Copyright Warning & Restrictions

The copyright law of the United States (Title 17, United States Code) governs the making of photocopies or other reproductions of copyrighted material.

Under certain conditions specified in the law, libraries and archives are authorized to furnish a photocopy or other reproduction. One of these specified conditions is that the photocopy or reproduction is not to be "used for any purpose other than private study, scholarship, or research." If a, user makes a request for, or later uses, a photocopy or reproduction for purposes in excess of "fair use" that user may be liable for copyright infringement,

This institution reserves the right to refuse to accept a copying order if, in its judgment, fulfillment of the order would involve violation of copyright law.

Please Note: The author retains the copyright while the New Jersey Institute of Technology reserves the right to distribute this thesis or dissertation

Printing note: If you do not wish to print this page, then select "Pages from: first page # to: last page #" on the print dialog screen

ABSTRACT

PATTERN DISCOVERY IN TREES: ALGORITHMS AND APPLICATIONS TO DOCUMENT AND SCIENTIFIC DATA MANAGEMENT

by
Chia-Yo Chang

Ordered, labeled trees are trees in which each node has a label and the left-to-right order of its children (if it has any) is fixed. Such trees have many applications in vision, pattern recognition, molecular biology and natural language processing.

In this dissertation we present algorithms for finding patterns in the ordered labeled trees. Specifically we study the largest approximately common substructure (LACS) problem for such trees. We consider a substructure of a tree T to be a connected subgraph of T. Given two trees T_1, T_2 and an integer d, the LACS problem is to find a substructure U_1 of T_1 and a substructure U_2 of T_2 such that U_1 is within distance d of U_2 and where there does not exist any other substructure V_1 of T_1 and V_2 of T_2 such that V_1 and V_2 satisfy the distance constraint and the sum of the sizes of V_1 and V_2 is greater than the sum of the sizes of U_1 and U_2. The LACS problem is motivated by the studies of document and RNA comparison.

We consider two types of distance measures: the general edit distance and a restricted edit distance originated from Selkow. We present dynamic programming algorithms to solve the LACS problem based on the two distance measures. The algorithms run as fast as the best known algorithms for computing the distance of two trees when the distance allowed in the common substructures is a constant independent of the input trees. To demonstrate the utility of our algorithms, we discuss their applications to discovering motifs in multiple RNA secondary structures.

Such an application shows an example of scientific data mining. We represent an RNA secondary structure by an ordered labeled tree based on a previously

proposed scheme. The patterns in the trees are substructures that can differ in both substitutions and deletions/insertions of nodes of the trees. Our techniques incorporate approximate tree matching algorithms and novel heuristics for discovery and optimization. Experimental results obtained by running these algorithms on both generated data and RNA secondary structures show the good performance of the algorithms. It is shown that the optimization heuristics speed up the discovery algorithm by a factor of 10. Moreover, our optimized approach is 100,000 times faster than the brute force method.

Finally we implement our techniques into a graphic toolbox that enables users to find repeated substructures in an RNA secondary structure as well as frequently occurring patterns in multiple RNA secondary structures pertaining to rhinovirus obtained from the National Cancer Institute. The system is implemented in C programming language and X windows and is fully operational on SUN workstations.

PATTERN DISCOVERY IN TREES: ALGORITHMS AND APPLICATIONS TO DOCUMENT AND SCIENTIFIC DATA MANAGEMENT

by
Chia-Yo Chang

A Dissertation
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy

Department of Computer and Information Science

May 1999

APPROVAL PAGE

PATTERN DISCOVERY IN TREES: ALGORITHMS AND APPLICATIONS TO DOCUMENT AND SCIENTIFIC DATA MANAGEMENT

Chia-Yo Chang

Dr. Jason T. L. Wang, Dissertation Advisor — Date
Associate Professor of Computer and Information Science Department,
NJIT, Newark, NJ

Dr. James A. McHugh, Committee Member — Date
Professor of Computer and Information Science Department,
NJIT, Newark, NJ

Dr. James M. Calvin, Committee Member — Date
Assistant Professor of Computer and Information Science Department,
NJIT, Newark, NJ

Dr. Pengcheng Shi, Committee Member — Date
Assistant Professor of Computer and Information Science Department,
NJIT, Newark, NJ

Dr. Frank C. D. Tsai, Committee Member — Date
Member of Technical Staff, AT&T Laboratories,
Middletown, NJ

BIOGRAPHICAL SKETCH

Author: Chia-Yo Chang

Degree: Doctor of Philosophy

Date: May 1999

Education:

- Doctor of Philosophy in Computer Science,
  New Jersey Institute of Technology, Newark, NJ, 1999
- Master of Science in Computer Science,
  New Jersey Institute of Technology, Newark, NJ, 1989
- Bachelor of Science in Mathematics,
  National Tsing Hua University, Hsinchu, Taiwan R.O.C, 1983

Publications:

J. T. L. Wang, K. Zhang, and C. Y. Chang, "Identifying Approximately Common Substructures in Trees Based on a Restricted Edit Distance," *Information Sciences*, Accepted.

C. Y. Chang, J. T. L. Wang, and R. K. Chang, "Scientific Data Mining: A Case Study," *International Journal of Software Engineering and Knowledge Engineering*, 8(1):77–96, 1998.

J. T. L. Wang and C. Y. Chang, "Fast Retrieval of Electronic Messages That Contain Mistyped Words or Spelling Errors," *IEEE Transactions on Systems, Man, and Cybernetics*, 27(3):441–451, June 1997.

C. Y. Chang and J. T. L. Wang, "An Algorithm for Identifying Structural Similarities in Electronic Documents," *Proceedings of the 3rd Joint Conference on Information Sciences*, Research Triangle Park, North Carolina, pp. 324–327, March 1997.

J. T. L. Wang, B. A. Shapiro, D. Shasha, K. Zhang, and C. Y. Chang, "Automated Discovery of Active Motifs in Multiple RNA Secondary Structures," *Proceedings of the 2nd International Conference on Knowledge Discovery and Data Mining*, Portland, Oregon, pp. 70–75, August 1996.

C. Y. Chang and J. T. L. Wang, "Scientific Data Mining: A Case Study," *Proceedings of the 8th International Conference on Software Engineering and Knowledge Engineering*, Lake Tahoe, Nevada, pp. 100–107, June 1996.

J. T. L. Wang, G. J. S. Chang, G. W. Chirn, C. Y. Chang, W. Wu, and F. Aljallad, "A Visualization Tool for Pattern Matching and Discovery in Scientific Databases," *Proceedings of the 8th International Conference on Software Engineering and Knowledge Engineering*, Lake Tahoe, Nevada, pp. 563–570, June 1996.

J. T. L. Wang and C. Y. Chang, "Fast Retrieval of Electronic Documents in Digital Libraries," *Proceedings of the 7th IEEE International Conference on Tools with Artificial Intelligence*, Washington, D.C., pp. 208–215, November 1995.

J. T. L. Wang, G. W. Chirn, C. Y. Chang, G. J. S. Chang, A. Noriega and K. Pysniak, "An Integrated Toolkit for Pattern Matching and Pattern Discovery in Scientific, Program and Document Databases," *Proceedings of the 7th International Conference on Software Engineering and Knowledge Engineering*, Rockville, Maryland, pp. 497, June 1995.

ACKNOWLEDGMENT

I am indebted to my advisor, Professor Jason T. L. Wang, for his guidance throughout this work. Also, I would like to thank Professor James McHugh, Professor James Calvin, Professor Pengcheng Shi and Dr. Frank Tsai for serving as committee members.

Most of all, I want to express my deepest appreciation to my parents, my brothers, my wife, Mei-Hsiu Ling, and my son, Eefine, my daughter, Eeshin. They have given me support during my most difficult days. Without their support and encouragement, my accomplishment would not be possible.

Finally, I would like to thank Dr. Gung-Wei Chirn and George Jyh-Shian Chang for their help on the toolbox presented in the dissertation.

TABLE OF CONTENTS

LIST OF TABLES

Table **Page**

LIST OF FIGURES

Figure **Page**

CHAPTER 1

INTRODUCTION

Ordered, labeled trees are trees in which each node has a label and the left-to-right order of its children (if it has any) is fixed.[1] Such trees have many applications in vision, pattern recognition, molecular biology, natural language processing, Web management, programming languages and natural language processing, including the representations of images [36], patterns [8, 32] , secondary structures of RNA [40], structured documents (e.g. HTML) [7], grammar parses [4] and sentences [15]. Identifying structural similarities in such trees helps in version management and change detection. For example, in software maintenance where programs are represented as parse trees [35], comparing the parse trees helps detect the changes to the code. As another example, a user of the World Wide Web may be interested in knowing changes in an HTML document. Such changes can be detected by representing the document as a tree based on its underlying markup language [7] and by comparing the old and new version of the document (tree). In hypertext authoring, a user may wish to find the common portions in the history list of an HTML document or a set of documents. This may be accomplished by finding the common substructures in the corresponding trees. Trees are frequently used in other disciplines as well.

A large amount of work has been performed for comparing two trees based on various distance measures [19, 28, 34, 50, 69]. [42, 48, 71] recently generalized one of the most commonly used distance measures, namely the edit distance, for both rooted and unrooted unordered trees. The work laid out a foundation that is useful for comparing graphs [41, 66].

[1] Throughout the dissertation, we shall refer to ordered labeled trees simply as trees when no ambiguity occurs.

In this dissertation we extend the previous work by considering the largest approximately common substructure (LACS) problem for ordered labeled trees. We consider a substructure of a tree T to be a connected subgraph of T. Given two trees T_1, T_2 and an integer d, the LACS problem is to find a substructure U_1 of T_1 and a substructure U_2 of T_2 such that U_1 is within distance d of U_2 and where there does not exist any other substructure V_1 of T_1 and V_2 of T_2 such that V_1 and V_2 satisfy the distance constraint and the sum of the sizes of V_1 and V_2 is greater than the sum of the sizes of U_1 and U_2. The LACS problem is motivated by the studies of document and RNA comparison.

We consider two types of distance measures: the general edit distance and a restricted edit distance originated from Selkow. We present dynamic programming algorithms to solve the LACS problem based on the two distance measures. The algorithms run as fast as the best known algorithms for computing the distance of two trees when the distance allowed in the common substructures is a constant independent of the input trees. To demonstrate the utility of our algorithms, we discuss their applications to discovering motifs in multiple RNA secondary structures.

Such an application shows an example of scientific data mining. We represent an RNA secondary structure by an ordered labeled tree based on a previously proposed scheme. The patterns in the trees are substructures that can differ in both substitutions and deletions/insertions of nodes of the trees. Various biologists [24, 40] represent RNA secondary structures as trees. Finding common patterns (also known as motifs) in these secondary structures helps both in predicting RNA folding [24] and in functional studies of RNA processing mechanisms [40].

Previous methods for detecting motifs in the RNA molecules (trees) are based on one of the following two approaches: (1) transforming the trees to sequences and then using sequence alignment algorithms [39]; (2) representing the molecules using a highly simplified tree structure and then searching for common nodes in the

trees [24]. Neither of the two approaches satisfactorily takes the full tree structure into account. By contrast, utilizing the proposed algorithm for pairs of trees enables one to locate tree-structured motifs occurring in multiple RNA secondary structures. Our experimental results concerning RNA classification show the significance of these motifs [61].

1.1 Related Work

A large amount of work has been performed for comparing trees based on various distance measures. Tai [46], for example, proposed to measure the distance between two trees by the minimum cost sequence of edit operations needed to transform one tree to the other. The set of allowable edit operations includes: (1) changing one node into another node (changing the label of the node); (2) deleting one node from a tree; and (3) inserting a node into a tree. Tai gave an algorithm for computing the distance of two trees, which runs in time $O(|T_1| \times |T_2| \times (depth(T_1))^2 \times (depth(T_2))^2)$. Zhang and Shasha [69] later presented a faster algorithm that runs in time $O(|T_1| \times |T_2| \times \min(depth(T_1), leaves(T_1)) \times \min(depth(T_2), leaves(T_2)))$.

In a more recent paper, Zhang, Shasha and Wang [70] extended the tree matching work to allow one of the trees (the pattern tree) to contain variable length don't cares (VLDCs). When matching the pattern tree with a data tree, the VLDCs may substitute for a portion of the data tree at no cost. Their algorithm runs as fast as the best algorithm for tree matching without VLDCs [69]. The authors implemented the algorithms into a graphics toolkit for querying lexical databases [62].

Cheng and Lu [8] studied a different distance measure based on node splitting and merging; the distance between two trees is defined as the minimum number of such operations needed to transform one tree to the other. The authors presented a tree matching algorithm based on this metric and applied it to waveform correlation.

Tanaka and Tanaka [50] considered a restricted edit distance with the condition that two disjoint subtrees must be matched with two disjoint subtrees. They argued that such a structure preserving matching is more meaningful when comparing two classification trees. The authors presented an algorithm to calculate the restricted distance of trees in time $O(|T_1| \times |T_2| \times \min(leaves(T_1), leaves(T_2)))$. [68] gave a faster algorithm that runs in time $O(|T_1| \times |T_2|)$.

Recently, Jiang *et al.* [19] proposed tree alignment as an alternative of tree matching. Two trees are aligned by inserting a minimum number of extra nodes to the two trees such that the resulting trees are topologically isomorphic. The metric is an upward extension of sequence alignment [44] commonly used in biocomputing. Other relevant distance measures for trees and efficient algorithms for computing the distance measures can be found in [17, 20, 67].

1.2 Our Algorithmic Strategy

In this dissertation we present two algorithms for the LACS problem for ordered labeled trees. One is based on the general edit distance and the other is based on a restricted edit distance, referred to as the top-down edit distance, originated from Selkow [33, 38]. The algorithms bear a spiritual kinship with the previous algorithms surveyed in section 1.1 that we and others have proposed for tree pattern *matching*. Nevertheless, fundamental differences exist because of the differences in the problem goals themselves. In matching problems, we are given a pattern and need to find a distance between the pattern and one or more objects; in discovery problems by contrast, we are given two objects and a "target" distance and are asked to find the largest portions of the objects that differ by at most that distance. Specializing the discovery problem to a pair of trees, we want to find the largest connected component from each tree such that the distance between them is under the target distance value.

Let us consider the connected component in one of the trees. Since it is connected, it must be rooted at a node n and is generated by cutting off some subtrees in the subtree rooted at n. This means that a naive algorithm for pattern discovery would have to consider all the subtree pairs and for each subtree pair all the possible cuts of its subtrees. Since the number of such possible cuts is exponential, the naive algorithm is clearly impractical.

Instead we use a compound form of dynamic programming called *selective memorization*. By compound, we mean that dynamic programming is applied

1. to compute sizes of common patterns between two subtree pairs given a set of cuts;

2. to find the cuttings that yield distances less than or equal to the target one;

3. to compute the optimal cuttings for distance k, $1 \leq k \leq d$, given the optimal cuttings for distances 0 to $k-1$.

The memorization is selective in the sense that we only memorize the necessary values. For example we do not compute the size values for cuttings that have a distance larger than the target distance. For cuttings that give distance values within the target value, we do not explicitly compute every one of them.

In the computation of an optimal solution for distance value k, we also have to solve a problem which is unique for trees. Consider a pair of subtrees s_1 and s_2 whose roots map to one another in the optimal solution for distance value k. Then, in general, there are several (more than one) subtrees of s_1 and s_2 pairs which map to one another. We have to determine how the distance value k should be distributed to these several subtree pairs so that we can obtain the optimal solution. We solve this problem by partitioning the subtrees of s_1, respectively s_2, into a forest and a subtree. We then compute the distance and size values from forest to forest and from subtree to subtree.

The rest of the dissertation is organized as follows. Chapter 2 describes the edit operations, edit distance, top-down edit distance, and theorems for establishing their relation. Chapter 3 presents the algorithms, theorems and the complexity for finding patterns in two trees based on the edit distance. Chapter 4 presents the algorithms, theorems and the complexity for finding patterns in two trees based on the top-down edit distance. Chapter 5 presents the algorithms, theorems and experiments for finding patterns in multiple trees. Chapter 6 describes a graphic toolbox for pattern discovery in RNA secondary structures. Chapter 7 concludes the dissertation and discusses our future work.

CHAPTER 2

BACKGROUND

In this chapter we first define the edit operations, the edit distance and mapping between two trees. Then we review Selkow's restricted edit distance, and establish the relationship between this restricted distance and the edit distance. Finally we define the problem of identifying the largest approximately common substructures of two trees and introduce the terms and notation used in the dissertation.

2.1 Edit Operations for Trees

The trees with which we are concerned are rooted, ordered, and labeled ones. Each node in the trees has a label and the left-to-right order of its children, if it has any, is fixed. The edit distance for trees is measured in terms of three types of edit operations: *relabel* a node, *delete* a node, and *insert* a node [46, 69]. Let Ω represent the alphabet of node labels. We represent the edit operations as $u \rightarrow v$, where each of u and v is either a node label in Ω or the null label $\lambda \notin \Omega$. We call $u \rightarrow v$ a relabeling operation if $u \neq \lambda$ and $v \neq \lambda$, a delete operation if $u \neq \lambda$ and $v = \lambda$, or an insert operation if $u = \lambda$ and $v \neq \lambda$. Let T_2 be the tree that results from the application of an edit operation $u \rightarrow v$ to tree T_1; this is written as $T_1 \Rightarrow T_2$ via $u \rightarrow v$. Figure 2.1 illustrates the edit operations.

Let S be a sequence $s_1, s_2, \ldots, s_k$ of edit operations. S transforms tree T to tree T' if there is a sequence of trees $T_0, T_1, \ldots, T_k$ such that $T = T_0$, $T' = T_k$ and $T_{i-1} \Rightarrow T_i$ via s_i, for $1 \leq i \leq k$. Our definition of edit operations is really a shorthand notation for the specification. Here is the specification in full detail. Consider a single edit operation, e.g., one that transforms T_{i-1} to T_i. If it is a relabeling operation, we specify the node to be relabeled in T_{i-1}. The same holds for a delete operation. An insert operation is a little more complicated. We must specify the parent p of the

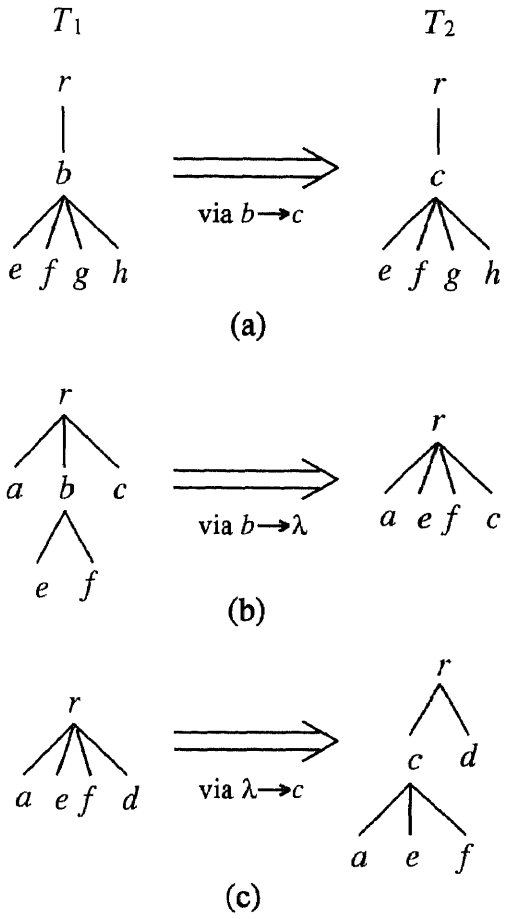


Figure 2.1 Examples illustrating the edit operations. (a) Relabeling to change one node label b to another c. (b) Deletion of a node. All children of the deleted node with label b become children of the parent labeled r. (c) Insertion of a node. A consecutive sequence of siblings among the children of the node labeled r, namely those with labels a, e, f, become the children of the newly inserted node labeled c.

node n to be inserted and which consecutive sequence of siblings among the children of p will be the children of n. If that consecutive sequence is empty, then we need to specify the position of n among the children of p. However, we continue to use our shorthand notation, because these other specifications are clear from the mapping structure defined below.

Let γ be a cost function that assigns to each edit operation $u \rightarrow v$ a nonnegative real number $\gamma(u \rightarrow v)$. For the purpose of this work, we assume $\gamma(u \rightarrow v) = 1$ if $u \neq v$ and 0 otherwise. We extend γ to a sequence of edit operations $S = s_1, s_2, \ldots, s_k$ by letting $\gamma(S) = \sum_{i=1}^{k} \gamma(s_i)$. The edit distance from tree T to tree T', denoted $tdist_e(T, T')$, is defined to be the cost of a minimum cost sequence of edit operations transforming T to T' [46, 69].

2.2 Mappings

The edit operations correspond to a *mapping* that is a graphical specification of which edit operations apply to each node in the two trees. For example, the mapping in Figure 2.2 shows a way to transform tree T_1 to tree T_2. It corresponds to the sequence: delete the nodes with labels a and m and then insert the nodes with labels a and m.

Let $t[i]$ represent the node of tree T whose position in the left-to-right postorder traversal of T is i. Formally, a mapping from tree T_1 to tree T_2 is a triple (M_e, T_1, T_2) (or simply M_e if the context is clear), where M_e is any set of ordered pairs of integers (i, j) satisfying the following conditions:

(1) $1 \leq i \leq |T_1|$ and $1 \leq j \leq |T_2|$ where $|.|$ represents the size, i.e. the number of nodes, of the indicated tree;

(2) for any pair of (i_1, j_1) and (i_2, j_2) in M_e,

   (a) $i_1 = i_2$ iff $j_1 = j_2$ (one-to-one condition);

(b) $t_1[i_1]$ is to the left of $t_1[i_2]$ iff $t_2[j_1]$ is to the left of $t_2[j_2]$ (sibling order preservation condition);

(c) $t_1[i_1]$ is an ancestor of $t_1[i_2]$ iff $t_2[j_1]$ is an ancestor of $t_2[j_2]$ (ancestor order preservation condition).

Thus, for example, the mapping in Figure 2.2 is {(1, 1), (2, 2), (3, 3), (4, 5), (6, 6), (8, 8)}. Let $l(t[i])$ represent the label of node $t[i]$. Let M_e be a mapping from T_1 to T_2. Let I and J be the sets of nodes in T_1 and T_2, respectively, not touched by any mapping line in M_e. Then we can define the cost of M_e:

$$\gamma(M_e) = \sum_{(i,j)\in M_e} \gamma(l(t_1[i]) \rightarrow l(t_2[j])) + \sum_{i\in I} \gamma(l(t_1[i]) \rightarrow \lambda) + \sum_{j\in J} \gamma(\lambda \rightarrow l(t_2[j])).$$

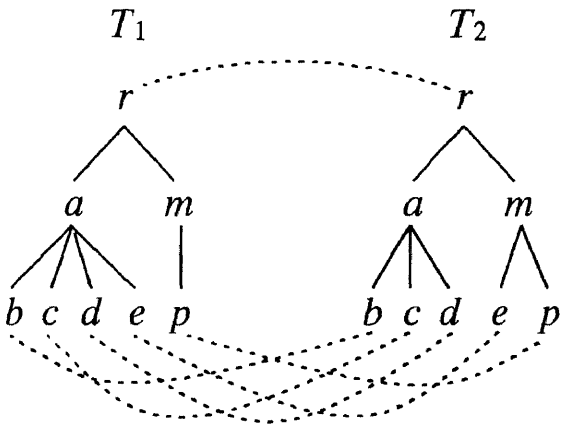


Figure 2.2 A mapping from tree T_1 to tree T_2 A dashed line from a node u in T_1 to a node v in T_2 indicates that u should be changed to v if $u \neq v$, or that u remains unchanged if $u = v$. The nodes of T_1 not touched by a dashed line are to be deleted and the nodes of T_2 not touched by a dashed line are to be inserted.

Given S, a sequence of edit operations from T_1 to T_2, it can be shown that there exists a mapping M_e from T_1 to T_2 such that $\gamma(M_e) \leq \gamma(S)$; conversely, for

any mapping M_e, there exists a sequence of edit operations S such that $\gamma(S) = \gamma(M_e)$ [69, Lemma 2].

Hence, we have

$$tdist_e(T_1, T_2) = \min \{\gamma(M_e) \mid M_e \text{ is a mapping from } T_1 \text{ to } T_2\}.$$

2.3 Selkow's Distance Measure

Selkow [38] proposed to impose the following restriction on the mapping conditions described in Section 2.2: if (i, j) is in M_e where neither $t_1[i]$ nor $t_2[j]$ is the root, then $(par(i), par(j))$[2] is also in M_e where $par(i)$ represents the postorder number of the parent of $t_1[i]$ and $par(j)$ represents the postorder number of the parent of $t_2[j]$ (parent-child order preservation). Due to its nature, we will refer to the resulting mapping as a *top-down edit mapping*. Figure 2.3 shows an example of a top-down edit mapping M_t from tree T_1 to tree T_2 given in Figure 2.2.

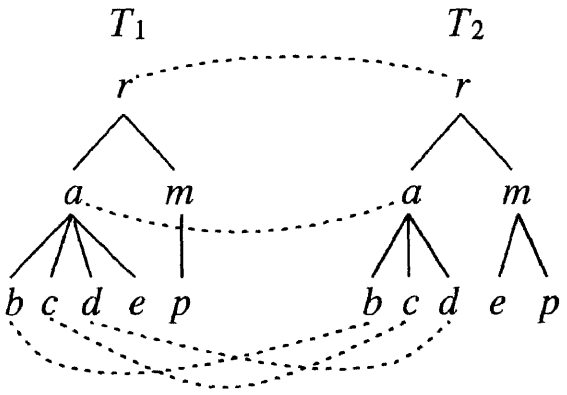


Figure 2.3 A top-down edit mapping from tree T_1 to tree T_2.

Three notes follow. First, the definition of top-down edit mappings implies that $(|T_1|, |T_2|)$ must be in M_t, i.e., the root of T_1 must be mapped to the root of T_2. Second, if a node n is to be deleted (inserted, respectively) in M_t, then the subtree

[2] $par(i)$ is undefined if $t[i]$ is the root.

rooted at n, if any, is to be deleted (inserted, respectively). Third, a top-down edit mapping is a mapping whereas a mapping may not satisfy the conditions of a top-down edit mapping. For example, the mapping in Figure 2.2 is *not* a top-down edit mapping since the nodes with labels a and m in T_1 are deleted while their children remain in the tree.

Let M_t be a top-down edit mapping from tree T_1 to tree T_2. Let I and J be the sets of nodes in T_1 and T_2, respectively, not touched by any mapping line in M_t. Then we can define the cost of M_t:

$$\gamma(M_t) = \sum_{(i,j)\in M_t} \gamma(l(t_1[i]) \rightarrow l(t_2[j])) + \sum_{i\in I} \gamma(l(t_1[i]) \rightarrow \lambda) + \sum_{j\in J} \gamma(\lambda \rightarrow l(t_2[j]))$$

The *top-down edit distance* from tree T_1 to tree T_2, denoted $tdist_t(T_1, T_2)$, is the cost of a minimum cost top-down edit mapping from T_1 to T_2. This restricted distance measure was originally proposed by Selkow [38] and will also be referred to as Selkow's distance, or simply the top-down distance, throughout the dissertation. Theorem 2.1 establishes the relationship between the edit distance of two trees T_1 and T_2 and their top-down edit distance.

Theorem 2.1. $tdist_e(T_1, T_2) \leq tdist_t(T_1, T_2)$.

Proof. By definition, $tdist_e(T_1, T_2)$ is the cost of a minimum cost mapping from T_1 to T_2 and $tdist_t(T_1, T_2)$ is the cost of a minimum cost top-down edit mapping from T_1 to T_2. Since any top-down edit mapping is also a mapping, the result follows. □

Intuitively, the top-down edit distance gives more weight to upper level nodes than to lower level nodes, i.e., the upper level nodes are more important than lower level ones. For example, in Figure 2.3, if we delete the node labeled a in T_1, then the subtree with the nodes labeled b, c, d, e must be removed, too. However, if we delete the node labeled, for example, d, then only d is removed. This is reasonable

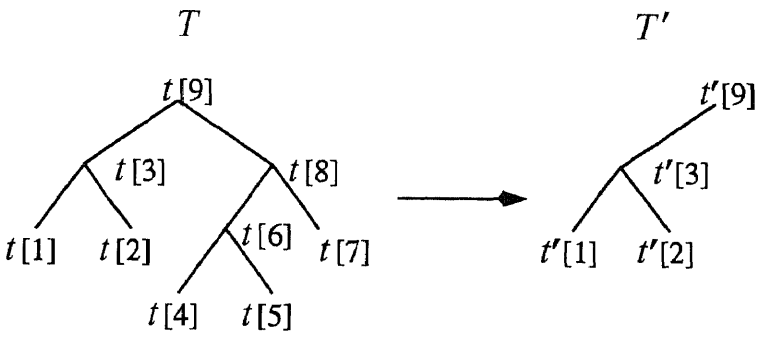


Figure 2.4 Cutting at node $t[8]$ to get T' from T.

when comparing two hierarchically structured documents or programs. For example, deleting a section of a document implies that one also removes all the paragraphs within the section whereas deleting a paragraph simply removes that paragraph, leaving the other paragraphs in the section intact.

2.4 Cut Operations

We define a substructure U of tree T to be a connected subgraph of T. That is, U is rooted at a node n in T and is generated by cutting off some subtrees in the subtree rooted at n. Formally, let $T[i]$ represent the subtree rooted at $t[i]$. The operation of cutting at node $t[i]$ means removing $T[i]$ (see Figure 2.4). A set S of nodes of $T[k]$ is said to be a set of consistent subtree cuts in $T[k]$ if (i) $t[i] \in S$ implies that $t[i]$ is a node in $T[k]$, and (ii) $t[i], t[j] \in S$ implies that neither is an ancestor of the other in $T[k]$. Intuitively, S is the set of all roots of the removed subtrees in $T[k]$.

We use $Cut(T, S)$ to represent the tree T with subtree removals at all nodes in S. Let $Subtrees(T)$ be the set of all possible sets of consistent subtree cuts in T. Given two trees T_1 and T_2 and an integer d, the size of the largest approximately common root-containing substructures within distance k, $0 \leq k \leq d$, of $T_1[i]$ and $T_2[j]$, denoted $tsize_\theta(T_1[i], T_2[j], k)$ (or simply $tsize_\theta(i, j, k)$ when the context is

clear), is

$$\max\{|Cut(T_1[i], S_1)| + |Cut(T_2[j], S_2)|\}$$

subject to

$$tdist_\theta(Cut(T_1[i], S_1), Cut(T_2[j], S_2)) \leq k$$

$$S_1 \in Subtrees(T_1[i])$$

$$S_2 \in Subtrees(T_2[j])$$

where θ is e (t, respectively) for the edit distance (top-down edit distance, respectively).

(**LACS[θ] Problem**) The LACS[θ] problem for $T_1[i]$ and $T_2[j]$ is to identify the largest approximately common substructure (LACS), within distance d, of $T_1[i]$ and $T_2[j]$, that is, to calculate $\max_{1\leq u\leq i, 1\leq v\leq j}\{tsize_\theta(T_1[u], T_2[v], d)\}$ and locate the $Cut(T_1[u], S_u)$ and $Cut(T_2[v], S_v)$, $S_u \in Subtrees(T_1[u])$, $S_v \in Subtrees(T_2[v])$ that achieve the maximum size. The LACS[θ] problem for T_1 and T_2 is to identify the largest approximately common substructure, within distance d, of T_1 and T_2, that is, to calculate $\max_{1\leq i\leq |T_1|, 1\leq j\leq |T_2|}\{tsize_\theta(T_1[i], T_2[j], d)\}$ and locate the substructure U_1 of T_1 and the substructure U_2 of T_2 that achieve the maximum size. Here the distance can be the edit distance (in which case $\theta = e$) or the top-dwon edit distance (in which case $\theta = t$).

We will focus on computing the maximum size in solving the LACS problem. By memorizing the size information during the computation and by a backtracking technique, one can find both the maximum size and one of the corresponding substructure pairs yielding the size in the same time and space complexity.

CHAPTER 3

FINDING PATTERNS IN TWO TREES BASED ON THE EDIT DISTANCE

In this chapter we first describe some basic properties and then present the dynamic programming algorithm to solve the LACS[e] problem, i.e. the problem for finding the largest approximately common substructure of two trees based on the edit distance.

3.1 Notation

We use $desc(i)$ to represent the set of postorder numbers of the descendants of the node $t[i]$ and $l(i)$ denotes the postorder number of the leftmost leaf of the subtree $T[i]$. When $T[i]$ is a leaf, $l(i) = i$. $T[i..j]$ is an ordered forest of tree T induced by the nodes numbered i to j inclusive (see Figure 3.1). If $i > j$, then $T[i..j] = \emptyset$. The definition of mappings for ordered forests is the same as for trees. Let F_1 and F_2 be two forests. The distance from F_1 to F_2, denoted $fdist_e(F_1, F_2)$, equals the cost of a minimum cost mapping from F_1 to F_2 [69].

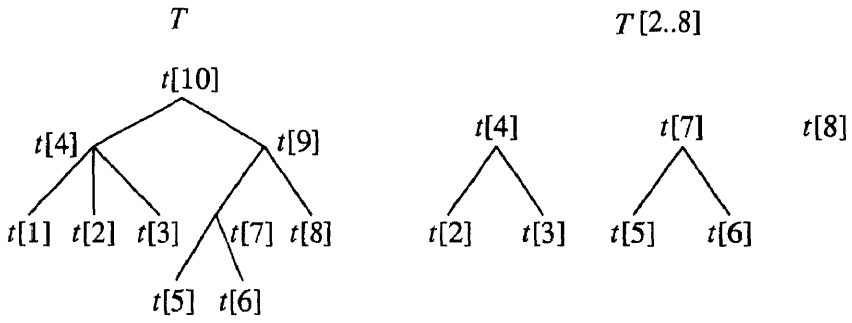


Figure 3.1 An induced ordered forest.

Let $F = T[i..j]$. A set S of nodes of F is said to be a set of consistent subtree cuts in F if (i) $t[p] \in S$ implies that $i \leq p \leq j$, and (ii) $t[p], t[q] \in S$ implies that neither is an ancestor of the other in F. We use $Cut(F, S)$ to represent

the sub-forest F with subtree removals at all nodes in S. Let $Subtrees(F)$ be the set of all possible sets of consistent subtree cuts in F. Define the size of the largest approximately common root-containing substructures, within distance k, of F_1 and F_2, denoted $fsize_e(F_1, F_2, k)$, to be $\max\{|Cut(F_1, S_1)|+|Cut(F_2, S_2)|\}$ subject to $fdist_e(Cut(F_1, S_1), Cut(F_2, S_2)) \leq k$, $S_1 \in Subtrees(F_1)$, $S_2 \in Subtrees(F_2)$. When $F_1 = T_1[l(i)..s]$ and $F_2 = T_2[l(j)..t]$, we also represent $fsize_e(F_1, F_2, k)$ by $fsize_e(l(i)..s, l(j)..t, k)$ if there is no confusion.

3.2 Basic Properties

Lemma 3.1. *Suppose $s \in desc(i)$ and $t \in desc(j)$. Then*

(i) $fsize_e(\emptyset, \emptyset, 0) = 0$;

(ii) $fsize_e(T_1[l(i)..s], \emptyset, 0) = 0$;

(iii) $fsize_e(\emptyset, T_2[l(j)..t], 0) = 0$.

Proof. Immediate from definitions. □

Lemma 3.2. *Suppose $s \in desc(i)$ and $t \in desc(j)$. Then for all k, $1 \leq k \leq d$,*

(i) $fsize_e(\emptyset, \emptyset, k) = 0$;

(ii)

$$fsize_e(T_1[l(i)..s], \emptyset, k) = \mathbf{max} \begin{cases} fsize_e(T_1[l(i)..s-1], \emptyset, k-1) + 1, \\ fsize_e(T_1[l(i)..l(s)-1], \emptyset, k); \end{cases}$$

(iii)

$$fsize_e(\emptyset, T_2[l(j)..t], k) = \mathbf{max} \begin{cases} fsize_e(\emptyset, T_2[l(j)..t-1], k-1) + 1, \\ fsize_e(\emptyset, T_2[l(j)..l(t)-1], k). \end{cases}$$

Proof. (i) follows from the definition. For (ii), suppose $S_1 \in Subtrees(T_1[l(i)..s])$ is a smallest set of consistent subtree cuts that maximizes $|Cut(T_1[l(i)..s], S_1)|$ where $fdist_e(Cut(T_1[l(i)..s], S_1), \emptyset) \leq k$. Then one of the following two cases must hold: (1) $t_1[s] \in S_1$; (2) $t_1[s] \notin S_1$. If (1) is true, then $fsize_e(T_1[l(i)..s], \emptyset, k) =$

$fsize_e(T_1[l(i)..l(s)-1], \emptyset, k)$; otherwise, $fsize_e(T_1[l(i)..s], \emptyset, k) = fsize_e(T_1[l(i)..s-1], \emptyset, k-1) + 1$. (iii) is proved similarly as for (ii). □

Lemma 3.3. *Suppose* $s \in desc(i)$ *and* $t \in desc(j)$. *If* $(l(s) \neq l(i)$ *or* $l(t) \neq l(j))$, *then*

$$fsize_e(l(i)..s, l(j)..t, 0) = \mathbf{max} \begin{cases} fsize_e(l(i)..l(s)-1, l(j)..t, 0), \\ fsize_e(l(i)..s, l(j)..l(t)-1, 0), \\ fsize_e(l(i)..l(s)-1, l(j)..l(t)-1, 0) \\ +tsize_e(s, t, 0). \end{cases}$$

Proof. Suppose $S_1 \in Subtrees(T_1[l(i)..s])$ and $S_2 \in Subtrees(T_2[l(j)..t])$ are two smallest sets of consistent subtree cuts that maximize $|Cut(T_1[l(i)..s], S_1)| + |Cut(T_2[l(j)..t], S_2)|$ where $fdist_e(Cut(T_1[l(i)..s], S_1), Cut(T_2[l(j)..t], S_2)) = 0$. Then at least one of the following cases must hold:

Case 1. $t_1[s] \in S_1$ (i.e., the subtree $T_1[s]$ is removed). So, $fsize_e(l(i)..s, l(j)..t, 0) = fsize_e(l(i)..l(s)-1, l(j)..t, 0)$.

Case 2. $t_2[t] \in S_2$ (i.e., the subtree $T_2[t]$ is removed). So, $fsize_e(l(i)..s, l(j)..t, 0) = fsize_e(l(i)..s, l(j)..l(t)-1, 0)$.

Case 3. $t_1[s] \notin S_1$ and $t_2[t] \notin S_2$ (i.e., neither $T_1[s]$ nor $T_2[t]$ is removed) (Figure 3.2). Let M_e be the mapping (with cost 0) from $Cut(T_1[l(i)..s], S_1)$ to $Cut(T_2[l(j)..t], S_2)$. In M_e, $T_1[s]$ must be mapped to $T_2[t]$ because otherwise we cannot have distance zero between $Cut(T_1[l(i)..s], S_1)$ and $Cut(T_2[l(j)..t], S_2)$. Therefore $fsize_e(l(i)..s, l(j)..t, 0) = fsize_e(l(i)..l(s)-1, l(j)..l(t)-1, 0)$ $+ tsize_e(s, t, 0)$.

Since these three cases exhaust all possible mappings yielding $fsize_e(l(i)..s, l(j)..t, 0)$, we take the maximum of the corresponding sizes, which gives the formula asserted by the lemma. □

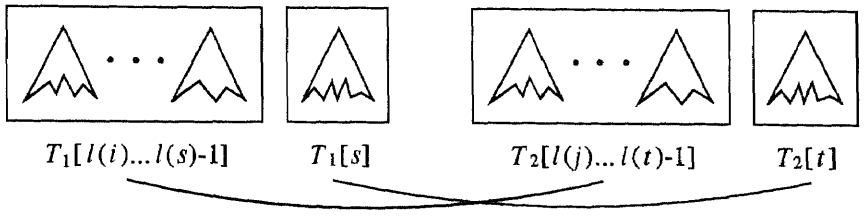


Figure 3.2 Illustration of the case in which one of $T_1[l(i)..s]$ and $T_2[l(j)..t]$ is a forest and neither $T_1[s]$ nor $T_2[t]$ is removed.

Lemma 3.4. *Suppose* $s \in desc(i)$ *and* $t \in desc(j)$. *Suppose both* $T_1[l(i)..s]$ *and* $T_2[l(j)..t]$ *are trees (i.e.,* $l(s) = l(i)$ *and* $l(t) = l(j)$*). Then*

$$fsize_e(l(i)..s, l(j)..t, 0) = \begin{cases} fsize_e(l(i)..s-1, l(j)..t-1, 0) + 2 & \text{if } t_1[s] = t_2[t], \\ 0 & \text{otherwise.} \end{cases}$$

Proof. Since $l(s) = l(i)$ and $l(t) = l(j)$, $T_1[l(i)..s] = T_1[s]$ and $T_2[l(j)..t] = T_2[t]$. First, consider the case where $t_1[s] = t_2[t]$. Suppose $S_1 \in Subtrees(T_1[s])$ and $S_2 \in Subtrees(T_2[t])$ are two smallest sets of consistent subtree cuts that maximize $|Cut(T_1[s], S_1)| + |Cut(T_2[t], S_2)|$ where $tdist_e(Cut(T_1[s], S_1), Cut(T_2[t], S_2)) = 0$. Let M_e be the mapping (with cost 0) from $Cut(T_1[s], S_1)$ to $Cut(T_2[t], S_2)$ (see Figure 3.3). Clearly, in M_e, $t_1[s]$ must be mapped to $t_2[t]$. Furthermore, the largest common root-containing substructure of $T_1[l(i)..s-1]$ and $T_2[l(j)..t-1]$ plus $t_1[s]$ (or $t_2[t]$) must be the largest common root-containing substructure of $T_1[s]$ and $T_2[t]$. This means that $fsize_e(l(i)..s, l(j)..t, 0) = fsize_e(l(i)..s-1, l(j)..t-1, 0) + 2$, where the 2 is obtained by including the two nodes $t_1[s]$ and $t_2[t]$.

Next consider the case where $t_1[s] \neq t_2[t]$ (i.e., the roots of the two trees $T_1[s]$ and $T_2[t]$ differ). In order to get distance zero between the two trees after applying cut operations to them, we have to remove both trees entirely. Thus, $fsize_e(l(i)..s, l(j)..t, 0) = 0$. □

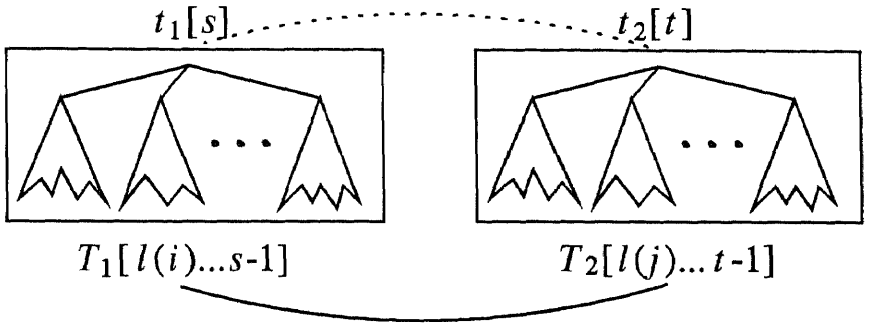


Figure 3.3 Illustration of the case in which both $T_1[l(i)..s]$ and $T_2[l(j)..t]$ are trees and $t_1[s] = t_2[t]$.

Lemma 3.5. *Suppose* $s \in desc(i)$ *and* $t \in desc(j)$. *If* $(l(s) \neq l(i)$ *or* $l(t) \neq l(j))$, *then for all* k, $1 \leq k \leq d$,

$$fsize_e(l(i)..s, l(j)..t, k) = \mathbf{max} \begin{cases} fsize_e(l(i)..l(s)-1, l(j)..t, k), \\ fsize_e(l(i)..s, l(j)..l(t)-1, k), \\ fsize_e(l(i)..s-1, l(j)..t, k-1)+1, \\ fsize_e(l(i)..s, l(j)..t-1, k-1)+1, \\ \max_{0\leq h\leq k}\{fsize_e(l(i)..l(s)-1, l(j)..l(t)-1, k-h) \\ +tsize_e(s,t,h)\}. \end{cases}$$

Proof. Suppose $S_1 \in Subtrees(T_1[l(i)..s])$ and $S_2 \in Subtrees(T_2[l(j)..t])$ are two smallest sets of consistent subtree cuts that maximize $|Cut(T_1[l(i)..s], S_1)| + |Cut(T_2[l(j)..t], S_2)|$ where $fdist_e(Cut(T_1[l(i)..s], S_1), Cut(T_2[l(j)..t], S_2)) \leq k$. Then at least one of the following cases must hold:

Case 1. $t_1[s] \in S_1$. So, $fsize_e(l(i)..s, l(j)..t, k) = fsize_e(l(i)..l(s)-1,\ l(j)..t,\ k)$.

Case 2. $t_2[t] \in S_2$. So, $fsize_e(l(i)..s, l(j)..t, k) = fsize_e(l(i)..s,\ l(j)..l(t)-1,\ k)$.

Case 3. $t_1[s] \notin S_1$ and $t_2[t] \notin S_2$. Let M_e be a minimum cost mapping from $Cut(T_1[l(i)..s], S_1)$ to $Cut(T_2[l(j)..t], S_2)$. There are three subcases to examine:

(a) $t_1[s]$ is not touched by a line in M_e. Then, $fsize_e(l(i)..s, l(j)..t, k) = fsize_e(l(i)..s-1,\ l(j)..t,\ k-1)+1$.

(b) $t_2[t]$ is not touched by a line in M_e. Then, $fsize_e(l(i)..s, l(j)..t, k) = fsize_e(l(i)..s,\ l(j)..t-1,\ k-1) + 1$.

(c) $t_1[s]$ and $t_2[t]$ are both touched by lines in M_e. Then $(s,t) \in M_e$. So, there exists an h such that $fsize_e(l(i)..s, l(j)..t, k) = fsize_e(l(i)..l(s)-1, l(j)..l(t)-1, k-h) + tsize_e(s,t,h)$. The value of h ranges from 0 to k. Therefore we take the maximum of the corresponding sizes, i.e., $fsize_e(l(i)..s, l(j)..t, k) = \max_{0\le h\le k}\{fsize_e(l(i)..l(s)-1, l(j)..l(t)-1, k-h) + tsize_e(s,t,h)\}$. □

Lemma 3.6. *Suppose $s \in desc(i)$ and $t \in desc(j)$. Suppose both $T_1[l(i)..s]$ and $T_2[l(j)..t]$ are trees (i.e., $l(s) = l(i)$ and $l(t) = l(j)$). Then for all k, $1 \le k \le d$,*

$$fsize_e(l(i)..s, l(j)..t, k) = \mathbf{max}\begin{cases} fsize_e(l(i)..s-1, l(j)..t, k-1)+1, \\ fsize_e(l(i)..s, l(j)..t-1, k-1)+1, \\ fsize_e(l(i)..s-1, l(j)..t-1, k-c)+2, \end{cases}$$

where

$$c = \begin{cases} 0 & \text{if } t_1[s] = t_2[t], \\ 1 & \text{otherwise.} \end{cases}$$

Proof. Since $T_1[l(i)..s]$ and $T_2[l(j)..t]$ are trees, $T_1[l(i)..s] = T_1[s]$ and $T_2[l(j)..t] = T_2[t]$. We first show that removing either $T_1[s]$ or $T_2[t]$ would not yield the maximum size. There are three cases to be considered:

Case 1. Both $T_1[s]$ and $T_2[t]$ are removed. Then, $fsize_e(l(i)..s, l(j)..t, k) = 0$. However, since $k \ge 1$, cutting at just the children of $t_1[s]$ and $t_2[t]$ would cause $fsize_e(l(i)..s, l(j)..t, k) \ge 2$. Therefore removing both $T_1[s]$ and $T_2[t]$ cannot yield the maximum size.

Case 2. Only $T_1[s]$ is removed. Then, $fsize_e(l(i)..s, l(j)..t, k) = tsize_e(\emptyset, T_2[t], k)$. Assume without loss of generality that $|T_2[t]| \ge k$. The above equation implies that we have to remove some subtrees from $T_2[t]$ so that there are no more than k nodes left in $T_2[t]$. Thus, $fsize_e(l(i)..s, l(j)..t, k) = tsize_e(\emptyset, T_2[t], k) = k$. On the other hand, if we just cut at the children of $t_1[s]$ and leave $t_1[s]$ in the tree, we would map $t_1[s]$ to

$t_2[t]$. This would lead to $fsize_e(l(i)..s, l(j)..t, k) \geq fsize_e(\emptyset, T_2[l(j)..t-1], k-1)+2$ $= k+1$. Thus, removing $T_1[s]$ alone cannot yield the maximum size.

Case 3. Only $T_2[t]$ is removed. The proof is similar to that in Case 2.

The above arguments lead to the conclusion that in order to obtain the maximum size, neither $T_1[s]$ nor $T_2[t]$ can be removed. Now suppose $S_1 \in Subtrees(T_1[s])$ and $S_2 \in Subtrees(T_2[t])$ are two smallest sets of consistent subtree cuts that maximize $|Cut(T_1[s], S_1)| + |Cut(T_2[t], S_2)|$ where $tdist_e(Cut(T_1[s], S_1), Cut(T_2[t], S_2)) \leq k$. Let M_e be a minimum cost mapping from $Cut(T_1[s], S_1)$ to $Cut(T_2[t], S_2)$. Then at least one of the following cases must hold:

Case 1. $t_1[s]$ is not touched by a line in M_e. So, $fsize_e(l(i)..s, l(j)..t, k) =$ $fsize_e(l(i)..s-1,\ l(j)..t,\ k-1)+1$.

Case 2. $t_2[t]$ is not touched by a line in M_e. So, $fsize_e(l(i)..s, l(j)..t, k) =$ $fsize_e(l(i)..s,\ l(j)..t-1,\ k-1)+1$.

Case 3. Both $t_1[s]$ and $t_2[t]$ are touched by lines in M_e. By the ancestor order preservation and sibling order preservation conditions on mappings (cf. Section 2.1), (s,t) must be in M_e. Thus, if $t_1[s] = t_2[t]$, we have $fsize_e(l(i)..s, l(j)..t, k) =$ $fsize_e(l(i)..s-1, l(j)..t-1, k)+2$; otherwise, since mapping $t_1[s]$ to $t_2[t]$ costs 1, we have $fsize_e(l(i)..s, l(j)..t, k) = fsize_e(l(i)..s-1, l(j)..t-1, k-1)+2$. □

3.3 The Algorithm

From Lemma 3.4 and Lemma 3.6, we observe that when s is on the path from $l(i)$ to i and t is on the path from $l(j)$ to j, we need not compute $tsize_e(s,t,k)$, $0 \leq k \leq d$, separately, since they can be obtained during the computation of $tsize_e(i,j,k)$. Thus, we will only consider nodes that are either the roots of the trees or having a left sibling. Let keynodes(T) contain all such nodes of a tree T, i.e., keynodes(T) = $\{k|$there exists no $k' > k$ such that $l(k) = l(k')\}$. For each $i \in$ keynodes(T_1) and $j \in$ keynodes(T_2), Procedure Find-Largest-1 in Figure 3.4 computes

Procedure Find-Largest-1

Input: $i, j, 0$.
Output: $tsize_e(s, t, 0)$ where $l(i) \leq s \leq i$ and $l(j) \leq t \leq j$.

```
fsize_e(∅, ∅, 0) := 0;
for s := l(i) to i do
   fsize_e(T1[l(i)..s], ∅, 0) := 0;
for t := l(j) to j do
   fsize_e(∅, T2[l(j)..t], 0) := 0;
for s := l(i) to i do
   for t := l(j) to j do
      if (l(s) ≠ l(i) or l(t) ≠ l(j)) then
         compute fsize_e(l(i)..s, l(j)..t, 0) as in Lemma 3.3;
      else begin      /* l(s) = l(i) and l(t) = l(j) */
         compute fsize_e(l(i)..s, l(j)..t, 0) as in Lemma 3.4;
         tsize_e(s, t, 0) := fsize_e(l(i)..s, l(j)..t, 0);
      end;
```

Figure 3.4 Algorithm for computing $tsize_e(s, t, 0)$.

$tsize_e(s, t, 0)$ for $l(i) \leq s \leq i$ and $l(j) \leq t \leq j$ and Procedure Find-Largest-2 in Figure 3.5 computes $tsize_e(s, t, k)$ for $1 \leq k \leq d$. The main algorithm is summarized in Figure 3.6.

Now, to calculate the size of the largest approximately common substructure (LACS[e]), within distance d, of $T_1[i]$ and $T_2[j]$, we build, in a bottom-up fashion, another array $\gamma(i, j, d)$, $1 \leq i \leq |T_1|$, $1 \leq j \leq |T_2|$, using $tsize_e(i, j, d)$ as follows. Let $L = \max_{1 \leq u \leq s} \gamma(i_u, j, d)$ where $i_1 \ldots i_s$ are the postorder numbers of the children of $t_1[i]$ or $L = 0$ if $t_1[i]$ is a leaf. Let $R = \max_{1 \leq v \leq t} \gamma(i, j_v, d)$ where $j_1 \ldots j_t$ are the postorder numbers of the children of $t_2[j]$ or $R = 0$ if $t_2[j]$ is a leaf. Calculate $\gamma(i, j, d) = \max\{tsize_e(i, j, d), L, R\}$. The size of LACS[e], within distance d, of $T_1[i]$ and $T_2[j]$ is $\gamma(i, j, d)$. The size of LACS[e], within distance d, of T_1 and T_2 is $\gamma(|T_1|, |T_2|, d)$.

Procedure Find-Largest-2

Input: i, j, d.
Output: $tsize_e(s,t,k)$ where $l(i) \leq s \leq i$, $l(j) \leq t \leq j$ and $1 \leq k \leq d$.

```
for k := 1 to d do
   fsize_e(∅, ∅, k) := 0;
for k := 1 to d do
   for s := l(i) to i do
      compute fsize_e(T_1[l(i)..s], ∅, k) as in Lemma 3.2 (ii);
for k := 1 to d do
   for t := l(j) to j do
      compute fsize_e(∅, T_2[l(j)..t], k) as in Lemma 3.2 (iii);
for k := 1 to d do
   for s := l(i) to i do
      for t := l(j) to j do
         if (l(s) ≠ l(i) or l(t) ≠ l(j)) then
            compute fsize_e(l(i)..s, l(j)..t, k) as in Lemma 3.5;
         else begin      /* l(s) = l(i) and l(t) = l(j) */
            compute fsize_e(l(i)..s, l(j)..t, k) as in Lemma 3.6;
            tsize_e(s, t, k) := fsize_e(l(i)..s, l(j)..t, k);
         end;
```

Figure 3.5 Algorithm for computing $tsize_e(s,t,k)$.

Algorithm Find-Largest[e]

Input: Trees T_1, T_2 and an integer d.
Output: $tsize_e(i,j,k)$ where $1 \leq i \leq |T_1|$, $1 \leq j \leq |T_2|$ and $0 \leq k \leq d$.

```
for i' := 1 to |keynodes(T_1)| do
   for j' := 1 to |keynodes(T_2)| do
      begin
         i := keynodes(T_1)[i'];
         j := keynodes(T_2)[j'];
         run Procedure Find-Largest-1 on input (i, j, 0);
         run Procedure Find-Largest-2 on input (i, j, d);
      end;
```

Figure 3.6 Algorithm for computing $tsize_e(i,j,k)$.

Theorem 3.1. *The space complexity of the algorithm is* $O(d \times |T_1| \times |T_2|)$ *and the time complexity of the algorithm is* $O(d^2 \times |T_1| \times |T_2| \times min(H_1, L_1) \times min(H_2, L_2))$.

Proof. We use an array to hold $fsize_e$, $tsize_e$ and γ, respectively. These arrays require $O(d \times |T_1| \times |T_2|)$ space.

Given $tsize_e(i, j, d)$, $1 \leq i \leq |T_1|$, $1 \leq j \leq |T_2|$, calculating $\gamma(i, j, d)$ requires $O(|T_1| \times |T_2|)$ time. For a fixed i and j, Procedure Find-Largest-1 requires $O(|T_1[i]| \times |T_2[j]|)$ time and Procedure Find-Largest-2 requires $O(d^2 \times |T_1[i]| \times |T_2[j]|)$ time. So the total time is bounded by

$$\begin{aligned}
& \sum_{i \in \text{keynodes}(T_1)} \sum_{j \in \text{keynodes}(T_2)} O(|T_1[i]| \times |T_2[j]|) + O(d^2 \times |T_1[i]| \times |T_2[j]|) \\
\leq\ & O(\sum_{i \in \text{keynodes}(T_1)} \sum_{j \in \text{keynodes}(T_2)} d^2 \times |T_1[i]| \times |T_2[j]|) \\
\leq\ & O(d^2 \times \sum_{i \in \text{keynodes}(T_1)} |T_1[i]| \times \sum_{j \in \text{keynodes}(T_2)} |T_2[j]|).
\end{aligned}$$

From [69, Theorem 2], the last term above is bounded by $O(d^2 \times |T_1| \times |T_2| \times \min(H_1, L_1) \times \min(H_2, L_2))$ where H_i, $i = 1, 2$, is the height of T_i and L_i is the number of leaves in T_i. □

When d is a constant, this is the same as the complexity of the best current algorithm for tree matching based on the edit distance [34, 69], even though the problem at hand appears to be harder than tree matching.

Note that to calculate $\max_{1 \leq i \leq |T_1|, 1 \leq j \leq |T_2|}\{tsize_e(i, j, 0)\}$, one could use a faster algorithm that runs in time $O(|T_1| \times |T_2|)$. However, the reason for considering the keynodes and the formulas as specified in Lemmas 3.3 and 3.4 is to prepare the optimal sizes from forests to forests and store these size values in the array to be used in calculating $tsize_e(s, t, k)$ for $k \neq 0$. Even if one could incorporate the faster algorithm into the Find-Largest[e] algorithm, the overall time complexity would not be changed, because the calculation of $tsize_e(s, t, k)$ for $k \neq 0$ dominates the cost.

CHAPTER 4

FINDING PATTERNS IN TWO TREES BASED ON THE TOP-DOWN EDIT DISTANCE

We will first describe some basic properties and then present the algorithm for solving the LACS[t] problem, i.e. the problem for finding the largest approximately common substructure of two trees based on the top-down edit distance.

4.1 Notation

We use $F[i]$ to represent the ordered forest obtained by deleting $t[i]$ from $T[i]$ (Figure 4.1). $|F|$ is the size of forest F. A set S of nodes of F is said to be a set of consistent subtree cuts in F if (i) $t[p] \in S$ implies that $t[p]$ is a node in F, and (ii) $t[p], t[q] \in S$ implies that neither is an ancestor of the other in F. We use $Cut(F, S)$ to represent the subforest F with subtree removals at all nodes in S. For example, consider Figure 4.1 again. $S = \{t[2], t[5], t[7]\}$ is a set of consistent subtree cuts in $F[9]$. $Cut(F[9], S)$ is obtained by cutting at nodes $t[2]$, $t[5]$ and $t[7]$ from $F[9]$.

Let $Subtrees(F)$ be the set of all possible sets of consistent subtree cuts in F. Let $fdist_t(F_1, F_2)$ be the top-down edit distance from forest F_1 to forest F_2. We

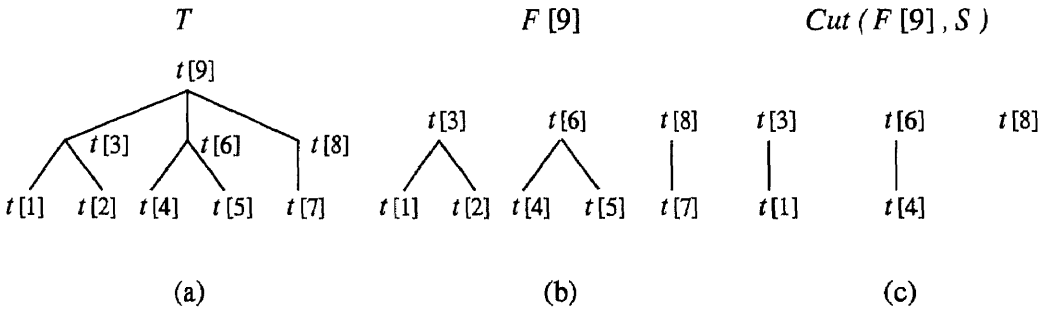


Figure 4.1 (a) A tree T. (b) Deleting node $t[9]$ to get $F[9]$ from T. (c) Cutting at nodes in $S = \{t[2], t[5], t[7]\}$ from $F[9]$.

define the size of the largest approximately common root-containing substructures, within distance k, of F_1 and F_2, denoted $fsize_t(F_1, F_2, k)$, as

$$\max\{|Cut(F_1, S_1)| + |Cut(F_2, S_2)|\}$$

subject to

$$fdist_t(Cut(F_1, S_1), Cut(F_2, S_2)) \leq k$$
$$S_1 \in Subtrees(F_1)$$
$$S_2 \in Subtrees(F_2)$$

4.2 Basic Properties

Let $t_1[i]$ be a node in tree T_1 and let $t_2[j]$ be a node in tree T_2. We use $t_1[i_1]$, $t_1[i_2]$, ..., $t_1[i_{m_i}]$ to represent the children of $t_1[i]$ and use $t_2[j_1], t_2[j_2], \ldots, t_2[j_{n_j}]$ to represent the children of $t_2[j]$. $F_1[i_s, i_t]$, $1 \leq s \leq t \leq m_i$, represents the forest consisting of the subtrees $T_1[i_s], \ldots, T_1[i_t]$. $F_1[i_1, i_{m_i}] = F_1[i]$; $F_1[i_p, i_{p-1}] = \emptyset$ for all $1 \leq p \leq m_i$. Note $F_1[i_p, i_p] = T_1[i_p] \neq F_1[i]$. Lemma 4.1 and Lemma 4.2 below summarize the formulas used for initializing $tsize_t$ and $fsize_t$ for all k, $0 \leq k \leq d$.

Lemma 4.1. *For all i, j, $1 \leq i \leq |T_1|$ and $1 \leq j \leq |T_2|$, and for any p, q, s, t, $1 \leq p \leq s \leq m_i$ and $1 \leq q \leq t \leq n_j$,*

(i) (a) $tsize_t(\emptyset, \emptyset, 0) = 0$,

(b) $fsize_t(\emptyset, \emptyset, 0) = 0$;

(ii)(a) $tsize_t(T_1[i], \emptyset, 0) = 0$,

(b) $tsize_t(\emptyset, T_2[j], 0) = 0$;

(iii) (a) $fsize_t(F_1[i_p, i_s], \emptyset, 0) = 0$,

(b) $fsize_t(\emptyset, F_2[j_q, j_t], 0) = 0$.

Proof. Immediate from definitions. □

Lemma 4.2. *For all* i, j, $1 \leq i \leq |T_1|$ *and* $1 \leq j \leq |T_2|$, *and for any* p, q, s, t, $1 \leq p \leq s \leq m_i$ *and* $1 \leq q \leq t \leq n_j$, *and for any* k, $1 \leq k \leq d$,

(i) (a) $tsize_t(\emptyset, \emptyset, k) = 0$,

(b) $fsize_t(\emptyset, \emptyset, k) = 0$;

(ii)(a) $tsize_t(T_1[i], \emptyset, k) = fsize_t(F_1[i], \emptyset, k-1) + 1$,

(b) $tsize_t(\emptyset, T_2[j], k) = fsize_t(\emptyset, F_2[j], k-1) + 1$;

(iii) (a) $fsize_t(F_1[i_p, i_s], \emptyset, k) = \mathbf{min}\{|F_1[i_p, i_s]|, k\}$,

(b) $fsize_t(\emptyset, F_2[j_q, j_t], k) = \mathbf{min}\{|F_2[j_q, j_t]|, k\}$.

Proof. (i) is obvious since the largest approximately common root-containing substructure of two empty trees or forests is empty. For (ii), in (a), the largest approximately common root-containing substructure of $F_1[i]$ and $\emptyset$ plus $t_1[i]$ is the largest approximately common root-containing substructure of $T_1[i]$ and $\emptyset$. This means that $tsize_t(T_1[i], \emptyset, k) = fsize_t(F_1[i], \emptyset, k-1) + 1$ where the 1 is obtained by including the node $t_1[i]$. (b) is proved similarly as for (a).

For (iii), in (a), if $k \leq |F_1[i_p, i_s]|$, since the allowed distance is at most k, we have to remove some subtrees from $F_1[i_p, i_s]$ until only k nodes are left in the forest. If $k > |F_1[i_p, i_s]|$, then all the nodes in $F_1[i_p, i_s]$ constitute the largest approximately common root-containing substructure of $F_1[i_p, i_s]$ and $\emptyset$. Thus the result follows. (b) is proved similarly as for (a). □

Lemma 4.3. *For all* i, j, $1 \leq i \leq |T_1|$ *and* $1 \leq j \leq |T_2|$, *and for any* s, t, $1 \leq s \leq m_i$ *and* $1 \leq t \leq n_j$,

$$fsize_t(F_1[i_1, i_s], F_2[j_1, j_t], 0) = \mathbf{max} \begin{cases} fsize_t(F_1[i_1, i_{s-1}], F_2[j_1, j_t], 0) \\ fsize_t(F_1[i_1, i_s], F_2[j_1, j_{t-1}], 0) \\ fsize_t(F_1[i_1, i_{s-1}], F_2[j_1, j_{t-1}], 0) \\ +tsize_t(i_s, j_t, 0) \end{cases}$$

Proof. Suppose $S_1 \in Subtrees(F_1[i_1, i_s])$ and $S_2 \in Subtrees(F_2[j_1, j_t])$ are two smallest sets of consistent subtree cuts that maximize $|Cut(F_1[i_1, i_s], S_1)| +$

$|Cut(F_2[j_1, j_t], S_2)|$ where $fdist_t(Cut\ (F_1[i_1, i_s],\ S_1),\ Cut(F_2[j_1, j_t],\ S_2)) = 0$. Then at least one of the following cases must hold:

Case 1. $t_1[i_s] \in S_1$ (i.e., the subtree $T_1[i_s]$ is removed). So,

$fsize_t(F_1[i_1, i_s], F_2[j_1, j_t], 0) = fsize_t(F_1[i_1, i_{s-1}], F_2[j_1, j_t], 0)$.

Case 2. $t_2[j_t] \in S_2$ (i.e., the subtree $T_2[j_t]$ is removed). So,

$fsize_t(F_1[i_1, i_s], F_2[j_1, j_t], 0) = fsize_t(F_1[i_1, i_s], F_2[j_1, j_{t-1}], 0)$.

Case 3. $t_1[i_s] \notin S_1$ and $t_2[j_t] \notin S_2$ (i.e., neither $T_1[i_s]$ nor $T_2[j_t]$ is removed) (Figure 4.2). Let M_t be the top-down edit mapping (with cost 0) from $Cut(F_1[i_1, i_s], S_1)$ to $Cut(F_2[j_1, j_t], S_2)$. In M_t, $T_1[i_s]$ must be mapped to $T_2[j_t]$ because otherwise we cannot have distance zero between $Cut(F_1[i_1, i_s], S_1)$ and $Cut(F_2[j_1, j_t], S_2)$. Therefore $fsize_t(F_1[i_1, i_s], F_2[j_1, j_t], 0) = fsize_t(F_1[i_1, i_{s-1}], F_2[j_1, j_{t-1}], 0) + tsize_t(i_s, j_t, 0)$.

Since these three cases exhaust all possible mappings yielding $fsize_t(F_1[i_1, i_s], F_2[j_1, j_t], 0)$, we take the maximum of the corresponding sizes, which gives the formula asserted by the lemma. □

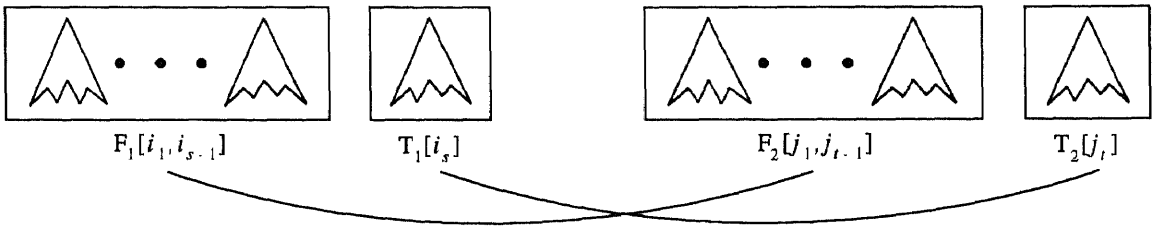


Figure 4.2 Illustration of the case in which one of $F_1[i_1, i_s]$ and $F_2[j_1, j_t]$ is a forest and neither $T_1[i_s]$ nor $T_2[j_t]$ is removed.

Lemma 4.4. *For all* i, j, $1 \le i \le |T_1|$ *and* $1 \le j \le |T_2|$,

$$tsize_t(i, j, 0) = \begin{cases} fsize_t(F_1[i], F_2[j], 0) + 2 & \text{if } l(t_1[i]) = l(t_2[j]) \\ 0 & \text{otherwise} \end{cases}$$

Proof. First, consider the case where $l(t_1[i]) = l(t_2[j])$. Suppose $S_1 \in Subtrees(T_1[i])$ and $S_2 \in Subtrees(T_2[j])$ are two smallest sets of consistent subtree cuts that

maximize $|Cut(T_1[i], S_1)| + |Cut(T_2[j], S_2)|$ where $tdist_t(Cut(T_1[i], S_1), Cut(T_2[j], S_2)) = 0$. Let M_t be the top-down edit mapping (with cost 0) from $Cut(T_1[i], S_1)$ to $Cut(T_2[j], S_2)$ (Figure 4.3). In M_t, $t_1[i]$ must be mapped to $t_2[j]$. Here is why. Suppose (i, k) and (h, j) are in M_t. Since $t_1[i]$ is the root of $T_1[i]$, $t_1[h]$ must be a proper descendant of $t_1[i]$. This implies that $t_2[j]$ is a proper descendant of $t_2[k]$ by the ancestor order preservation condition on mappings described in Section 2.2. This is impossible because $t_2[j]$ is the root of $T_2[j]$. So, $h = i$. By symmetry, $k = j$ and hence $(i, j) \in M_t$.

Next, note that the largest common root-containing substructure of $F_1[i]$ and $F_2[j]$ plus $t_1[i]$ (or $t_2[j]$) must be the largest common root-containing substructure of $T_1[i]$ and $T_2[j]$. This means that $tsize_t(i, j, 0) = fsize_t(F_1[i], F_2[j], 0) + 2$, where the 2 is obtained by including the two nodes $t_1[i]$ and $t_2[j]$.

Finally consider the case where $l(t_1[i]) \neq l(t_2[j])$ (i.e., the labels of the roots of the two trees $T_1[i]$ and $T_2[j]$ differ). In order to get distance zero between the two trees after applying cut operations to them, we have to remove both trees entirely. Thus, $tsize_t(i, j, 0) = 0$. □

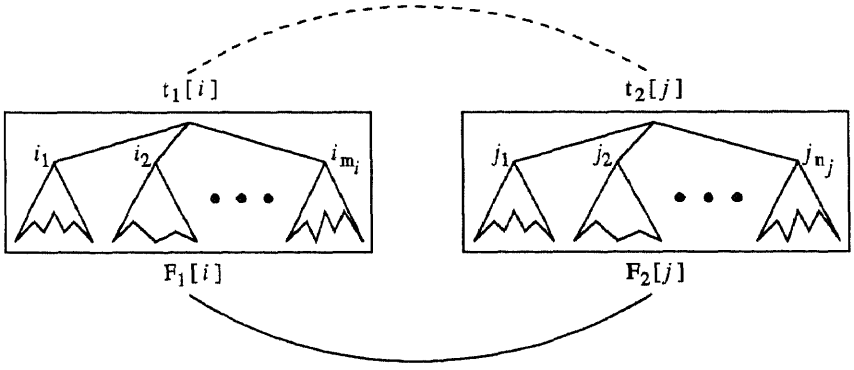


Figure 4.3 Illustration of the case in which we have two trees $T_1[i]$ and $T_2[j]$ and $l(t_1[i]) = l(t_2[j])$.

The following two lemmas show the recurrence equations for calculating $fsize_t$ and $tsize_t$ when the allowed distance $k > 0$.

Lemma 4.5. *For all* i, j, $1 \leq i \leq |T_1|$ *and* $1 \leq j \leq |T_2|$, *and for any* s, t, $1 \leq s \leq m_i$ *and* $1 \leq t \leq n_j$, *and for any* k, $1 \leq k \leq d$,

$$fsize_t(F_1[i_1, i_s], F_2[j_1, j_t], k) = \mathbf{max} \begin{cases} fsize_t(F_1[i_1, i_{s-1}], F_2[j_1, j_t], k), \\ fsize_t(F_1[i_1, i_s], F_2[j_1, j_{t-1}], k), \\ \max_{0 \leq h \leq k}\{fsize_t(F_1[i_1, i_{s-1}], F_2[j_1, j_t], k-h) \\ \qquad +tsize_t(T_1[i_s], \emptyset, h)\}, \\ \max_{0 \leq h \leq k}\{fsize_t(F_1[i_1, i_s], F_2[j_1, j_{t-1}], k-h) \\ \qquad +tsize_t(\emptyset, T_2[j_t], h)\}, \\ \max_{0 \leq h \leq k}\{fsize_t(F_1[i_1, i_{s-1}], F_2[j_1, j_{t-1}], k- \\ \qquad h) + tsize_t(i_s, j_t, h)\}. \end{cases}$$

Proof. Suppose $S_1 \in Subtrees(F_1[i_1, i_s])$ and $S_2 \in Subtrees(F_2[j_1, j_t])$ are two smallest sets of consistent subtree cuts that maximize $|Cut(F_1[i_1, i_s], S_1)| + |Cut(F_2[j_1, j_t], S_2)|$ where $fdist_t(Cut\ (F_1[i_1,\ i_s],\ S_1),\ Cut(F_2[j_1,\ j_t],\ S_2)) \leq k$. Then, at least one of the following cases must hold:

Case 1. $t_1[i_s] \in S_1$ (i.e., $T_1[i_s]$ is removed). So, $fsize_t(F_1[i_1, i_s], F_2[j_1, j_t], k) = fsize_t(F_1[i_1, i_{s-1}],\ F_2[j_1, j_t],\ k)$.

Case 2. $t_2[j_t] \in S_2$ (i.e., $T_2[j_t]$ is removed). So, $fsize_t(F_1[i_1, i_s], F_2[j_1, j_t], k) = fsize_t(F_1[i_1, i_s],\ F_2[j_1, j_{t-1}],\ k)$.

Case 3. $t_1[i_s] \notin S_1$ and $t_2[j_t] \notin S_2$ (i.e., neither $T_1[i_s]$ nor $T_2[j_t]$ is removed). Let M_t be a minimum cost top-down edit mapping from $Cut(F_1[i_1, i_s], S_1)$ to $Cut(F_2[j_1, j_t], S_2)$. There are three subcases to be considered:

(a) $t_1[i_s]$ is not touched by a line in M_t. So, $T_1[i_s]$ must be mapped to an empty tree. We may remove nodes from $T_1[i_s]$ until only h nodes are left where $0 \leq h \leq k$. Therefore $fsize_t(F_1[i_1, i_s], F_2[j_1, j_t], k) = \max_{0 \leq h \leq k}\{fsize_t(F_1[i_1, i_{s-1}], F_2[j_1, j_t], k - h) + tsize_t(T_1[i_s], \emptyset, h)\}$.

(b) $t_2[j_t]$ is not touched by a line in M_t. So, $fsize_t(F_1[i_1, i_s], F_2[j_1, j_t], k) = \max_{0 \leq h \leq k}\ \{fsize_t(F_1[i_1,\ i_s],\ F_2[j_1,\ j_{t-1}],\ k - h) + tsize_t(\emptyset,\ T_2[j_t],\ h)\}$.

(c) $t_1[i_s]$ and $t_2[j_t]$ are both touched by lines in M_t. Then, $T_1[i_s]$ must be mapped to $T_2[j_t]$. The allowed distance h between $T_1[i_s]$ and $T_2[j_t]$ can vary from 0 to k. So, $fsize_t(F_1[i_1, i_s], F_2[j_1, j_t], k) = \max_{0 \le h \le k} \{fsize_t(F_1[i_1, i_{s-1}], F_2[j_1, j_{t-1}], k-h) + tsize_t(i_s, j_t, h)\}$. □

Lemma 4.6. *For all* i, j, $1 \le i \le |T_1|$ *and* $1 \le j \le |T_2|$, *and for any* k, $1 \le k \le d$,

$$tsize_t(i, j, k) = fsize_t(F_1[i], F_2[j], k - c) + 2$$

where

$$c = \begin{cases} 0 & \text{if } l(t_1[i]) = l(t_2[j]) \\ 1 & \text{otherwise} \end{cases}$$

Proof. We first show that removing either $T_1[i]$ or $T_2[j]$ would not yield the maximum size. There are three cases to be considered:

Case 1. Both $T_1[i]$ and $T_2[j]$ are removed. Then $tsize_t(i, j, k) = 0$. However, since $k \ge 1$, cutting at just the children of $t_1[i]$ and $t_2[j]$ would cause $tsize_t(i, j, k) \ge 2$. Therefore removing both $T_1[i]$ and $T_2[j]$ cannot yield the maximum size.

Case 2. Only $T_1[i]$ is removed. Then $tsize_t(i, j, k) = tsize_t(\emptyset, T_2[j], k)$. Assume without loss of generality that $|T_2[j]| \ge k$. The above equation implies that we have to remove some subtrees from $T_2[j]$ so that there are no more than k nodes left in $T_2[j]$. Thus, $tsize_t(i, j, k) = tsize_t(\emptyset, T_2[j], k) = k$. On the other hand, if we just cut at the children of $t_1[i]$ and leave $t_1[i]$ in the tree, we would map $t_1[i]$ to $t_2[j]$. This would lead to $tsize_t(i, j, k) \ge fsize_t(\emptyset, F_2[j], k-1) + 2 = k + 1$. Thus, removing $T_1[i]$ alone cannot yield the maximum size.

Case 3. Only $T_2[j]$ is removed. The proof is similar to that in Case 2.

The above arguments lead to the conclusion that in order to obtain the maximum size, neither $T_1[i]$ nor $T_2[j]$ can be removed. Now suppose $S_1 \in Subtrees(T_1[i])$ and $S_2 \in Subtrees(T_2[j])$ are two smallest sets of consistent

Procedure Find-Largest[t]
Input: Trees T_1, T_2 and an integer d.
Output: $tsize_t(i,j,k)$ where $1 \leq i \leq |T_1|$, $1 \leq j \leq |T_2|$ and $0 \leq k \leq d$.
initialize $fsize_t()$ and $tsize_t()$ for $0 \leq k \leq d$ as in Lemma 4.1 and Lemma 4.2;
for $i := 1$ **to** $|T_1|$ **do**
  for $j := 1$ **to** $|T_2|$ **do begin**
    for $s := 1$ **to** m_i **do**
      for $t := 1$ **to** n_j **do**
        compute $fsize_t(F_1[i_1, i_s], F_2[j_1, j_t], 0)$ as in Lemma 4.3;
    compute $tsize_t(i,j,0)$ as in Lemma 4.4;
  end;
for $k := 1$ **to** d **do**
  for $i := 1$ **to** $|T_1|$ **do**
    for $j := 1$ **to** $|T_2|$ **do begin**
      for $s := 1$ **to** m_i **do**
        for $t := 1$ **to** n_j **do**
          compute $fsize_t(F_1[i_1, i_s], F_2[j_1, j_t], k)$ as in Lemma 4.5;
      compute $tsize_t(i,j,k)$ as in Lemma 4.6;
    end;

Figure 4.4 Procedure for computing $tsize_t(i,j,k)$.

subtree cuts that maximize $|Cut(T_1[i], S_1)| + |Cut(T_2[j], S_2)|$ where $tdist_t(Cut(T_1[i], S_1), Cut(T_2[j], S_2)) \leq k$. Let M_t be a minimum cost top-down edit mapping from $Cut(T_1[i], S_1)$ to $Cut(T_2[j], S_2)$. By the mapping conditions, $t_1[i]$ must be mapped to $t_2[j]$ in M_t. Thus, if $l(t_1[i]) = l(t_2[j])$, $tsize_t(i,j,k) = fsize_t(F_1[i], F_2[j], k) + 2$; otherwise $tsize_t(i,j,k) = fsize_t(F_1[i], F_2[j], k-1) + 2$. □

4.3 The Algorithm

We use dynamic programming to compute $tsize_t(i,j,k)$. The $fsize_t$ values computed and used in the algorithm are stored in a temporary array that is freed once the corresponding $tsize_t$ is computed. The $tsize_t$ values are stored in a permanent array. Figure 4.4 summarizes the procedure for computing $tsize_t(i,j,k)$ for $0 \leq k \leq d$.

Note that $tsize_t(i,j,k)$ represents the size of the largest approximately common root-containing substructures, within distance k, of $T_1[i]$ and $T_2[j]$. To calculate the size of the largest approximately common substructure (LACS[t]), within distance d, of $T_1[i]$ and $T_2[j]$, we build, in a bottom-up fashion, another array $\beta(i,j,d)$, $1 \leq i \leq |T_1|$, $1 \leq j \leq |T_2|$, using $tsize_t(i,j,d)$ as follows. Let $L = \max_{1 \leq u \leq m_i} \beta(i_u, j, d)$ where $i_1 \ldots i_{m_i}$ are the postorder numbers of the children of $t_1[i]$ or $L = 0$ if $t_1[i]$ is a leaf. Let $R = \max_{1 \leq v \leq n_j} \beta(i, j_v, d)$ where $j_1 \ldots j_{n_j}$ are the postorder numbers of the children of $t_2[j]$ or $R = 0$ if $t_2[j]$ is a leaf. Calculate $\beta(i,j,d) = \max\{tsize_t(i,j,d), L, R\}$. The size of the largest approximately common substructure (LACS[t]), within distance d, of $T_1[i]$ and $T_2[j]$ is $\beta(i,j,d)$. The size of the largest approximately common substructure (LACS[t]), within distance d, of T_1 and T_2 is $\beta(|T_1|, |T_2|, d)$.

Theorem 4.1. *The algorithm presented above correctly solves the LACS[t] problem. The space complexity of the algorithm is* $O(d \times |T_1| \times |T_2|)$ *and the time complexity of the algorithm is* $O(d^2 \times |T_1| \times |T_2|)$.

Proof. The correctness of the algorithm follows by observing that Procedure Find-Largest[t] correctly computes $tsize_t(i,j,k)$, $1 \leq i \leq |T_1|$, $1 \leq j \leq |T_2|$, $0 \leq k \leq d$, and the bottom-up procedure described above correctly computes $\beta(i,j,d)$. We use an array to hold $fsize_t$, $tsize_t$ and β, respectively. These arrays require $O(d \times |T_1| \times |T_2|)$ space.

By Lemma 4.3, computing $fsize_t(F_1[i_1, i_s], F_2[j_1, j_t], 0)$ takes $O(1)$ time. By Lemma 4.4, computing $tsize_t(i,j,0)$ also takes $O(1)$ time. So the time required for the $k = 0$ case is

$$\sum_{i=1}^{|T_1|} \sum_{j=1}^{|T_2|} O(m_i \times n_j) \leq O(|T_2| \times \sum_{i=1}^{|T_1|} m_i) \leq O(|T_1| \times |T_2|)$$

By Lemma 4.5, computing $fsize_t(F_1[i_1, i_s], F_2[j_1, j_t], k)$ takes $O(k)$ time. By Lemma 4.6, computing $tsize_t(i,j,k)$ takes $O(1)$ time. So the time required for the $k > 0$

case is

$$\sum_{k=1}^{d}\sum_{i=1}^{|T_1|}\sum_{j=1}^{|T_2|} O(k \times m_i \times n_j) \leq O(|T_2| \times \sum_{k=1}^{d}\sum_{i=1}^{|T_1|} k \times m_i)$$

$$\leq O(|T_1| \times |T_2| \times \sum_{k=1}^{d} k) \leq O(d^2 \times |T_1| \times |T_2|)$$

Finally computing the array β takes $O(|T_1| \times |T_2|)$ time. Therefore the total time used by our algorithm is $O(d^2 \times |T_1| \times |T_2|)$. □

When d is a constant, the complexity is the same as that of the best known algorithm for computing the top-down edit distance of two trees due to Selkow [38], even though the LACS[t] problem addressed here appears to be harder than finding the distance.

It should be pointed out that by memorizing the size information during the computation and by a backtracking technique, one can find both the maximum size and one of the corresponding substructure pairs yielding the size in the same time and space complexity. To see this, notice that in considering $tsize_t(i, j, k)$ during the backtracking, one checks $l(t_1[i])$ and $l(t_2[j])$ and determines $fsize_t(F_1[i], F_2[j], k-c)$ (cf. Lemma 4.4 and Lemma 4.6). When considering $fsize_t(F_1[i], F_2[j], k)$, one re-calculates the recurrence formulas in Lemma 4.3 and Lemma 4.5 and finds out which case yields the maximum value, thus determining the mapping corresponding to that case.

4.4 Implementation

We have implemented the proposed algorithm into a graphics toolbox for comparing structured documents such as SGML and HTML [7]. Our approach is to transform the documents into ordered labeled trees based on the underlying markup language and then compare the documents using the proposed algorithm. Specifically, we represent the paragraph content of a document as a leaf node in the corresponding

tree structure.[3] The contents of the paragraphs are encoded into signatures. Each signature is an integer obtained by hashing the content of the corresponding paragraph. Thus, if two paragraphs are the same, their signatures remain the same. When the hash function chosen is perfect [13], two equivalent signatures also imply the equivalence of the corresponding paragraphs. Likewise, the section and subsection titles associated with the non-leaf nodes of the tree structure are encoded into signatures. These signatures become the labels of the nodes. We then use the proposed algorithm to compare two trees (documents), and display/highlight their common portion through the graphical interface.

Based on the top-down edit distance, our system allows paragraph deletes, inserts and mismatches to exist when finding the common portion of two documents. In addition, the system is able to detect a paragraph move, which is an important operation in document editing [63]. The underlying heuristic works by first finding the mapping between two trees. For nodes not touched by mapping lines, we check their signatures. Two paragraphs are related and labeled as "moved paragraphs" if they have the same signature.

Our work requires a grammar based on which documents or programs can be parsed into trees. Some electronic documents such as line-based documents may not hold structural tags as those of SGML. Some documents, such as Latex, have very liberal syntax rules, rendering the document comparison difficult. In such situations, we create an artificial grammar embeddable to the documents, so that they are parsed based on the same rules. The UNIX "diff" program is a good tool that complements our system. We have demonstrated the system in combination with "diff" in several

[3]There are trade-offs concerning the leaf representation in terms of running-time, mappings obtained, and semantics of the comparison. The choice is application-dependent and we feel that representing paragraphs as leaves is suitable for document editing. In our implementation, when the user wants to see a more detailed comparison between two paragraphs, he/she can invoke the UNIX "diff" program to do so.

conferences [54, 60, 63]. The executable code has been used by several researchers in their labs and is available from the authors.

CHAPTER 5
FINDING PATTERNS IN MULTIPLE TREES

In this chapter we present techniques for finding patterns in multiple trees (more precisely, RNA secondary structures). The distance measure used is the edit distance. Finding approximately common patterns or frequently occurring patterns in RNA secondary structures is an important problem in computational biology [6, 23, 30, 31]. For example, in predicting secondary structures for a given mRNA, one may first find a set of 'optimal' and 'suboptimal' structures using existing algorithms [72]. Then to determine which one among these structures is closest to the one occurring naturally, one may search for common patterns in the structures [24]. These patterns are assumed to be more stable in the intramolecular interactions between nucleotides and therefore are more likely to be present in the actual structure. Finding common patterns in secondary structures of different RNA molecules is useful as well. Often, the information obtained from such patterns, in conjunction with results obtained from sequence alignments, helps to conduct the phylogenetic study of the structure for a class of sequences [21, 30, 40, 65].

To find the common patterns in RNA secondary structures by a computer, we need a suitable representation for the structures. In this dissertation, we adopt the tree representation proposed in [40]. Figure 5.1 illustrates an RNA secondary structure and its tree representation. The structure is decomposed into five terms: stem, hairpin, bulge, internal loop and multibranch loop [45]. A stem is a portion of the RNA in which nucleotides are paired: G is paired with C and A is paired with U. The nucleotides in the other four terms are unpaired. A bulge is a half loop that sticks out of a stem. A hairpin is a closed loop connecting with a stem. It's like a direct turn from one side of the stem to the other. An internal loop is connected with two stems; it's a loop that does not make a direct turn like a hairpin. A multibranch loop is connected with multiple stems or loops, cf. Figure 5.1(a).

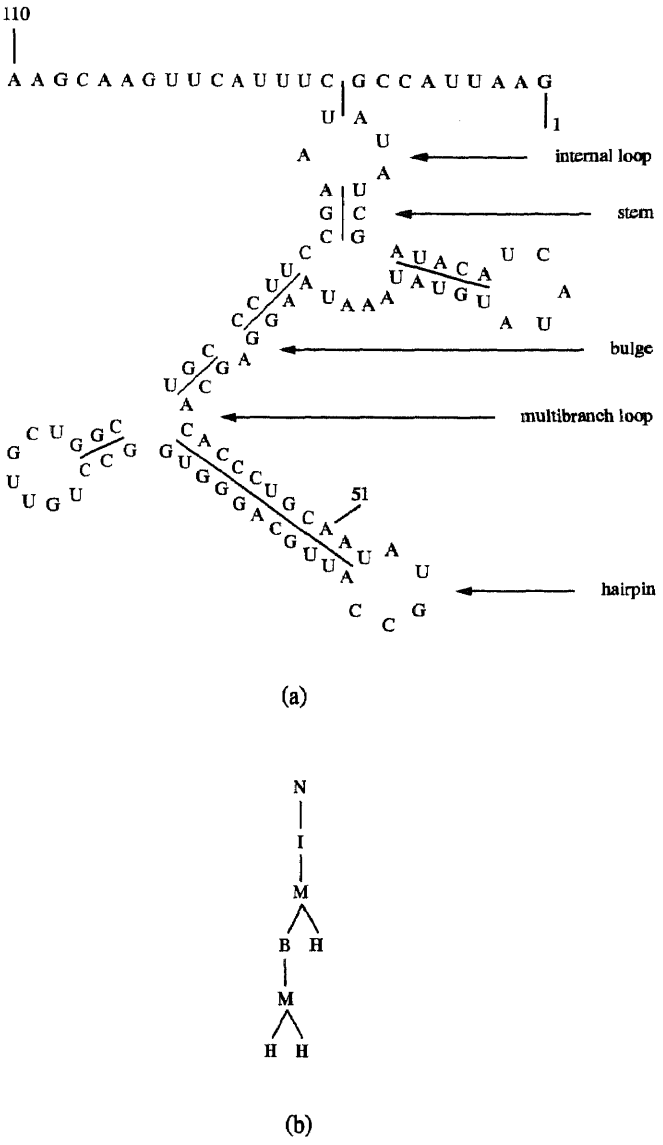


Figure 5.1 Illustration of a typical RNA secondary structure and its tree representation. (a) Normal polygonal representation of the structure. (b) Tree representation of the structure.

In our tree representation in Figure 5.1(b), H represents hairpin nodes, I represents internal loops, B represents bulge loops, M represents multibranch loops, R represents helical stem regions, which are shown as connecting arcs, and N is a special node used to make sure the tree is connected. The tree is an ordered one in which the order among siblings is significant [40]. The representation allows one to

encode detailed information of RNA by associating each node with a property list. Common properties may include sizes of loop components, sequence information and energy.

5.1 Common Patterns in Trees

We consider a pattern in a tree T to be a *substructure* of T, viz., a subtree U of T with certain nodes being cut. Cutting at a node n in U means removing n and all its descendants i.e., removing the subtree rooted at n. We say a tree T contains a pattern M within distance d, or M occurs in T within distance d, if there exists a subtree U of T such that the *minimum* distance between M and U is less than or equal to d, allowing zero or more cuttings at nodes in U. There is no cost associated with the cuttings. Formally, let $someroots(U)$ represent a set of nodes in U, where for any two nodes $m, n \in someroots(U)$, neither is an ancestor of the other. Let $cut(U, someroots(U))$ represent the resulting tree after we remove the subtrees rooted at nodes in $someroots(U)$. We denote the distance between M and U, allowing zero or more cuttings at nodes in U, as $\delta_{cut}(M, U)$, where

$$\delta_{cut}(M, U) = \min_{someroots(U)} \{\delta(M, cut(U, someroots(U)))\}.$$

M occurs in T within distance d if there exists a subtree U of T such that $\delta_{cut}(M, U) \leq d$.

Example 5.1. (Common patterns in trees) Consider the set $\mathcal{S}$ of three trees in Figure 5.2(a). Suppose only exactly coinciding substructures occurring in all the three trees and having size greater than 2 are considered as 'common patterns.' Then $\mathcal{S}$ contains two common patterns shown in Figure 5.2(b). If substructures having size greater than 4 and occurring in all the three trees within distance one are considered as common patterns, i.e., one relabeling, insertion or deletion of a

node is allowed in matching a pattern with a tree, then $\mathcal{S}$ contains two common patterns shown in Figure 5.2(c).

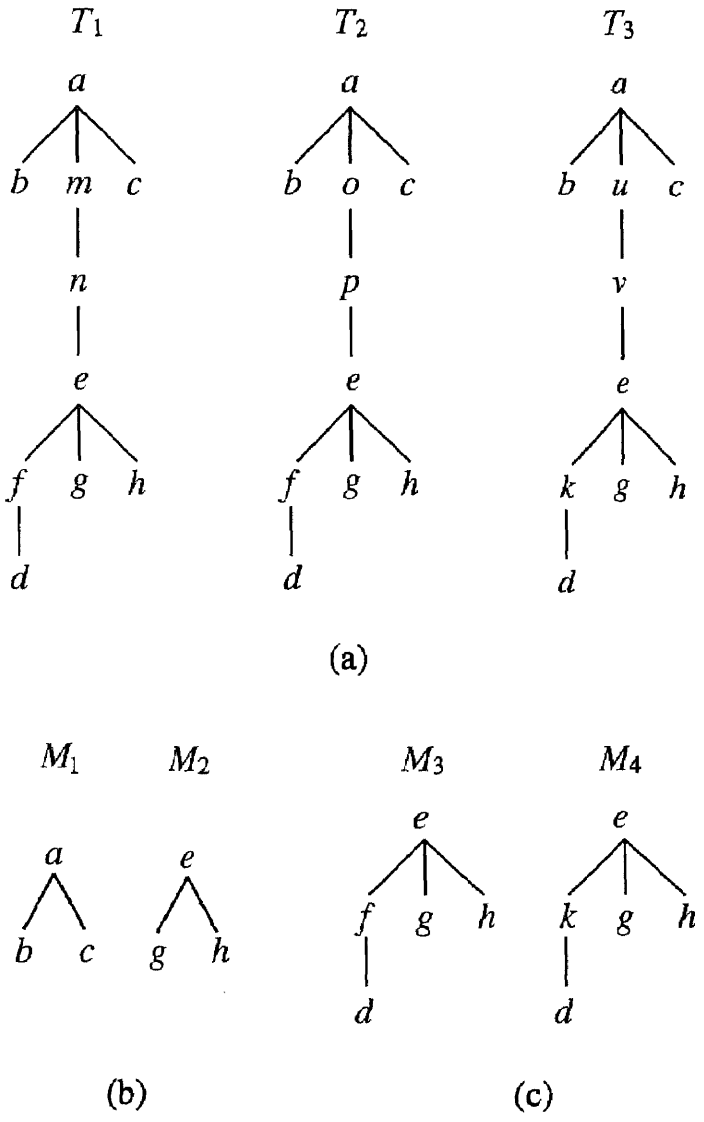


Figure 5.2 (a) The set $\mathcal{S}$ of three trees. (b) Two patterns exactly occurring in all the three trees. (c) Two patterns approximately occurring, within discovery operation.

To discover such approximately common patterns in a database of trees, our overall strategy is first to find candidate patterns among a small sample. From

the candidates, locate a set of promising patterns. Then check if the promising patterns exist in all the trees. Some previously published techniques may solve similar problems when the patterns meet some restrictions, for example, when they are exact matches [24], their topologies are known in advance [9, 30], or they are translated into sequences where the tree structure is not considered during searching [21, 39]. In contrast, our approach can discover approximately common substructures without prior knowledge of their topologies, positions, or occurrence frequency in the trees. In the next section we formalize the discovery queries of interest and then present algorithms to process the queries.

5.2 The Basic Queries and Algorithms

5.2.1 Basic Query Type

Given a database $\mathcal{D}$ of trees, there exist various requirements on the sizes and forms of patterns to be sought. The following parameters appear to be most significant; all these parameter values are specified by the user:

- the form of patterns, in our case the substructures of trees;
- the minimum size of a pattern of interest *Size*, in our case the number of nodes in the pattern;
- the distance metric, in our case edit distance between trees with unit cost;
- the allowed distance *Dist*;
- the minimum occurrence number *Occur* with respect to the distance and size of a chosen pattern. The occurrence number, or *activity*, of a pattern M is the number of trees in $\mathcal{D}$ that contain M within the allowed distance. Formally, we say the occurrence number of a pattern M with respect to distance d and set $\mathcal{S}$, denoted $occurrence\_no^d_{\mathcal{S}}(M)$, is k if there are k trees in $\mathcal{S}$ that contain M

within distance d. For example, consider again the set $\mathcal{S}$ of trees in Figure 5.2. Then $occurrence\_no^0_{\mathcal{S}}(M_1) = occurrence\_no^0_{\mathcal{S}}(M_2) = 3$; $occurrence\_no^1_{\mathcal{S}}(M_3) = occurrence\_no^1_{\mathcal{S}}(M_4) = 3$.

The basic query is to find the patterns M where M is within the allowed distance $Dist$ of at least $Occur$ trees in the database $\mathcal{D}$ and $|M| \geq Size$. We can use the results of this query in several ways. For example, natural scientists may attempt to evaluate whether the common patterns indeed occur in the actual structure of RNA molecules. Computer scientists may use the patterns to classify RNA sequences into one family or another.

5.2.2 Query Processing Algorithms

Our approach is a two phase process:

1. Find candidate patterns among a small sample $\mathcal{A}$ of the trees and search for promising patterns from the candidates with respect to the sample.

2. Evaluate the activity of the promising patterns in all of the database $\mathcal{D}$ to determine which promising patterns approximately characterize $\mathcal{D}$ as a whole.

Phase 1 consists of two subphases. In subphase A, we look for the candidate patterns in the sample. In subphase B, we store the candidate patterns in an index structure and locate the promising patterns by traversing the index structure.

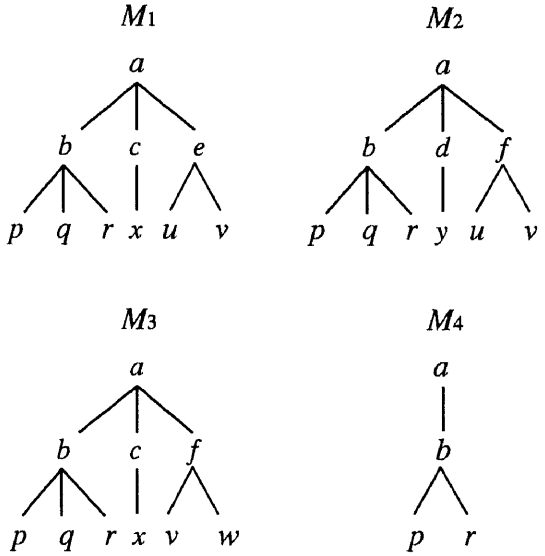
5.2.2.1 Subphase A of Phase 1: In subphase A of phase 1, we divide the sample $\mathcal{A}$ into a collection of disjoint tree pairs. Any such collection has $|\mathcal{A}|/2$ pairs, where $|\mathcal{A}|$ represents the size, i.e. the number of trees, in the sample $\mathcal{A}$. For each pair of trees T_1 and T_2, we use the algorithms described in Chapter 3 to find the largest approximately common substructures, within the allowed distance $Dist$, between $T_1[i]$ and $T_2[j]$ for all $1 \leq i \leq |T_1|$, $1 \leq j \leq |T_2|$. The asymptotic time complexity

of the algorithm is $O(Dist^2 \times |T_1| \times |T_2| \times \min\{H_1, L_1\} \times \min\{H_2, L_2\})$ where H_i, $i = 1, 2$, is the height of T_i and L_i is the number of leaves in T_i. Let $\mathcal{C}$ contain the found substructures M where $|M| \geq Size$; these substructures constitute candidate patterns. For each candidate pattern M, the tree pairs from which M is discovered are recorded.

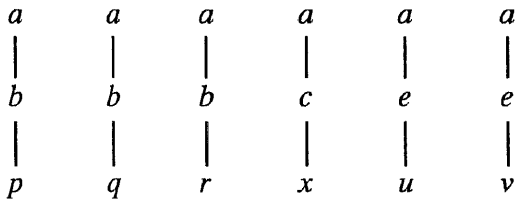
5.2.2.2 Subphase B of Phase 1: In this subphase, we store the candidate patterns into a prefix tree PRET, which is similar to Kosaraju's suffix tree for trees [22]. Each candidate pattern M is decomposed into a collection of paths, called *p-strings*. Each p-string contains a sequence of nodes starting at the root of M and ending at a leaf of M. Figure 5.3(a) shows four candidate patterns; Figure 5.3(b) shows the p-strings for one of the patterns.

The p-strings of candidate patterns are inserted into the PRET as into a trie [3, Section 5.3] except that if a node has only one child, we collapse the child with the parent and label the edge going down from the parent with a substring instead of a single character. Figure 5.4 illustrates an example, in which the nodes with the labels 7 and 8 show the result of a collapsing. For each node v in the PRET, let $string(v)$ be the string on the edge labels from the root to v. We associate v with two fields: $pattern(v)$ and $count(v)$. The field $pattern(v)$ tells which candidate patterns contain $string(v)$; $string(v)$ may be a p-string or a prefix of some p-string in the candidate patterns. The field $count(v)$ shows the number of sample trees that contain $string(v)$ within the allowed distance. This field is calculated by traversing the PRET in a bottom-up fashion, e.g., by a postorder traversal, and counting the values in the field $pattern(v)$ during the traversal. Specifically, the counting algorithm works as follows. Recall that the tree pairs from which a candidate pattern is discovered are recorded. For each $string(v)$, we add up the numbers of distinct trees from which

the candidate patterns in the field $pattern(v)$ are discovered and assign the sum to the field $count(v)$.



(a)



(b)

Figure 5.3 (a) Four candidate patterns M_1, M_2, M_3 and M_4. (b) M_1's p-strings.

The PRET can be constructed asymptotically in $O(N)$ time and space where N is the total length of all p-strings contained in the candidate patterns in $\mathcal{C}$ [18, 22]. After constructing the PRET, we traverse the PRET in a top-down fashion, pruning

unlikely candidate patterns using the optimization heuristics described in the next section and finding a set of promising patterns. As we will see, a promising pattern may be a candidate pattern or a substructure of a candidate pattern.

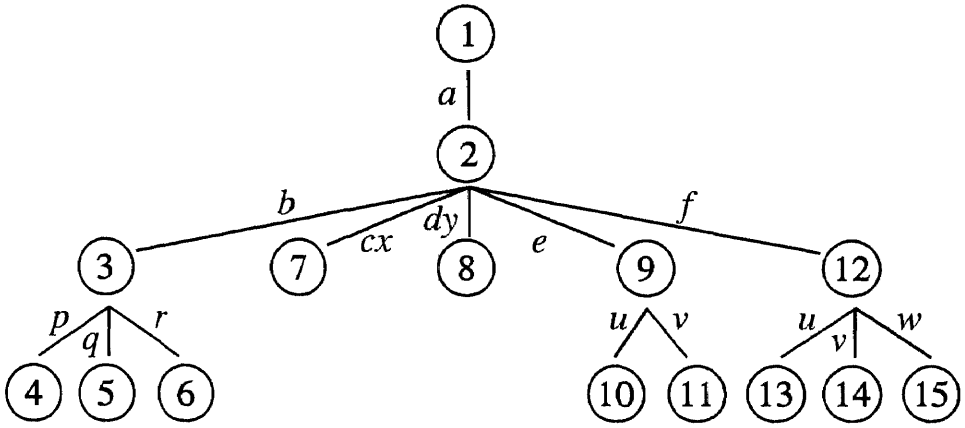


Figure 5.4 Illustration of the PRET, constructed by inserting the p-strings of the patterns shown in Figure 5.3(a) into the PRET. Each node in the PRET is labeled by its preorder number.

5.2.2.3 Phase 2: In this phase, we evaluate the promising patterns with respect to the entire database $\mathcal{D}$. In checking whether a promising pattern M occurs in a tree $T \in \mathcal{D}$ within the allowed distance, we add variable length don't cares, denoted as VLDCs, to M as the new root and leaves to form a VLDC pattern V and then compare V with T using the tree pattern matching algorithm developed in [70]. A VLDC, conventionally denoted by "$*$", can be matched, at no cost, with a path or portion of a path in T. The algorithm calculates the minimum distance between V and T after implicitly computing an optimal substitution for the VLDCs in V, allowing zero or more cuttings at nodes from T; see Figure 5.5.

The motivation for considering only the patterns found promising in the sample is that comparing a VLDC pattern V with a tree T requires a dynamic programming approach that can take, in the worst case, $O(|V| \times |T| \times \min\{H_V, L_V\} \times \min\{H_T, L_T\})$

time. Here H_V (H_T, respectively) is the height and L_V (L_T, respectively) is the number of leaves of V (T, respectively). Screening out those unlikely candidate patterns may save significant time in the overall computation. For example, in our empirical study to be presented in Section 5.4, it is shown experimentally that screening out the unlikely candidate patterns can save about 9/10 of the running time.

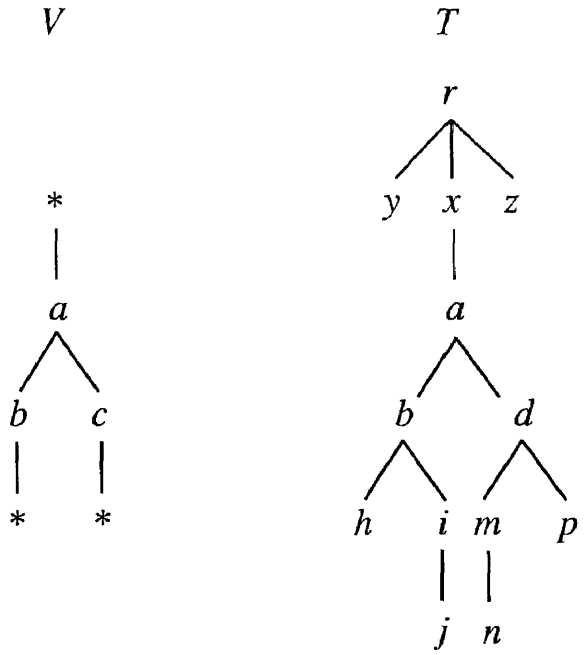


Figure 5.5 Matching a VLDC pattern V and a tree T (both the pattern and tree are hypothetical ones solely used for illustration purposes). The root $*$ in V would be matched with nodes r, x in T, and the two leaves $*$ in V would be matched with nodes i, j and m, n in T, respectively. Nodes y, z, h, p in T would be cut. The distance of V and T would be 1 (representing the cost of changing c in V to d in T).

5.3 Optimization Heuristics

In phase 2 of our query processing algorithms, we compare the promising patterns with the entire database. The main question from an optimization point of view

is when to give up on a candidate and how to find the promising patterns. Our strategies are as follows.

5.3.1 Pruning Unlikely Candidates

We consider the least possible occurrence number a qualifying pattern should have in the sample. Let $\alpha = Occur/|\mathcal{D}|$ and we require that a qualifying pattern M should appear in at least $\alpha \times |\mathcal{D}|$ trees in the database $\mathcal{D}$. This implies that M will appear in at least $\alpha \times |\mathcal{A}|$ trees in the sample on the average. We are interested in patterns having α values greater than 0.5, e.g., $\alpha = 0.6$. Since the sample may not reflect the whole database exactly however, we consider patterns that appear in at least $(2\alpha - 1) \times |\mathcal{A}|$ elements of the sample. We use the following analysis to justify this.

Worst Case Analysis. Let $\mathcal{A}_1$ be the set of sample trees containing M within *Dist* and let $\mathcal{A}_2$ be the set of sample trees not containing M within *Dist*. Obviously, $\mathcal{A}_1 \cup \mathcal{A}_2 = \mathcal{A}$. Recall that we find candidate patterns from $|\mathcal{A}|/2$ sample tree pairs. Thus, in the worst case, among the $|\mathcal{A}|/2$ sample tree pairs, $(1 - \alpha) \times |\mathcal{A}|$ pairs were chosen in such a way that each of them has one tree taken from $\mathcal{A}_1$ and the other tree taken from $\mathcal{A}_2$. From these $(1 - \alpha) \times |\mathcal{A}|$ pairs, one cannot discover M. One can only discover M from the other $|\mathcal{A}|/2 - ((1 - \alpha) \times |\mathcal{A}|) = (\alpha - 0.5) \times |\mathcal{A}|$ pairs, where both sample trees of each pair must contain M within *Dist*. Therefore the occurrence number of M with respect to the sample $\mathcal{A}$ is, in the worst case, $2 \times (\alpha - 0.5) \times |\mathcal{A}| = (2\alpha - 1) \times |\mathcal{A}|$.

Based on the above analysis, if a candidate pattern's occurrence number with respect to the sample $\mathcal{A}$ is below $(2\alpha - 1) \times |\mathcal{A}|$, it's unlikely that the candidate will

be a qualifying pattern, i.e., satisfy the *Occur* constraint with respect to the whole database. Therefore we discard it.

5.3.2 Eliminating Redundant Calculation of Occurrence Numbers

Observe that the most expensive operation in the query processing algorithms is to find the occurrence number of a pattern with respect to the entire database, since that entails comparing the pattern with all trees in the database. To reduce the number of such comparisons, we introduce the notion of subpatterns for trees.

Definition 5.1. (Subpatterns for trees) Let T and T' be two trees. An *embedded mapping* M_b from T to T' is a mapping from T to T' satisfying the following conditions:

1. all nodes in T are touched by a mapping line in M_b;
2. for any pair (i, j) in M_b, $t[i] = t'[j]$;
3. for any two pairs (i_1, j_1) and (i_2, j_2) in M_b, $t[i_1]$ is a parent of $t[i_2]$ iff $t'[j_1]$ is a parent of $t'[j_2]$.

T is said to be a *subpattern* of T', or T' is a *superpattern* of T, if there exists an embedded mapping from T to T'.

As an example, consider again the tree patterns in Figure 5.3. M_4 is a subpattern of M_1, M_2 and M_3. Also each p-string P, which itself can be treated as a pattern, is a subpattern of the candidate pattern containing P.

Proposition 5.1. *If M is a subpattern of M', or M' is a superpattern of M, then for any distance parameter k,*

$$occurrence\_no_D^k(M) \geq occurrence\_no_D^k(M').$$

Proof. Let $T \in \mathcal{D}$ be a tree such that M' occurs in T with distance j, $0 \leq j \leq k$. By definition, there exists a subtree U of T such that $\delta_{cut}(M', U) = j$. Since M is a subpattern of M', there must exist a subtree V of T such that V is a subpattern of U and $\delta_{cut}(M, V) \leq j$. This implies that $occurrence\_no^j_{\mathcal{D}}(M) \geq occurrence\_no^j_{\mathcal{D}}(M')$, and hence the result follows.

Thus if M' is in the final output set, then we need not bother evaluating M, since it will be too. If M is not in the final output set, then M' won't be either, since its occurrence number will be even lower.

Let $occurrence\_set^k_{\mathcal{D}}(M)$ denote the set of all trees in $\mathcal{D}$ that contain M within distance k, i.e., $|occurrence\_set^k_{\mathcal{D}}(M)| = occurrence\_no^k_{\mathcal{D}}(M)$.

Proposition 5.2. *If M and M' are subpatterns of M'', or M'' is a superpattern of M and M', then for any distance parameter k,*

$$occurrence\_set^k_{\mathcal{D}}(M'') \subseteq (occurrence\_set^k_{\mathcal{D}}(M) \cap occurrence\_set^k_{\mathcal{D}}(M')).$$

Proof. For any tree $T \in \mathcal{D}$, if $T \in occurrence\_set^k_{\mathcal{D}}(M'')$, by definition, there exists a subtree U of T such that $\delta_{cut}(M'', U) = j$ for some integer $j \leq k$. Since M and M' are subpatterns of M'', there must exist two subtrees V and W of T such that V and W are subpatterns of U and $\delta_{cut}(M, V) \leq j$ and $\delta_{cut}(M', W) \leq j$ respectively. Therefore $T \in occurrence\_set^k_{\mathcal{D}}(M)$ and $T \in occurrence\_set^k_{\mathcal{D}}(M')$, and hence the result follows.

Thus if $|(occurrence\_set^k_{\mathcal{D}}(M) \cap occurrence\_set^k_{\mathcal{D}}(M'))| < Occur$, we can eliminate M'' from consideration since its occurrence number will be even lower.

5.3.3 Our Optimized Approach

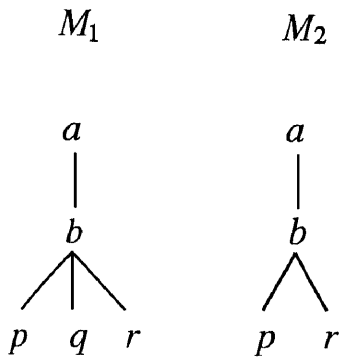
To incorporate the preceding optimization heuristics into our query processing algorithms, we traverse the prefix tree PRET described in Section 5.2.2.2 in a top-down fashion, e.g., by a preorder traversal. When visiting a node u in the PRET, check the field $count(u)$. This field tells the number of sample trees that contain $string(u)$ within the allowed distance $Dist$. Suppose that, based on the worst-case analysis described in Section 5.3.1, it is estimated that the occurrence number of $string(u)$ with respect to the whole database is below the required activity $Occur$. Let M_i, $1 \leq i \leq n_u$, be the candidate patterns, identified by the field $pattern(u)$, that contain $string(u)$. Let c be the first character of the substring on the edge between u and its parent. We remove the subtree rooted at c from M_i, i.e., cut at node c in M_i. Intuitively, this results in the removal of the unlikely portions from each of these candidate patterns. This may result in some new candidate patterns which were not discovered in subphase A of phase 1 of our query processing algorithms described in Section 5.2.2.1.

Effectively, we discard the candidate patterns M_i, $1 \leq i \leq n_u$, containing strict superpatterns of $string(u)$. The reason for doing so is that, by Propositions 5.1 and 5.2, these patterns have lower occurrence numbers than $string(u)$, and as a consequence the occurrence numbers are much smaller than $Occur$. This creates another set of new candidate patterns composed of only parts of the patterns M_i. Note that after cutting at c in the M_i, there is no need to visit the descendants of u in the PRET. Upon completing the traversal of the PRET, we enumerate all possible cuttings in the resulting patterns and pick the patterns M where $|M| \geq Size$ and M's estimated occurrence number with respect to the database is greater than $Occur$. This gives a set of promising patterns.

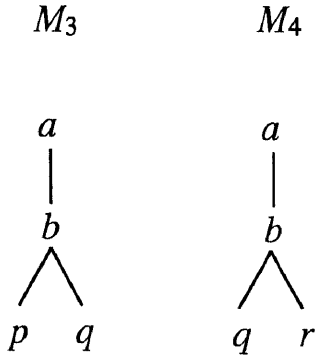
Table 5.1 Actions taken during traversing the PRET in Figure 5.3; *number*(v) represents the preorder number of the node v in the PRET.

number(v)	*count*(v)	*pattern*(v)	actions taken when visiting node v
1	–	—	go to the next node, numbered 2
2	4	M_1, M_2, M_3, M_4	go to the next node, numbered 3
3	4	M_1, M_2, M_3, M_4	go to the next node, numbered 4
4	4	M_1, M_2, M_3, M_4	go to the next node, numbered 5
5	3	M_1, M_2, M_3	go to the next node, numbered 6
6	4	M_1, M_2, M_3, M_4	go to the next node, numbered 7
7	2	M_1, M_3	cut at c in M_1 and M_3; go to the next node, numbered 8
8	1	M_2	cut at d in M_2; go to the next node, numbered 9
9	1	M_1	cut at e in M_1; ignore the descendants of node 9; go to the next node, numbered 12
12	2	M_2, M_3	cut at f in M_2 and M_3; the traversal ends

Example 5.2. (Illustration of the Two Phase Optimized Approach) Consider again the four patterns in Figure 5.3(a). Assume that these are the only patterns obtained from the sample. Also assume that the field *count*(v) for each node v of the PRET in Figure 5.4 is as shown in Table 5.1. Now suppose that, based on the worst-case analysis described in Section 5.3.1, a qualifying pattern must occur in at least three sample trees. Table 5.1 illustrates the steps and the actions taken at each step during traversing the PRET in Figure 5.4. Figure 5.6(a) shows the resulting patterns. Suppose a qualifying pattern must have size greater than 3. Figure 5.6(b) shows the patterns obtained by cutting at nodes in the patterns of Figure 5.6(a) that satisfy the size constraint. We then compare the patterns in Figure 5.6 with sample trees and choose the promising ones to compare with all trees in the database.



(a)



(b)

Figure 5.6 (a) Resulting patterns after completing the traversal of the PRET in Figure 5.4. (b) Patterns obtained by cutting at nodes in the patterns in (a); M_3 is obtained by cutting at r in M_1 and M_4 is obtained by cutting at p in M_1.

5.4 Performance Analysis

We carried out a series of experiments to evaluate the effectiveness and speed of our approach. The speed is measured by elapsed CPU time. The programs were written in C and run on a SUN SPARC workstation under the SUN operating system

version 4.1.2. The data was a set of randomly generated 80 trees. To make the experiments manageable, the size of the trees was fixed at 15. Each node label of the generated trees was drawn randomly from the range A to Z. To gain a better understanding of the performance of our algorithms, we also tested the algorithms on RNA secondary structures. Eighty secondary structures, represented as trees, were selected randomly from the database in the National Cancer Institute [39, 40]. The sizes of the secondary structures ranged from 10 to 15. Table 5.2 shows the parameters and base values used in the experiments. It was observed that the results obtained from the generated and real data, for both the base values and other parameter values, are rather consistent. As a consequence, we only present here the results for the RNA molecules with the base values.

Referring to Table 5.2, the trees in the sample were chosen randomly from the database. The parameter *NumSample* in the table indicates the number of samples chosen for each database. In the experiments presented here, 20 samples were chosen randomly. Each time one sample was used in running the database and the average was plotted. The sample size was obtained by multiplying *DBSize* by *SizeRatio*.

Table 5.2 Experimental parameters and base values.

Parameter	Value	Description
DBSize	80	# of trees in a database
NumSample	20	# of samples tested for a database
SizeRatio	30%	Ratio between sample size and database size
Size	5	Minimum size of an interesting pattern
Dist	1	Allowed distance between a pattern and a tree
Occur	70	Minimum occurrence number of an interesting pattern

The metric used to evaluate the effectiveness of our algorithms is

$$HitRatio = \frac{NumDiscovered}{TotalNum} \times 100\%$$

where *NumDiscovered* is the number of interesting patterns discovered by our techniques. *HitRatio* stands for the percentage of the interesting patterns obtained from the exhaustive search method. By exhaustive search, we mean selecting as candidates all patterns in the database that satisfy the size constraint. One would like this percentage to be as high as possible. It should be pointed out that we have rejected approximately occurring patterns that never appear in the database yet satisfy the *Dist* and *Occur* constraints in favor of those that obey the constraints and do appear in the database. This is a theoretical limitation of our work that we have introduced to save computation time, though this also seems pragmatically to be a reasonable approach.

Figure 5.7 compares the effectiveness of our optimized approach with a non-optimized approach for varying sample sizes. Figure 5.8 compares their running times with that of the exhaustive search method. It can be seen in Figure 5.7 that very few qualifying patterns were missed by the two proposed optimization heuristics. Both heuristics sped up the discovery algorithm by a factor of 10. Moreover, our optimized approach was 100,000 times faster than the brute force method (Figure 5.8).

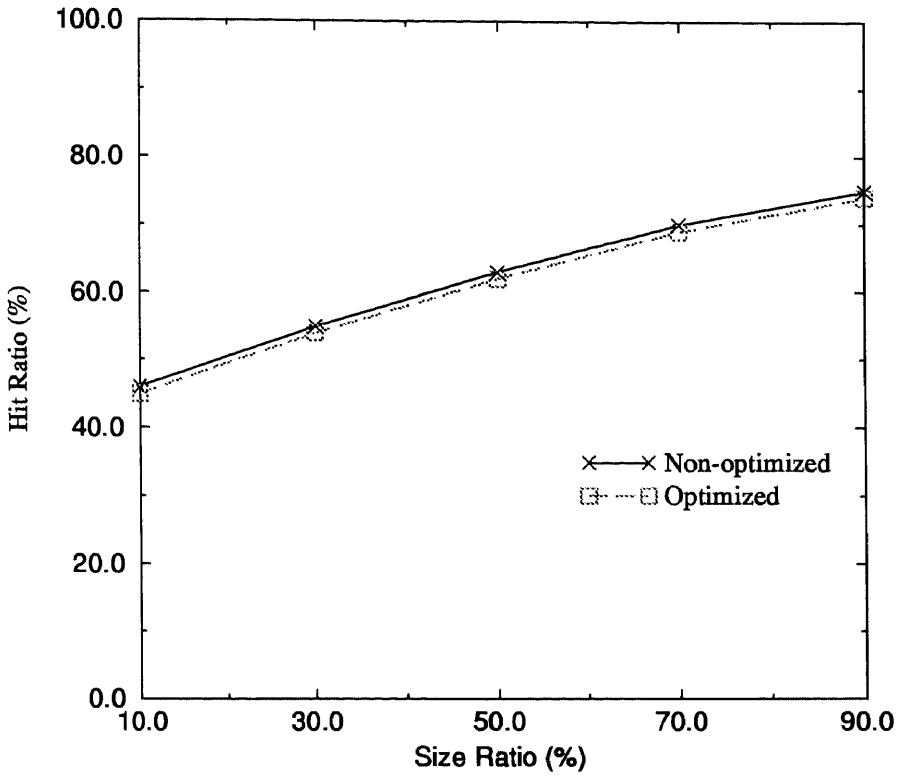


Figure 5.7 Performance of the pruning techniques for varying sample sizes.

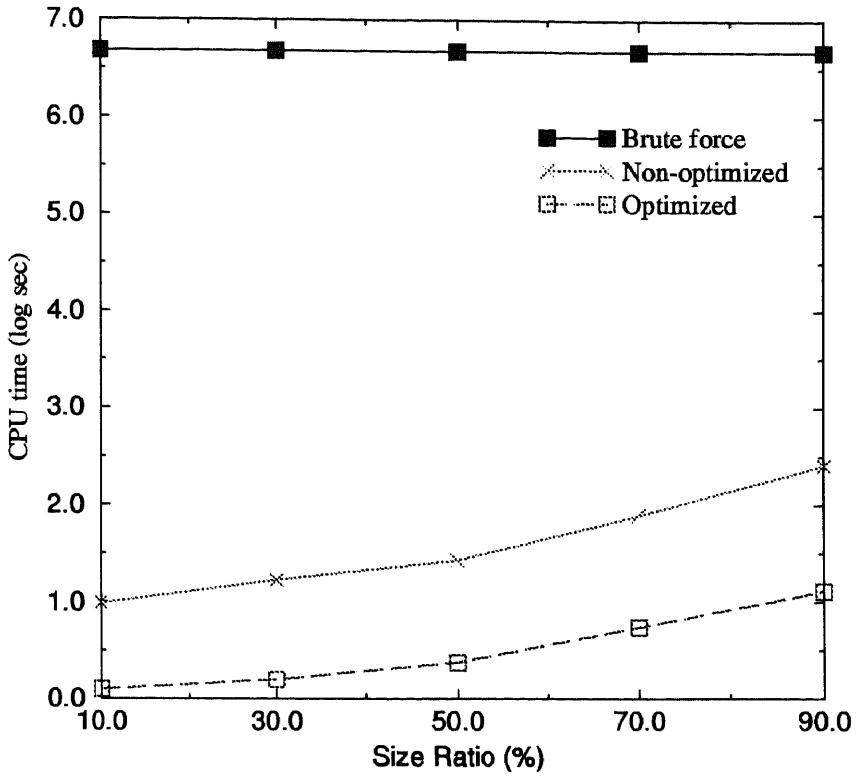


Figure 5.8 Comparison of the running times between the brute force method, our optimized approach and the approach without optimizations.

CHAPTER 6

A TOOLBOX FOR PATTERN DISCOVERY IN RNA SECONDARY STRUCTURES

We have applied our algorithm to find motifs in multiple RNA secondary structures. In this experiment, we examined three phylogenetically related families of mRNA sequences chosen from GenBank [5] pertaining to the poliovirus, human rhinovirus and coxsackievirus. Each family contained two sequences, as shown in Table 6.1.

Table 6.1 Data used in the experiment.

Family	Sequence	# of trees	File #
poliovirus	polio3 sabin strain	3,026	file 1
	pol3mut	3,000	file 2
human rhinovirus	rhino 2	3,000	file 3
	rhino 14	3,000	file 4
coxsackievirus	cox5	3,000	file 5
	cvb305pr	2,999	file 6

Under physiological conditions, i.e., at or above the room temperature, these RNA molecules do not take on only a single structure. They may change their conformation between structures with similar free energies or be trapped in local minima. Thus, one has to consider not only the optimal structure but all structures within a certain range of free energies. On the other hand, a loose rule of thumb is that the "real" structure of an RNA molecule appears in the top 5% - 10% of suboptimal structures of the sequence based on the ranking of their energies with the minimum energy one (i.e. the optimal one) being at the top. Therefore, we folded the 5' non-coding region of the selected mRNA sequences and collected (roughly) the top 3,000 suboptimal structures for each sequence. We then transformed these suboptimal structures into trees using the algorithms described in [39, 40].

Figure 5.1 illustrates an RNA secondary structure and its tree representation. The structure is decomposed into five terms: stem, hairpin, bulge, internal loop and multi-branch loop [40]. In the tree, H represents hairpin nodes, I represents internal loops, B represents bulge loops, M represents multi-branch loops, R represents helical stem regions (shown as connecting arcs) and N is a special node used to make sure the tree is connected. The tree is considered to be an ordered one where the ordering is imposed based upon the 5' to 3' nature of the molecule. The resulting trees for each mRNA sequence selected from GenBank were stored in a separate file, where the trees had between 70 and 180 nodes (cf. Table 6.1). Each tree is represented by a fully parenthesized notation where the root of every subtree precedes all the nodes contained in the subtree. Thus, for example, the tree depicted in Figure 5.1(b) is represented as (N(I(M(B(M(H)(H)))(H)))).

For each pair of trees T_1, T_2 in a file, we ran the algorithm Find-Largest[e] on T_1, T_2, finding the size of the largest approximately common substructures, within distance 1, for each subtree pair $T_1[i]$ and $T_2[j]$, $1 \leq i \leq |T_1|$ and $1 \leq j \leq |T_2|$, and locating one of the corresponding substructure pairs yielding the size. These substructures constituted candidate motifs. Then we calculated the occurrence number[4] of each candidate motif M by adding variable length don't cares (VLDCs) to M as the new root and leaves to form a VLDC pattern V and then comparing V with each tree T in the file using the pattern matching technique developed in [70]. (A VLDC (conventionally denoted by "*") can be matched, at no cost, with a path or portion of a path in T. The technique calculates the minimum distance between V and T after implicitly computing an optimal substitution for the VLDCs in V, allowing zero or more cuttings at nodes from T (see Figure 5.5).) This way we can

[4]The occurrence number of a motif M with respect to distance k refers to the number of trees of the file in which M approximately occurs (i.e. these trees approximately contain M) within distance k.

locate the motifs approximately occurring in all (or the majority of) the trees in the file.[5]

We have incorporated the presented algorithms into a visualization toolbox for pattern matching and discovery in scientific [55, 57, 58, 62] and document databases [53, 59]. Figure 6.1, Figure 6.2 and Figure 6.3 show the screen layout of the toolbox. In Figure 6.1, the lines connecting two nodes mean the two nodes are relabeled. The colored nodes are nodes that need to be deleted or inserted. The executable software packages of the toolbox are available from the author. For details, please visit the Web site at http://www.cis.njit.edu/~discdb.

[5]One can speed up this method by encoding the candidate motifs into a suffix tree and then using the statistical sampling and optimization techniques described in [61] to find the motifs.

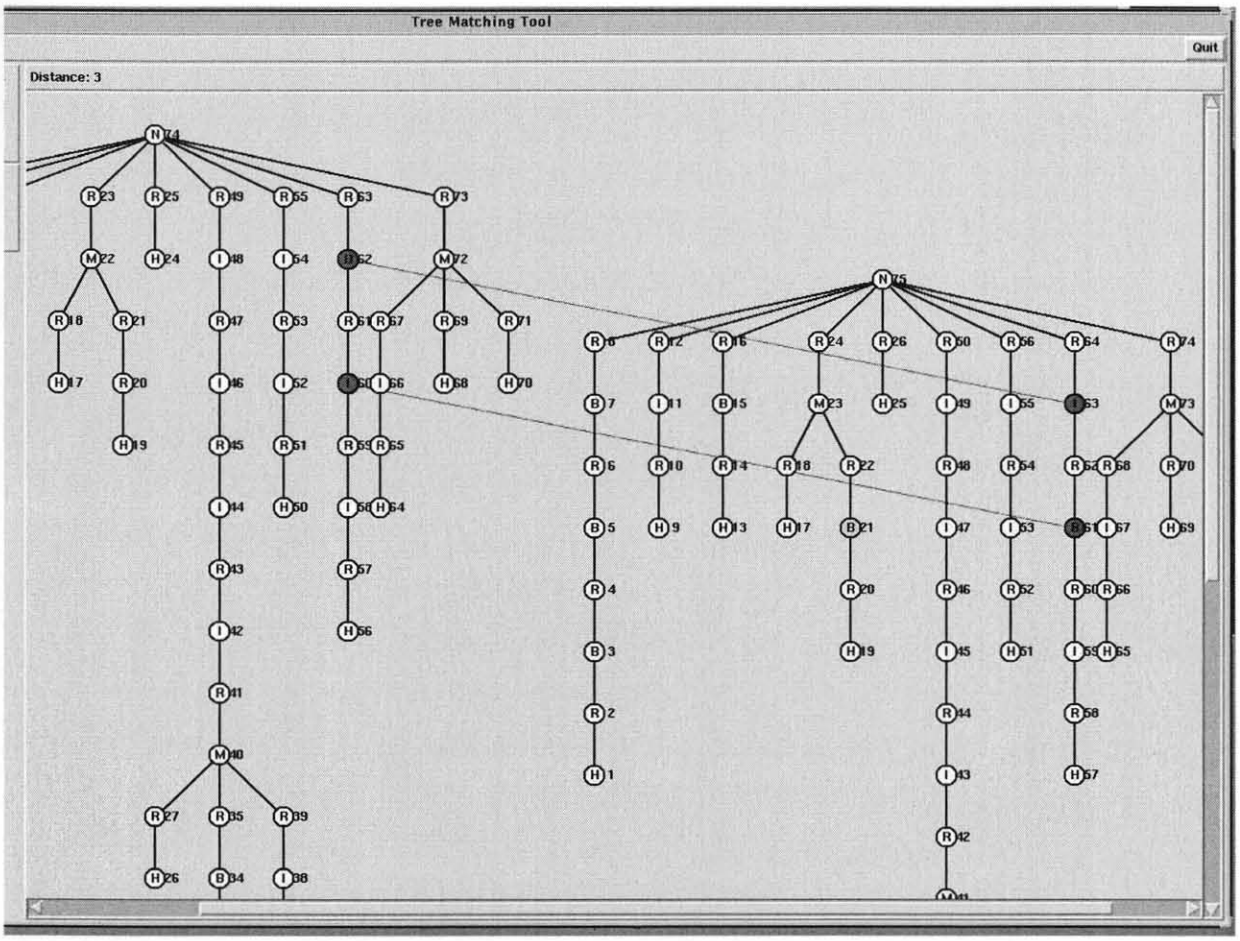


Figure 6.1 Illustration of the differences between two RNA secondary structures pertaining to rhinovirus.

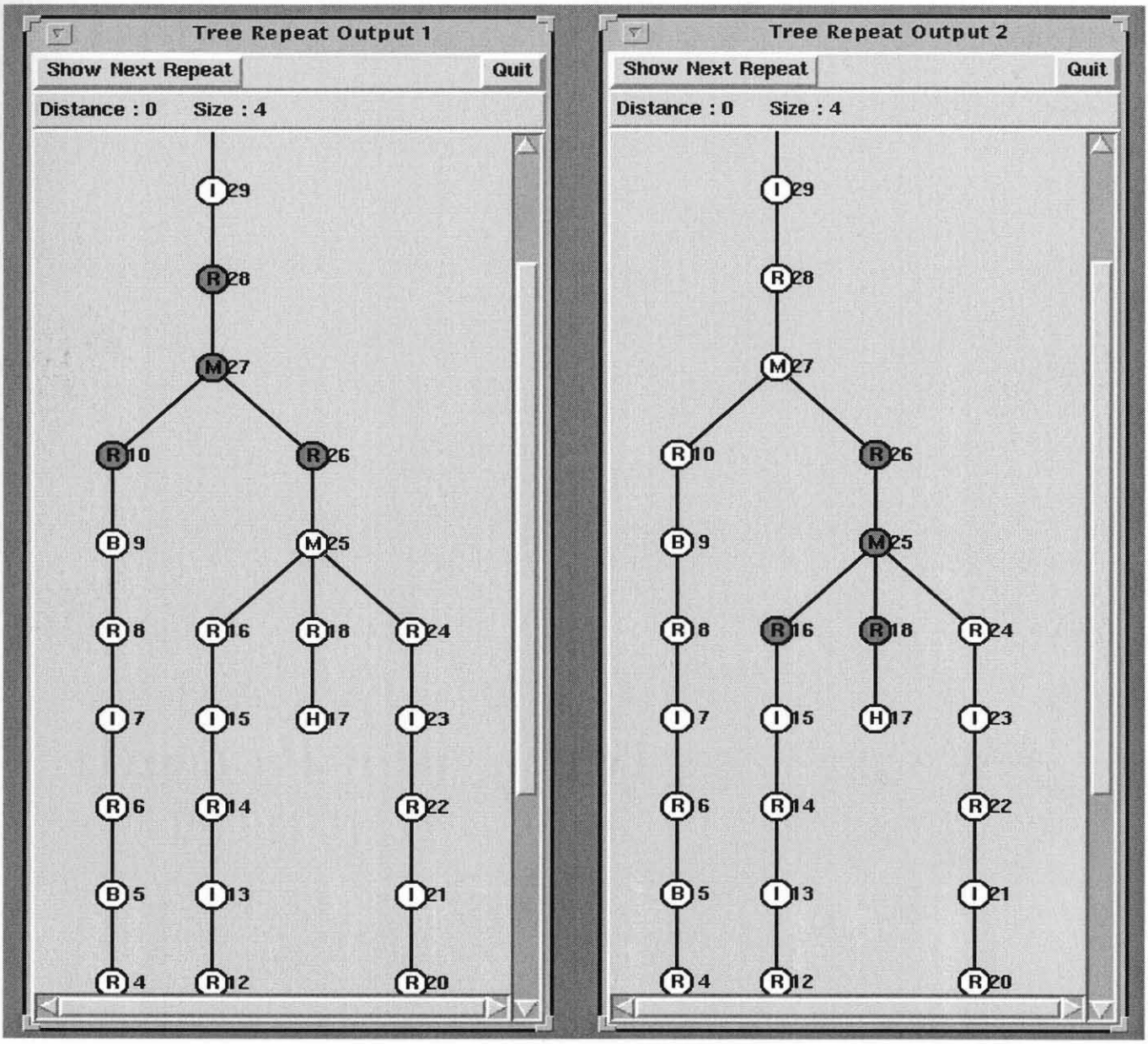


Figure 6.2 Illustration of the discovery of repeatedly occurring sub-structures within one RNA secondary structure.

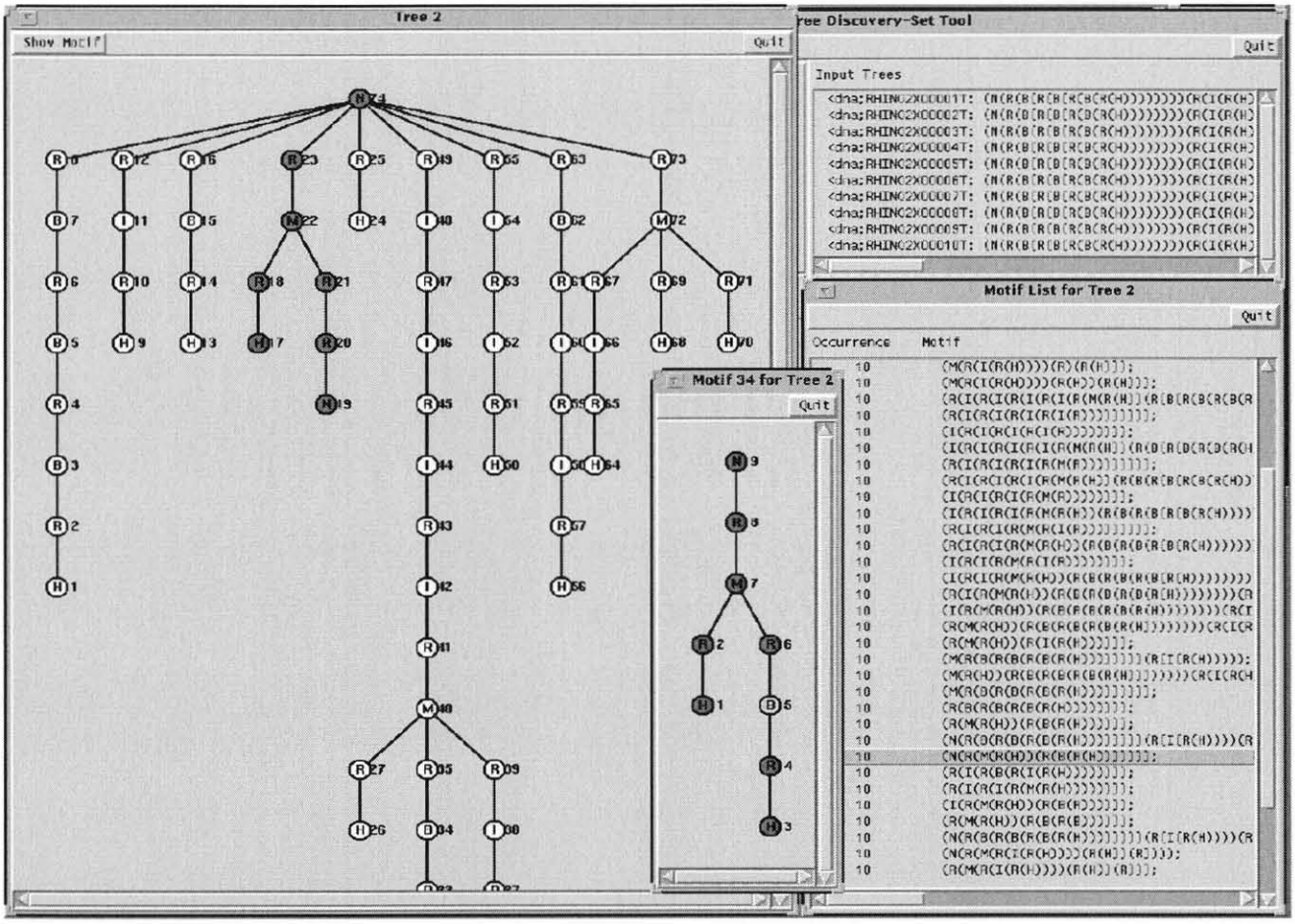


Figure 6.3 Illustration of the discovery of frequently occurring sub-structures within a set of RNA secondary structures.

CHAPTER 7

CONCLUSIONS

The algorithms presented in the dissertation assume a unit cost for all edit operations. In practice, a more refined non-unit cost function can reflect more subtle differences in the RNA secondary structures [40]. It would then be interesting to score the measures in detecting common substructures or repeats in trees. Another interesting problem is to find a largest consensus motif T_3 in two input trees T_1 and T_2 where T_3 is a largest tree such that each of T_1 and T_2 has a substructure that is within a given distance to T_3. A comparison of the different types of common substructures (see also [25, 26, 27]), probably based on different metrics (e.g. [47, 48, 49]), as well as their applications remains to be explored.

This dissertation presented an example of scientific data mining: the discovery of common patterns in a database of trees. The strategy we proposed here is first to find candidate patterns satisfying structural constraints of size in a small sample. From the candidates, we locate a set of promising patterns. Then evaluate the promising patterns on the whole database. We applied the proposed techniques to discovering approximately common patterns in both generated data and RNA secondary structures. Our experimental results demonstrated the good performance of the proposed algorithms.

Our work on scientific data mining is continuing. We have two main goals.

1. We would like to extend the discovery framework presented here to two and three dimensional graphs. Finding common substructures in such graphs has many applications in biology and chemistry [71]. Recently, we have obtained some preliminary results along this direction [64].

2. We want to apply the discovered substructures to clustering molecules. We are currently extending our pattern discovery algorithms to cluster proteins and RNA secondary structures.

The substructures defined in the dissertation are generated by applying the "cut" operations to nodes in the trees. Cutting at node n from tree T means removing n and all its descendants, i.e., removing the subtree rooted at n. In [69], a relevant operation, called "prune", was defined. Pruning at node n from tree T means removing only the descendants of n; n itself remains in T. Thus, a pruning never eliminates the entire tree. The formulas and algorithms presented in the dissertation generalize easily to the "prune" case, though the resulting substructures differ as the definitions for the two operations are different. We also plan to study applications of the algorithms based on the pruning operation.

APPENDIX A

PROGRAM LISTINGS

This appendix contains:

- Programs that find the largest common substructures between two trees.
- Programs that find the frequently occurring patterns in a set of trees.

```
/*****************************************************************/
/*                                                               */
/*     (c) copyright 1999                                        */
/*         All rights reserved                                   */
/*     Programs written by Chia-Yo Chang (NJIT)                  */
/*                                                               */
/*     Permission to use, copy, modify, and distribute this      */
/*     software and its documentation for any purpose and without */
/*     fee is hereby granted, provided that this copyright       */
/*     notice appears in all copies.   Programmer(s) makes no    */
/*     representations about the suitability of this             */
/*     software for any purpose.  It is provided "as is" without */
/*     express or implied warranty.                              */
/*                                                               */
/*     These programs are to find the largest common             */
/*     substructures between two trees.                          */
/*****************************************************************/


#include <stdio.h>
#include <string.h>
#define min2(x,y) ( ((x) < (y))? (x) : (y) )
#define DELETECOST 1
#define INSERTCOST 1
#define RELABELCOST 1

#define MISMATCH 1
#define MATCH 2
#define CUT 3
#define BOTHCUT 4
#define RELABEL 5
```

```
#define TREE 6

#define TREE_SIZE 200
#define THRESHOLD 1
#define MAXTREE 400

/* ----- internal data structure ----- */
typedef struct
{
   char label[TREE_SIZE];
   int left_most[TREE_SIZE], comp_set[TREE_SIZE];
   int treesz, setsz;
} Tree;

typedef struct
{
   int l, r, tag;
} Pair;

char datatree[MAXTREE],pattree[MAXTREE];
int pre_post[TREE_SIZE];
int treedist[TREE_SIZE][TREE_SIZE][THRESHOLD];
int tempdist[TREE_SIZE][TREE_SIZE][THRESHOLD];
Tree tree1,tree2;
int MUTATION;
int interestsize;

/*
 * ================================================================
 *
 * Definition for internal data stucture and functions of stack.
 *
 * ================================================================
 */
typedef struct
{
   Pair p;
   short ty, err;
} Triple;

typedef struct
{
   int stp;
   Triple st[TREE_SIZE];
```

```
} Stack;

Stack S;

void initS();
void pushS();
Triple popS();
short emptyS();

/*
 * ================================================================
 *
 * Definition for internal data stucture and functions of queue.
 *
 * ================================================================
 */
typedef struct
{
   int q_length;
   Pair q_elems[TREE_SIZE];
} Queue;

Queue M;

void initQ();
void enqueue();
void printQ();
void printCUT();

/*
 * ================================================================
 *
 * Definition for internal data stucture of tree.
 *
 * ================================================================
 */
void evadist();
void evamap();
int tree_dist();
void mapping();
short max_pos();
FILE *In1, *In2, *outfile;
```

```
/*
 * ================================================================
 *
 * Main()
 *
 * ================================================================
 */
main(argc, argv)
int argc;
char *argv[];
{
   int dist;
   char tree1st[30], tree2nd[30], output[30], name[30];

   do{printf("%% Enter the file name of T1\n");
     printf("  (an example file can be found in file FILE1;\n");
     printf("  maximum size of trees is 200) [FILE1]: ");
     gets(name);
     if(name[0]==0)strcpy(tree1st,"FILE1");
     else strcpy(tree1st, name);
   } while((In1=fopen(tree1st, "r"))==NULL);
   do{printf("%% Enter the file name of T2\n");
     printf("  (an example file can be found in file FILE2;\n");
     printf("  maximum size of trees is 200) [FILE2]: ");
     gets(name);
     if(name[0]==0)strcpy(tree2nd,"FILE2");
     else strcpy(tree2nd, name);
   } while((In2=fopen(tree2nd, "r"))==NULL);
   do{printf("\n%% Enter the number of mutations allowed in
        searching\n");
     printf("  for similar motifs (default is 2; maximum number
        is 5) [2] ? ");
     gets(name);
     if (name[0] == '\0') MUTATION=0;
     else MUTATION=atoi(name);
   } while(MUTATION<0);
   do{printf("\n%% Enter the minimum size of interesting motifs\n");
     printf("  (default is 10) [10] ? "); gets(name);
     if (name[0] == '\0') interestsize=10;
     else interestsize = atoi(name);
   } while(interestsize<0);
   do{printf("\n%% Where the result should be stored (enter the
        file name) [data.out] ? ");
     gets(name);
```

```
    if(name[0]==0) strcpy(output,"data.out");
    else strcpy(output, name);
  } while((outfile=fopen(output, "w"))==NULL);
    fgets(datatree,MAXTREE,In1);fclose(In1);
    fgets(pattree,MAXTREE,In2);fclose(In2);
    prepareLD();

  if(interestsize==0)
    fprintf(outfile,"Interesting size = the largest one\n");
  else
    fprintf(outfile,"Interesting size = %d\n",interestsize);
  fprintf(outfile,"Number of mutations allowed = %d\n\n",MUTATION);

  fprintf(outfile, "T1\n"); fprintf(outfile, "%s\n", datatree);
  fprintf(outfile, "T2\n"); fprintf(outfile, "%s\n", pattree);
  dist=tree_dist();
  fclose(outfile);
  exit(0);
}


prepareLD()
{
  int lgh, i, k, m, n, stack[999], maxlgh;
  int index, st[999], p, gap;

  int hash(char tree_array[], int *idx)
  {
    int value=0;
    int factor=1;
    while (tree_array[*idx] != ')' && tree_array[*idx] != '(')
    {
      value += (int) tree_array[*idx] * factor;
      (*idx)++;
      factor++;
    }
    (*idx)--;
    return (value % 10000);
  }

  lgh=strlen(datatree);
  st[0]=0; index=p=1; gap=0;
  m=n=k=1;
  tree1.setsz=stack[0]=0;
```

```
  tree1.label[0]='x';
  for(i=0;i<lgh;i++){
    if(datatree[i]=='(') continue;
    if(datatree[i]==')')
    {
      tree1.left_most[n]=stack[--k];
      tree1.label[n++]=stack[--k];
      if(stack[k-1]!=tree1.left_most[n-1])
        tree1.comp_set[m++]=n-1;
    }
    else
    {
      stack[k++]=hash(datatree, &i);
      stack[k++]=n;
    }
  }
  tree1.treesz=maxlgh=n-1;
  tree1.setsz=m-1;
  lgh=strlen(pattree);
  m=n=k=1;
  st[0]=0; index=p=1; gap=0;
  tree2.setsz=stack[0]=0;
  tree2.label[0]='x';
  for(i=0;i<lgh;i++)
  {
    if(pattree[i]=='(') continue;
    if(pattree[i]==')')
    {
      tree2.left_most[n]=stack[--k];
      tree2.label[n++]=stack[--k];
      if(stack[k-1]!=tree2.left_most[n-1])
        tree2.comp_set[m++]=n-1;
    }
    else
    {
      stack[k++]=hash(pattree, &i);
      stack[k++]=n;
    }
  }
  tree2.treesz=n-1;
  tree2.setsz=m-1;
  if(n-1>maxlgh) maxlgh=n-1;
}
```

```
/*
 * ------------------------------------------------------------------
 *
 * tree_dist --
 *
 * Function of tree editing distance algorithm.
 *
 * Inputs:
 *      Two Normlized tree. (See tree.c for normlized tree)
 *
 * outputs:
 *      Editing distance between two trees.
 *
 * Results:
 *      Matrix 'treedist' and 'tempdist'.  In 'treedist' are all
 *      distance between any subtree to any subtree. In
 *      'tempdist' are all forest to forest distance D[1..i,1..j].
 *
 * Side Effects:
 *      See above.
 *
 * ------------------------------------------------------------------
 */
int tree_dist()
{
   register int i, j;
   int n, m, k, maxi_size, maxi_candi[3], flag=0;

   n=tree1.setsz;
   m=tree2.setsz;
   for(i=1; i<=n; i++)
     for(j=1; j<=m; j++)
       evadist(tree1.comp_set[i],tree2.comp_set[j],MUTATION);
   if(interestsize==0)
   {
     maxi_size=0;
     for(i=0; i< 3; i++) maxi_candi[i] = 0;
     for(i=1; i<=tree1.treesz; i++)
       for(j=1; j<=tree2.treesz; j++)
         if(treedist[i][j][MUTATION] > maxi_size)
         {
           maxi_size=treedist[i][j][MUTATION];
           maxi_candi[0]=i; maxi_candi[1]=j; maxi_candi[2]=MUTATION;
```

```
        }
    if(maxi_size!=0)
    {
      fprintf(outfile, "$$$$$$$$$$ largest approximately common
        substructure $$$$$$$$$$\n");
      fprintf(outfile,"tree size between T1(%d) and T2(%d) is
        %d\n", maxi_candi[0], maxi_candi[1],
        treedist[maxi_candi[0]][maxi_candi[1]][MUTATION]);
      mapping(maxi_candi[0],maxi_candi[1],MUTATION);
      printQ(outfile,&M);
      fprintf(outfile,"$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
    }
    else
      fprintf(outfile,"$$$$$$$$$$ Largest common substructure
        equals 0 $$$$$$$$$$\n");
  }
  else
  {
    for(i=tree1.treesz; i>=1; i--)
      for(j=tree2.treesz; j>=1; j--)
        if(treedist[i][j][MUTATION] > interestsize)
        {
          flag=1;
          fprintf(outfile,"$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
          fprintf(outfile,"tree size between T1(%d) and T2(%d)
            is %d\n", i, j, treedist[i][j][MUTATION]);
          mapping(i, j, MUTATION);
          printQ(outfile,&M);
          fprintf(outfile,"$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n");
          fprintf(outfile,"\n");
        }
     if(flag==0)
       fprintf(outfile,"$$$$$$$$$$ No answer is found for size
         %d $$$$$$$$$$\n",interestsize);
  }
  return(treedist[tree1.treesz][tree2.treesz][MUTATION]);
}


/*
 * ----------------------------------------------------------------
 *
 * evadist --
 *
```

```
 * function to compute tree distance between subtree rooted at k
 * and subtree rooted at l assuming that all subtree to subtree
 * distance needed in computation is available in 'treedist' matrix.
 *
 * Results:
 *      All subtree to subtree distance for subtree pair such that
 *      they have same left most leave decandants.  Result are in
 *      'treedist' matrix.
 *
 * Side Effects:
 *      In 'tempdist' matrix, from l(k)-1, l(l)-1 to k,l will
 *      be all forests distance generated in computation.
 *
 * ---------------------------------------------------------------
 */
void evadist(k, l, d)
int k, l, d;
{
   int lk, ll, m, tmpi, tmpj, dis;
   register int i, j, h;
   int item[10];
   register int mmin;

   tempdist[0][0][0]=0;
   lk=tree1.left_most[k];
   ll=tree2.left_most[l];
   tmpi=tree1.left_most[lk]; tmpj=tree2.left_most[ll];
   for(m=0; m<=d; m++)
     for(i=tmpi-1; i<=k; ++i)
       for(j=tmpj-1; j<=l; ++j)
         tempdist[i][j][m]=0;
   for(i=lk; i<=k; ++i)
      tempdist[i][0][0]=0;
   for(j=ll; j<=l; ++j)
      tempdist[0][j][0]=0;
   for(i=lk; i<=k; ++i)
     for(j=ll; j<=l; ++j)
     {
       if(tree1.left_most[i] != lk || tree2.left_most[j] != ll){
         tmpi=tree1.left_most[i]; tmpj=tree2.left_most[j];
         item[0]=tempdist[tmpi-1][j][0];
         item[1]=tempdist[i][tmpj-1][0];
         item[2]=tempdist[tmpi-1][tmpj-1][0]+treedist[i][j][0];
         tempdist[i][j][0]=max(item, 3);
```

```
    }
    else{
      item[2]=((tree1.label[i]==tree2.label[j])?
        tempdist[i-1][j-1][0]+2:0);
      tempdist[i][j][0]=item[2];
      treedist[i][j][0]=tempdist[i][j][0];
    }
  }
dis=d;
for(m=1; m<=dis; m++)
  tempdist[0][0][m]=0;
for(m=1; m<=dis; m++)
  for(i=lk; i<=k; ++i)
  {
    tmpi=tree1.left_most[i];
    item[0]=tempdist[i-1][0][m-1]+1;
    item[1]=tempdist[tmpi-1][0][m];
    tempdist[i][0][m]=max(item, 2);
  }
for(m=1; m<=dis; m++)
  for(j=ll; j<=l; ++j)
  {
    tmpj=tree2.left_most[j];
    item[0]=tempdist[0][j-1][m-1]+1;
    item[1]=tempdist[0][tmpj-1][m];
    tempdist[0][j][m]=max(item, 2);
  }
for(m=1; m<=dis; m++)
  for(i=lk; i<=k; ++i)
    for(j=ll; j<=l; ++j)
    {
      if(tree1.left_most[i]!=lk || tree2.left_most[j]!=ll)
      {
        tmpi=tree1.left_most[i]; tmpj=tree2.left_most[j];
        item[0]=tempdist[tmpi-1][j][m];
        item[1]=tempdist[i][tmpj-1][m];
        item[2]=tempdist[i-1][j][m-1]+1;
        item[3]=tempdist[i][j-1][m-1]+1;
        for(h=0;h<=m;h++)
           item[4+h]=tempdist[tmpi-1][tmpj-1][h]+
             treedist[i][j][m-h];
        tempdist[i][j][m]=max(item, m+5);
      }
      else
```

```
        {
          item[0]=tempdist[i-1][j][m-1]+1;
          item[1]=tempdist[i][j-1][m-1]+1;
          item[2]=((tree1.label[i]==tree2.label[j])?
            tempdist[i-1][j-1][m]+2:tempdist[i-1][j-1][m-1]+2);
          tempdist[i][j][m]=max(item, 3);
          treedist[i][j][m]=tempdist[i][j][m];
        }
      }
}


/*
 * ----------------------------------------------------------------
 *
 * mapping --
 *
 * Function to produce mapping between subtree i and subtree j
 * assuming that 'tree-dist' has been called before it.
 *
 * Results:
 *      Mapping between subtree i and subtree j, the result is
 *      in 'M'.
 *
 * Side Effects:
 *      'tempdist' matrix will be changed.
 *
 * ----------------------------------------------------------------
 */
void mapping(i, j, d)
int i, j, d;
{
   int k, l, n, m, dis, e;
   Triple t;

   n=tree1.treesz;
   m=tree2.treesz;
   dis=d;

   initS(&S);
   initQ(&M);
   if((i==n) && (j==m) && (dis==MUTATION))
       evamap(n, m, dis);
   else
```

```
    {
      evadist(i, j, dis);
      evamap(i, j, dis);
    }
    while(!emptyS(&S))
    {
      t=popS(&S);
      if(t.ty != 0 )
      {
        k=t.p.l;
        l=t.p.r;
        e=t.err;
        evadist(k, l, e);
        evamap(k, l, e);
      }
      else
        enqueue(t.p, &M);
    }
}


/*
 * ------------------------------------------------------------------
 *
 * evamap --
 *
 * Function to produce mapping.  The function back tracking the
 * forest distance matrix 'tempdist'.
 *
 * Results:
 *      None.
 *
 * Side Effects:
 *      Push some new triple to stack 'S'. Triple (i, j, 0) means
 *      that pair (i, j) will be in mapping.  Triple (i, j, 1)
 *      means that best mapping between subtree i and subtree j
 *      will be part of current mapping.
 *
 * ------------------------------------------------------------------
 */
void evamap(k, l, d)
int k, l, d;
{
   int lk, ll, dis, tmpi, tmpj;
```

```
int i, j, h;
int item[10];
short pos, tmp_pos;
Triple t;

lk=tree1.left_most[k];
ll=tree2.left_most[l];

i=k;
j=l;
dis=d;

while((i >= lk || j >= ll) && (dis >= 0))
{
  if(dis==0)
  {
    if(i<lk)
    {
      for(h=j;h>=ll;h--)
      {
        t.p.l=-1;
        t.p.r=h;
        t.p.tag=CUT;
        t.err=0;
        t.ty=0;
        pushS(t, &S);
      }
      break;
    }
    if(j<ll)
    {
      for(h=i;h>=lk;h--)
      {
        t.p.l=h;
        t.p.r=-1;
        t.p.tag=CUT;
        t.err=0;
        t.ty=0;
        pushS(t, &S);
      }
      break;
    }
    if(tree1.left_most[i] != lk || tree2.left_most[j] != ll)
    {
```

```
  tmpi=tree1.left_most[i]; tmpj=tree2.left_most[j];
  item[0]=tempdist[tmpi-1][j][0];
  item[1]=tempdist[i][tmpj-1][0];
  item[2]=tempdist[tmpi-1][tmpj-1][0]+treedist[i][j][0];
  pos=max_pos(item, 3);
  if(pos==0)
  {
    t.p.l=i;
    t.p.r=-1;
    t.p.tag=CUT;
    t.err=0;
    t.ty=0;
    pushS(t, &S);
    i=tmpi-1;
  }
  else if(pos==1)
  {
    t.p.l=-1;
    t.p.r=j;
    t.p.tag=CUT;
    t.err=0;
    t.ty=0;
    pushS(t, &S);
    j=tmpj-1;
  }
  else
  {
    t.p.l=i;
    t.p.r=j;
    t.p.tag=MATCH;
    t.err=0;
    t.ty=1;
    pushS(t, &S);
    i=tmpi-1;
    j=tmpj-1;
  }
}
else
{
  item[2]=((tree1.label[i]==tree2.label[j])?
    tempdist[i-1][j-1][0]+2:0);
  if(tree1.label[i]==tree2.label[j])
  {
    t.p.l=i;
```

```
        t.p.r=j;
        t.p.tag=MATCH;
        t.err=0;
        t.ty=0;
        pushS(t, &S);
        i=i-1;
        j=j-1;
      }
      else
      {
        t.p.l=i;
        t.p.r=j;
        t.p.tag=BOTHCUT;
        t.err=0;
        t.ty=0;
        pushS(t, &S);
        i=0;
        j=0;
      }
    }
  }
  else
  { /* dis != 0 */
    if(j<ll)
    {
      tmpi=tree1.left_most[i];
      item[0]=tempdist[i-1][0][dis-1]+1;
      item[1]=tempdist[tmpi-1][0][dis];
      pos=max_pos(item, 2);
      if(pos==0)
      {
        t.p.l=i;
        t.p.r=-1;
        t.p.tag=MISMATCH;
        t.err=dis-1;
        t.ty=0;
        pushS(t, &S);
        i=i-1;
        dis=dis-1;
      }
      else if(pos==1)
      {
        t.p.l=i;
        t.p.r=-1;
```

```
    t.p.tag=CUT;
    t.err=dis;
    t.ty=0;
    pushS(t, &S);
    i=tmpi-1;
  }
}
else if(i<lk)
{
  tmpj=tree2.left_most[j];
  item[0]=tempdist[0][j-1][dis-1]+1;
  item[1]=tempdist[0][tmpj-1][dis];
  pos=max_pos(item, 2);
  if(pos==0)
  {
    t.p.l=-1;
    t.p.r=j;
    t.p.tag=MISMATCH;
    t.err=dis-1;
    t.ty=0;
    pushS(t, &S);
    j=j-1;
    dis=dis-1;
  }
  else if(pos==1)
  {
    t.p.l=-1;
    t.p.r=j;
    t.p.tag=CUT;
    t.err=dis;
    t.ty=0;
    pushS(t, &S);
    j=tmpj-1;
  }
}
else if(tree1.left_most[i]!=lk || tree2.left_most[j]!=ll)
{
  tmpi=tree1.left_most[i]; tmpj=tree2.left_most[j];
  item[0]=tempdist[tmpi-1][j][dis];
  item[1]=tempdist[i][tmpj-1][dis];
  item[2]=tempdist[i-1][j][dis-1]+1;
  item[3]=tempdist[i][j-1][dis-1]+1;
  for(h=0;h<=dis;h++)
    item[4+h]=tempdist[tmpi-1][tmpj-1][h]+
```

```
    treedist[i][j][dis-h];
pos=max_pos(item, dis+5);
if(pos==0)
{
  t.p.l=i;
  t.p.r=-1;
  t.p.tag=CUT;
  t.err=dis;
  t.ty=0;
  pushS(t, &S);
  i=tmpi-1;
}
else if(pos==1)
{
  t.p.l=-1;
  t.p.r=j;
  t.p.tag=CUT;
  t.err=dis;
  t.ty=0;
  pushS(t, &S);
  j=tmpj-1;
}
else if(pos==2)
{
  t.p.l=i;
  t.p.r=-1;
  t.p.tag=MISMATCH;
  t.err=dis-1;
  t.ty=0;
  pushS(t, &S);
  i=i-1;
  dis=dis-1;
}
else if(pos==3)
{
  t.p.l=-1;
  t.p.r=j;
  t.p.tag=MISMATCH;
  t.err=dis-1;
  t.ty=0;
  pushS(t, &S);
  j=j-1;
  dis=dis-1;
}
```

```
  else
  {
    tmp_pos=pos-4;
    t.p.l=i;
    t.p.r=j;
    t.p.tag=TREE;
    t.err=dis-tmp_pos;
    t.ty=1;
    pushS(t, &S);
    i=tmpi-1;
    j=tmpj-1;
    dis=tmp_pos;
  }
}
else
{
  item[0]=tempdist[i-1][j][dis-1]+1;
  item[1]=tempdist[i][j-1][dis-1]+1;
  item[2]=((tree1.label[i]==tree2.label[j])?
    tempdist[i-1][j-1][dis]+2:tempdist[i-1][j-1][dis-1]+2);
  pos=max_pos(item, 3);
  if(pos==0)
  {
    t.p.l=i;
    t.p.r=-1;
    t.p.tag=MISMATCH;
    t.err=dis-1;
    t.ty=0;
    pushS(t, &S);
    i=i-1;
    dis=dis-1;
  }
  else if(pos==1)
  {
    t.p.l=-1;
    t.p.r=j;
    t.p.tag=MISMATCH;
    t.err=dis-1;
    t.ty=0;
    pushS(t, &S);
    j=j-1;
    dis=dis-1;
  }
  else
```

```
        {
          if(tree1.label[i]==tree2.label[j]) t.p.tag=MATCH;
          else t.p.tag=RELABEL;
          if(tree1.label[i]!=tree2.label[j]) dis=dis-1;
          t.p.l=i;
          t.p.r=j;
          t.ty=0;
          pushS(t, &S);
          i=i-1;
          j=j-1;
        }
      }
    }
  }
}


int max(item, count)
int *item, count;
{
   int i, m;

   m=item[0];
   for(i=1; i<count; i++)
   {
     if(m < item[i]) m=item[i];
   }

   return m;
}


/*
 * ----------------------------------------------------------------
 *
 * max_pos --
 *
 * Give the position of the maximum elements in array "item".
 *
 * Results:
 *      Potion of the maximum elements.
 *
 * Side Effects:
 *      None.
```

```
 *
 * ----------------------------------------------------------------
 */
short max_pos(item, count)
int *item;
int count;
{
   int i, m, pos;

   m=item[0];
   pos=0;
   for(i=0; i<count; i++)
   {
     if(m<item[i])
     {
       pos=i;
       m=item[i];
     }
   }
    return pos;
}


/*
 * ----------------------------------------------------------------
 *
 * initS --
 *
 * Initialize the stack 'S' to be empty.
 *
 * Results:
 *      None.
 *
 * Side Effects:
 *      See above.
 *
 * ----------------------------------------------------------------
 */
void initS(S)
Stack *S;
{
   S->stp=-1;
}
```

```
/*
 * ------------------------------------------------------------
 *
 * pushS --
 *
 * Push one triple element 't' into stack 'S'.
 *
 * Results:
 *      None.
 *
 * Side Effects:
 *      See above.
 *
 * ------------------------------------------------------------
 */
void pushS(t, S)
Triple t;
Stack *S;
{
   if(S->stp>=TREE_SIZE)
   {
     printf("stack overflow!\n");
     exit(1);
   }

   S->st[++S->stp]=t;
}


/*
 * ------------------------------------------------------------
 *
 * popS --
 *
 * Pop one triple element out from stack 'S'.
 *
 * Results:
 *      Return one triplr element.
 *
 * Side Effects:
 *      See above.
 *
 * ------------------------------------------------------------
```

```
 */
Triple popS(S)
Stack *S;
{
   if(S->stp<0)
   {
     printf("stack underflow!\n");
     exit(1);
   }

   return(S->st[S->stp--]);
}


/*
 * --------------------------------------------------------------
 *
 * emptyS --
 *
 * Boolean funtion to test if stack 'S' is empty.
 *
 * Results:
 *      Return 1 when 'S' is empty, else  return 0.
 *
 * Side Effects:
 *      none.
 *
 * --------------------------------------------------------------
 */
short emptyS(S)
Stack *S;
{
   if(S->stp<0)
     return(1);
   else
     return(0);
}


/*
 * --------------------------------------------------------------
 *
 * initQ --
 *
```

```
 * Initialize 'Q' to be empty.
 *
 * Results:
 *      None.
 *
 * Side Effects:
 *      See above.
 *
 * ----------------------------------------------------------------
 */
void initQ(Q)
Queue *Q;
{
   Q->q_length=0;
}


/*
 * ----------------------------------------------------------------
 *
 * enqueue --
 *
 * Insert pair 'p' to 'Q'.
 *
 * Results:
 *      None.
 *
 * Side Effects:
 *      See above.
 *
 * ----------------------------------------------------------------
 */
void enqueue(p, Q)
Pair p;
Queue *Q;
{
   if(Q->q_length>=TREE_SIZE)
   {
     printf("queue overflow!\n");
     exit(1);
   }

   Q->q_elems[Q->q_length]=p;
   Q->q_length+=1;
```

```
}


/*
 * ------------------------------------------------------------------
 *
 * printQ --
 *
 * Print content in 'Q'.
 *
 * Results:
 *      None.
 *
 * Side Effects:
 *      See above.
 *
 * ------------------------------------------------------------------
 */
void printQ(fp, Q)
FILE *fp;
Queue *Q;
{
   int i;

   for(i=0; i<Q->q_length; i++)
   {
     switch (Q->q_elems[i].tag)
     {
       case MISMATCH:
         if(Q->q_elems[i].r == -1)
           fprintf(fp, "DELETE T1(%d)\n", Q->q_elems[i].l);
         else
           fprintf(fp, "INSERT T2(%d)\n", Q->q_elems[i].r);
         break;

       case MATCH:
         fprintf(fp, "MATCH T1(%d)  T2(%d)\n", Q->q_elems[i].l,
           Q->q_elems[i].r);
         break;

       case CUT:
         if(Q->q_elems[i].r == -1)
           fprintf(fp, "CUT T1(%d)\n", Q->q_elems[i].l);
         else
```

```
            fprintf(fp, "CUT T2(%d)\n", Q->q_elems[i].r);
          break;

        case BOTHCUT:
          fprintf(fp, "BOTH CUT T1(%d) T2(%d)\n", Q->q_elems[i].l,
            Q->q_elems[i].r);
          break;

        case RELABEL:
          fprintf(fp, "RELABEL T1(%d) %c and T2(%d) %c\n",
            Q->q_elems[i].l, tree1.label[Q->q_elems[i].l],
            Q->q_elems[i].r, tree2.label[Q->q_elems[i].r]);
          break;
      }
    }
}
```

```
/*****************************************************************/
/*                                                               */
/*     (c) copyright 1999                                        */
/*         All rights reserved                                   */
/*     Programs written by Chia-Yo Chang (NJIT)                  */
/*                                                               */
/*     Permission to use, copy, modify, and distribute this      */
/*     software and its documentation for any purpose and without */
/*     fee is hereby granted, provided that this copyright       */
/*     notice appears in all copies.   Programmer(s) makes no    */
/*     representations about the suitability of this             */
/*     software for any purpose.  It is provided "as is" without */
/*     express or implied warranty.                              */
/*                                                               */
/*     These programs are to find the frequently occurring       */
/*     patterns in a set of trees.                               */
/*****************************************************************/

include <stdio.h>
#include <string.h>
#include <memory.h>
#define min2(x,y) ( ((x) < (y))? (x) : (y) )
#define DELETECOST 1
#define INSERTCOST 1
#define RELABELCOST 1

#define MISMATCH 1
#define MATCH 2
#define CUT 3
#define BOTHCUT 4
#define RELABEL 5
#define TREE 6

/* ----- constant ----- */
#define TREE_SIZE 300
#define THRESHOLD 10
#define MAXTREE 1000

/* ----- internal data structure ----- */
typedef struct
{
    char label[TREE_SIZE];
    int left_most[TREE_SIZE], comp_set[TREE_SIZE];
```

```
    int treesz, setsz;
} Tree;

typedef struct
{
    int l, r, tag;
} Pair;

char T1tree[MAXTREE],T2tree[MAXTREE];
int parent1[TREE_SIZE], parent2[TREE_SIZE];
int treedist[TREE_SIZE][TREE_SIZE][THRESHOLD];
int treedist1[TREE_SIZE][TREE_SIZE];
int tempdist[TREE_SIZE][TREE_SIZE][THRESHOLD];
int tempdist1[TREE_SIZE][TREE_SIZE];
Tree tree1,tree2;
int MUTATION;
int interestsize;

/*
 * ============================================================
 *
 * stack.h ---
 *
 * Definition for internal data stucture and functions of stack.
 *
 * ============================================================
 */

typedef struct
{
    Pair p;
    short ty, err;
} Triple;

typedef struct
{
    int stp;
    Triple st[TREE_SIZE];
} Stack;

Stack S;

void initS();
void pushS();
```

```
Triple popS();
short emptyS();

/*
 * ================================================================
 *
 * queue.h ---
 *
 * Definition for internal data stucture and functions of queue.
 *
 * ================================================================
 */

typedef struct
{
    int q_length;
    Pair q_elems[TREE_SIZE];
} Queue;

Queue M;

void initQ();
void enqueue();
void printQ();
void printCUT();

/*
 * ================================================================
 *
 * tree.h ---
 *
 * Definition for internal data stucture of tree.
 *
 * ================================================================
 */
void evadist();
void evamap();
int tree_dist();
void mapping();
short max_pos();
long int Xn;
struct link{
     int child;
     struct link *next;
```

```
};
struct tree_node{
     int lvalue;
     char lable;
     struct link *tree_ptr;
};
struct tree_node tree[MAXTREE +1];
int DATALINES, maxsize, minsize, occur, MAXSAMPLE;
char datafile[30], output[30];
FILE *tree1fp, *tree2fp, *candidate, *vldc_candidate, *outfile;
int tree1_size, tree2_size;
float PROB_THRESHOLD;
#define MAXOPTLINE 200
#define CREATE 1
#define USED 0
#define EQUAL 0
#define NONEQUAL 1
#define MAXVLDCLINE  1000
#define MAXDATALINE 50000
#define TRUE 0
#define FALSE 1
#define NEW(x)  (struct x *)malloc(sizeof(struct x))
int VLDCLINE, flag=0;
int left_hand, right_hand;

/*#define DISPLAY  1*/
#ifdef DISPLAY
   char K=0,A[4]={'-','\\','|','/'};
#endif
#define SPEED  100
#define SPEED1  1


main(argc, argv)
int argc;
char *argv[];
{
   int dist, line_index=0, sample_ratio;
   int T1index, T2index;
   char name[20];
   FILE *fp, *fp1, *fp2;
   extern int optind;

   if(optind!=argc-4)
```

```
{
  perror(optind<argc-4 ? "too many arguments":"missing
    file name");
  exit(1);
}
T1index=T2index=0; Xn=23311;
strcpy(datafile,argv[1]);
if((fp=fopen(datafile, "r"))==NULL)
  printf("Cannot open file [%s]\n", datafile); exit(1);
fp1=fopen("tmpdata", "w");
while(fgets(T1tree, MAXTREE, fp) != NULL)
{
  if(T1tree[0] == '(')
  {
    fprintf(fp1,"%s", T1tree);
    line_index++;
  }
} fclose(fp1);
fclose(fp); DATALINES = line_index;
if(line_index>200)
{
  printf("Err: Too many trees\n");
  exit(0);
}
if(line_index==0)
{
  printf("Err: Empty file!! \n");
  exit(0);
}
minsize=atoi(argv[2]);
if(minsize <= 0)
{
  printf("Minimum size must greater than zero\n"); exit(1);
}
occur=atoi(argv[3]);
if(occur <= 0 || occur >DATALINES)
{
  printf("Occurrence number must in between zero and %d\n",
    DATALINES);
  exit(1);
}
MUTATION=atoi(argv[4]);
if(MUTATION < 0 || MUTATION >5)
{
```

```
    printf("Mutation only allowed from 0 to 5\n");
    exit(1);
  }

  printf("  Occurrence number     Motif\n");
  printf("  -----------------     -----\n");

  if(DATALINES<20) MAXSAMPLE=DATALINES;
  else if(DATALINES<100 && DATALINES>=20)
    MAXSAMPLE=(int)(0.3*DATALINES);
  else MAXSAMPLE=(int)(0.1*DATALINES);

  #ifdef DISPLAY
    printf("%c  Running.....       0 motif found\b\b\b\b\b\b\b
      \b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b",
      A[K++]);
    fflush(stdout);
  #endif

  create_sample();
  candidate = fopen("tmp00c", "w");
  vldc_candidate = fopen("tmp00v", "w");
  fp1 = fopen("tmpsample", "r");
  while (fgets(T1tree, MAXTREE, fp1) != NULL)
  {
    if(fgets(T2tree, MAXTREE, fp1)==NULL) break;
    prepareLD(); dist=tree_dist();
  }
  if(flag==0)
    printf("$$$$$$$$$$ No answer is found $$$$$$$$$$\n");
  fclose(fp2); fclose(fp1);
  fclose(candidate); fclose(vldc_candidate);
  uniq_candidate(); uniq_vldc();
  system("rm -f tmp00c"); system("rm -f tmp00v");
  system("rm -f tmptree1"); system("rm -f tmptree2");
  vldc();
  system("rm -f tmpsample"); system("rm -f tmpvldc");
  system("rm -f tmpcandidate"); system("rm -f tmpdata");

  exit(0);
}


uniq_candidate()
```

```
{
   int i, FLAG, line1, line2;
   FILE *fp1, *fp2, *uniq;
   char tree[10], T1[MAXTREE + 1], T2[MAXTREE + 1];

   fp1=fopen("tmp00c", "r"); fp2 = fopen("tmp00c", "r");
   uniq=fopen("tmpcandidate", "w"); line1 = 0;
   while(fgets(tree, 10, fp1) != NULL)
   {
     fgets(T1, MAXTREE, fp1);
     fseek(fp2, 0, 0); line2 = 0;
     FLAG=NONEQUAL;
     while(line2<line1)
     {
       fgets(tree, 10, fp2);
       fgets(T2, MAXTREE, fp2); line2++;
       if(strcmp(T1, T2)==0)
       {
         FLAG=EQUAL; break;
       }
     }
     if(FLAG==NONEQUAL)
     {
       fprintf(uniq, "%s", tree);
       fprintf(uniq, "%s", T1);
     }
     line1++;
   }
   fclose(fp1); fclose(fp2); fclose(uniq);
}


uniq_vldc()
{
   int i, FLAG, line1, line2;
   FILE *fp1, *fp2, *uniq;
   char tree[10], T1[MAXTREE + 1], T2[MAXTREE + 1];

   fp1=fopen("tmp00v", "r");
   fp2=fopen("tmp00v", "r");
   uniq = fopen("tmpvldc", "w"); line1 = 0;
   while(fgets(tree, 10, fp1)!=NULL)
   {
     fgets(T1, MAXTREE, fp1);
```

```
    fseek(fp2, 0, 0); line2 = 0;
    FLAG=NONEQUAL;
    while(line2<line1)
    {
      fgets(tree, 10, fp2);
      fgets(T2, MAXTREE, fp2); line2++;
      if(strcmp(T1, T2)==0){
        FLAG=EQUAL; break;
      }
    }
    if(FLAG==NONEQUAL){
      fprintf(uniq, "%s", tree);
      fprintf(uniq, "%s", T1);
    }
    line1++;
  }
  fclose(fp1); fclose(fp2); fclose(uniq);
}


float random_generator()
{
  float random;

  Xn=(25173*Xn+13849)%65536;
  random=Xn/65536.0;
  return(random);
}


create_sample()
{
   int i=0, j, line_index=0, line[MAXDATALINE], FLAG;
   FILE *fp, *sample;
   int temp;

   sample=fopen("tmpsample", "w");
   fp=fopen("tmpdata", "r");
   for(i=0;i<MAXSAMPLE;i++)
   {
     if(fgets(T1tree, MAXTREE, fp)==NULL) break;
     fprintf( sample,"%s", T1tree);
   }
   fclose(fp);
```

```
   fclose(sample);
}


filetotree(tree_index)
int tree_index;
{
   int index, par_index, par_lvalue;
   struct link *lptr, *ptr;
   FILE *fp;
   int table[MAXTREE +1];

   if(tree_index==1) fp=fopen("tmptree1", "r");
   else fp=fopen("tmptree2", "r");
   for(index=0; index<=MAXTREE; index++)
   {
     table[index]=0;
     tree[index].lvalue=0;
     tree[index].lable=' ';
     while(tree[index].tree_ptr!=NULL)
     {
       ptr = tree[index].tree_ptr;
       tree[index].tree_ptr = tree[index].tree_ptr->next;
       free(ptr);
     }
   }
   index=1;
   while(feof(fp)==0)
   {
     fscanf(fp, "%d %c ", &tree[index].lvalue, &tree[index].lable);
     table[tree[index].lvalue]=index;
     if(index!=1)
     {
       if(tree_index==1) par_lvalue=parent1[tree[index].lvalue];
       else par_lvalue=parent2[tree[index].lvalue];
       par_index=table[par_lvalue];
       if(tree[par_index].tree_ptr==NULL){
         tree[par_index].tree_ptr=NEW(link);
         tree[par_index].tree_ptr->child=index;
         tree[par_index].tree_ptr->next=NULL;
       }
       else{
         lptr=NEW(link);
         lptr->child=index;
```

```
            lptr->next=tree[par_index].tree_ptr;
            tree[par_index].tree_ptr=lptr;
          }
        }
        index++;
      }
      fclose(fp);
      if(tree_index==1) tree1_size=index-1;
      else tree2_size=index-1;
}


treetopre(i)
int i;
{
      struct link *link_ptr;
      int child_index;

      if(tree[i].tree_ptr==NULL)
      {
        fprintf(candidate, "(%c)", tree[i].lable);
        return;
      }
      fprintf(candidate, "(%c", tree[i].lable);
      link_ptr=tree[i].tree_ptr;
      while(link_ptr!=NULL){
        child_index=link_ptr->child;
        treetopre(child_index);
        link_ptr=link_ptr->next;
      }
      fprintf(candidate,")");
}


create_candidate(i, j, k)
int i, j, k;
{
      left_hand=right_hand=0;
      tree1fp=fopen("tmptree1", "w");
      tree2fp=fopen("tmptree2", "w");
      mapping(i, j, k); printQ(&M);
      fclose(tree1fp); fclose(tree2fp);
      if(left_hand>=minsize)
      {
```

```
      fprintf(candidate, "tree1\n");
      filetotree(1); treetopre(1);
      fprintf(candidate, ";\n");
    }
}


add_vldc_tree(tree_index)
int tree_index;
{
   int i, tree_size;

   tree[0].lable='*';
   tree[0].lvalue=0;
   tree[0].tree_ptr=NEW(link);
   tree[0].tree_ptr->child=1;
   tree[0].tree_ptr->next=NULL;
   if(tree_index==1) tree_size=tree1_size;
   else tree_size=tree2_size;
   tree[tree_size+1].lable='*';
   tree[tree_size+1].lvalue=0;
   tree[tree_size+1].tree_ptr=NULL;
   for(i=1; i<=tree_size; i++)
     if(tree[i].tree_ptr==NULL)
     {
       tree[i].tree_ptr=NEW(link);
       tree[i].tree_ptr->child=tree_size+1;
       tree[i].tree_ptr->next=NULL;
     }
}


create_vldc_candidate(i, j, k)
int i, j, k;
{
   left_hand=right_hand=0;
   tree1fp=fopen("tmptree1", "w");
   tree2fp=fopen("tmptree2", "w");
   mapping(i, j, k); printQ(&M);
   fclose(tree1fp); fclose(tree2fp);
   if(left_hand>=minsize)
   {
     fprintf(vldc_candidate, "tree1\n");
     filetotree(1); add_vldc_tree(1); vldc_treetopre(0);
```

```
    fprintf(vldc_candidate, ";\n");
  }
}


vldc_treetopre(i)
int i;
{
  struct link *link_ptr;
  int child_index;

  if(tree[i].tree_ptr==NULL)
  {
    fprintf(vldc_candidate, "(%c)", tree[i].lable);
    return;
  }
  fprintf(vldc_candidate, "(%c", tree[i].lable);
  link_ptr=tree[i].tree_ptr;
  while(link_ptr!=NULL){
    child_index=link_ptr->child;
    vldc_treetopre(child_index);
    link_ptr=link_ptr->next;
  }
  fprintf(vldc_candidate,")");
}


prepareLD()
{
  int lgh, i, k, m, n, stack[999],maxlgh;
  int index, st[999], p, r, s, t, gap;
  int POST[TREE_SIZE];

  lgh=strlen(T1tree);
  st[0]=0; index=p=1; gap=0;
  m=n=k=r=s=t=1;
  tree1.setsz=stack[0]=0;
  tree1.label[0]='x';
  for(i=0;i<lgh;i++)
  {
    if(T1tree[i]=='(') continue;
    if(T1tree[i]==')')
    {
      POST[st[--p]]=n;
```

```
    tree1.left_most[n]=stack[--k];
    tree1.label[n++]=stack[--k];
    if(stack[k-1]!=tree1.left_most[n-1])
      tree1.comp_set[m++]=n-1;
  }
  else
  {
    stack[k++]=T1tree[i];
    stack[k++]=n; st[p++]=index++;
  }
}
tree1.treesz=maxlgh=n-1; tree1.setsz=m-1;
for(i=0;i<lgh;i++)
{
  if(T1tree[i]=='(') continue;
  if(T1tree[i]==')')
  {
    --s; parent1[t++]=stack[s-1];
  }
  else stack[s++]=POST[r++];
}

lgh=strlen(T2tree);
m=n=k=r=s=t=1;
st[0]=0; index=p=1; gap=0;
tree2.setsz=stack[0]=0;
tree2.label[0]='x';
for(i=0;i<lgh;i++)
{
  if(T2tree[i]=='(') continue;
  if(T2tree[i]==')')
  {
    POST[st[--p]]=n;
    tree2.left_most[n]=stack[--k];
    tree2.label[n++]=stack[--k];
    if(stack[k-1]!=tree2.left_most[n-1])
      tree2.comp_set[m++]=n-1;
  }
  else
  {
    stack[k++]=T2tree[i];
    stack[k++]=n; st[p++]=index++;
  }
}
```

```
    tree2.treesz=n-1; tree2.setsz=m-1;
    if(n-1>maxlgh) maxlgh=n-1;
    for(i=0;i<lgh;i++)
    {
      if(T2tree[i]=='(') continue;
      if(T2tree[i]==')')
      {
        --s; parent2[t++]=stack[s-1];
      }
      else stack[s++]=POST[r++];
    }
}


/*
 * ------------------------------------------------------------
 *
 * tree_dist --
 *
 * Function of tree editing distance algorithm.
 *
 * Inputs:
 *      Two Normlized tree. (See tree.c for normlized tree)
 *
 * outputs:
 *      Editing distance between two trees.
 *
 * Results:
 *      Matrix 'treedist' and 'tempdist'.  In 'treedist' are all
 *      distance between any subtree to any subtree.
 *      In 'tempdist' are all forest to forest distance
 *      D[1..i,1..j].
 *
 * Side Effects:
 *      See above.
 *
 * ------------------------------------------------------------
 */
int tree_dist()
{
    register int i, j;
    int n, m, k, size;
    struct link *ptr;
```

```
    size=2*minsize-MUTATION;
    n=tree1.setsz;
    m=tree2.setsz;
    for(i=1; i<=n; i++)
      for(j=1; j<=m; j++)
        evadist(tree1.comp_set[i],tree2.comp_set[j],MUTATION);
    for(i=tree1.treesz; i>=1; i--)
      for(j=tree2.treesz; j>=1; j--)
        if(treedist[i][j][MUTATION] >= size )
        {
          flag=1;
          create_candidate(i, j, MUTATION);
          create_vldc_candidate(i, j, MUTATION);
        }
    for(i=0; i <= MAXTREE; i++)
    {
      while(tree[i].tree_ptr != NULL)
      {
        ptr = tree[i].tree_ptr;
        tree[i].tree_ptr = tree[i].tree_ptr->next;
        free(ptr);
      }
    }

    return(treedist[tree1.treesz][tree2.treesz][MUTATION]);
}


/*
 * ----------------------------------------------------------------
 *
 * evadist --
 *
 * function to compute tree distance between subtree rooted at k
 * and subtree rooted at l assuming that all subtree to subtree
 * distance needed in computation is available in 'treedist' matrix.
 *
 * Results:
 *      All subtree to subtree distance for subtree pair such that
 *      they have same left most leave decandants.  Result are in
 *      'treedist' matrix.
 *
 * Side Effects:
 *      In 'tempdist' matrix, from l(k)-1, l(l)-1 to k,l will
```

```
 *      be all forests distance generated in computation.
 *
 * ---------------------------------------------------------------
 */
void evadist(k, l, d)
int k, l, d;
{
   int lk, ll, m, tmpi, tmpj, dis;
   register int i, j, h;
   int item[10];
   register int mmin;

   #ifdef DISPLAY
     static short c=0;
   #endif

   tempdist[0][0][0]=0;
   lk=tree1.left_most[k];
   ll=tree2.left_most[l];
   tmpi=tree1.left_most[lk]; tmpj=tree2.left_most[ll];
   for(m=0; m<=d; m++)
     for(i=tmpi-1; i<=k; ++i)
       for(j=tmpj-1; j<=l; ++j)
         tempdist[i][j][m]=0;
   for(i=lk; i<=k; ++i)
     tempdist[i][0][0]=0;
   for(j=ll; j<=l; ++j)
     tempdist[0][j][0]=0;
   for(i=lk; i<=k; ++i)
     for(j=ll; j<=l; ++j)
     {
       #ifdef DISPLAY
          if(++c>SPEED)
            {
              c=0;
              if(K==4) K=0;
              printf("%c\b",A[K++]); fflush(stdout);
            }
       #endif

       if(tree1.left_most[i]!=lk || tree2.left_most[j]!=ll)
       {
         tmpi=tree1.left_most[i]; tmpj=tree2.left_most[j];
         item[0]=tempdist[tmpi-1][j][0];
```

```
      item[1]=tempdist[i][tmpj-1][0];
      item[2]=tempdist[tmpi-1][tmpj-1][0]+treedist[i][j][0];
      tempdist[i][j][0]=max(item, 3);
    }
    else
    {
      item[2]=((tree1.label[i]==tree2.label[j])
        ?tempdist[i-1][j-1][0]+2:0);
      tempdist[i][j][0]=item[2];
      treedist[i][j][0]=tempdist[i][j][0];
    }
}
dis=d;
for(m=1; m<=dis; m++)
  tempdist[0][0][m]=0;
for(m=1; m<=dis; m++)
  for(i=lk; i<=k; ++i)
  {
    tmpi=tree1.left_most[i];
    item[0]=tempdist[i-1][0][m-1]+1;
    item[1]=tempdist[tmpi-1][0][m];
    tempdist[i][0][m]=max(item, 2);
  }
for(m=1; m<=dis; m++)
  for(j=ll; j<=l; ++j)
  {
    tmpj=tree2.left_most[j];
    item[0]=tempdist[0][j-1][m-1]+1;
    item[1]=tempdist[0][tmpj-1][m];
    tempdist[0][j][m]=max(item, 2);
  }
for(m=1; m<=dis; m++)
  for(i=lk; i<=k; ++i)
    for(j=ll; j<=l; ++j)
    {
      if(tree1.left_most[i]!=lk || tree2.left_most[j]!=ll)
      {
        tmpi=tree1.left_most[i]; tmpj=tree2.left_most[j];
        item[0]=tempdist[tmpi-1][j][m];
        item[1]=tempdist[i][tmpj-1][m];
        item[2]=tempdist[i-1][j][m-1]+1;
        item[3]=tempdist[i][j-1][m-1]+1;
        for(h=0;h<=m;h++)
          item[4+h]=tempdist[tmpi-1][tmpj-1][m]
```

```
                +treedist[i][j][m-h];
            tempdist[i][j][m]=max(item, m+5);
          }
          else
          {
            item[0]=tempdist[i-1][j][m-1]+1;
            item[1]=tempdist[i][j-1][m-1]+1;
            item[2]=((tree1.label[i]==tree2.label[j])
              ?tempdist[i-1][j-1][m]+2:tempdist[i-1][j-1][m-1]+2);
            tempdist[i][j][m]=max(item, 3);
            treedist[i][j][m]=tempdist[i][j][m];
  }
        }
}


/*
 * ----------------------------------------------------------------
 *
 * mapping --
 *
 * Function to produce mapping between subtree i and subtree j
 * assuming that 'tree-dist' has been called before it.
 *
 * Results:
 *      Mapping between subtree i and subtree j,
 *      the result is in 'M'.
 *
 * Side Effects:
 *      'tempdist' matrix will be changed.
 *
 * ----------------------------------------------------------------
 */
void mapping(i, j, d)
int i, j, d;
{
   int k, l, n, m, dis, e;
   Triple t;

   n=tree1.treesz;
   m=tree2.treesz;
   dis=d;

   initS(&S);
```

```
    initQ(&M);
    if((i==n)&&(j==m)&&(dis==MUTATION))
      evamap(n, m, dis);
    else
    {
      evadist(i, j, dis);
      evamap(i, j, dis);
    }
    while(!emptyS(&S))
    {
      t=popS(&S);
      if(t.ty!=0)
      {
        k=t.p.l;
        l=t.p.r;
        e=t.err;
        evadist(k, l, e);
        evamap(k, l, e);
      }
      else
        enqueue(t.p, &M);
    }
}


/*
 * ----------------------------------------------------------------
 *
 * evamap --
 *
 * Function to produce mapping.  The function back tracking the
 * forest distance matrix 'tempdist'.
 *
 * Results:
 *      None.
 *
 * Side Effects:
 *      Push some new triple to stack 'S'. Triple (i, j, 0) means
 *      that pair (i, j) will be in mapping.  Triple (i, j, 1)
 *      means that best mapping between subtree i and subtree j
 *      will be part of current mapping.
 *
 * ----------------------------------------------------------------
 */
```

```
void evamap(k, l, d)
int k, l, d;
{
   int lk, ll, dis, tmpi, tmpj;
   int i, j, h;
   int item[10];
   short pos, tmp_pos;
   Triple t;

   lk=tree1.left_most[k];
   ll=tree2.left_most[l];

   i=k;
   j=l;
   dis=d;

   while((i>=lk) || (j>=ll) && (dis>=0))
   {
     if(dis==0)
     {
       if(i<lk)
       {
         for(h=j;h>=ll;h--)
         {
           t.p.l=-1;
           t.p.r=h;
           t.p.tag=CUT;
           t.err=0;
           t.ty=0;
           pushS(t, &S);
         }
         break;
       }
       if(j<ll)
       {
         for(h=i;h>=lk;h--)
         {
           t.p.l=h;
           t.p.r=-1;
           t.p.tag=CUT;
           t.err=0;
           t.ty=0;
           pushS(t, &S);
         }
```

```
  break;
}
if(tree1.left_most[i] != lk || tree2.left_most[j] != ll)
{
  tmpi=tree1.left_most[i]; tmpj=tree2.left_most[j];
  item[0]=tempdist[tmpi-1][j][0];
  item[1]=tempdist[i][tmpj-1][0];
  item[2]=tempdist[tmpi-1][tmpj-1][0]+treedist[i][j][0];
  pos=max_pos(item, 3);
  if(pos==0)
  {
    t.p.l=i;
    t.p.r=-1;
    t.p.tag=CUT;
    t.err=0;
    t.ty=0;
    pushS(t, &S);
    i=tmpi-1;
  }
  else if(pos == 1)
  {
    t.p.l=-1;
    t.p.r=j;
    t.p.tag=CUT;
    t.err=0;
    t.ty=0;
    pushS(t, &S);
    j=tmpj-1;
  }
  else
  {
    t.p.l=i;
    t.p.r=j;
    t.p.tag=MATCH;
    t.err=0;
    t.ty=1;
    pushS(t, &S);
    i=tmpi-1;
    j=tmpj-1;
  }
}
else
{
  item[2]=((tree1.label[i]==tree2.label[j])
```

```
      ?tempdist[i-1][j-1][0]+2:0);
    if(tree1.label[i]==tree2.label[j])
    {
      t.p.l=i;
      t.p.r=j;
      t.p.tag=MATCH;
      t.err=0;
      t.ty=0;
      pushS(t, &S);
      i=i-1;
      j=j-1;
    }
    else
    {
      t.p.l=i;
      t.p.r=j;
      t.p.tag=BOTHCUT;
      t.err=0;
      t.ty=0;
      pushS(t, &S);
      i=0;
      j=0;
    }
  }
}
else{
  if(j<ll)
  {
    tmpi=tree1.left_most[i];
    item[0]=tempdist[i-1][0][dis-1]+1;
    item[1]=tempdist[tmpi-1][0][dis];
    pos=max_pos(item, 2);
    if(pos==0)
    {
      t.p.l=i;
      t.p.r=-1;
      t.p.tag=MISMATCH;
      t.err=dis-1;
      t.ty=0;
      pushS(t, &S);
      i=i-1;
      dis=dis-1;
    }
    else if(pos==1)
```

```
  {
    t.p.l=i;
    t.p.r=-1;
    t.p.tag=CUT;
    t.err= is;
    t.ty=0;
    pushS(t, &S);
    i=tmpi-1;
  }
}
else if(i<lk)
{
  tmpj=tree2.left_most[j];
  item[0]=tempdist[0][j-1][dis-1]+1;
  item[1]=tempdist[0][tmpj-1][dis];
  pos=max_pos(item, 2);
  if(pos==0)
  {
    t.p.l=-1;
    t.p.r=j;
    t.p.tag=MISMATCH;
    t.err=dis-1;
    t.ty=0;
    pushS(t, &S);
    j=j-1;
    dis=dis-1;
  }
  else if(pos==1)
  {
    t.p.l=-1;
    t.p.r=j;
    t.p.tag=CUT;
    t.err=dis;
    t.ty=0;
    pushS(t, &S);
    j=tmpj-1;
  }
}
else if (tree1.left_most[i]!=lk || tree2.left_most[j]!=ll)
{
  tmpi=tree1.left_most[i]; tmpj=tree2.left_most[j];
  item[0]=tempdist[tmpi-1][j][dis];
  item[1]=tempdist[i][tmpj-1][dis];
  item[2]=tempdist[i-1][j][dis-1]+1;
```

```
item[3]=tempdist[i][j-1][dis-1]+1;
for(h=0;h<=dis;h++)
  item[4+h]=tempdist[tmpi-1][tmpj-1][h]
    +treedist[i][j][dis-h];
pos=max_pos(item, dis+5);
if(pos==0)
{
  t.p.l=i;
  t.p.r=-1;
  t.p.tag=CUT;
  t.err=dis;
  t.ty=0;
  pushS(t, &S);
  i=tmpi-1;
}
else if(pos==1)
{
  t.p.l=-1;
  t.p.r=j;
  t.p.tag=CUT;
  t.err=dis;
  t.ty=0;
  pushS(t, &S);
  j=tmpj-1;
}
else if(pos==2)
{
  t.p.l=i;
  t.p.r=-1;
  t.p.tag=MISMATCH;
  t.err=dis-1;
  t.ty=0;
  pushS(t, &S);
  i=i-1;
  dis=dis-1;
}
else if(pos==3)
{
  t.p.l=-1;
  t.p.r=j;
  t.p.tag=MISMATCH;
  t.err=dis-1;
  t.ty=0;
  pushS(t, &S);
```

```
      j=j-1;
      dis=dis-1;
    }
    else
    {
      tmp_pos=pos-4;
      t.p.l=i;
      t.p.r=j;
      t.p.tag=TREE;
      t.err=dis-tmp_pos;
      t.ty=1;
      pushS(t, &S);
      i=tmpi-1;
      j=tmpj-1;
      dis=tmp_pos;
    }
  }
  else
  {
    item[0]=tempdist[i-1][j][dis-1]+1;
    item[1]=tempdist[i][j-1][dis-1]+1;
    item[2]=((tree1.label[i]==tree2.label[j])
      ?tempdist[i-1][j-1][dis]+2:tempdist[i-1][j-1][dis-1]+2);
    pos=max_pos(item, 3);
    if(pos==0)
    {
      t.p.l=i; /* mismatch */
      t.p.r=-1;
      t.p.tag=MISMATCH;
      t.err=dis-1;
      t.ty=0;
      pushS(t, &S);
      i=i-1;
      dis=dis-1;
    }
    else if (pos == 1) {
      t.p.l=-1; /* mismatch */
      t.p.r=j;
      t.p.tag=MISMATCH;
      t.err=dis-1;
      t.ty=0;
      pushS(t, &S);
      j=j-1;
      dis=dis-1;
```

```
        }
        else {
          if(tree1.label[i]==tree2.label[j]) t.p.tag=MATCH;
          else t.p.tag=RELABEL;
          if(tree1.label[i]!=tree2.label[j]) dis=dis-1;
          t.p.l=i;
          t.p.r=j;
          t.ty=0;
          pushS(t, &S);
          i=i-1;
          j=j-1;
        }
      }
    }
  }
}


int max(item, count)
int *item, count;
{
   int i, m;

   m=item[0];
   for(i=1; i<count; i++)
   {
     if(m<item[i]) m=item[i];
   }

   return m;
}


int min(item, count)
int *item, count;
{
   int i, m;

   m=item[0];
   for(i=1; i<count; i++)
   {
     if(m>item[i]) m=item[i];
   }
```

```
   return m;
}


/*
 * ----------------------------------------------------------------
 *
 * max_pos --
 *
 * Give the position of the maximum elements in array "item".
 *
 * Results:
 *      Potion of the maximum elements.
 *
 * Side Effects:
 *      None.
 *
 * ----------------------------------------------------------------
 */
short max_pos(item, count)
int *item;
int count;
{
   int i, m, pos;

   m=item[0];
   pos=0;
   for(i=0; i<count; i++)
   {
     if(m<item[i])
     {
       pos=i;
       m=item[i];
     }
   }
    return pos;
}


/*
 * ----------------------------------------------------------------
 *
 * initS --
 *
```

```
 * Initialize the stack 'S' to be empty.
 *
 * Results:
 *      None.
 *
 * Side Effects:
 *      See above.
 *
 * ----------------------------------------------------------------
 */
void initS(S)
Stack *S;
{
   S->stp=-1;
}


/*
 * ----------------------------------------------------------------
 *
 * pushS --
 *
 * Push one triple element 't' into stack 'S'.
 *
 * Results:
 *      None.
 *
 * Side Effects:
 *      See above.
 *
 * ----------------------------------------------------------------
 */
void pushS(t, S)
Triple t;
Stack *S;
{
   if(S->stp>=TREE_SIZE)
   {
     printf("stack overflow!\n");
     exit(1);
   }

   S->st[++S->stp]=t;
}
```

```
/*
 * --------------------------------------------------------------
 *
 * popS --
 *
 * Pop one triple element out from stack 'S'.
 *
 * Results:
 *      Return one triplr element.
 *
 * Side Effects:
 *      See above.
 *
 * --------------------------------------------------------------
 */
Triple popS(S)
Stack *S;
{
   if(S->stp<0)
   {
     printf("stack underflow!\n");
     exit(1);
   }

   return(S->st[S->stp--]);
}


/*
 * --------------------------------------------------------------
 *
 * emptyS --
 *
 * Boolean funtion to test if stack 'S' is empty.
 *
 * Results:
 *      Return 1 when 'S' is empty, else  return 0.
 *
 * Side Effects:
 *      none.
 *
 * --------------------------------------------------------------
```

```
 */
short emptyS(S)
Stack *S;
{
   if(S->stp<0) return(1);
   else return(0);
}


/*
 * --------------------------------------------------------------------
 *
 * initQ --
 *
 * Initialize 'Q' to be empty.
 *
 * Results:
 *      None.
 *
 * Side Effects:
 *      See above.
 *
 * --------------------------------------------------------------------
 */
void initQ(Q)
Queue *Q;
{
   Q->q_length=0;
}


/*
 * --------------------------------------------------------------------
 *
 * enqueue --
 *
 * Insert pair 'p' to 'Q'.
 *
 * Results:
 *      None.
 *
 * Side Effects:
 *      See above.
 *
```

```
 * ------------------------------------------------------------------
 */
void enqueue(p, Q)
Pair p;
Queue *Q;
{
   if(Q->q_length>=TREE_SIZE)
   {
     printf("queue overflow!\n");
     exit(1);
   }

   Q->q_elems[Q->q_length]=p;
   Q->q_length+=1;
}


/*
 * ------------------------------------------------------------------
 *
 * printQ --
 *
 * Print content in 'Q'.
 *
 * Results:
 *      None.
 *
 * Side Effects:
 *      See above.
 *
 * ------------------------------------------------------------------
 */
void printQ(Q)
Queue *Q;
{
   int i;

   for(i=Q->q_length-1; i>=0; i--)
   {
     switch (Q->q_elems[i].tag)
     {
       case MISMATCH:
         if(Q->q_elems[i].r == -1)
         {
```

```
            fprintf(tree1fp, "%d %c ", Q->q_elems[i].l,
              tree1.label[Q->q_elems[i].l]);
            left_hand++;
          }
          else
          {
            fprintf(tree2fp, "%d %c ", Q->q_elems[i].r,
              tree2.label[Q->q_elems[i].r]);
            right_hand++;
          }
          break;

        case MATCH:
          fprintf(tree1fp, "%d %c ", Q->q_elems[i].l,
            tree1.label[Q->q_elems[i].l]);
          fprintf(tree2fp, "%d %c ", Q->q_elems[i].r,
            tree2.label[Q->q_elems[i].r]);
          left_hand++; right_hand++;
          break;

        case CUT:
          break;

        case BOTHCUT:
          break;

        case RELABEL:
          fprintf(tree1fp, "%d %c ", Q->q_elems[i].l,
            tree1.label[Q->q_elems[i].l]);
          fprintf(tree2fp, "%d %c ", Q->q_elems[i].r,
            tree2.label[Q->q_elems[i].r]);
          left_hand++; right_hand++;
          break;
      }
    }
}


/*
 * ----------------------------------------------------------------
 *
 * printQ --
 *
 * Print content in 'Q'.
```

```
 *
 * Results:
 *      None.
 *
 * Side Effects:
 *      See above.
 *
 * ----------------------------------------------------------------------
 */
void printCUT(fp, Q)
FILE *fp;
Queue *Q;
{
   int i;

   for(i=0; i<Q->q_length; i++)
     fprintf(fp, "( %d )\n", Q->q_elems[i].l);
}


vldc()
{
   FILE *fp1, *fp2, *fp3;
   char T3tree[MAXTREE + 1], tree[20];
   int i, dist, vldc_line=0, vldc_count[MAXVLDCLINE];
   int opt_occur, line, p=0;

   #ifdef DISPLAY
     static short c=0;
   #endif

   if(2*occur>=DATALINES)
     opt_occur=(int)(2.0*occur/DATALINES*MAXSAMPLE-MAXSAMPLE);
   for(i=0; i<MAXVLDCLINE; i++) vldc_count[i] = 0;
   fp1=fopen("tmpvldc", "r");
   fp2=fopen("tmpdata", "r");
   fp3=fopen("tmpcandidate", "r");
   vldc_line=0;
   while(fgets(tree, 10, fp1) != NULL)
   {
     fgets(tree, 10, fp3);
     fgets(T2tree, MAXTREE, fp1);
     fgets(T3tree, MAXTREE, fp3);
     fseek(fp2, 0, 0); line=0;
```

```
      while(fgets(T1tree, MAXTREE, fp2)!=NULL)
      {
        if((++line==MAXSAMPLE+1)&&(2*occur>=DATALINES))
        {
          if(vldc_count[vldc_line]<opt_occur) goto next;
        }
        prepareLD();
        dist=vldc_dist();
        if(dist<=MUTATION) vldc_count[vldc_line]++;
      }
      if(vldc_count[vldc_line]>=occur)
      {
        printf("\t%3d\t\t%s\n", vldc_count[vldc_line], T3tree);
        p++;
        #ifdef DISPLAY
          if(++c>SPEED1)
          {
            c=0; if(K==4)K=0;
            printf("%c  Running..... %6d motif found\b\b\b\b\b\b\b\b
              \b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b",
              A[K++],p);
            fflush(stdout);
          }
        #endif
      }
next:  vldc_line++;
    }
    VLDCLINE=vldc_line;
    fclose(fp1); fclose(fp2); fclose(fp3);
    printf("-------------------------------------------------
      \n");
    printf("%6d motif found\n",p);
}


int vldc_dist()
{
    register int i, j;
    int n, m;

    n=tree1.setsz;
    m=tree2.setsz;
    for(i=1; i<=n; i++)
      for(j=1; j<=m; j++)
```

```
      dist(tree1.comp_set[i],tree2.comp_set[j]);
   return(treedist1[tree1.treesz][tree2.treesz]);
}


dist(k, l)
int k, l;
{
   int lk, ll, tmpi, tmpj, h, MAX=10000;
   register int i, j;
   int item[10];
   register int mmin;

   lk=tree1.left_most[k];
   ll=tree2.left_most[l];
   for(i=0; i<=k; ++i)
     for(j=0; j<=l; ++j)
       tempdist1[i][j]=0;
   for(i=lk; i<=k; ++i)
     tempdist1[i][0]=0;
   for(j=ll; j<=l; ++j)
     tempdist1[0][j]=tempdist1[0][j-1]
       +((tree2.label[j]=='*')?0:DELETECOST);
   for(i=lk; i<=k; ++i)
     for(j=ll; j<=l; ++j)
     {
       if(tree1.left_most[i]!=lk || tree2.left_most[j]!=ll)
       {
         tmpi=tree1.left_most[i]; tmpj=tree2.left_most[j];
         item[0]=((tmpi-1>=lk)?tempdist1[tmpi-1][j]:
           tempdist1[0][j]);
         item[1]=tempdist1[i][j-1]
           +((tree2.label[j]=='*')?0:DELETECOST);
         item[2]=((i-1>=lk)?(tempdist1[i-1][j] + INSERTCOST):
           tempdist1[0][j]+INSERTCOST);
         item[3]=((tmpi-1>=lk && tmpj-1>=ll)?
           (tempdist1[tmpi-1][tmpj-1]+treedist1[i][j]):
           (tmpi-1>=lk)? (tempdist1[tmpi-1][0]+treedist1[i][j]):
           (tmpj-1>=ll)? (tempdist1[0][tmpj-1]+treedist1[i][j]):
           (tempdist1[0][0]+treedist1[i][j]));
           tempdist1[i][j] = min(item, 4);
       }
       else
       {
```

```
        if(tree2.label[j]!='*')
        {
          item[0]=tempdist1[i][j-1]+DELETECOST;
          item[1]=((i-1>=lk)?(tempdist1[i-1][j]+INSERTCOST):
            (tempdist1[0][j]+INSERTCOST));
          item[2]=((i-1>=lk && j-1>=ll)?(tempdist1[i-1][j-1]+
            ((tree1.label[i]==tree2.label[j])?0:RELABELCOST)):
            (i-1>=lk)?(tempdist1[i-1][0]+
            ((tree1.label[i]==tree2.label[j])?0:RELABELCOST)):
            (j-1>=ll)?(tempdist1[0][j-1]+
            ((tree1.label[i]==tree2.label[j])?0:RELABELCOST)):
            (tempdist1[0][0]+
            ((tree1.label[i]==tree2.label[j])?0:RELABELCOST)));
          item[3]=tempdist1[0][j];
          tempdist1[i][j]=min(item, 4);
        }
        else
        {
          item[4]=MAX;
          item[0]=tempdist1[i][j-1];
          item[1]=((i-1>=lk)?(tempdist1[i-1][j]+INSERTCOST):
            (tempdist1[0][j]+INSERTCOST));
          item[2]=((i-1>=lk && j-1>=ll)?tempdist1[i-1][j-1]:
            (i-1>=lk)?tempdist1[i-1][0]:
            (j-1>=ll)?tempdist1[0][j-1]:
            tempdist1[0][0]);
          item[3]=tempdist1[0][j];
          mmin=min(item, 4);
          if(i!=lk)
          {
            for(h=i-1;h>=lk;h=tree1.left_most[h]-1)
              item[4]=min2(treedist1[h][j], item[4]);
          }
          tempdist1[i][j]=min2(item[4], mmin);
        }
        treedist1[i][j]=tempdist1[i][j];
      }
    }
}
```

REFERENCES

1. R. Agrawal. Tutorial: Database mining. In *Proceedings of the 13th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 75–76, Minneapolis, MN, May 1994.

2. R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 207–216, Washington, D.C., May 1993.

3. A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms.* Addison-Wesley, Reading, MA, 1983.

4. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison Wesley Publishing Company, Reading, MA, 1986.

5. C. Burks, M. Cassidy, M. J. Cinkosky, K. E. Cumella, P. Gilna, J. E.-D. Hayden, G. M. Keen, T. A. Kelley, M. Kelly, D. Kristofferson, and J. Ryals. GenBank. *Nucleic Acids Research*, 19:2221–2225, 1991.

6. T. R. Cech. Conserved sequences and structures of group I introns: building an active site for RNA catalysis – a review. *Gene*, 73:259–271, 1988.

7. G. J. S. Chang, G. Patel, L. Relihan, and J. T. L. Wang. A graphical environment for change detection in structured documents. In *Proceedings of the 21st Annual International Computer Software and Application Conference*, pages 536–541, Washington D.C., August 1997.

8. Y. C. Cheng and S. Y. Lu. Waveform correlation by tree matching. *IEEE Trans. Pattern Anal. Machine Intell.*, 7:299–305, May 1985.

9. C. Chevalet and B. Michot. An algorithm for comparing RNA secondary structures and searching for similar substructures. *Comput. Applic. Biosci.*, 8:215–225, 1992.

10. D. E. Cooke and S. A. Starks. The software architecture for the analysis of geographic and remotely sensed data. In *Proceedings of the Workshop on Increasing the Practical Impact of Formal Methods for Computer-Aided Software Development*, Monterey, CA, Sep. 1995.

11. K. M. Currey and B. A. Shapiro. Secondary structure computer prediction of the polio virus 5' non-coding region is improved with a genetic algorithm. *Comput. Applic. Biosci.*, 13(1):1-12, 1997.

12. U. Fayyad, D. Haussler, and P. Stolorz. KDD for science data analysis: Issues and examples. In *Proceedings of the 2nd International Conference on Knowledge Discovery and Data Mining*, pages 50–56, Portland, Oregon, Aug. 1996.

13. E. A. Fox, L. S. Heath, Q. F. Chen, and A. M. Daoud. Practical minimal perfect hash functions for large databases. *Communications of the ACM*, 35(1):105–121, Jan. 1992.

14. W. J. Frawley, G. Piatetsky-Shapiro, and C. J. Matheus. Knowledge discovery in databases: An overview. In G. Piatetsky-Shapiro and W. J. Frawley, editors, *Knowledge Discovery in Databases*, pages 1–27. AAAI/MIT Press, 1991.

15. R. Grishman. *Computational Linguistics*. Cambridge University Press, New York, 1986.

16. J. Han, Y. Cai, and N. Cercone. Data-driven discovery of quantitative rules in relational databases. *IEEE Transactions on Knowledge and Data Engineering*, 5(1):29–40, Feb. 1993.

17. J. Hein, T. Jiang, L. Wang, and K. Zhang. On the complexity of comparing evolutionary trees. *Discrete Applied Mathematics*, 71:153–169, 1996.

18. L. C. K. Hui. Color set size problem with applications to string matching. In A. Apostolico, M. Crochemore, Z. Galil, and U. Manber, editors, *Combinatorial Pattern Matching, Lecture Notes in Computer Science, 644*, pages 230–243. Springer-Verlag, 1992.

19. T. Jiang, L. Wang, and K. Zhang. Alignment of trees – an alternative to tree edit. In M. Crochemore and D. Gusfield, editors, *Combinatorial Pattern Matching, Lecture Notes in Computer Science, 807*, pages 75–86. Springer-Verlag, 1994.

20. D. Keselman and A. Amir. Maximum agreement subtree in a set of evolutionary trees – metrics and efficient algorithms. In *Proceedings of the 35th IEEE Annual Symposium on Foundations of Computer Science*, pages 758–769, Santa Fe, NM, 1994.

21. D. A. M. Konings and P. Hogeweg. Pattern analysis of RNA secondary structure – similarity and consensus of minimal energy folding. *J. Mol. Biol.*, 207:597–614, 1989.

22. S. R. Kosaraju. Efficient tree pattern matching. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science*, pages 178–183, Oct. 1989.

23. S.-Y. Le, K. M. Currey, R. Nussinov, and J. V. Maizel. Studies of frequently recurring substructures in human alpha-like globin mRNA precursors. *Comput. Methods Med.*, 20:563–582, 1987.

24. S.-Y. Le, J. Owens, R. Nussinov, J.-H. Chen, B. A. Shapiro, and J. V. Maizel. RNA secondary structures: comparison and determination of frequently

recurring substructures by consensus. *Comput. Applic. Biosci.*, 5(3):205–210, 1989.

25. S. Liu and E. Tanaka. A largest common similar substructure problem for trees embedded in a plane. Technical Report of the Institute of Electronics, Information and Communication Engineers, COMP 95–74, Jan. 1996.

26. S. Liu and E. Tanaka. Largest common similar substructures of rooted and unordered trees. *Mem. Grad. School Sci. & Technol., Kobe Univ.*, 14-A:107–119, 1996.

27. S. Liu and E. Tanaka. The largest common similar substructure problem. *IEICE Trans. Fundamentals*, E80-A:643–650, 1997.

28. S. Y. Lu. A tree-matching algorithm based on node splitting and merging. *IEEE Trans. Pattern Anal. Machine Intell.*, 6(2):249–256, Mar. 1984.

29. H. Mannila. Foundations of data mining. Tutorial in the 6th Annual Symposium on Combinatorial Pattern Matching, Espoo, Finland, July 1995.

30. H. Margalit, B. A. Shapiro, A. B. Oppenheim, and J. V. Maizel Jr. Detection of common motifs in RNA secondary structures. *Nucleic Acids Res.*, 17(12):4829–4845, 1989.

31. B. Michot and J.-P. Bachellerie. Secondary structure of the 5' external transcribed spacer of vertebrate pre-rRNA – presence of phylogenetically conserved features. *Eur. J. Biochem.*, 195:601–609, 1991.

32. B. Moayer and K. S. Fu. A tree system approach for fingerprint pattern recognition. *IEEE Trans. Pattern Anal. Machine Intell.*, 8:376–387, May 1986.

33. A. S. Noetzel and S. M. Selkow. An analysis of the general tree-editing problem. In D. Sankoff and J. B. Kruskal, editors, *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*, pages 237–252. Addison-Wesley, Reading, MA, 1983.

34. K. Ohmori and E. Tanaka. A unified view on tree metrics. In *Preprint of the Workshop on Syntactic and Structural Pattern Recognition* (Barcelona, 1986). Syntactic and Structural Pattern Recognition, Eds. G. Ferrate *et al.*, Springer, 1988.

35. C. V. Ramamoorthy and W. T. Tsai. Advances in software engineering. *IEEE Computer*, 29(10):47–58, October 1996.

36. H. Samet. Distance transform for images represented by quadtrees. *IEEE Trans. Pattern Anal. Machine Intell.*, 4(3):298–303, May 1982.

37. D. Sankoff and J. B. Kruskal, editors. *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison.* Addison-Wesley, Reading, MA, 1983.

38. S. M. Selkow. The tree-to-tree editing problem. *Information Processing Letters*, 6(6):184–186, Dec. 1977.

39. B. A. Shapiro. An algorithm for comparing multiple RNA secondary structures. *Comput. Applic. Biosci.*, 4(3):387–393, 1988.

40. B. A. Shapiro and K. Zhang. Comparing multiple RNA secondary structures using tree comparisons. *Comput. Applic. Biosci.*, 6(4):309–318, 1990.

41. L. G. Shapiro and R. M. Haralick. Structural descriptions and inexact matching. *IEEE Trans. Pattern Anal. Machine Intell.*, 3(5):504–519, Sep. 1981.

42. D. Shasha, J. T. L. Wang, K. Zhang, and F. Y. Shih. Exact and approximate algorithms for unordered tree matching. *IEEE Transactions on Systems, Man and Cybernetics*, 24(4):668–678, April 1994.

43. A. Silberschatz, M. Stonebraker, and J. D. Ullman. Database systems: Achievements and opportunities. *Communications of the ACM*, 34(10):94–109, 1991.

44. T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195–197, 1981.

45. G. M. Studnicka, G. M. Rahn, I. W. Cummings, and W. Salser. Computer method for predicting the secondary structure of single-stranded RNA. *Nucleic Acids Res.*, 5:3365–3387, 1978.

46. K.-C. Tai. The tree-to-tree correction problem. *J. ACM*, 26(3):422–433, 1979.

47. E. Tanaka. The metric between rooted and ordered trees based on strongly structure preserving mapping and its computing method. *IECE Trans.*, J67-D(6):722–723, 1984.

48. E. Tanaka. A metric between unrooted and unordered trees and its bottom-up computing method. *IEEE Trans. Pattern Anal. Machine Intell.*, 16(12):1233–1238, Dec. 1994.

49. (a) E. Tanaka and K. Tanaka. A metric on trees and its computing method. *IECE Trans.*, J65-D(5): 511–518, 1982. (b) Correction to "A metric on trees and its computing method." *IEICE Trans.*, J76-D-I(11):635, 1993.

50. E. Tanaka and K. Tanaka. The tree-to-tree editing problem. *International Journal of Pattern Recognition and Artificial Intelligence*, 2(2):221–240, 1988.

51. Z. Tu, N. M. Chapman, G. Hufnagel, S. Tracy, B. A. Shapiro, J. R. Romero, W. H. Barry, L. Zhao, and K. M. Currey. The cardiovirulent phenotype of coxsackievirus B3 is determined at a single site in the genomic 5' non-translated region. *J. Virology*, 69:4607–4618, 1995.

52. R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *J. ACM*, 21(1):168–173, Jan. 1974.

53. J. T. L. Wang and C.-Y. Chang. Fast retrieval of electronic messages that contain mistyped words or spelling errors. *IEEE Transactions on Systems, Man, and Cybernetics*, 27(3):441–451, June 1997.

54. J. T. L. Wang, G.-W. Chirn, C.-Y. Chang, G. Chang, A. Noriega, and K. Pysniak. An integrated toolkit for pattern matching and pattern discovery in scientific, program and document databases. In *Proceedings of the 7th International Conference on Software Engineering and Knowledge Engineering*, page 497, Rockville, Maryland, June 1995.

55. J. T. L. Wang, G. J. S. Chang, G.-W. Chirn, C.-Y. Chang, W. Wu, and F. Aljallad. A visualization tool for pattern matching and discovery in scientific databases. In *Proceedings of the 8th International Conference on Software Engineering and Knowledge Engineering*, pages 563–570, Lake Tahoe, Nevada, Jun. 1996.

56. J. T. L. Wang, G.-W. Chirn, T. G. Marr, B. A. Shapiro, D. Shasha, and K. Zhang. Combinatorial pattern discovery for scientific data: Some preliminary results. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, pages 115–125, Minneapolis, Minnesota, May 1994.

57. J. T. L. Wang, T. G. Marr, D. Shasha, B. A. Shapiro, and G.-W. Chirn. Discovering active motifs in sets of related protein sequences and using them for classification. *Nucleic Acids Research*, 22(14):2769–2775, 1994.

58. J. T. L. Wang, T. G. Marr, D. Shasha, B. A. Shapiro, G.-W. Chirn, and T. Y. Lee. Complementary classification approaches for protein sequences. *Protein Engineering*, 9(5):381–386, 1996.

59. J. T. L. Wang and P. A. Ng. TEXPROS: An intelligent document processing system. *International Journal of Software Engineering and Knowledge Engineering*, 2(2):171–196, June 1992.

60. J. T. L. Wang, D. Shasha, G. J. S. Chang, L. Relihan, K. Zhang, and G. Patel. Structural matching and discovery in document databases. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, pages 560–563, Tucson, AZ, May 1997.

61. J. T. L. Wang, B. A. Shapiro, D. Shasha, K. Zhang, and C.-Y. Chang. Automated discovery of active motifs in multiple RNA secondary structures. In *Proceedings of the 2nd International Conference on Knowledge Discovery and Data Mining*, pages 70–75, Portland, Oregon, August 1996.

62. J. T. L. Wang, K. Zhang, K. Jeong, and D. Shasha. A system for approximate tree matching. *IEEE Transactions on Knowledge and Data Engineering*, 6(4):559–571, August 1994.

63. J. T. L. Wang, K. Zhang, and D. Shasha. Pattern matching and pattern discovery in scientific, program, and document databases. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, page 487, San Jose, California, May 1995.

64. X. Wang, J. T. L. Wang, D. Shasha, B. A. Shapiro, S. Dikshitulu, I. Rigoutsos, and K. Zhang, Automated discovery of active motifs in three dimensional molecules. In *Proceedings of the 3rd International Conference on Knowledge Discovery and Data Mining*, pages 89–95, Newport Beach, California, Aug. 1997.

65. S. Wolfram. *Principles of Nucleic Acids Structure*, pages 331–332. Springer-Verlag, New York, 1984.

66. A. K. Wong, M. You, and S. C. Chang. An algorithm for graph optimal monomorphism. *IEEE Transactions on Systems, Man and Cybernetics*, 20:628–639, 1990.

67. W. Yang. Identifying syntactic differences between two programs. *Software – Practice and Experience*, 21(7):739–755, July 1991.

68. K. Zhang. Algorithms for the constrained editing distance between ordered labeled trees and related problems. *Pattern Recognition*, 28(3):463–474, 1995.

69. K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM Journal on Computing*, 18(6):1245–1262, Dec. 1989.

70. K. Zhang, D. Shasha, and J. T. L. Wang. Approximate tree matching in the presence of variable length don't cares. *Journal of Algorithms*, 16(1):33–66, Jan. 1994.

71. K. Zhang, J. T. L. Wang, and D. Shasha. On the editing distance between undirected acyclic graphs. *International Journal of Foundations of Computer Science*, 7(1):43–57, March 1996.

72. M. Zuker. On finding all suboptimal foldings of an RNA molecule. *Science*, 244:48–52, 1989.