_1 \neq u_2$, $v_1 \neq v_2$) are leaves, and $\ell(u_1) = \ell(u_2)$ or $\ell(v_1) = \ell(v_2)$, $W[u, v]$ can be computed without creating an edge $\{(u_1, v_2), (u_2, v_1)\}$ for any u_1, u_2, v_1, and v_2 in the construction of $G_{(u,v)}$.*

Proof. If there exists a mapping M which includes two pairs (u_1, v_1), (u_2, v_2) between T_1 and T_2, it is implied that there also exists a mapping M' which includes

two pairs (u_1, v_2), (u_2, v_1) instead of (u_1, v_1), (u_2, v_2) between T_1 and T_2. Now, the following equality holds:

$$\begin{aligned} score(M) &= \sum_{(x,y)\in M} W[x,y] \\ &= \sum_{(x,y)\in M\setminus\{(u_1,v_1),(u_2,v_2)\}} W[x,y] + f(u_1,v_1) + f(u_2,v_2) \\ &= \sum_{(x,y)\in M\setminus\{(u_2,v_1),(u_1,v_2)\}} W[x,y] + f(u_2,v_1) + f(u_1,v_2) \\ &= \sum_{(x,y)\in M'} W[x,y] \\ &= score(M') \end{aligned}$$

Hence, the score of a mapping including two pairs (u_1, v_2), (u_2, v_1) is equal to that of a mapping including two pairs (u_1, v_1), (u_2, v_2). Therefore, we have only to create an edge $\{(u_1, v_1), (u_2, v_2)\}$ without creating an edge $\{(u_1, v_2), (u_2, v_1)\}$. □

Heuristic technique (4) u_1, u_2, v_1, and v_2 ($u_1 \neq u_2,\ v_1 \neq v_2$) are leaves, and all labels of them are different.

In this case, we need not create an edge $\{(u_1, v_2), (u_2, v_1)\}$ for any u_1, u_2, v_1, and v_2 in the construction of $G_{(u,v)}$ for the same reason as in the case (3).

Proposition 3.2.5. *Suppose that $u_1, u_2 \in T_1$ and $v_1, v_2 \in T_2$. If u_1, u_2, v_1, and v_2 ($u_1 \neq u_2,\ v_1 \neq v_2$) are leaves, and $\ell(u_1), \ell(u_2), \ell(v_1)$, and $\ell(v_2)$ are different, $W[u, v]$ can be computed without creating an edge $\{(u_1, v_2), (u_2, v_1)\}$ for any u_1, u_2, v_1, and v_2 in the construction of $G_{(u,v)}$.*

Proposition 3.2.5 can be proved in the same way as the proof of the Proposition 3.2.4, so we omit the proof of Proposition 3.2.5.

The idea of (3) focusing on the same labeled leaves is extended to the isomorphic subtrees. If T_1 and T_2 are isomorphic including label information, we write $T_1 \approx T_2$.

Heuristic technique (5) $T_1(u_1) \approx T_1(u_2)$ ($u_1 \neq u_2$) or $T_2(v_1) \approx T_2(v_2)$ ($v_1 \neq v_2$).

In this case, we need not create an edge $\{(u_1, v_2), (u_2, v_1)\}$ for any u_1, u_2, v_1, and v_2 in the construction of $G_{(u,v)}$ because of the following reason. When $T_1(u_1) \approx T_1(u_2)$ or $T_2(v_1) \approx T_2(v_2)$, the score of mapping $\{(T_1(u_1), T_2(v_2)), (T_1(u_2), T_2(v_1))\}$ is equal to that of mapping $\{(T_1(u_1), T_2(v_1)),(T_1(u_2), T_2(v_2))\}$. Therefore, we have only to create an edge $\{(u_1, v_1), (u_2, v_2)\}$ without creating an edge $\{(u_1, v_2), (u_2, v_1)\}$ (see also Figure 3.8).

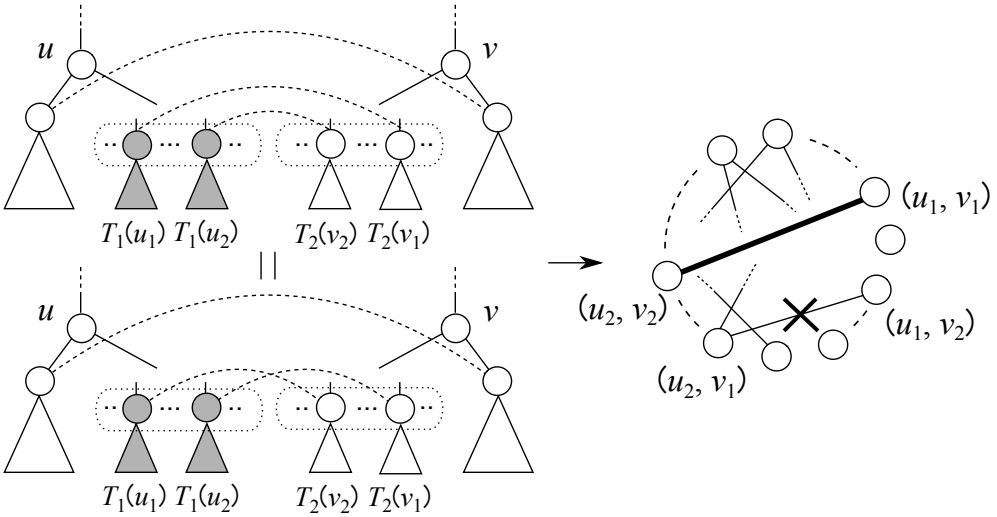


Figure 3.8: Example of heuristic technique (5). $T_1(u_1)$ and $T_1(u_2)$ are isomorphic including label information. In this case, we need not create $\{(u_1, v_2), (u_2, v_1)\}$.

Proposition 3.2.6. *Suppose that $u_1, u_2 \in T_1$ and $v_1, v_2 \in T_2$. If $T_1(u_1) \approx T_1(u_2)$ $(u_1 \neq u_2)$ or $T_2(v_1) \approx T_2(v_2)$ $(v_1 \neq v_2)$, $W[u, v]$ can be computed without creating an edge $\{(u_1, v_2), (u_2, v_1)\}$ for any u_1, u_2, v_1, and v_2 in the construction of $G_{(u,v)}$.*

Proof. If there exists a mapping M which includes two pairs (u_1, v_1), (u_2, v_2) between T_1 and T_2, it is implied that there also exists a mapping M' which includes two pairs (u_1, v_2), (u_2, v_1) instead of (u_1, v_1), (u_2, v_2) between T_1 and T_2. Moreover, from the assumption, the score of mapping $\{(T_1(u_1), T_2(v_2)), (T_1(u_2), T_2(v_1))\}$ is equal to that of mapping $\{(T_1(u_1), T_2(v_1)), (T_1(u_2), T_2(v_2))\}$, that is, $W[u_1, v_1] = W[u_2, v_1]$ and $W[u_2, v_2] = W[u_1, v_2]$. Now, the following equality holds,

$$
\begin{aligned}
score(M) &= \sum_{(x,y)\in M} W[x, y] \\
&= \sum_{(x,y)\in M\setminus\{(u_1,v_1),(u_2,v_2)\}} W[x, y] + W[u_1, v_1] + W[u_2, v_2] \\
&= \sum_{(x,y)\in M\setminus\{(u_2,v_1),(u_1,v_2)\}} W[x, y] + W[u_2, v_1] + W[u_1, v_2] \\
&= \sum_{(x,y)\in M'} W[x, y] \\
&= score(M')
\end{aligned}
$$

Thus, the score of a mapping including two pairs (u_1, v_2), (u_2, v_1) is equal to that of a mapping including two pairs (u_1, v_1), (u_2, v_2). Therefore, we have only to create an edge $\{(u_1, v_1), (u_2, v_2)\}$ without creating an edge $\{(u_1, v_2), (u_2, v_1)\}$. □

It is expensive to determine whether two graphs are isomorphic or not but we can solve the problem easier when the two graphs are trees. Though various algorithms are invented for the problem, we employ an algorithm introduced in [38]. The algorithm transforms the tree isomorphism problem into the comparison of simple numerical sequences.

From the propositions 3.2.2 $\sim$ 3.2.6, we have the following theorem.

Theorem 3.2.7. *DpCliqueEdit with the heuristic* (1) $\sim$ (5) *computes dist*(T_1, T_2) *without violating the optimality of the solution.*

It should be noted that we can use the heuristic techniques only if DP is introduced to the clique-based approach.

3.3 Experimental Results

In order to evaluate the efficiency of the improved method and heuristic techniques, we applied CliqueEdit, UwCliqueEdit, and DpCliqueEdit to comparison of real tree-structured data. As the tree-structured data, we employed CSLOGS dataset which consists of Weblogs files [57] and glycan structures that were obtained from KEGG/Glycan database [28].

It is to be noted that, as far as we know, there exists no other publicly available program for exactly computing the unordered tree edit distance and thus we only compared these methods. From the result given in [24], it is considered that CliqueEdit has similar efficiency [19] to the A^*-algorithm for unordered tree edit distance [24].

We implemented CliqueEdit, UwCliqueEdit, and DpCliqueEdit using C++ language and compared UwCliqueEdit and DpCliqueEdit with the previous method CliqueEdit. In the implementations of CliqueEdit and DpCliqueEdit, MWCQ [42] was used as a maximum vertex weighted clique algorithm, while in that of UwCliqueEdit, MCS [52] was used as a maximum clique algorithm. **DpCliqueEdit-A**, **DpCliqueEdit-B**, **DpCliqueEdit-C**, **DpCliqueEdit-D**, and **DpCliqueEdit-E** represent DpCliqueEdit without heuristics, with heuristics (1)(2), with heuristics (1)(2)(3), with heuristics (1)(2)(3)(5), and with all heuristics, respectively. The preliminary version of DpCliqueEdit in [6] is equivalent to DpCliqueEdit-B. We

Table 3.1: CPU time for comparing Weblogs data obtained from SUBLOGS3 dataset. Average CPU time (sec.) per Weblogs pair is shown for each case. The maximum number of children of each node is limited to smaller than or equal to 3. "-" denotes that there exist at least one hard case for which the program could not output a solution within 60 minutes (= 3600 seconds). Bold face indicates the best results for each case.

total # nodes	Clique Edit	UwClique Edit	DpClique Edit-A	DpClique Edit-B	DpClique Edit-C	DpClique Edit-D	DpClique Edit-E
$30 \sim 34$	**0.003**	**0.003**	0.010	0.007	0.007	0.007	0.008
$35 \sim 39$	**0.009**	**0.009**	0.024	0.018	0.017	0.018	0.018
$40 \sim 44$	0.061	**0.023**	0.068	0.038	0.032	0.031	0.033
$45 \sim 49$	0.218	0.100	0.155	0.059	0.053	0.051	**0.050**
$50 \sim 54$	2.928	0.129	0.370	0.222	0.119	0.117	**0.095**
$55 \sim 59$	2.189	0.809	2.965	0.210	0.173	0.182	**0.167**
$60 \sim 64$	-	39.940	-	20.450	0.542	1.904	**0.297**
$65 \sim 69$	-	17.380	-	-	2.230	1.106	**0.662**
$70 \sim 74$	-	-	-	-	3.589	1.423	**1.024**
$75 \sim 79$	-	-	-	-	-	1.895	**1.566**
$80 \sim 84$	-	-	-	-	-	-	**2.625**
$85 \sim 89$	-	-	-	-	-	46.920	**10.550**
$90 \sim 94$	-	-	-	-	-	-	**50.570**
$95 \sim 99$	-	-	-	-	-	-	**64.980**

performed computational experiments using a PC with 2.66 GHz Intel Core i7 CPU and 3.88 GB RAM running under the Mac OS X operating system. In this chapter, we focus only on the computational efficiency and do not conduct computational experiments for evaluating the performance (i.e., accuracy of comparison) of CliqueEdit, UwCliqueEdit, and DpCliqueEdit because these methods compute the same distances, the performance of CliqueEdit was already evaluated in the previous work [19], and the tree edit distance is the most established distance measure for trees [14].

For evaluation of the methods, we used the standard weighting scheme (i.e., $f(u, v) = 2$ for $\ell(u) = \ell(v)$, $f(u, v) = 1$ for $\ell(u) \neq \ell(v)$) corresponding to the unit cost edit distance.

3.3.1 CSLOGS Dataset

In this experiment, we use Weblogs data with a specified range of the total number of nodes (i.e., the sum of the numbers of nodes in T_1 and T_2) and measured the average CPU time (user time) per pair. We randomly selected trees from CSLOGS

Table 3.2: CPU time for comparing Weblogs data obtained from SUBLOGS5 dataset. Average CPU time (sec.) per Weblogs pair is shown for each case. The maximum number of children of each node is limited to smaller than or equal to 5. "-" denotes that there exist at least one hard case for which the program could not output a solution within 60 minutes (= 3600 seconds). Bold face indicates the best results for each case.

total # nodes	Clique Edit	UwClique Edit	DpClique Edit-A	DpClique Edit-B	DpClique Edit-C	DpClique Edit-D	DpClique Edit-E
$30 \sim 34$	0.010	**0.005**	0.029	0.011	0.010	0.011	0.008
$35 \sim 39$	0.244	0.034	0.357	0.114	0.027	0.025	**0.016**
$40 \sim 44$	44.090	4.067	41.000	4.630	2.890	1.913	**0.028**
$45 \sim 49$	35.380	7.310	19.140	3.075	1.079	0.994	**0.101**
$50 \sim 54$	-	-	-	-	-	-	**0.141**
$55 \sim 59$	-	53.750	-	-	12.580	12.260	**0.423**
$60 \sim 64$	-	-	-	-	-	-	**17.240**
$65 \sim 69$	-	-	-	-	-	-	**10.850**

and created two sub-datasets called SUBLOGS3 and SUBLOGS5 in this chapter. Each sub-dataset has 15000 trees whose sizes are restricted to smaller than or equal to 80, where the maximum number of children of each node is limited to smaller than or equal to 3 and 5 in SUBLOGS3 and SUBLOGS5, respectively. The percentage of the number of trees in which the maximum number of children of each node are restricted to 3 and 5 is about 65% and 81% of the total number of them in CSLOGS. Unbalanced cases in which the size of one structure was smaller than 1/3 of the other structure were excluded.

The results of the computational experiments we performed with SUBLOGS3 and SUBLOGS5 are shown in Table 3.1 and Table 3.2, respectively. From Table 3.1, it is seen that UwCliqueEdit is the fastest for small trees. However, as the total number of nodes of input trees becomes larger, it takes longer CPU time for UwCliqueEdit to solve the problem, and there exist hard cases for which UwCliqueEdit could not output a solution within 60 minutes. For non-small trees, although most methods could not solve the problem in 60 minutes in some cases, DpCliqueEdit-E could in 60 minutes for all cases we selected in this experiment. Similarly, from Table 3.2, we find that UwCliqueEdit is faster than any other method for small trees and DpCliqueEdit-E is the fastest for non-small trees.

Table 3.3: CPU time for comparing glycans. Average CPU time (sec.) per glycan pair is shown for each case. Bold face indicates the best results for each case.

total # nodes	Clique Edit	UwClique Edit	DpClique Edit-A	DpClique Edit-B	DpClique Edit-C	DpClique Edit-D	DpClique Edit-E
$30 \sim 34$	**0.002**	0.003	0.013	0.006	0.006	0.006	0.009
$35 \sim 39$	**0.004**	0.007	0.027	0.011	0.011	0.012	0.017
$40 \sim 44$	0.056	0.035	0.107	0.026	**0.019**	0.021	0.029
$45 \sim 49$	0.064	0.036	0.126	0.031	**0.030**	0.031	0.040
$50 \sim 54$	0.078	0.049	0.228	0.039	**0.037**	0.039	0.051
$55 \sim 59$	1.987	0.433	8.968	0.108	0.088	**0.086**	0.096
$60 \sim 64$	2.746	4.949	1.780	0.167	0.163	**0.149**	0.177
$65 \sim 69$	64.290	9.303	39.460	0.381	0.364	**0.328**	0.357
$70 \sim 74$	58.690	0.099	1.337	0.545	**0.436**	0.463	0.501
$75 \sim 79$	2.441	0.918	4.051	0.953	**0.752**	0.754	0.781
$80 \sim 84$	7.150	6.570	44.630	2.516	2.268	**1.620**	1.653
$85 \sim 89$	237.700	28.030	21.110	3.205	3.205	**2.413**	2.490
$90 \sim 94$	303.200	1211.000	1710.000	38.810	26.300	**8.165**	9.475

3.3.2 Glycan Structures

As in the previous work [19], we randomly selected 100 pairs of glycan structures with a specified range of the total number of nodes (i.e., the sum of the numbers of nodes in T_1 and T_2) and measured the average CPU time (user time) per pair. Unbalanced cases mentioned in Section 3.3.1 were excluded. For each of the ranges in $60 \sim 79$, we took the average over 20 pairs because there did not exist an enough number of pairs, where we could use 19 pairs among 20 pairs for the range of $70 \sim 74$ because there was a hard case for which DpCliqueEdit-A could not output a solution within 60 minutes. For the ranges of $80 \sim 84$, $85 \sim 89$, and $90 \sim 94$, only 9, 5, and 4 pairs were available, respectively. We could use only 4 pairs among 5 pairs for the range of $85 \sim 89$ and 2 pairs among 4 pairs for the range of $90 \sim 94$ because there were hard cases for which DpCliqueEdit-A could not output a solution within 60 minutes.

The result of the computational experiment is shown in Table 3.3. From this table, it is seen that from DpCliqueEdit-B to DpCliqueEdit-E are much faster than CliqueEdit, UwCliqueEdit, and DpCliqueEdit-A for non-small glycan structures. In particular, DpCliqueEdit-D is the fastest for comparison of large glycan structures. Although UwCliqueEdit is faster than CliqueEdit in most cases, it is not fast for comparison of large glycan structures because it constructs larger and denser graph $\hat{G}$ as G becomes larger and thus MCS does not work efficiently. Besides, DpCliqueEdit-A is not fast despite the fact that DpCliqueEdit-B $\sim$ DpCliqueEdit-

Table 3.4: CPU time for comparing glycans for each hard case. Average CPU time (sec.) per glycan pair is shown for each case. CPU time (sec.) per glycan pair is shown for each hard case. "-" denotes that the program could not output a solution within 60 minutes (= 3600 seconds). Bold face indicates the best results for each case.

glycan pair	total # nodes	Clique Edit	UwClique Edit	DpClique Edit-A	DpClique Edit-B	DpClique Edit-C	DpClique Edit-D	DpClique Edit-E
{G04520, G04682}	35	693.400	-	223.900	225.800	**0.020**	**0.020**	**0.020**
{G04520, G05248}	36	1124.000	-	284.500	285.900	**0.020**	**0.020**	**0.020**
{G03769, G04682}	71	-	491.400	-	-	10.910	**0.490**	0.520
{G03769, G04520}	72	-	59.080	-	-	11.800	**0.420**	0.450
{G03769, G05248}	72	-	17.380	-	-	56.500	**0.600**	0.630
{G03769, G05297}	72	-	17.590	-	-	56.560	**0.600**	0.630
{G03655, G03769}	88	108.600	277.400	-	300.700	31.170	**5.610**	6.430
{G03769, G04206}	91	844.100	1397.000	-	5.870	5.250	5.830	**5.120**
{G03769, G11847}	91	132.100	911.500	-	108.400	82.880	28.120	**22.200**

E are faster than the other methods. This is because DpCliqueEdit repeatedly solves instances of the maximum vertex weighted clique problem as sub-problems, so that it takes long CPU time if the heuristic techniques are not introduced. Since the heuristic techniques proposed in this chapter cannot be used without using DP, DP needs to be introduced in order to reduce the computation time. Although CliqueEdit and UwCliqueEdit are faster than DpCliqueEdit-B $\sim$ DpCliqueEdit-E for small glycan structures, comparison of large glycan structures is more crucial because it takes a large amount of time.

Table 3.4 shows the results on pairs of trees (i.e., hard cases) for which some of the examined methods could not compute the distance within 60 minutes. From this table, though there is no great difference between DpCliqueEdit-B $\sim$ DpCliqueEdit-E except for the range of 90 $\sim$ 94 in Table 3.3, we find that DpCliqueEdit-D and DpCliqueEdit-E are much faster than the other methods in hard cases. This implies that DpCliqueEdit-D and DpCliqueEdit-E utilize the existence of the same labeled leaves, different labeled leaves, and isomorphic subtrees, and thus need much shorter time for MWCQ. It takes long CPU time for DpCliqueEdit-A to output a solution in most cases. This is because it costs much CPU time to construct $G_{(u,v)}$ and solve the maximum vertex weighted clique problem repeatedly. It is also seen that there exist some instances which UwCliqueEdit can solve within 60 minutes whereas CliqueEdit or DpCliqueEdit-B cannot solve within 60 minutes. However, UwCliqueEdit is not faster than CliqueEdit and DpCliqueEdit-B for comparison of large glycan structures.

Although DpCliqueEdit-E is the most useful for comparison of Weblogs data, there is no great difference between DpCliqueEdit-D and DpCliqueEdit-E for com-

parison of glycan structures. In CSLOGS, there are 13,361 unique Web page [57] and each Web page is assigned to each node as a label, so that there exist many leaves with different labels. Therefore, heuristic (4) works efficiently.

From the results of these computational experiments, we can conclude that DpCliqueEdit-E is the most useful of the proposed clique-based methods.

3.4 Discussions

In this chapter, we proposed an improved clique-based method by introducing DP and several heuristic techniques for computing the tree edit distance between rooted unordered trees. DP and the heuristic techniques are very useful and then the improved method is much faster than the previous method in most cases of comparison of real tree-structured data. In particular, for hard instances of comparison of glycan structures, the improved method achieved more than 100 times speed-up. Although the improved method is not faster for comparison of small glycans, it is not crucial because comparison of large glycan structures takes much longer CPU time than that of small glycans.

Although the improved method is much faster than the previous method, there still exist cases for which it takes long CPU time. In particular, it takes long CPU time if there exist long subtrees (i.e., there exist many nodes but few leaves) because the heuristics (1)(2) proposed in this chapter can reduce the computation time only if there exist nodes with one child in the long subtrees, and the heuristics (3)(4)(5) cannot well contribute to reduction of the number of edges in such cases and thus the maximum vertex weighted clique algorithm does not work efficiently. How to cope with such cases is left for future work.

Moreover, in order to achieve further speed-up, we should develop an improved algorithm for the maximum vertex weighted clique problem because an improvement of the efficiency of clique finding directly leads to an improvement of the efficiency of our proposed algorithm. In particular, a maximum vertex weighted clique solver specialized for properties of weighted graphs generated by the clique-based algorithm might be useful for the tree edit distance problem. How to develop such an algorithm is also left as future work.

In addition to the future work mentioned above, some modifications are needed for application to analysis of tree-structured data used in computational biology. Although we have used the unit cost edit distance in computational experiments, more suitable cost functions should be used for analysis of biological and other objects. Development of cost functions suitable to individual applications is also left for future work.

As for the application to glycan data, we focused only on the computational efficiency of the proposed methods in the computational experiments since the use-

fulness of tree edit distance for similar glycan structure search has already been shown in [19]. On the other hand, although it is not easy to relate the structures of glycans to their functions as mentioned in Chapter 1, recent studies indicate that there exist relationships between the structures and their functions [36]. Fortunately, the improved clique-based method enables us to compute the tree edit distance among all glycans in KEGG database, which is a big step for analyzing glycans data. Furthermore, we can obtain the mapping between input glycans using a standard traceback technique. Thus, for example, the results might be useful for analyzing the changes in glycan structures that are caused by knocking out the genes encoding glycosyltransferases.

Chapter 4

Similar Subtree Search Using Extended Tree Inclusion

This chapter 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. As mentioned in Chapter 3, 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.

4.1 Background

As mentioned in Chapter 1, the comparison of tree-structured data has numerous applications in various scientific areas and data processing. Therefore, comparison of the tree-structured data is important and the tree edit distance has been well-studied as one of the most widely used measures for it. However, the tree edit distance problem is NP-hard for unordered trees.

To cope with this hardness, several approaches have been taken. The first one is the use of branch-and-bound techniques [24, 41]. 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 [12]; 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 [8, 14, 59] (see also [27] 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 [60]. An example of a method that combines approaches three and four is the *alignment of trees distance* introduced by Jiang et al. [25], 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 [31] 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 4.3. 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 chapter 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.

4.2 Unordered Tree Inclusion

Here we review the unordered tree inclusion problem and the algorithm of [31]. We use the notation in Section 2.1. 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 [31] 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 4.1 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 Figure 4.1. 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 $S(v_2) = \{\ \emptyset,\ \{u_4\},\ \{u_5\},\ \{u_4, u_5\},\ \{u_2\},\ \{u_3\},\ \{u_2, u_3\},\ \{u_1\}\ \}$.

4.3 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 Figure 4.2. 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

Algorithm 4.1 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**
      $S\Delta \leftarrow S\Delta \cup \{\{u\}\}$;
  $S(v) \leftarrow S \cup S\Delta$.

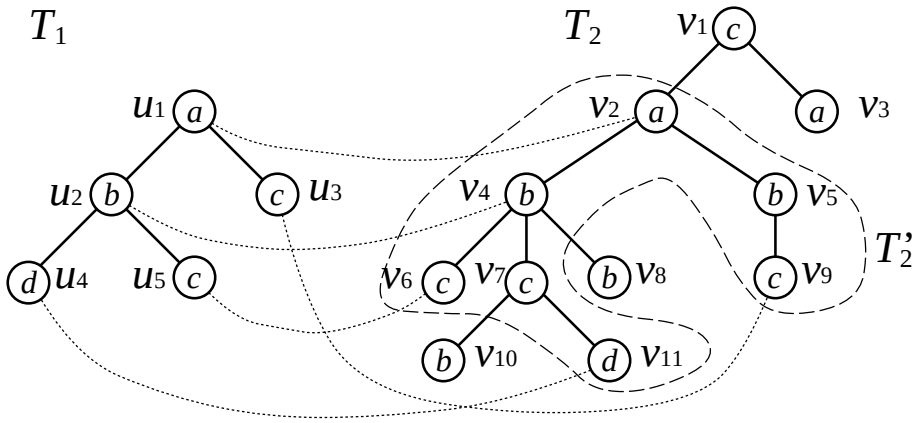


Figure 4.1: An example of unordered tree inclusion.

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, 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 Figure 4.1, 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

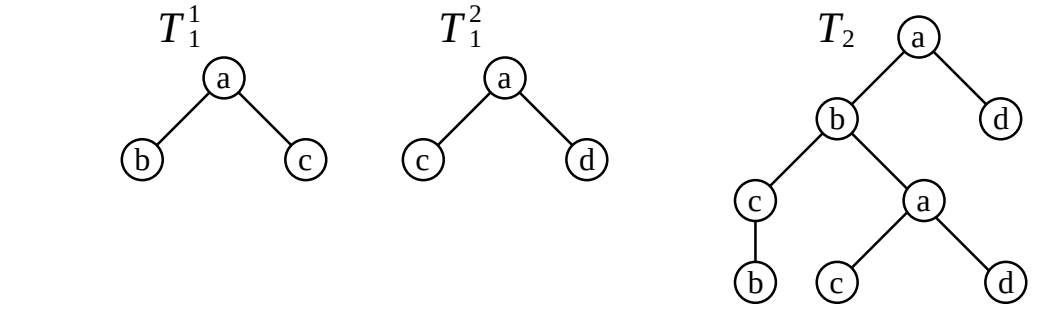


Figure 4.2: Both T_1^1 and T_1^2 are included in T_2. However, T_1^2 looks like a more similar subtree since it is isomorphic to a rooted subtree of T_2.

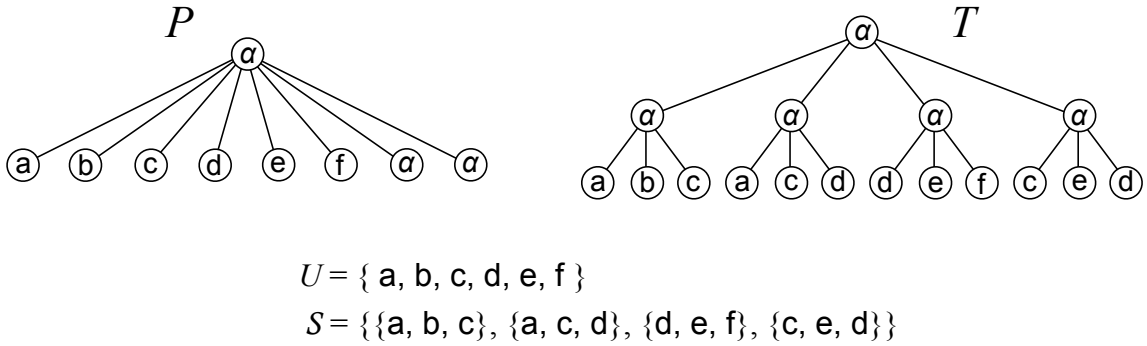


Figure 4.3: Example of reduction from Exact 3-Set Cover. P and T are corresponding pattern and text trees, respectively.

is *embedded* into T_2 (with cost $D_{\prec}(T_1, T_2)$). It should be noted that the problem **MinCostIncl** is NP-hard. The hardness is proved using a reduction from Exact 3-Set Cover (X3C) as follows (see also Figure 4.3):

Proposition 4.3.1. *Let P and T be pattern and text trees, respectively. MinCostIncl remains NP-hard even if $h(T) = 2$ and $h(P) = 1$, where $h(T')$ denotes the height of tree T'.*

Proof. Let $\mathcal{S} = \{S_1, \ldots, S_m\}$ over U ($n = |U|$) be an instance of X3C ($|S_i| = 3$ for all i), which asks whether there exists $\mathcal{S}' \subseteq \mathcal{S}$ such that $|\mathcal{S}'| = n/3$ and $\bigcup_{S_i \in \mathcal{S}'} S_i = U$, where we can assume without loss of generality that $n \bmod 3 = 0$.

Let T_i be a tree of height 1 with 3 leaves where the set of labels of the leaves is the same as S_i. Then, T is constructed by letting $r(T_i)$s be the children of $r(T)$. We assign the same label (say, α) to all internal nodes.

P has $n + m - (n/3)$ leaves. The set of labels of n leaves is the same as U. The other nodes have label α.

Suppose that $D_{\prec}(P, T) \leq n/3$ holds under the unit-cost edit operation model. Let $L(T')$ be the leaves of a tree T'. In this case, P is included in T and there exists a set of T_j such that $\bigcup_{1 \leq j \leq n/3} L(T_j) = U$. Here, $L(T_j) = S_j$, so that $\bigcup_{1 \leq j \leq n/3} S_j = U$ holds. This means that there exists $|\mathcal{S}'| \subseteq \mathcal{S}$ such that $|\mathcal{S}'| = n/3$ and $\bigcup_{S_j \in \mathcal{S}'} S_j = U$. Therefore, if $D_{\prec}(P, T) \leq n/3$ holds, then X3C has a solution. Conversely, we assume that X3C has a solution. In this case, we can see from the construction of T_is that there exits a set of T_j such that $\bigcup_{1 \leq j \leq n/3} L(T_j) = U$. Then, we embed P into T by inserting $r(P_j)$s with label α as children of $r(P)$ such that $L(T(r(P_j))) = L(T_j)$. Here, the total number of inserted nodes is $n/3$. Thus, if X3C has a solution, $D_{\prec}(P, T) \leq n/3$ holds under the unit-cost edit operation model. Therefore, $D_{\prec}(P, T) \leq n/3$ holds under the unit-cost edit operation model if and only if X3C has a solution.

□

4.3.1 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 [31] 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 4.2. 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 $T_2(v) - \{v\}$.

Figure 4.4 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

Algorithm 4.2 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)
      $w_v(\{u\}) \leftarrow \min(\ w_v(\{u\}),\ w_{v_i}(\{u\}) + \delta(-, \ell(v))\ )$.

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.

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.

Theorem 4.3.2. *$D_{\prec}(T_1, T_2)$ 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 Figures 4.4 and 4.5). (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).

Algorithm 4.3 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**
  $W_v(R) \leftarrow U_v(R)$;
  $w_v(R) \leftarrow \min(\ w_v(R), W_v(R) + \delta(-, \ell(v))\ )$;

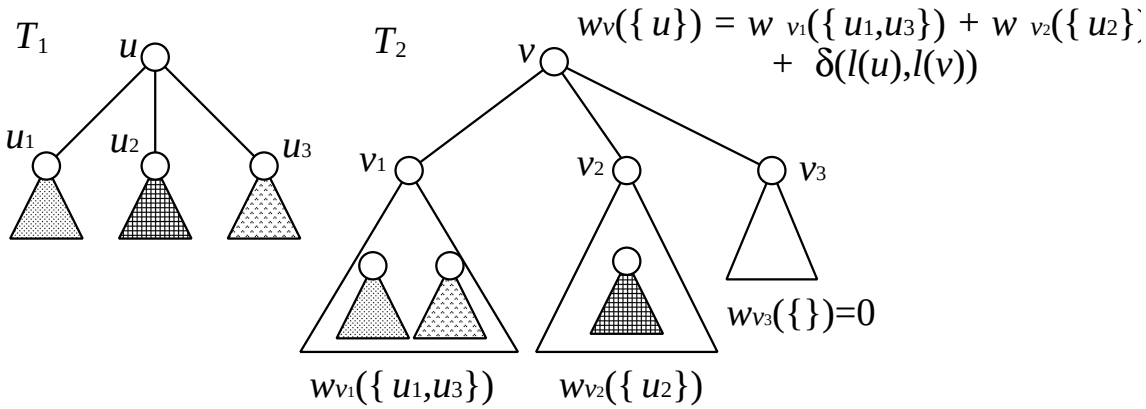


Figure 4.4: Illustrating the computation of minimum-cost tree inclusion. Here, u is mapped to v.

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 $O(2^D mn)$. □

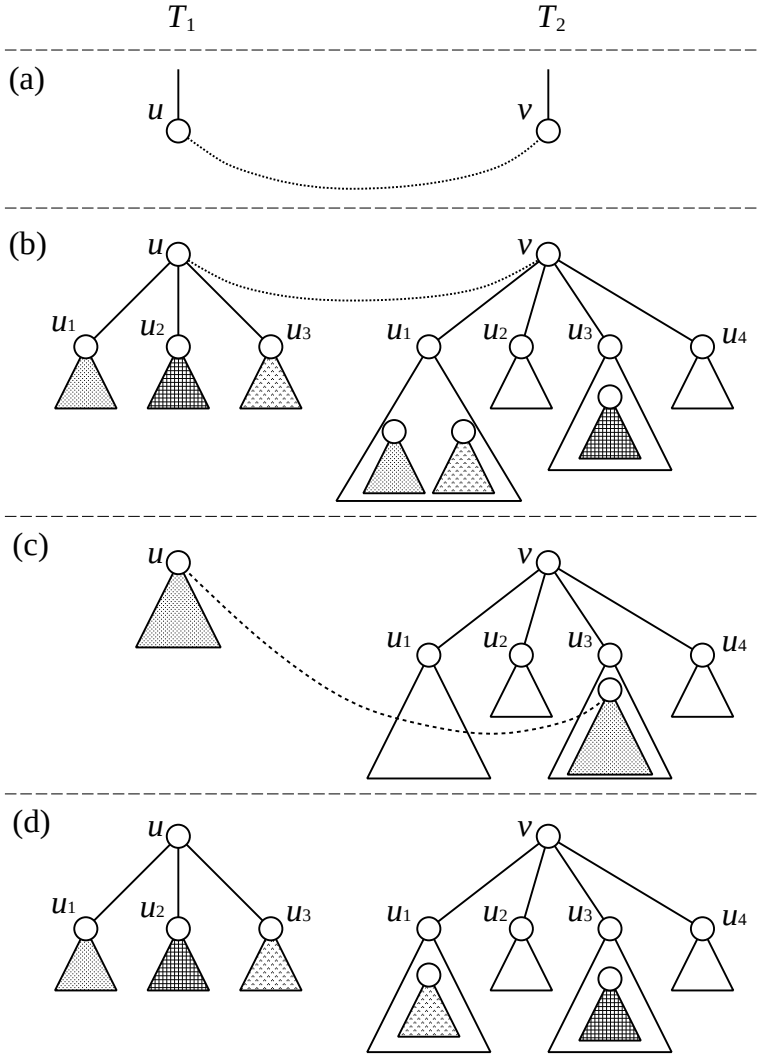


Figure 4.5: The four cases in the proof of Theorem 4.3.2.

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 [17] (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.

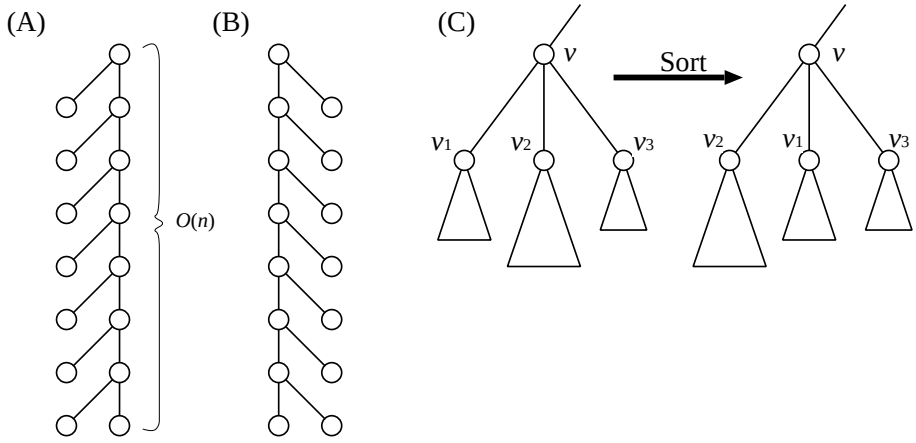


Figure 4.6: Improving the space complexity. (A) needs $O(n+2^D mn)$ space, whereas (B) needs $O(n+2^D m)$ space. To minimize the space, it is enough to sort the children as in (C).

4.3.2 Improvement of 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 $H = \max_{(x_0,\cdots,x_h)} |\{x_i \mid x_i \text{ is not the leftmost child}\}|$. For example, H is $O(n)$ for Figure 4.6(A), whereas H is $O(1)$ for Figure 4.6(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 Figure 4.6(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 4.3.3. *$D_{\prec}(T_1, T_2)$ 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.

4.4 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 [61]. However, if we limit the number K of nodes that may be deleted from the pattern 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 4.4.1. *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 ($e = 2.718...$).*

Proof. According to Lemma 2.1 (a) in [15], the number is at most $\frac{1}{K+1} \cdot \binom{(K+1)D}{K}$. By the comments immediately before Lemma 2.1 in [15], 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 4.4.1 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 Figure 4.7).

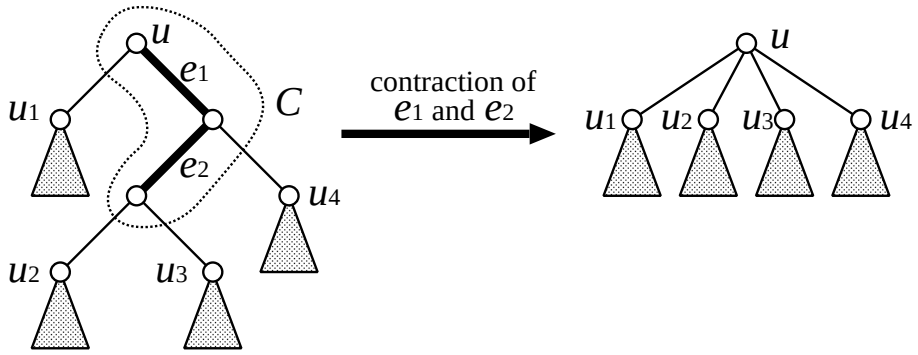


Figure 4.7: A contraction of a connected subtree. In this case, $chd(u, C) = \{u_1, u_2, u_3, u_4\}$.

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 4.4.

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 4.5. Although the meaning of $W_v^k(R)$ is slightly different from that in the above procedure, we obtain the same score.

Theorem 4.4.2. *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 Figure 4.7). 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

Algorithm 4.4 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**
          $w_v^k(\{u\}) \leftarrow \min(\ w_v^k(\{u\}),\ w_{v_i}^k(\{u\}) + \delta(-, \ell(v))\ )$.

Algorithm 4.5 Procedure MinCostIwdSub

```
for k = 0 to K do
  for all R ⊆ chd(u, C) do U_v^k(R) ← +∞;
U_v^0(∅) ← 0;
for i = 1 to |chd(v)| do
  for all R ⊆ chd(u, C) do
    for k = 0 to K do Ŵ_v^k(R) ← +∞;
  for all R^1 ⊆ chd(u, C) do
    for all R^2 ⊆ chd(u, C) \ R^1 do
      for k^1 = 0 to K do
        for k^2 = 0 to K − k^1 do
          k ← k^1 + k^2;
          Ŵ_v^k(R^1 ∪ R^2) ← min( Ŵ_v^k(R^1 ∪ R^2), U_v^{k^1}(R^1) + w_{v_i}^{k^2}(R^2) );
  for k = 0 to K do
    for all R ⊆ chd(u, C) do U_v^k(R) ← Ŵ_v^k(R);
for k = 0 to K do
  for all R ⊆ chd(u, C) do
    W_v^k(R) ← U_v^k(R);
    w_v^k(R) ← min( w_v^k(R), W_v^k(R) + δ(−, ℓ(v)) );
```

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 4.4.1 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 Figure 4.7. However, if we let $K = 2$ in Theorem 4.4.2, 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 4.4.1. *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.

4.5 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 [31], we used both synthetic data and real data. To obtain synthetic data, we modified the tree generation algorithm in [55] so that the resulting tree size and the maximum outdegree can be specified. As for real data, we used Weblogs data [57] and glycan data taken from the KEGG database [29]. 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.

All experiments were performed on a PC cluster with Intel(R) Xeon(R) CPU E5-2690 2.90GHz 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.

4.5.1 Processing Efficiency

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 Figure 4.8. We can observe from this figure that the 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,

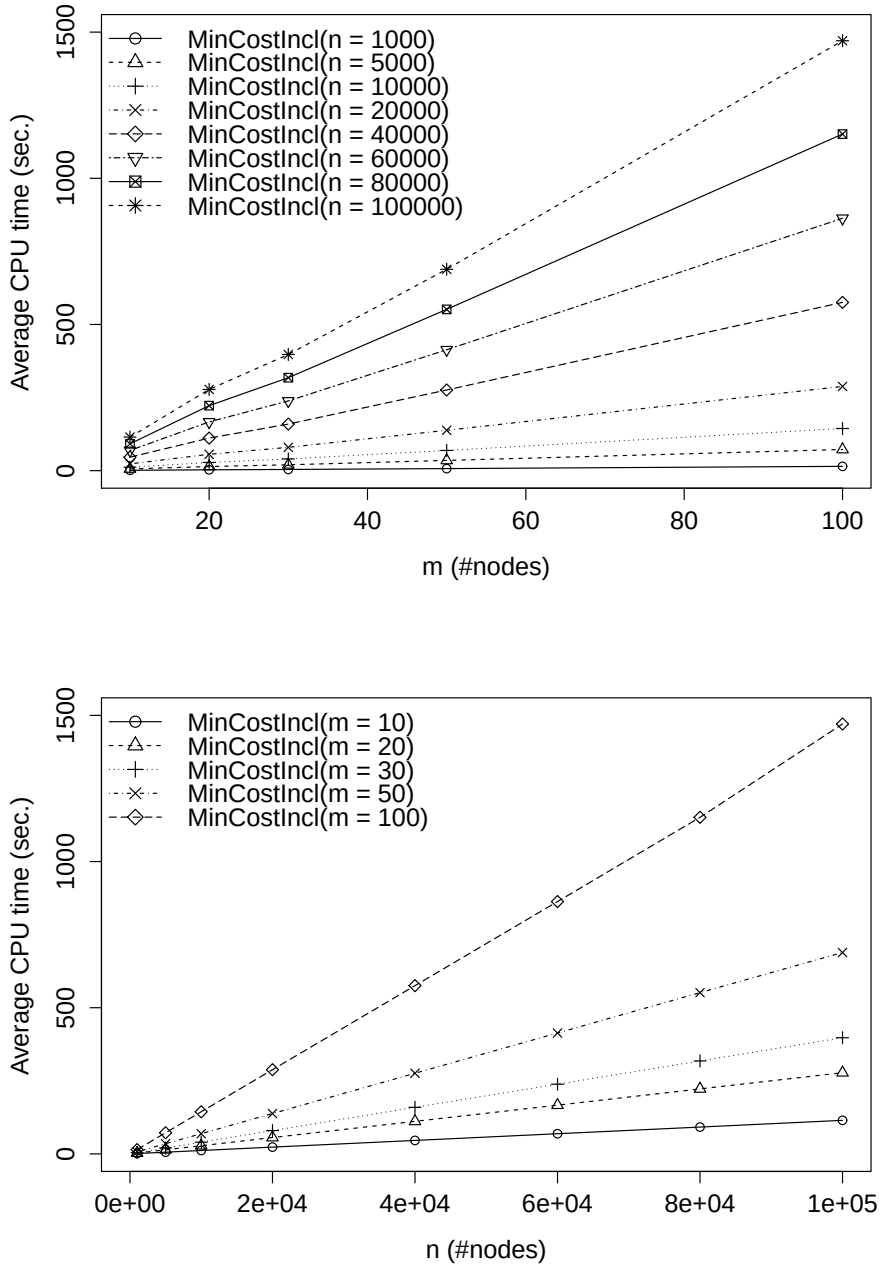


Figure 4.8: Execution times of MinCostIncl for varying sizes of pattern trees (top) and text trees (bottom). Note that $m \leq n$ must hold from the definition of MinCostIncl.

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 [41] 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 Figure 4.9. 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 Figure 4.9, whereas MinCostIncl works even for text tree whose size is 100,000 with linear increase of CPU time as shown in Figure 4.8. 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 chapter 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 $(m, n) = (100, 10000)$ and $(100, 20000)$, 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). Figure 4.10 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 [31] 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 4.1. 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 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 4.5.2.

Processing efficiency of MinCostInclWithDel was also examined by using syn-

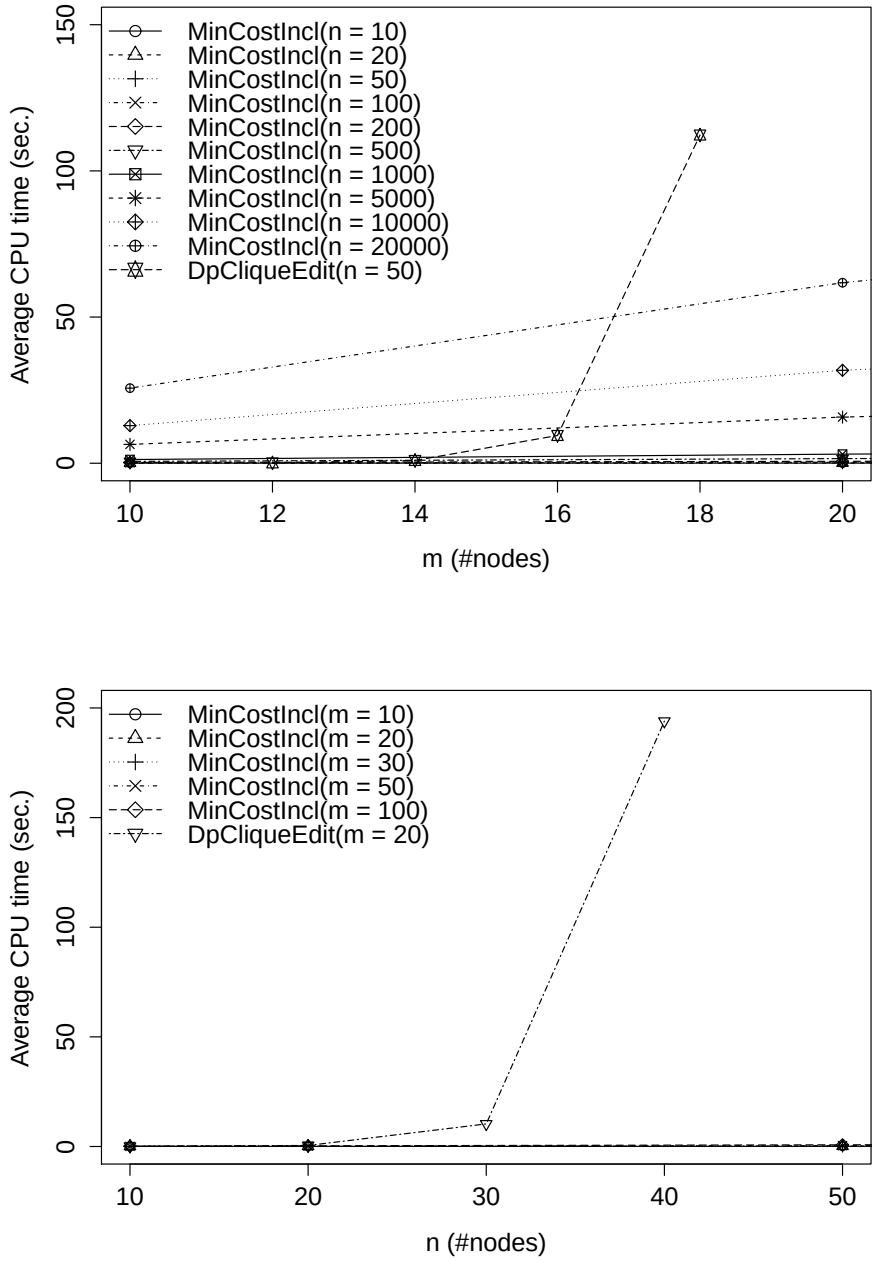


Figure 4.9: Comparison between MinCostIncl and DpCliqueEdit for varying sizes of pattern trees (top) and text trees (bottom).

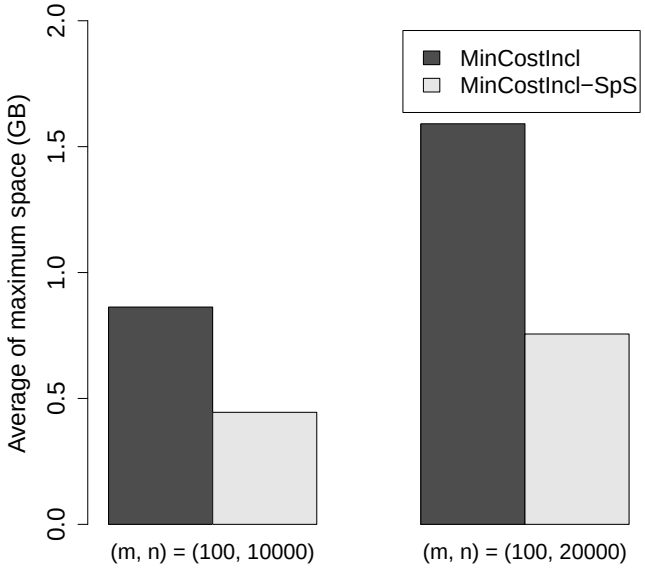


Figure 4.10: The reduction in memory usage for large text data.

Table 4.1: 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

thetic data. The results are shown in Figure 4.11, 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.

Results for Real Data

We evaluated the efficiency of MinCostIncl for another type of real data: Weblogs data [57]. In this case, we selected trees with $50-500$ nodes and maximum outdegree

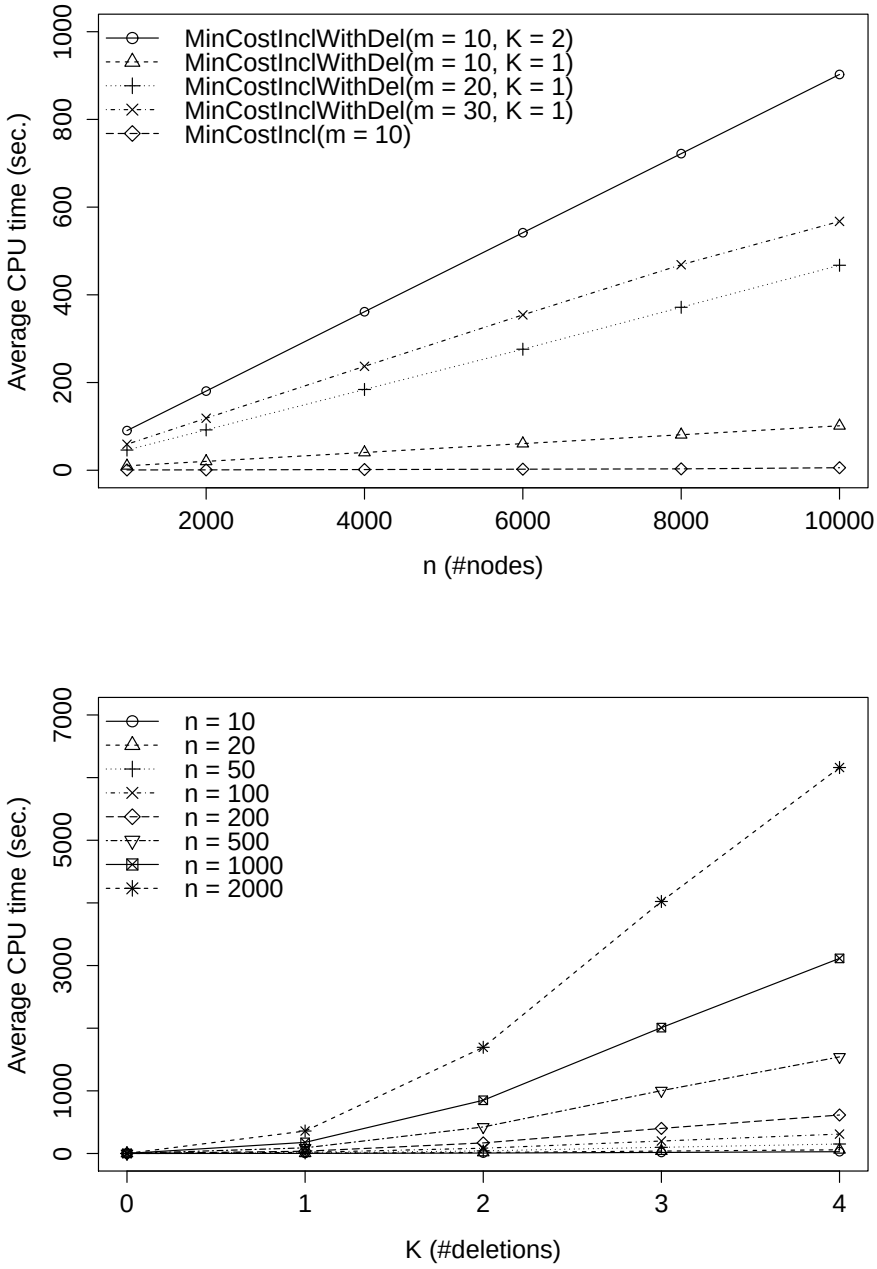


Figure 4.11: Execution times for MinCostInclWithDel with varying n (top) and varying K with $m = 10$ (bottom).

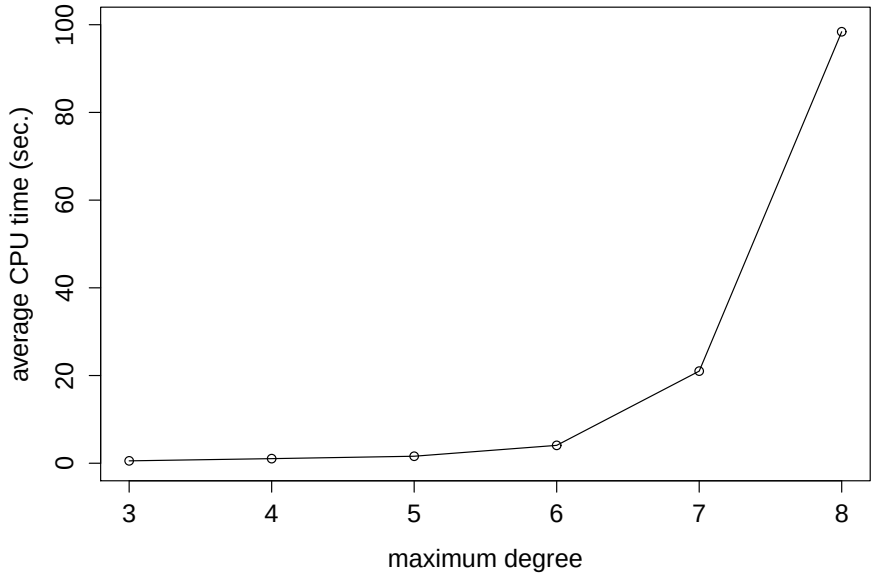


Figure 4.12: Execution times for weblogs data.

50, which resulted in a data set of 2481 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 Figure 4.12. 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.

4.5.2 Tree-Based Bibliographic Matching

To evaluate the usefulness of MinCostIncl in a practical setting, we applied it to bibliographic matching [13, 33], 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, which are then combining into the record-level similarity [13]. 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, 55], where corresponding fields between

Table 4.2: 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 4.3: Bibliographic data in Leipzig benchmark dataset

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

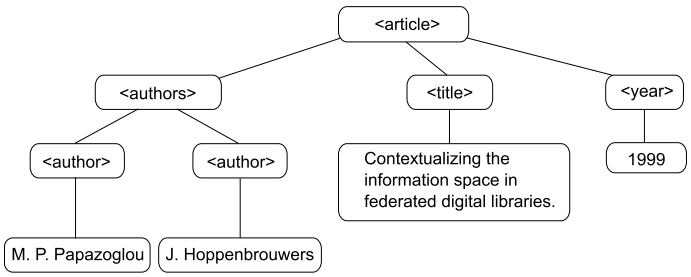
bibliographic records are mapped to each other and their similarity is calculated simultaneously.

Data Set

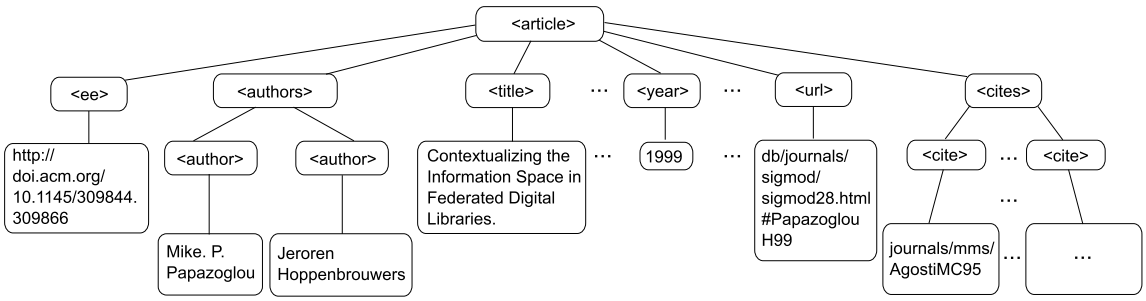
We used bibliographic data included in the benchmark datasets[1] provided by the authors of [33]. It contains bibliographic data selected from DBLP, Google Scholar (Scholar, for short), and the ACM digital libraries (ACM, for short). The benchmark dataset designed for bibliographic matching between the 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 4.2 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 4.2). 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 4.3.

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



(a) a pattern tree from ACM



(b) a text tree in DBLP

Figure 4.13: Pattern and text trees for bibliographic search.

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. Figure 4.13 (a) 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. Figure 4.13 (b) 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 4.2 shows the selected numbers of article pairs for each dataset.

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 subsection, we counted the mappings of tags between pattern and text trees. Table 4.4 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.

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 [45] 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 subsection 4.5.2, were ordered according to the minimum-cost of tree inclusion computed by MinCostIncl and the tree edit distance computed by RTED. The average CPU time of MinCostIncl and RTED for each candidate pair is shown in Table 4.5. Notice that although RTED is much faster than MinCostIncl, MinCostIncl is still reasonably fast.

Figure 4.14 shows the Receiver Operating Characteristic (ROC) curves for ACM-DBLP and Scholar-DBLP, where an ROC curve is a measure of the binary classi-

Table 4.4: 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 4.5: Execution times of MinCostIncl and RTED

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

fication 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

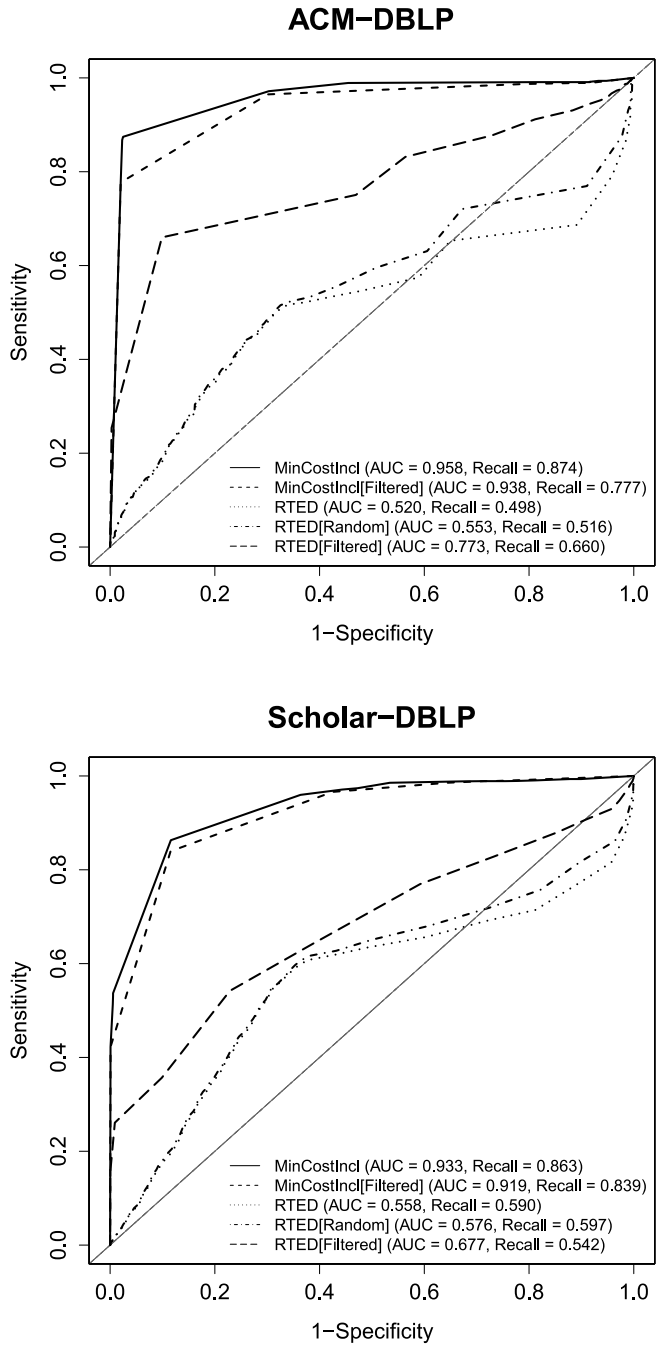


Figure 4.14: Comparison of accuracy for ACM-DBLP data (top) and Scholar-DBLP data (bottom).

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 Figure 4.13 (b). 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 Figure 4.14 shows 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. Therefore, it is suitable for the bibliographic matching without various preprocessing steps involving adjusting the left-to-right node orderings or extra node removal.

4.5.3 Application to Glycan Data

We evaluated the efficiency of MinCostIncl and MinCostInclWithDel for glycan data in the KEGG database [29], including comparison with DpCliqueEdit (proposed in Chapter 3, [41]). As in [41], 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 Figure 4.15. 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 [61] 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.

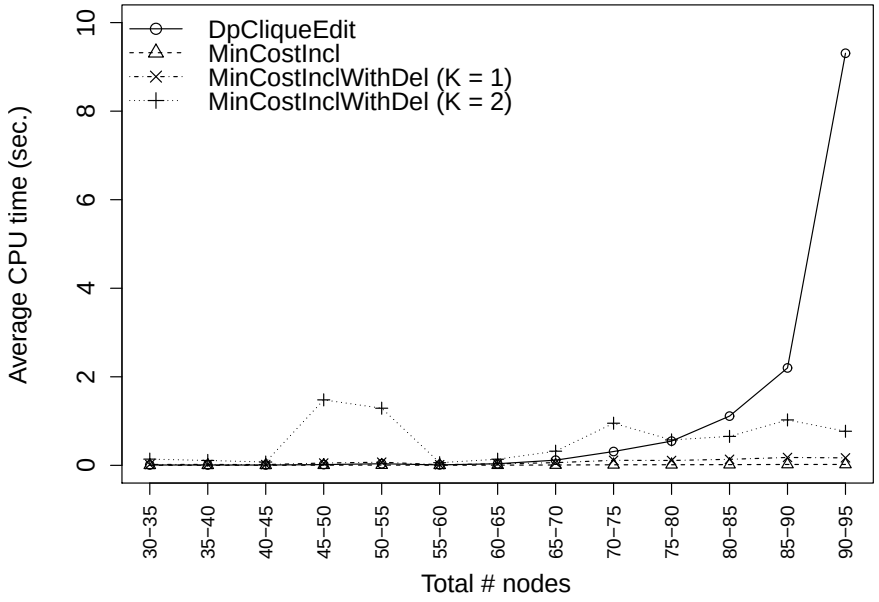


Figure 4.15: Execution times for glycan data.

4.6 Discussions

In this chapter, 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 [31] 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.

As for the application to glycan data, we focused only on the computational efficiencies of MinCostIncl and MinCostInclWithDel in the same way as Chapter 3. As mentioned in Chapter 1, glycans consist of the core structures and the terminal elaborations, and they often contribute to the glycan functions independently. For example, while the core structures protect their binding protein from proteases and aqueous solvent regardless of their terminal elaborations, a certain kind of hormone activity is regulated through recognition of modification of terminal elaborations by the mannose receptors. Thus, we often need to focus on not the whole structures but the substructures of them. Therefore, tree inclusion is useful for analysis of glycans. However, the original tree inclusion can not deal with the substitution operations to pattern trees, so that any difference of monosaccharide is never detected. The ABO blood group system is one of the most famous example of such difference in terminal elaborations. The extended tree inclusion can detect such slight differences in substructures, and so it is highly effective for glycan studies. Furthermore, the proposed algorithms for computing the cost of inclusion can be applied to glycan database searching. Actually, a Web application for glycan searching has been developed [8] and opened to the public as a part of KEGG Glycan database. The application is based on a tree-matching algorithm proposed in [7] and it returns a set of glycans listed according to a similarity score when a pattern glycan is input. However, this application does not deal with similar substructure searching. Therefore, it is worth applying the extended tree inclusion and its cost to database searching.

In this chapter, although we focus only on comparison of glycans, further applications to other tree-structured biological data are left as future work. For example, MinCostIncls might be useful for similar RNA secondary substructure searching, since RNA secondary structures are sometimes regarded as unordered trees [39] as well as the fact that MinCostIncls can compute new measure (i.e., the cost of tree inclusion) for comparison of them.

Chapter 5

Conclusion

5.1 Summary

In this thesis, first we described the tree edit distance, the tree inclusion, and other related work. Then, we dealt with two research topics for analyzing the tree-structured data: computation of the similarity between unordered trees using the tree edit distance and similar subtree searching via the extended unordered tree inclusion.

In Chapter 2, we briefly reviewed the measures for (dis-)similarity between two rooted trees and the algorithms for computing them. First, we introduced the tree edit distance which is one of the most widely used measures, and then we described A^*-algorithm, a fixed-parameter algorithm, an ILP-based algorithm, and a clique-based algorithm for the unordered tree edit distance problem. Then, as related work, we also introduced restricted versions of the tree edit distance: the constrained edit distance and less-constrained edit distance. Although efficient algorithms have been proposed for the restricted distances, the less-constrained case has been proved to be MAX SNP-hard. Furthermore, we described some approximation algorithms for the unordered tree edit distance and the largest common subtrees. They have been developed in order to reduce the computational cost by relaxing the optimality of solutions. Finally, we explained other variants of the tree edit distance: the tree alignment and the tree inclusion. The problems of computing these measures are also NP-hard. However, they can be solved in polynomial time if the maximum degree of input trees and the maximum degree of pattern tree are bounded, respectively.

In Chapter 3, we proposed an improved clique-based method, which is extended from a previous clique-based method by introducing DP scheme and several heuristic techniques, for computing the tree edit distance between rooted unordered trees. We also proved that the proposed method computes the tree edit distance exactly in

spite of introducing heuristic techniques. In computational experiments, the introduced DP scheme and heuristic techniques were verified to be very useful, especially for the cases that trees have many leaves or many isomorphic subtrees. In particular, for hard instances of comparison of glycan structures, the improved method achieved more than 100 times speed-up for large cases. Although the improved method was applied to Weblogs data, it took long CPU time to compute the tree edit distance when some nodes have many child nodes. However most biological data such as glycans and RNA secondary structures have few such internal nodes, so that such case is less of an issue in computational biology.

In Chapter 4, we have extended the concept of unordered tree inclusion to take the costs of insertions and substitutions into account. The time complexity of the proposed algorithm for computing the cost is $O(2^{2D}mn)$ for unordered cases, where m, n and D denote the size of the pattern tree, the size of the text tree, and the maximum outdegree of the pattern tree, respectively. Additionally, we showed that the space complexity of the algorithms can be reduced from $O(2^D mn)$ to $O(n + 2^D m \log(n))$ if traceback is not required. We also extended the algorithm to allow a small number of deletions in the pattern tree. The time complexity of the resulting algorithm is $O((eD)^K K^{1/2} 2^{2(DK+D-K)} mn)$, where K is the number of allowed deletion operations and e is the base of the natural logarithm. Computational experiments on large synthetic and real datasets showed that our proposed algorithm is fast and scalable. As for glycan data, we compared the computational efficiency of MinCostIncl and MinCostInclWithDel to DpCliqueEdit. From the result, the cost of inclusion can be computed much faster than the tree edit distance for large cases even when MinCostInclWIthDel with $K = 2$ is applied.

5.2 Future Directions

Through this thesis, we have developed computational methods and performed theoretical analysis for tree-structured data and network data appearing in the fields of computational biology. Although the effectiveness of each proposed method are verified via computational experiments, further improvements of these approaches are still left for future work.

For the unordered tree edit distance problem, although our proposed method DpCliqueEdit outperform the previous clique-based method, there still exist cases for which it takes long CPU time. For example, if input trees have long subtrees, DpCliqueEdit does not work efficiently since none of the introduced heuristic techniques cope with the cases well. Moreover, in order to achieve further speed-up, we should improve the algorithm for the maximum vertex weighted clique problem since the efficiency of DpCliqueEdit depends on that of a clique-finding algorithm. Additionally, a cost function which is suitable for tree-structured biological data is

also needed in order to analyze biological objects.

For the unordered tree inclusion problem, we showed that our proposed algorithm MinCostIncl is fast and scalable via computational experiments on large synthetic and real datasets. On the other hand, although a space-efficient version of the algorithm, MinCostIncle-SpS, can save memory usage actually, it does not allow traceback. Therefore, to make MinCostIncl-SpS allow traceback is left as an open problem. Another open problem is to further improve the time complexity of MinCostInclWithDel, in which a small number of deletions are allowed.

As for applications to tree-structured biological data, we showed that our proposed algorithms are effective for glycan data. DpCliqueEdit enables us to compute the edit distance among all pairs of glycans in KEGG database. On the other hand, MinCostIncl, MinCostIncl-SpS, and MinCostInclWithDel can compute the new measure for similar subtree searching. However, in this thesis, we focused only on the computational efficiency of our algorithms. Therefore, further applications to glycans are left as future work. First, we should apply DpCliqueEdit and MinCostIncls for analyzing glycan functions based on their structures. It should be noted that although the usefulness of tree edit distance for similar structure search has already been shown in [19], to relate the structures to their functions is still challenging. However, relationships between them are indicated in some computational studies. For example, a clustering method for glycan functions based on their structures has been proposed [36]. The method employs q-gram as a similarity measure and shows high classification accuracy. Thus, we need to perform this kind of clustering analysis using the tree edit distance and the cost of tree inclusion instead of q-gram and compare their accuracies.

Furthermore, in order to make DpCliqueEdit and MinCostIncls widely used in the field of glycobiology, we need to extend them to database searching and provide them as a Web application. Actually, a Web application for glycan database searching called KCaM has been developed [8]. The application is based on a tree-matching algorithm with constraints on deletion and insertion operations. Although the constraints reduce the computational cost, only one child of a node u is set as a child of the parent of u and the other children are deleted in a deletion operation (an insertion operation is the inverse of a deletion). Therefore, some cases of mapping are ignored. On the other hand, DpCliqueEdit and MinCostIncl take all cases into account and return dissimilarity between input tree-structured data, so that it is worth providing a Web application for glycan searching based on the proposed algorithms.

Applications to other tree-structured biological data are also left as future work. For RNA secondary structures, many studies for analyzing the structures and their functions have been done and the results are stored in databases. For example, Rfam is one of the most widely used databases [43] and it provides 2450 RNA families with various information such as their secondary structures and Gene Ontology

(GO) annotations [9]. Therefore, a clustering analysis of RNA functions based on their structure information can be done using the tree edit distance and the cost of inclusion as similarity measures. It is true that the similarity computation can be done much faster than the cases of glycans since the RNA secondary structures are generally regarded as ordered trees, so that efficient algorithms for unordered trees are not required. However, they are sometimes regarded as unordered trees [39]. Thus, in such cases, DpCliqueEdit and MinCostIncls are expected to be useful. On the other hand, for phylogenetic trees, it is not clear whether DpCliqueEdit and MinCostIncls can be applied or not since we suppose that inputs are general rooted, labeled, and unordered trees but all internal nodes in phylogenetic trees are unlabeled. Therefore, an extension of the algorithms for phylogenetic trees is left as future work.

Bibliography

[1] T. Akutsu. A relation between edit distance for ordered trees and edit distance for Euler strings. *Information Processing Letters*, 100:105–109, 2006.

[2] 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. *Theoretical Computer Science*, 470:10–22, 2013.

[3] T. Akutsu, D. Fukagawa, and A. Takasu. Improved approximation of the largest common subtle of two unordered trees of bounded height. *Information Processing Letters*, 109:165–170, 2008.

[4] T. Akutsu, D. Fukagawa, and A. Takasu. Approximating tree edit distance through string edit distance. *Algorithmica*, 57(2):325–348, 2010.

[5] T. Akutsu, D. Fukagawa, A. Takasu, and T. Tamura. Exact algorithms for computing tree edit distance between unordered trees. *Theoretical Computer Science*, 421:352–364, 2011.

[6] T. Akutsu, T. Mori, T. Tamura, D. Fukagawa, A. Takasu, and E. Tomita. An improved clique-based method for computing edit distance between unordered trees and its application to comparison of glycan structures. In *The 4th International Workshop on Intelligent Informatics in Biology and Medicine, A Part of Proceedings of the 5th International Conference on Complex, Intelligent and Software Intensive Systems, 2011*, pages 536–540.

[7] K.F. Aoki, A. Yamaguchi, Y. Okuno, T. Akutsu, N. Ueda, M. Kanehisa, and H. Mamitsuka. Efficient tree-matching methods for accurate carbohydrate database queries. *Genome Informatics*, 14:134–143.

[8] 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 Asids Research*, 32:W267–W272, 2004.

[9] M. Ashburner, C.A. Ball, J.A. Blake, D. Botstein, H. Butler, J.M. Cherry, A.P. Davis, K. Dolinski, S.S. Dwight, J.T. Eppig, M.A. Harris, D.P. Hill, L. Issel-Tarver, A. Kasarskis, S. Lewis, J.C. Matese, J.E. Richardson, M. Ringwald, G.M. Rubin, and G. Sherlock. Gene Ontology: tool for the unification of biology. *Nature Genetics*, 25:25–29, 2000.

[10] N. Augsten, D. Barbosa, M.H. Böhlen, and T. Palpanas. TASM: Top-k approximate subtree matching. In *Proceedings of ICDE*, pages 353–364, 2010.

[11] N. Augsten, M.H. Böhlen, C.E. Dyreson, and J. Gamper. Approximate joins for data-centric xml. In *Proceedings of ICDE*, pages 814–823, 2008.

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

[13] M. Bilenko and R.J. Mooney. Adaptive duplicate detection using learnable string similarity measures. In *Proceedings of KDD*, pages 39–48, 2003.

[14] P. Bille. A survey on tree edit distance and related problems. *Theoretical Computer Science*, 337:217–239, 2005.

[15] C. Borgs, J. Chayes, J. Kahn, and L. Lovász. Left and right convergence of graphs with bounded degree. *Random Structures and Algorithms*, 42:1–28, 2013.

[16] E.D. Demaine, S. Mozes, B. Rossman, and O. Weimann. An optimal decomposition algorithm for tree edit distance. *ACM Transactions on Algorithms*, 6:1, 2009.

[17] R. Durbin, S.R. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis - Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press, 1998.

[18] J. Felsenstein. *Inferring Phylogenies*. Sinauer Associates, Inc., Sunderland, Massachusetts, 2004.

[19] D. Fukagawa, T. Tamura, A. Takasu, E. Tomita, and T. Akutsu. A clique-based method for the edit distance between unordered trees and its application to analysis of glycan structures. *BMC Bioinformatics*, 12(Suppl 1), S13, 2011.

[20] M.N. Garofalakis and A. Kumar. XML stream processing using tree-edit distance embeddings. *ACM Trans. Database System*, 30:279–332, 2005.

[21] S. Guha, H.V. Jagadish, N. Koudas, D. Srivastava, and T. Yu. Approximate xml joins. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management on Data*, pages 287–298, 2002.

[22] M.M. Halldórsson and K. Tanaka. Approximation and special cases of common subtrees and editing distance. pages 75–84, 1996.

[23] 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. *Theoretical Computer Science*, 411:2423–2432.

[24] Y. Horesh, R. Mehr, and R. Unger. Designing an A* algorithm for calculating edit distance between rooted-unordered trees. *Journal of Computational Biology*, 13:1165–1176, 2006.

[25] T. Jiang, L. Wang, and K. Zhang. Alignment of trees - an alternative to tree edit. *Theoretical Computer Science*, 143:137–148, 1995.

[26] K. Kailing, H.-P. Kriegel, S. Schönauer, and T. Seidl. Efficient similerity search for hierarchical data in large databases. In *Proceedings of the 9th International Conference on Extending Database Technology (EDBT)*, pages 676–693, 2004.

[27] T. Kan, S. Higuchi, and K. Hirata. Segmental mapping and distance for rooted labeled ordered trees. *Fundamenta Informaticae*, 132:461–483, 2014.

[28] M. Kanehisa, S. Goto, M. Furumichi, M. Tanabe, and M. Hirakawa. KEGG for representation and analysis of molecular networks. *Nucleic Acids Research*, 38:D355–D360, 2010.

[29] M. Kanehisa, S. Goto, M. Furumichi, M. Tanabe, and M. Hirakawa. Data, information, knowledge and principle: back to metabolism in KEGG. *Nucleic Acids Research*, 42:D199–D205, 2014.

[30] P. Kilpeläinen. *Tree Matching Problems with Applications to Structured Test Datasets*. PhD thesis, University of Helsinki, Department of Computer Science, November 1992.

[31] P. Kilpeläinien and H. Mannila. Ordered and unordered tree inclusion. *SIAM Journal on Computing*, 24:340–356, 1995.

[32] S. Kondo, K. Otaki, M. Ikeda, and A. Yamamoto. Fast computation of the tree edit distance between unordered trees using IP solvers,. In *Proceedings of the 17th International Conference on Discovery Science*, LNAI 8777, pages 156–167, 2014.

[33] H. Köpcke, A. Thor, and E. Rahm. Evaluation of entity resolution approaches on real-world match problems. In *Proceedings of the VLDB Endowment*, volume 3, pages 484–493, 2010.

[34] T. Kuboyama. *Matching and Learning in Trees*. PhD thesis, University of Tokyo, 2007.

[35] F. Li, H. Wang, J. Li, and H. Gao. A survey on tree edit distance lower bound estimation techniques for similarity join on xml data. *ACM SIGMOD Record*, 42(4):29–39, 2013.

[36] L. Li, W-K. Ching, T. Yamaguchi, and K.F. Aoki-Kinoshita. A weighted q-gram method for glycan structure classification. *BMC Bioinformatics*, 11 (Suppl 1): S33, 2011.

[37] C.L. Lu, Z.-Y. Su, and C.Y. Tang. A new measure of edit distance between labeled trees. In *Proc. 7th Annual International Conference on Computing and Combinatorics (COCOON) (Lecture Notes in Computer Science Vol. 2108)*. Springer, 2001.

[38] J. Matoušek and J. Nešetři. *Invitation to Discrete Mathematics*. Oxford University Press, New York, 1998.

[39] N. Milo, S. Zakov, E. Katzenelson, E. Bachmat, Y. Dinitz, and M. Ziv-Ukelson. Unrooted unordered homeomorphic subtree alignment of RNA trees. *Algorithms for Molecular Biology*, 8(13), 2013.

[40] T. Mori, A. Takasu, J. Jansson, J. Hwang, T. Tamura, and T. Akutsu. Similar subtree search using extended tree inclusion. *IEEE Transactions on Knowledge and Data Engineering*. in press.

[41] 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. *Journal of Computational Biology*, 19:1089–1104, 2012.

[42] T. Nakamura and E. Tomita. Efficient algorithms for finding a maximum clique with maximum vertex weight. Technical Report UEC-TR-CAS3-2005 (in Japanese), the University of Electro-Communications, 2005.

[43] E.P. Nawrocki, S.W. Burge, A. Bateman, J. Daub, R.Y. Eberhardt, S.R. Eddy, E.W. Floden, P.P. Gardner, T.A. Jones, J. Tate, and R.D. Finn. Rfam 12.0: updates to the RNA families database. *Nucleic Acids Research*, 43(D1):D130–D137, 2014.

[44] H. Ogawa. Labeled point pattern matching by Delaunay triangulation and maximal cliques. *Pattern Recognition*, pages 35–40, 1986.

[45] M. Pawlik and N. Augsten. RTED: A robust algorithm for the tree edit distance. In *Proceedings of the VLDB Endowment*, volume 5, pages 334–345, 2013.

[46] M. Pelillo, K. Siddiqi, and S. W. Zucker. Matching hierarchical structures using association graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 21:1105–1119, 1999.

[47] T. Richter. A new measure of the distance between ordered trees and its applicatios. Technical Report 85166-cs, University of Bonn, 1997.

[48] K.-C. Tai. The tree-to-tree correction problem. *Journal of ACM*, 26:4220–4433, 1979.

[49] M.E. Taylor and K. Drickamer. *Introduction to Glycobiology.* Oxford University Press, 2003.

[50] E. Tomita, T. Akutsu, and T. Matsunaga. Efficient algorithms for finding maximum and maximal cliques: Effective tools for bioinformatics. In *Biomedical Engineering, Trends in Electronics, Communications and Software*, pages 625–640. Vienna: InTech, 2011.

[51] E. Tomita and T. Seki. An efficient branch-and-bound algorithm for finding a maximum clique. In *Proceedings of the 4th International Conference on Discrete Mathematics and Theoretical Computer Science (Lecture Notes in Computer Science Vol. 2731)*, pages 278–289, 2003.

[52] E. Tomita, Y. Sutani, T. Higashi, S. Takahashi, and M. Wakatsuki. A simple and faster branch-and-bound algorithm for finding a maximum clique. In *Proceedings of the 4th International Workshop on Algorithms and Computation (Lecture Notes in Computer Science Vol. 5942)*, pages 191–203, 2010.

[53] A. Torsello and E.R. Hancock. Computing approximate tree edit distance using relaxation labeling. *Pattern Recognition Letters*, 24:1089–1097, 2003.

[54] P.H. Winston. *Artificial Intelligence.* Addison Wesley, New York, 3rd edition, 1992.

[55] R. Yang, P. Kalnis, and A.K.H. Tung. Similarity evaluation on tree-structured data. In *SIGMOD conference*, pages 754–765, 2005.

[56] K.-C. Yu, E.L. Ritman, and E. Higgns. System for the analysis and visualization of large 3D anatomical trees. *Computers in Biology and Medicine*, 27:1802–1830, 2007.

[57] M.J. Zaki. Efficiently mining frequent trees in a forest: Algorithms and applications. *IEEE Transactions on Knowledge and Data Engineering*, 17(8):1021–1035, 2005.

[58] K. Zhang. Algorithms for the constrained editing problem between ordered labeled trees and related problems. *Pattern Recognition*, 28:463–474, 1995.

[59] K. Zhang. A constrained edit distance between unordered labeled trees. *Algorithmica*, 15:205–222, 1996.

[60] K. Zhang and T. Jiang. Some MAX SNP-hard results concerning unordered labeled trees. *Information Processing Letters*, 49:249–254, 1994.

[61] K. Zhang, R. Statman, and D. Shasha. On the editing distance between unordered labeled trees. *Information Processing Letters*, 42:133–139, 1992.

List of Publications

Journal Paper

1. 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", *Journal of Computational Biology*, vol. 19(10), pp. 1089–1104, 2012.

2. T. Mori, M. Flöttmann, M. Krantz, T. Akutsu and E. Klipp, "Stochastic simulation of Boolean *rxncon* models: towards quantitative analysis of large signaling networks", *BMC Systems Biology*, 9(45), 2015.

3. T. Mori, A. Takasu, J. Jansson, J. Hwang, T. Tamura and T. Akutsu, "Similar subtree search using extended tree inclusion", *IEEE Transactions on Knowledge and Data Engineering*, in press.

4. T. Hasegawa, T. Mori, R. Yamaguchi, S. Imoto, S. Miyano and T. Akutsu, "An efficient data assimilation schema for restoration and extension of gene regulatory networks using time-course observation data", *Journal of Computational Biology*, vol. 21(11), pp. 785–798, 2014.

5. T. Hasegawa, T. Mori, R. Yamaguchi, T. Shimamura, S. Miyano, S. Imoto and T. Akutsu, "Genomic data assimilation using a higher moment filtering technique for restoration of gene regulatory networks", *BMC Systems Biology*, 9(14), 2015.

6. T. Hasegawa, A. Niida, T. Mori, T. Shimamura, R. Yamaguchi, S. Miyano, T. Akutsu, and S. Imoto, "A likelihood-free filtering method via approximate Bayesian computational in evaluating biological simulation models", *Computational Statistics and Data Analysis*, in press.

Conference Paper

1. T. Akutsu, T. Mori, T. Tamura, D. Fukagawa, A. Takasu and E. Tomita, "An improved clique-based method for computing edit distance between unordered trees and its application to comparison of glycan structures", *International Conference on Complex, Intelligent and Software Intensive Systems* (CISIS), pp. 536–540, 2011.