INFORMATION TO USERS

The most advanced technology has been used to photograph and reproduce this manuscript from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book. These are also available as one exposure on a standard 35mm slide or as a 17" x 23" black and white photographic print for an additional charge.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

U·M·I

University Microfilms International
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
313/761-4700 800/521-0600

Order Number 9016327

The editing distance between trees: Algorithms and applications

Zhang, KaiZhong, Ph.D.
New York University, 1989

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

THE EDITING DISTANCE BETWEEN TREES: ALGORITHMS AND APPLICATIONS

by

KaiZhong Zhang

July 1989

A dissertation in the Department of Computer Science submitted to the faculty of the Graduate School of Arts and Sciences in partial fulfillment of the requirements for the degree of Doctor of Philosophy at New York University.

Approved: Dennis E. Shasha

Dennis E. Shasha

Research Advisor

ABSTRACT

Trees are a ubiquitous building block in computer science and related fields. Examples are grammar parses, image descriptions, secondary structures of RNA molecules, and many other phenomena. Comparing trees is therefore useful to compare scenes, parses, and so on.

This thesis presents algorithms for tree comparison and applications of those algorithms. We consider the distance between two labeled trees to be the weighted number of editing operations (insert, delete, and modify) to transform one tree to another. We show that for unordered trees this is a NP-Complete problem.

For ordered trees we present a simple fast dynamic programming algorithm that is significantly better than the best previous published algorithms. We then show that our method provides a general technique for solving other related tree problems (e.g. approximate tree matching). We also present efficient parallel algorithms on the assumption that the costs be unit.

One of our applications is to compare secondary structures of RNA molecules. We describe another application to vision that uses tree comparison to compare shapes.

We have also implemented some of the algorithms in the form of a tree comparison toolkit. The preliminary version of the toolkit has been used at the U.S. National Cancer Institute for the comparison of RNA secondary structures.

ACKNOWLEDGMENTS

I am deeply indebted to Professor Dennis Shasha for his guidance and constant encouragement throughout this work.

Also, I would like to thank Professor Robert Hummel for his valuable suggestions. Among other things, he introduced and guided me in a research project on parallel image processing.

I would like to express my appreciation to Professor Zvi Kedem for his valuable suggestions and support.

I would like to thank Dr. Bruce Shapiro and Dr. Ruth Nussinov, both at NIH, for introducing me to the problem of comparing RNA secondary structures in Molecular Biology. I also would like to thank Professor Rick Statman, of Carnegie Mellon, for his suggestion concerning the NP-completeness of unordered tree editing problem. Thank to Professor Haim Wolfson for the discussion concerning vision stuff.

Finally, I would like to thank Mr. Tsong-Li Wang for his help on the toolkit manual. Special thanks are given to my friends, especially to Mr Yongtao You, who helped me in many ways during my years at Courant Institute.

Most of all, I want to express my appreciation to my wife, Jinfei Wang, to my daughter, Maomao, and to my parents and sisters, without whose support and encouragement my accomplishments would not be possible.

TABLE OF CONTENT

CHAPTER 1

Introduction

1.1. The Tree Editing Problem

Ordered labeled trees are trees whose nodes are labeled and in which the left-to-right order among siblings is significant. In this thesis, unless otherwise stated, all trees are ordered labeled trees. The editing operation we consider in this thesis will be (i) relabel, --change the label of a tree node; (ii) delete, --delete a tree node and make all the children of the deleted node be the children of the parent of the deleted node; and (iii) insert, --inverse of deletion.

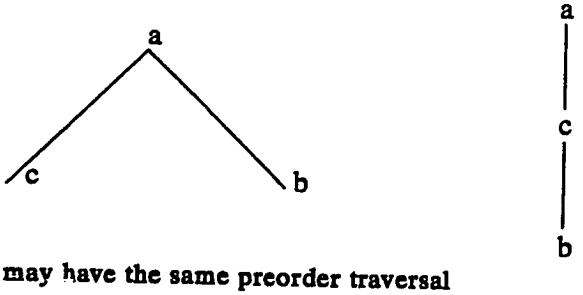
Given two trees, the question to ask is that what is the minimum weighted number (the precise meaning will be defined later) of editing operations that will transform one tree to another (see Figure 1.1). This gives a natural way to measure the distance between trees.

1.2. Tree Distance as a Generalization of String Distance

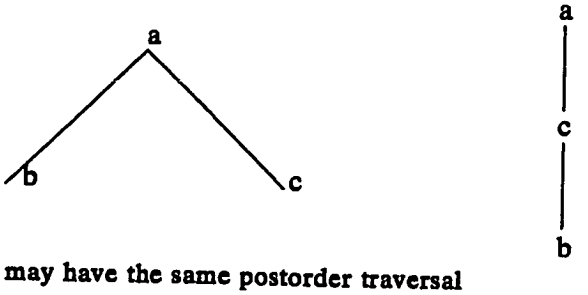
Much of the inspiration for the problems considered in this thesis comes from string comparison algorithms. In string comparison algorithms, the problem is to find the fewest number of mismatches, inserts, and deletes, to transform one string to another.

Figure 1.1 An example of string editing
The editing sequence is: insert(C), delete(C), relabel(B to C), insert(A).

We basically generalize this problem to trees. One conceivable approach to tree comparison would be to take the preorder or postorder traversal of trees and apply string comparison. But this fails because two trees can have the same preorder or postorder traversal yet be different (see Figure 1.2).



(a) Different trees may have the same preorder traversal



(b) Different trees may have the same postorder traversal

Figure 1.2 Different trees may have the same preorder or postorder traversal

Another approach is to take the parenthesized expression describing each tree and then to apply string comparison. However, this may reverse the actual distances (see Figure 1.3).

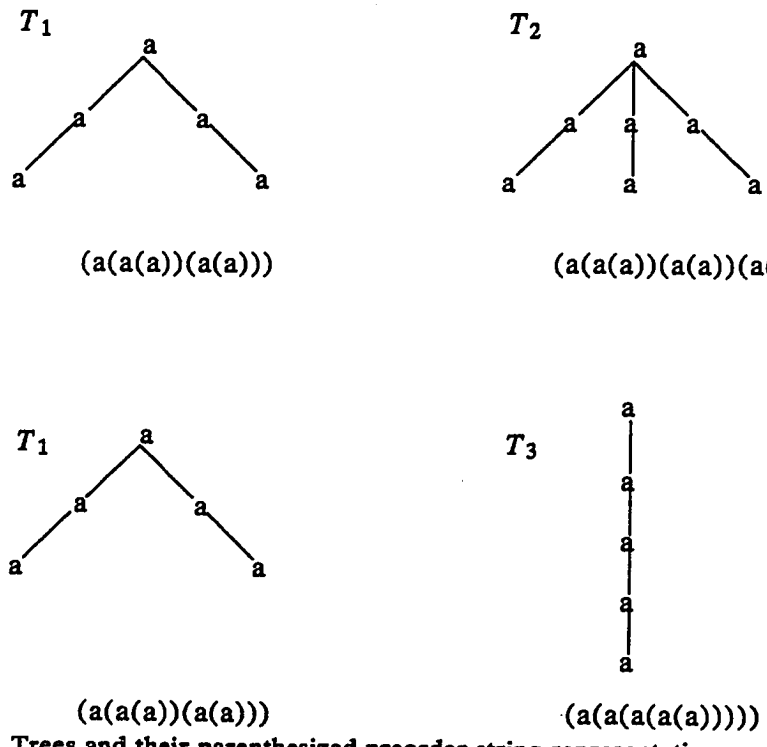


Figure 1.3 Trees and their parenthesized preorder string representations

In Figure 1.3, the tree distance between T_1 and T_2 is 2 and the tree distance between T_1 and T_3 is 4. And it seems that T_2 is closer to T_1 than T_3 is. However if we represent them in the parenthesized preorder string forms, the string distance of T_2 and T_1 (6) is larger than the string distance of T_3 and T_1 (4).

Moreover, it is far from clear what it means to remove a parenthesis when mapping one tree to the other. Also the editing sequence that transforms one string to the other will not give you a sequence to transform one tree to the other.

1.3. Why Tree Editing is Useful

Since trees can represent grammar parses, image descriptions, RNA secondary structures and many other phenomena, comparing such trees is a way to compare scenes, parses, and so on.

As an example, consider the secondary structure comparison problem for RNA. Because RNA is a single strand of nucleotides, it folds back onto itself into a shape that is topologically a tree (called its secondary structure). Each node of this tree contains several nucleotides. Nodes have colorful labels such as "bulge" and "hairpin". Various researchers [ALKBO-87], [BSSBWD-87, DD-87] have observed that the secondary structure influences translation rates (from RNA to proteins). Because different sequences can produce similar secondary structures [DA-82], [SK-76], comparisons among secondary structures are necessary to understanding the comparative functionality of different RNA's.

1.4. Previous Work

1.4.1. [L-79]

In [L-79], Lu gave an algorithm for the editing distance between trees. The author claimed that the complexity is $O(N \times M)$. However the algorithm contains a fatal mistake.

Let S and T be two trees. Suppose S has s children and let S_1, ... S_s be the s subtrees. Suppose T has t children and let T_1, ... T_t be the t subtrees. The algorithm does not consider the case that one subtree of S, i.e. S_i, could be mapped to more than one subtree in S, i.e. T_k, .. T_l. (We will define mapping in the next chapter, but intuitively if node n in one tree is mapped to m in another, then n's label is changed to the label of m.) However in general it is possible that one of the subtrees will map to more than one subtree.

Consider the example in Figure 1.4. If all editing costs are 1, then $D(T_1,T_2)$ should be 1, i.e. delete $T_1[3]$. However the Lu's algorithm gives 2.

Figure 1.4 $D(T_1,T_2) = 1$

1.4.2. [T-79]

In [T-79] Tai gave a correct algorithm. The paper suggested many definitions that we have used. However, the algorithm was very complicated, requiring 20 journal pages to describe. Our first discovery was a simpler, asymptotically faster algorithm then Tai's. We also improved the space complexity. The space complexity in [T-79] will impose a serious problem for the implementation when the size of tree is larger than ten or twenty.

1.5. Our Approach

Intuitively, the problem with [T-79] is that in any intermediate step it requires that the substructures always be trees. By introducing the distance between ordered forests and by careful elimination of certain subtree-to-subtree distance calculations, we improve both the time and space complexity as compared to his approach.

In style, our algorithm resembles algorithms for computing the distance between strings. In fact, our algorithm specializes directly to the dynamic programming string distance algo-

rithm when the input is a string. Also our method provide a general framework for tree editing problems. This enables us to generalize the technique to some other problems.

If all editing costs are unit, the problem becomes unit cost tree editing problem. For the unit cost tree editing problem, we develop some subtle properties of the distance matrices and the tree structure. Examples of these properties are monotonicity of the distance matrices and certain rules of inference about tree and forest distances. These allow us to design more efficient algorithms.

We will present following results. Let trees T_1 and T_2 have numbers of levels L_1 and L_2 respectively. Let k be the actual distance between T_1 and T_2. Let N be $min(|T_1|, |T_2|)$. The asymptotic running times (assuming a concurrent-read concurrent-write parallel random access machine for algorithm 2 and algorithm 3) are:

Algorithm	Time	Processors
Algorithm 0	$\|T_1\| \times \|T_2\| \times L_1 \times L_2$	1
Algorithm 1	$k^2 \times N \times min(L_1, L_2)$	1
Algorithm 2 (parallel)	$k \times log(k) \times log(N)$	$k^2 \times N$
Algorithm 3 (parallel)	$(k^2 \times log(k)) + log(N)$	$k^2 \times N$

1.6. Organization of This Thesis

In chapters 2, we will first give the definition of the problem and then define the concept of mapping. We then give our simple algorithm for computing the editing distance between two trees. We also show how to construct the best mapping.

In chapter 3, we extend the technique to other problems. Specifically we will solve the problem of approximate tree matching and constraint tree editing.

In chapter 4, we will show that with a minor modification we can define the problem of editing distance between two unordered tree. We then show that the problem to compute the editing distance between two unordered trees is NP-Complete [1].

In chapter 5, we will restrict the editing costs to be unit. In this case a faster algorithm is possible. We give one simple sequential algorithm and two slightly different parallel algorithms.

In chapter 6, we will consider the applications of tree editing distance. We first show that RNA secondary structures can be represented by ordered labeled trees and tree editing distance can be used to compare of RNA secondary structures [2]. We then describe an application to the correlation of waveforms. We also describe how to use tree distance to measure the distance between shapes that could be used in computer vision.

In chapter 7, we will describe a toolkit that implements some of the tree editing algorithms.

[1] - This result is obtained with the help from Rick Statman of Carnegie Mellon.

[2] - This is a joint result with Bruce Shapiro of NIH.

CHAPTER 2

Algorithm for the Editing Distance between Trees

2.1. Definitions

2.1.1. Editing Operations and Editing Distance between Trees

Let us consider three kind operations. Changing node n means changing the label on n. Deleting a node n means making the children of n become the children of the parent of n and then removing n. Inserting is the complement of delete. This means that inserting n as the child of n' will make n the parent of a consecutive subsequence of the current children of n'. Figures 2.1, 2.2, and 2.3 illustrate these editing operations.

1. Relabel: To change one node label to another.

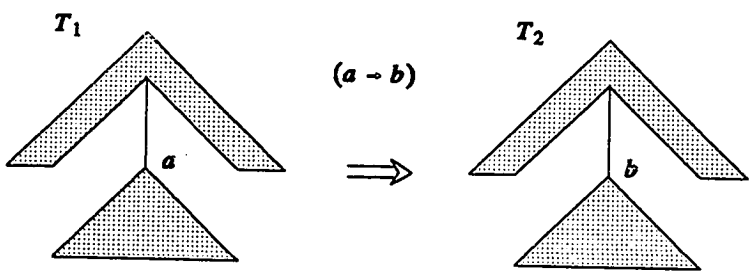


Figure 2.1 Relabeling

2. Delete: To delete a node.
(All children of the deleted node b become children of the parent a.)

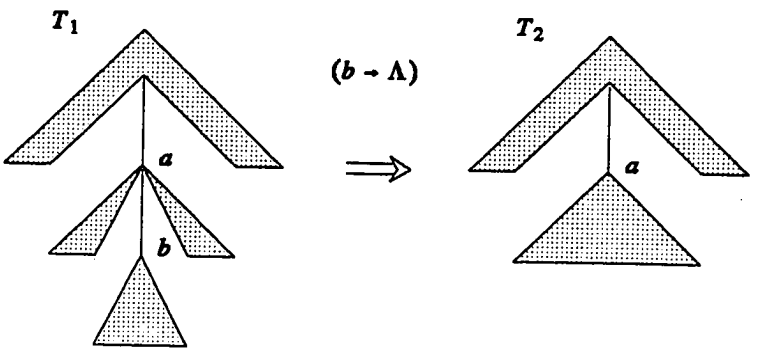


Figure 2.2 Deletion

3. Insert: To insert a node.
(A consecutive sequence of siblings among the children of a become the children of b.)

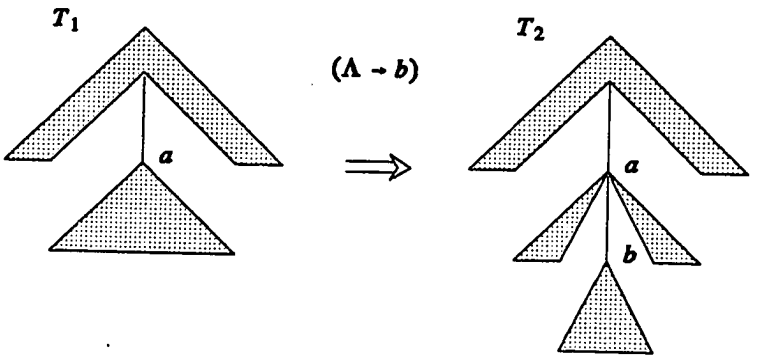


Figure 2.3 Insertion

Following [WF-74] and [T-79], we represent an edit operation as a pair $(a,b) \neq (\Lambda,\Lambda)$, sometimes written as $a \to b$, where a is either Λ or a label of a node in tree T_1 and b is either Λ or a label of a node in tree T_2. We call $a \to b$ a change operation if $a \neq \Lambda$ and $b \neq \Lambda$; a delete operation if $b = \Lambda$; and an insert operation if $a = \Lambda$. Since many nodes may have the same label, this notation is potentially ambiguous. It could be made precise by identifying the nodes as well as their labels. However, in this paper, which node is meant will always be

clear from the context.

Let S be a sequence $s_1, \ldots, s_k$ of edit operations. An S-derivation from A to B is a sequence of trees $A_0, \ldots A_k$ such that $A=A_0$, $B=A_k$, and $A_{i-1} \rightarrow A_i$ via s_i for $1 \leq i \leq k$.

Let γ be a cost function which assigns to each edit operation $a \rightarrow b$ a nonnegative real number $\gamma(a \rightarrow b)$. This cost can be different for different nodes, so it can be used to give greater weights to, for example, the higher nodes in a tree than to lower nodes.

We constrain γ to be a distance metric. That is,

i) $\gamma(a \rightarrow b) \geq 0$; $\gamma(a \rightarrow a) = 0$;

ii) $\gamma(a \rightarrow b) = \gamma(b \rightarrow a)$; and

iii) $\gamma(a \rightarrow c) \leq \gamma(a \rightarrow b) + \gamma(b \rightarrow c)$.

We extend γ to the sequence S by letting $\gamma(S) = \sum_{i=1}^{|S|} \gamma(s_i)$. Formally the distance between T_1 and T_2 is defined as:

$\delta(T_1,T_2) = \min \{\gamma(S) \mid S$ is an edit operation sequence taking T_1 to $T_2\}$. The definition of γ makes δ a distance metric also.

2.1.2. Mapping

Let T_1 and T_2 be two trees with N_1 and N_2 nodes respectively. Suppose that we have an ordering for each tree, then $T[i]$ means the ith node of tree T in the given ordering.

The edit operations give rise to a mapping which is a graphical specification of what edit operations apply to each node in the two trees (or two ordered forests). The mapping in Figure 2.4 shows a way to transform T_1 to T_2. It corresponds to the sequence (delete(node with label c), insert(node with label c)).

Consider the following diagram of a mapping:

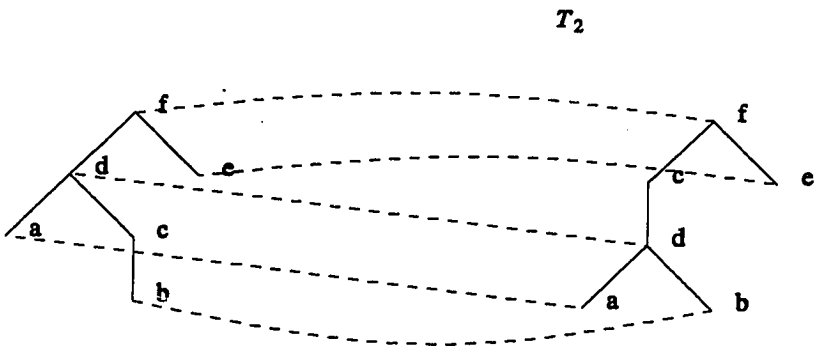


Figure 2.4 Mapping

A dotted line from $T_1[i]$ to $T_2[j]$ indicates that $T_1[i]$ should be changed to $T_2[j]$ if $T_1[i] \neq T_2[j]$, or that $T_1[i]$ remains unchanged if $T_1[i]=T_2[j]$. The nodes of T_1 not touched by a dotted line are to be deleted and the nodes of T_2 not touched are to be inserted. The mapping shows a way to transform T_1 to T_2.

Formally we define a triple (M, T_1, T_2) to be a mapping from T_1 to T_2, where M is any set of pair of integers (i,j) satisfying:[1]

(1) $1 \leq i \leq N_1, 1 \leq j \leq N_2$;

(2) For any pair of (i_1,j_1) and (i_2,j_2) in M,

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

(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 preserved)

(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 preserved)

[1] - Note that our definition of mapping is different from the definition in [T-79]. We believe that our definition is more natural because it does not depend on any traversal ordering of the tree.

We will use M instead of (M,T_1,T_2) if there is no confusion. Let M 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 line in M. Then we can define the cost of M:

$$\gamma(M)=\sum_{(i,j)\in M}\gamma(T_1[i]\rightarrow T_2[j])+\sum_{i\in I}\gamma(T_1[i]\rightarrow\Lambda)+\sum_{j\in J}\gamma(\Lambda\rightarrow T_2[j])$$

Mappings can be composed. Let M_1 be a mapping from T_1 to T_2 and let M_2 be a mapping from T_2 to T_3. Define

$M_1 \circ M_2=\{(i,j)\mid \exists\ k$ s.t. $(i,k)\in M_1$ and $(k,j)\in M_2\}$

Lemma 2.1:

(1) $M_1 \circ M_2$ is a mapping

(2) $\gamma(M_1 \circ M_2) \leq \gamma(M_1) + \gamma(M_2)$

Proof: (1) follows from the definition of mapping.

(2) Let M_1 be the mapping from T_1 to T_2. Let M_2 be the mapping from T_2 to T_3. Let $M_1 \circ M_2$ be the composed mapping from T_1 to T_3 and let I and J be the corresponding deletion and insertion sets. Three general situations occur. $(i,j)\in M_1 \circ M_2$, $i\in I$, or $j\in J$. In each case this corresponds to an editing operation $\gamma(x\rightarrow y)$ where x and y may be nodes or may be Λ. In all such cases, the triangle inequality on the distance metric γ ensures that $\gamma(x\rightarrow y)\leq\gamma(x\rightarrow z)+\gamma(z\rightarrow y)$. □

The relation between a mapping and a sequence of edit operation is as follows:

Lemma 2.2: Given S, a sequence $s_1, \ldots, s_k$ of edit operations from T_1 to T_2, there exists a mapping M from T_1 to T_2 such that $\gamma(M)\leq\gamma(S)$. Conversely, for any mapping M, there exists a sequence of editing operations such that $\gamma(S)=\gamma(M)$.

Proof: The first part can be proved by induction on k. The base case is $k=1$. This case holds, because any single editing operation preserves the ancestor and sibling relationships in

the mapping. In the general case, let S_1 be the sequence $s_1, \ldots, s_{k-1}$ of edit operations. There exist a mapping M_1 such that $\gamma(M_1) \leq \gamma(S_1)$. Let M_2 be the mapping for s_k. From Lemma 2.1, we have that

$\gamma(M_1 \circ M_2) \leq \gamma(M_1)+\gamma(M_2) \leq \gamma(S)$.

To construct the sequence of editing operations, simply perform all the deletes indicated by the mapping (i.e. all nodes in T_1 having no lines attached to them are deleted), then all relabelings, then all inserts. □

Hence, $\delta(T_1,T_2) = \min\{\gamma(M) | M \text{ is a mapping from } T_1 \text{ to } T_2\}$

There has been previous work on this problem. Tai [T-79] gave the best published algorithm for the problem. [Z-83] is an improvement of [T-79], giving better sequential time and space than [T-79]. Our new algorithm is much simpler than [T-79] and [Z-83], gives better time and space than both of them, and extends to related problems. The algorithm of Lu [L-79] does not solve this problem for trees of more than two levels.

2.2. A Simple New Algorithm

This algorithm, unlike [T-79], [L-79], and [Z-83], will, in its intermediate steps, consider the distance between two ordered forests. At first sight one may think that this will complicate the work for us, but it will in fact make matters easier.

We use a postorder numbering of the nodes in the trees. In the postordering, $T_1[1 .. i]$ and $T_2[1 .. j]$ will generally be forests as in the following Figure 2.5. (The edges are those in the subgraph of the tree induced by the vertices.) Fortunately, the definition of mapping for ordered forests is the same as for trees.

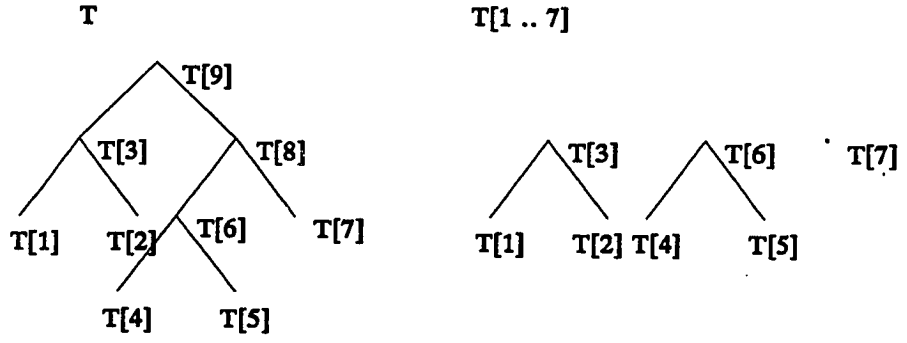


Figure 2.5 Left-to-right postorder numbering

2.2.1. Notation

Let $T[i]$ be the ith node in the tree according to the left-to-right postorder numbering. $l(i)$ is the number of the leftmost leaf descendant of the subtree rooted at $T[i]$. When $T[i]$ is a leaf, $l(i)=i$. The parent of $T[i]$ is denoted $p(i)$. We define $p^0(i)=i$, $p^1(i)=p(i)$, $p^2(i)=p(p^1(i))$ and so on. Let $anc(i)=\{p^k(i) \mid 0 \le k \le depth(i)\}$.

$T[i..j]$ is the ordered subforest of T induced by the nodes numbered i to j inclusive (Figure 2.5). If $i>j$, then $T[i..j] = \varnothing$. $T[1..i]$ will be referred to as *forest(i)*, when the tree T referred to is clear. $T[l(i)..i]$ will be referred to as *tree*(i). *Size(i)* is the number of nodes in *tree*(i).

The distance between $T_1[i'..i]$ and $T_2[j'..j]$ is denoted $dist(T_1[i'..i],T_2[j'..j])$ or $dist(i'..i,j'..j)$ if the context is clear. We use a more abbreviated notation for certain special cases. The distance between $T_1[1..i]$ and $T_2[1..j]$ is sometimes denoted *forestdist(i,j)*. The distance between the subtree rooted at i and the subtree rooted at j is sometimes denoted *treedist(i,j)*.

2.2.2. New Algorithm

We first present three lemmas and then give our new algorithm.

Recall that $anc(i) = \{p^k(i) \mid 0 \le k \le depth(i)\}$

Lemma 2.3:

(i) $forestdist(\varnothing,\varnothing)=0$

(ii) $forestdist(T_1[l(i_1)..i],\varnothing) = forestdist(T_1[l(i_1)..i-1],\varnothing)+\gamma(T_1[i]\rightarrow\Lambda)$

(iii) $forestdist(\varnothing,T_2[l(j_1)..j]) = forestdist(\varnothing,T_2[l(j_1)..j-1])+\gamma(\Lambda\rightarrow T_2[j])$
where $i_1 \in anc(i)$ and $j_1 \in anc(j)$

Proof: (i) requires no edit operation. In (ii) and (iii), the distances correspond to the cost of deleting or inserting the nodes in $T_1[l(i_1)..i]$ and $T_2[l(j_1)..j]$) respectively. □

Lemma 2.4. Let $i_1 \in anc(i)$ and $j_1 \in anc(j)$. Then

$forestdist(l(i_1)..i,l(j_1)..j)=$

$$\min\begin{cases} forestdist(l(i_1)..i-1,l(j_1)..j)+\gamma(T_1[i]\rightarrow\Lambda) \\ forestdist(l(i_1)..i,l(j_1)..j-1)+\gamma(\Lambda\rightarrow T_2[j]) \\ forestdist(l(i_1)..l(i)-1,l(j_1)..l(j)-1)+forestdist(l(i)..i-1,l(j)..j-1)+\gamma(T_1[i]\rightarrow T_2[j]) \end{cases}$$

Proof: We compute $forestdist(l(i_1)..i,l(j_1)..j)$ for $l(i_1)\le i\le i_1$ and $l(j_1)\le j\le j_1$. We are trying to find a minimum-cost map M between $forest(l(i_1)..i)$ and $forest(l(j_1)..j)$. The map can be extended to $T_1[i]$ and $T_2[j]$ in three ways.

(1) $T_1[i]$ is not touched by a line in M. Then $(i,\Lambda) \in M$. So, $forestdist(l(i_1)..i,l(j_1)..j)$ $= forestdist(l(i_1)..i-1,l(j_1)..j)+\gamma(T_1[i]\rightarrow\Lambda)$.

(2) $T_2[j]$ is not touched by a line in M. Then $(\Lambda,j) \in M$. So, $forestdist(l(i_1)..i,l(j_1)..j)$ $= forestdist(l(i_1)..i,l(j_1)..j-1)+\gamma(\Lambda\rightarrow T_2[j])$.

(3) $T_1[i]$ and $T_2[j]$ are both touched by lines in M. Then $(i,j) \in M$. Here is why. Suppose (i,k) and (h,j) are in M. if $l(i_1)\le h\le l(i)-1$, then i is to the right of h so k must

be to the right of j by the sibling condition on mappings. This is impossible in $forest(l(j_1)..j)$. Similarly, if i is a proper ancestor of h, then k must be a proper ancestor of j by the ancestor condition on mappings. This too is impossible. So, $h=i$. By symmetry, $k=j$ and $(i,j) \in M$.

Now, by the ancestor condition on mapping, any node in the subtree rooted at $T_1[i]$ can only be touched by a node in the subtree rooted at $T_2[j]$. Hence,

$forestdist(l(i_1)..i, l(j_1)..j) = forestdist(l(i_1)..l(i)-1, l(j_1)..l(j)-1)$

$+ forestdist(l(i)..i-1, l(j)..j-1) + \gamma(T_1[i] \rightarrow T_2[j])$. Figure 2.6 shows the situation.

Since these three cases express all the possible mappings yielding $forestdist(l(i_1)..i, l(j_1)..j)$, we take the minimum of these three costs. Thus,

$forestdist(l(i_1)..i, l(j_1)..j) =$

$$\min \begin{cases} forestdist(l(i_1)..i-1, l(j_1)..j) + \gamma(T_1[i] \rightarrow \Lambda) \\ forestdist(l(i_1)..i, l(j_1)..j-1) + \gamma(\Lambda \rightarrow T_2[j]) \\ forestdist(l(i_1)..l(i)-1, l(j_1)..l(j)-1) + forestdist(l(i)..i-1, l(j)..j-1) + \gamma(T_1[i] \rightarrow T_2[j]) \end{cases}$$

□.

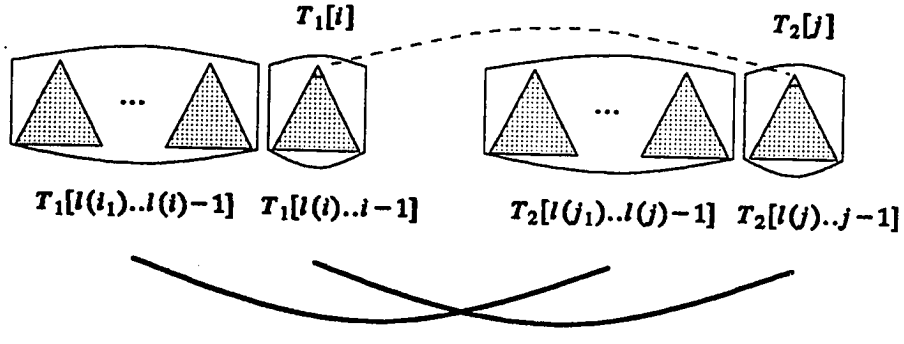


Figure 2.6 Case 3 of Lemma 2.4.

Lemma 2.5 Let $i_1 \in anc(i)$ and $j_1 \in anc(j)$. Then

(1) if $l(i)=l(i_1)$ and $l(j)=l(j_1)$

$$forestdist(l(i_1)..i, l(j_1)..j) = \min \begin{cases} forestdist(l(i_1)..i-1, l(j_1)..j) + \gamma(T_1[i] \to \Lambda) \\ forestdist(l(i_1)..i, l(j_1)..j-1) + \gamma(\Lambda \to T_2[j]) \\ forestdist(l(i_1)..i-1, l(j_1)..j-1) + \gamma(T_1[i] \to T_2[j]) \end{cases}$$

(2) if $l(i) \neq l(i_1)$ or $l(j) \neq l(j_1)$ (i.e., otherwise)

$$forestdist(l(i_1)..i, l(j_1)..j) = \min \begin{cases} forestdist(l(i_1)..i-1, l(j_1)..j) + \gamma(T_1[i] \to \Lambda) \\ forestdist(l(i_1)..i, l(j_1)..j-1) + \gamma(\Lambda \to T_2[j]) \\ forestdist(l(i_1)..l(i)-1, l(j_1)..l(j)-1) + treedist(i,j) \end{cases}$$

Proof: By Lemma 2.4, if $l(i)=l(i_1)$ and $l(j)=l(j_1)$ then, since $forestdist(l(i_1)..l(i)-1, l(j_1)..l(j)-1) = forestdist(\varnothing,\varnothing) = 0$, (1) follows immediately.

Because the distance is the cost of a minimal cost mapping, we know $forestdist(l(i_1)..i, l(j_1)..j) \leq forestdist(l(i_1)..l(i)-1, l(j_1)..l(j)-1) + treedist(i,j)$ since the latter formula represents a particular (and therefore possibly suboptimal) mapping of $forest(l(i_1)..i)$ to $forest(l(j_1)..j)$. For the same reason, $treedist(i,j) \leq forestdist(l(i)..i-1, l(j)..j-1) + \gamma(T_1[i] \to T_2[j])$. Lemma 2.4 and these two inequalities imply that the substituting of $treedist(i,j)$ for $forestdist(l(i)..i-1, l(j)..j-1) + \gamma(T_1[i] \to T_2[j])$ in (2) is correct. □

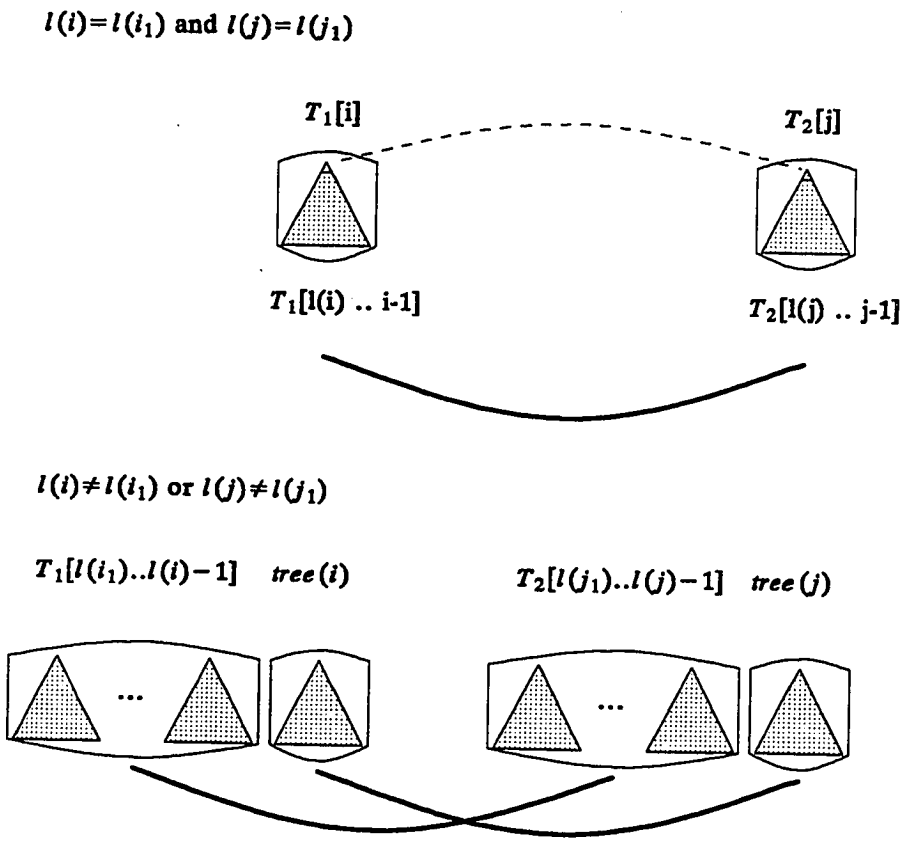


Figure 2.7 The two situations of Lemma 2.5.

Lemma 2.5 has three important implications.

First the formulas it yields suggest that we can use a dynamic programming style algorithm to solve the tree distance problem.

Second, from (2) of Lemma 2.5 we observe that in order to compute *treedist*(i_1,j_1) we need in advance almost all values of *treedist*(i,j) where i is the root of a subtree containing i_1 and j is the root of a subtree containing j_1. This suggests a bottom up procedure for computing all subtree pairs.

Third, from (1) in Lemma 2.5 we can observe that when i is in the path from $l(i_1)$ to i_1 and j

is in the path from $l(j_1)$ to j_1, we do not need to compute *treedist*(i,j) separately. These subtree distances can be obtained as a byproduct of computing *treedist*(i_1,j_1).

These implications lead to the following definition and then our new algorithm. Let us define the set *LR_keyroots* of tree T as follows:

$$LR\_keyroots(T) = \{k \mid \text{there exists no } k' > k \text{ such that } l(k) = l(k')\}$$

That is, if k is in *LR_keyroots*(T) then either k is the root of T or $l(k) \neq l(p(k))$, i.e. k has a left sibling. Intuitively, this set will be the roots of all the subtrees of tree T that need separate computations.

Consider trees T_1 and T_2 in Figure 2.4, from the above definition one can see that *LR_keyroots*$(T_1) = \{3, 5, 6\}$ and *LR_keyroots*$(T_2) = \{2, 5, 6\}$.

It is easy to see that there is a linear time algorithm to compute the function $l()$ and the set *LR_keyroots*. We can also assume that the result is in array l and *LR_keyroots*. Further in array *LR_keyroots* the order of the elements is in increasing order.

We are now ready to give our new simple algorithm.

Input: Tree T_1 and T_2.

Output: *Tree_dist*(i,j), where $1 \leq i \leq |T_1|$ and $1 \leq j \leq |T_2|$.

Preprocessing

(To compute $l()$, *LR_keyroots*$(T_1)[]$ and *LR_keyroots*$(T_2)[]$)

Main loop

```
for i':=1 to |LR_keyroots(T1)|
  for j':=1 to |LR_keyroots(T2)|
    i=LR_keyroots(T1)[i'];
    j=LR_keyroots(T2)[j'];
```

Compute *treedist*(i,j);

We use dynamic programming to compute *treedist*(i,j). The forestdist values computed and used here are put in a temporary array that is freed once the corresponding treedist is computed. The treedist values are put in the permanent treedist array.

The computation of *treedist*(i,j).

forestdist$(\emptyset,\emptyset)=0$;

for $i_1:=l(i)$ to i

forestdist$(T_1[l(i)..i_1],\emptyset)$ = *forestdist*$(T_1[l(i)..i_1-1],\emptyset)+\gamma(T_1[i_1]\rightarrow\Lambda)$

for $j_1:=l(j)$ to j

forestdist$(\emptyset,T_2[l(j)..j_1])$= *forestdist*$(\emptyset,T_2[l(j)..j_1-1])+\gamma(\Lambda\rightarrow T_2[j_1])$

for $i_1:=l(i)$ to i

for $j_1:=l(j)$ to j

if $l(i_1)=l(i)$ and $l(j_1)=l(j)$ then

forestdist$(T_1[l(i)..i_1],T_2[l(j)..j_1])$= min {

forestdist$(T_1[l(i)..i_1-1],T_2[l(j)..j_1])+\gamma(T_1[i_1]\rightarrow\Lambda)$,

forestdist$(T_1[l(i)..i_1],T_2[l(j)..j_1-1])+\gamma(\Lambda\rightarrow T_2[j_1])$,

forestdist$(T_1[l(i)..i_1-1],T_2[l(j)..j_1-1])+\gamma(T_1[i_1]\rightarrow T_2[j_1])$}

treedist(i_1,j_1) = *forestdist*$(T_1[l(i)..i_1],T_2[l(j)..j_1])$ /* put in permanent array */

else

forestdist$(T_1[l(i)..i_1],T_2[l(j)..j_1])$= min {

forestdist$(T_1[l(i)..i_1-1],T_2[l(j)..j_1])+\gamma(T_1[i_1]\rightarrow\Lambda)$,

forestdist$(T_1[l(i)..i_1],T_2[l(j)..j_1-1])+\gamma(\Lambda\rightarrow T_2[j_1])$,

forestdist$(T_1[l(i)..l(i_1)-1],T_2[l(j)..l(j_1)-1])$ + *treedist*(i_1,j_1))}

Theorem 1: The basic algorithm is correct.

Proof: We will prove that for any pair (i,j) such that $i \in LR\_keyroots(T_1)$ and $j \in LR\_keyroots(T_2)$, the following invariants hold.

1) Immediately before the computation of $treedist(i,j)$, all distances $treedist(i_1,j_1)$, where $l(i) \le i_1 \le i$ and $l(j) \le j_1 \le j$ and either $l(i) \ne l(i_1)$ or $l(j) \ne l(j_1)$, are available. In other words, $treedist(i_1,j_1)$ is available if i_1 is in the subtree of $tree(i)$ but not in the path from $l(i)$ to i and j_1 is in the subtree of $tree(j)$ but not in the path from $l(j)$ to j.

2) Immediately after the computation of treedist(i,j), all distances $treedist(i_1,j_1)$, where $l(i) \le i_1 \le i$ and $l(j) \le j_1 \le j$ are available.

We first show that if (1) is true then (2) is true. From Lemma 2.5 we know that all required subtree-to-subtree distances are available. (We need all treedist(i_1,j_1) such that $l(i) \le i_1 \le i$ and $l(j) \le j_1 \le j$ and either $l(i) \ne l(i_1)$ or $l(j) \ne l(j)$, and by (1) all these distances are available). We compute each $treedist(i_1,j_1)$, where $l(i_1)=l(i)$ and $l(j_1)=l(j)$ in the if part and add it to the permanent treedist array. So, (2) holds.

Let us show that (1) always holds. Suppose $l(i_1) \ne l(i)$. Let i_1' be the lowest ancestor of i_1 such that $i_1' \in LR\_keyroots(T_1)$. Since $l(i_1')=l(i_1) \ne l(i)$, $i_1' \ne i$. Since $i \in LR\_keyroots(T_1)$, $i_1' \le i$. So $i_1' < i$. Let j_1' be the lowest ancestor of j_1 such that $j_1' \in LR\_keyroots(T_2)$. Since $j \in LR\_keyroots(T_2)$, $j_1' \le j$. Hence $i_1' + j_1' < i + j$. This means that $treedist(i_1',j_1')$ will have already been computed before $treedist(i,j)$ because in the main loop $LR\_keyroots(T_1)[]$ and $LR\_keyroots(T_2)[]$ are in increasing order. Hence $treedist(i_1,j_1)$ is available after the computation of $treedist(i_1',j_1')$. □

As an example, consider tree T_1 and T_2 in Figure 2.4. For simplicity, assume that all insert, delete and change (of labels) operations will cost 1. Figure 2.8 shows the result of applying our new algorithm to T_1 and T_2. The matrix below $tree\_dist(i,j)$ is the result of

temporary array produced by the computation of *tree_dist*(*i*,*j*). (Out of 36 possible *tree_dist* arrays, only 9 -- those corresponding to pairs of keyroots -- are explicitly computed.) The matrix below tree_dist is the final result. The value in the lower right corner (2) is the distance between T_1 and T_2.

tree_dist(3,2)

```
0 1
1 0
2 1
```

tree_dist(3,5)

```
0 1
1 1
2 2
```

tree_dist(3,6)

```
0 1 2 3 4 5 6
1 1 1 2 3 4 5
2 2 2 2 2 3 4
```

tree_dist(5,2)

```
0 1
1 1
```

tree_dist(5,5)

```
0 1
1 0
```

tree_dist(5,6)

```
0 1 2 3 4 5 6
1 1 2 3 4 4 5
```

tree_dist(6,2)

```
0 1
1 1
2 1
3 2
4 3
5 4
6 5
```

tree_dist(6,5)

```
0 1
1 1
2 2
3 3
4 4
5 4
6 5
```

tree_dist(6,6)

```
0 1 2 3 4 5 6
1 0 1 2 3 4 5
2 1 0 1 2 3 4
3 2 1 2 3 4 5
4 3 2 1 2 3 4
5 4 3 2 3 2 3
6 5 4 3 3 3 2
```

tree_dist

```
0 1 2 3 1 5
1 0 2 3 1 5
2 1 2 2 2 4
3 3 1 2 4 4
1 1 3 4 0 5
5 5 3 3 5 2
```

Figure 2.8 The result of computation for T_1 and T_2 in Figure 2.4.

2.3. Some Aspects of Our Algorithm

2.3.1. Complexity

Lemma 2.6. $|LR\_keyroots(T)| \leq |leaves(T)|$.

Proof: We will prove that for any $i,j \in LR\_keyroots(T)$, $l(i) \neq l(j)$.

Let $i,j \in LR\_keyroots(T)$ and $i<j$. If $l(i)=l(j)$ from $i<j$ we know that i is in the path from $l(j)$ to j. By the definition of $l(j)$, i has no left_sibling. This contradicts the assertion that $i \in LR\_keyroots(T)$. Hence each leaf is the leftmost descendant of at most one member of $LR\_keyroots(T)$. So, $|LR\_keyroots(T)| \leq |leaves(T)|$. □

Because not all subtree-to-subtree distances need be computed, the number of such calculation a node participates in is less than its depth. Instead, it is the node's *collapsed depth*.

$$LR\_colldepth(i) = |anc(i) \cap LR\_keyroots(T)|$$

We define the collapsed depth of tree T as follows.

$$LR\_colldepth(T) = \max_i LR\_colldepth(i)$$

By the definition and Lemma 2.6 we can see that $LR\_colldepth(i) \leq \min(depth(T), leaves(T))$ for $1 \leq i \leq |T|$. Hence $LR\_colldepth(T) \leq min(depth(T), leaves(T))$.

Lemma 2.7: $\sum_{i=1}^{|LR\_keyroots(T)|} Size(i) = \sum_{j=1}^{N_1} |LR\_colldepth(j)|$

Proof: Consider when node j is counted in the first summation: in the subtrees corresponding to each of its ancestors that is in $LR\_keyroots(T)$. By the definition of $LR\_colldepth()$, j is counted $LR\_colldepth(j)$ times.

Theorem 2. The time complexity is

$O(|T_1| \times |T_2| \times min(depth(T_1), leaves(T_1)) \times min(depth(T_2), leaves(T_2)))$.

The space complexity is $O(|T_1| \times |T_2|)$.

Proof: Let us consider the space complexity first. We use a permanent array for treedist and a temporary array for forestdist. Each of these two arrays requires space $O(|T_1| \times |T_2|)$.

Consider the time complexity of our algorithm. The preprocessing takes linear time. The subtree distance dynamic programming algorithm takes $Size(i) \times Size(j)$ for the subtree rooted at $T_1[i]$ and the subtree rooted at $T_2[j]$. We have a main loop that calls this subroutine several times. So the time is:

$$\sum_{i=1}^{|LR\_keyroots(T_1)|} \sum_{j=1}^{|LR\_keyroots(T_2)|} Size(i) \times Size(j)$$

$$= \sum_{i=1}^{|LR\_keyroots(T_1)|} Size(i) \times \sum_{j=1}^{|LR\_keyroots(T_2)|} Size(j).$$

By Lemma 2.6, the above equals

$$\sum_{i=1}^{N_1} LR\_colldepth(i) \times \sum_{j=1}^{N_2} LR\_colldepth(j).$$

This is less than

$$|T_1| \times |T_2| \times LR\_colldepth(T_1) \times LR\_colldepth(T_2)$$

By the definition of LR_colldepth, we have that the time complexity is

$$O(|T_1| \times |T_2| \times min(depth(T_1), leaves(T_1)) \times min(depth(T_2), leaves(T_2))).$$

□

These time and space complexities are an improvement over the $O(|T_1| \times |T_2| \times depth(T_1)^2 \times depth(T_2)^2)$ time and space complexity of [T-79].

Note: If we use a right-to-left postorder numbering for tree nodes and define similar functions $r(i)$, $RL\_keyroots(T)$ and $RL\_colldepth(i)$, we can have the same result as above. The complexity will be $\sum_{i=1}^{N_1} RL\_colldepth(i) \times \sum_{j=1}^{N_2} RL\_colldepth(j)$.

Clearly, using the left-to-right or right-to-left postorder numberings give same worst-case time complexity. However, in practice it may be beneficial to choose the ordering that gives the lower of the following two products: $\sum_{i=1}^{N_1} LR\_colldepth(i) \times \sum_{j=1}^{N_2} LR\_colldepth(j)$ and $\sum_{i=1}^{N_1} RL\_colldepth(i) \times \sum_{j=1}^{N_2} RL\_colldepth(j)$.

2.3.2. Mapping

A natural question to ask is to give a mapping that yields the distance computed. This is important since in some applications people want to know the editing sequence itself instead of distance. Also given two trees, one may ask what is the largest common substructure of these two trees. This is analogous to the longest common substring problem for strings and can be obtained easily by choosing the relabeling cost to be 2, the insert cost be 1, and the delete cost be 1. The largest common substructure is then the subgraph induced by the nodes from mapping pairs where the two nodes in the pair are the same. We can find the mapping in the same sequential time and space complexity as finding the distance. The mapping is produced by our toolkit.

In order to get the best mapping, we will need the *treedist* array. So we can run our basic algorithm to compute array *treedist*.

Let $root_1 = |T_1|$ and $root_2 = |T_2|$, and $lm_1 = l(root_1)$ and $lm_2 = l(root_2)$.

Let $i = root_1$ and $j = root_2$, and M be a queue of pairs of integers.

Let S be a stack of pairs of integers. We then can do the following until $i < lm_1$ or $j < lm_2$.

Let $item_1$ = $forestdist(i-1,j)$ + $\gamma(T_1[i]\rightarrow\Lambda)$, and $item_2$ = $forestdist(i,j-1)$ + $\gamma(\Lambda\rightarrow T_2[j])$.

If $l(i)=lm_1$ and $l(j)=lm_2$, let $item_3$ = $forestdist(i-1,j-1)+\gamma(T_1[i]\rightarrow T_2[j])$

If $l(i)\neq lm_1$ or $l(j)\neq lm_2$, let $item_3$ = $forestdist(l(i)-1,l(j)-1)+treedist(i,j)$

If $item_1=min(item_1,item_2,item_3)$, then let $i=i-1$.

Else if $item_2=min(item_1,item_2,item_3)$, then let $j=j-1$.

Else if $item_3=min(item_1,item_2,item_3)$, then there are two cases.

Case 1: $l(i)=lm_1$ and $l(j)=lm_2$. So, let $i=i-1$ and $j=j-1$ and put (i,j) in M. This means (i,j) is in the best mapping.

Case 2: $l(i)\neq lm_1$ or $l(j)\neq lm_2$. So, let $i=l(i)-1$ and $j=l(j)-1$ and put (i,j) in S. This means that we have to repeat the same procedure for subtree pair ($tree1(i)$,$tree2(j)$). (That is, i will become $root_1$ and j will become $root_2$.)

The above procedure can be formalized to the following program.

```
Mapping computation
Compute treedist array using basic algorithm.
Initial S to (|T1|,|T2|) and initial M to empty.
while (not_empty(S))
begin
  (m,n)=pop(S);
  treedist(m,n); (* to get forestdist array *)
  mapping(m,n);
end

Here is the computation of mapping(m,n)
lm=l(m); ln=l(n);
i=m; j=n;
```

```
While (i≥lm and j≥ln)
begin
  item1 = forestdist(i-1,j) + γ(T1[i]→Λ);
  item2 = forestdist(i,j-1) + γ(Λ→T2[j]);
  if (l(i)=lm and l(j)=ln)
    item3 = forestdist(i-1,j-1) +γ(T1[i]→T2[j]);
  else  (* l(i)≠lm or l(j)≠ln *)
    item3 = forestdist(l(i)-1,l(j)-1)+treedist(i,j);
  if (item1=min(item1,item2,item3))
    i=i-1;
  else if (item2=min(item1,item2,item3))
    j=j-1;
  else (* item3=min(item1,item2,item3) *)
    if (l(i)=lm and l(j)=ln)
    begin
      i=i-1;
      j=j-1;
      push_queue((i,j),M);
    end
    else (* l(i)≠lm or l(j)≠ln *)
    begin
      i=l(i)-1;
      j=l(j)-1;
      push_stack((i,j),S);
    end
end
```

2.3.3. Parallel Implementation

A straightforward transformation of our algorithm to a parallel one yields an algorithm with time complexity $O(N_1+N_2)$ whereas [T-79] and [Z-83] have time complexity $O((N_1+N_2) \times (depth(T_1)+depth(T_2)))$. Our algorithm uses $O(\min(|T_1|,|T_2|) \times leaves(T_1) \times leaves(T_2))$ processors.*

The algorithm computes in "waves" for all subtree pairs $tree(i)$ and $tree(j)$, where $i \in LR\_keyroots(T_1)$ and $j \in LR\_keyroots(T_2)$, simultaneously. We start at wave 0. At wave k, for each such subtree pair $tree(i)$ and $tree(j)$, compute $forestdist(l(i)..i_1, l(j)..j_1)$, where $(i_1-l(i))+(j_1-l(j))=k$.

We now present the parallel algorithm in detail. (When the PARBEGIN - PAREND construct surrounds one or more for loops, it means that every setting of the iterators in the enclosed for loops can be executed in parallel. The semantics are those of the sequential program ignoring this construct.)

In the algorithm $dist[i,j]$ is the array for the computation of $treedist(i,j)$. Therefore $dist[i,j][p,q]$ is the distance $forestdist(l(i)..p, l(j)..q)$ and is the p,qth member of the array computing $treedist(i,j)$.

* Actually, by controlling the starting point of each treedist computation more carefully, we can reduce the processor bound to $O(\min(|T1|,|T2|) \times min(depth(T_1), leaves(T_1)) \times min(depth(T_2), leaves(T_2))$. The algorithm is more complicated however.

```
Algorithm Parallel Distance

begin

PARBEGIN
  for i':=1 to |LR_keyroots(T1)|
    for j':=1 to |LR_keyroots(T2)|
      i:=LR_keyroots1[i']
      j:=LR_keyroots2[j']
      dist[i,j][l(i)-1,l(j)-1]:=0 /* initializes temporary array for each tree dist */
PAREND


for k:=0 to N-1
 PARBEGIN
   for i':=1 to |LR_keyroots(T1)|
     for j':=1 to |LR_keyrcots(T2)|
       i:=LR_keyroots1[i']
       j:=LR_keyroots2[j']
       dist[i,j][l(i)+k,l(j)-1]:=
           dist[i,j][l(i)+k-1,l(j)-1]+γ(T1[l[i]+k]→Λ)
 PAREND


for k:=0 to M-1
 PARBEGIN
   for i':=1 to |LR_keyroots(T1)|
     for j':=1 to |LR_keyroots(T2)|
       i:=LR_keyroots1[i']
       j:=LR_keyroots2[j']
       dist[i,j][l(i)-1,l(j)+k]:=
           dist[i,j][l(i)-1,l(j)+k-1]+γ(Λ→T1[l[j]+k])
 PAREND
```

```
for k:=0 to N+M-2
 PARBEGIN
  for i':=1 to |LR_keyroots(T1)|
   for j':=1 to |LR_keyroots(T2)|
    i:=LR_keyroots1[i']
    j:=LR_keyroots2[j']
    for i1-l(i)+j1-l(j)=k  where l(i)≤i1≤i, l(j)≤j1≤j
     if l(i)=l(i1) and l(j)=l(j1) then
      dist[i,j][i1,j1]:=min{
       dist[i,j][i1-1,j1]+γ(T1[i1]→Λ)
       dist[i,j][i1,j1-1]+γ(Λ→T2[j1])
       dist[i,j][i1-1,j1-1]+γ(T1[i1]→T2[j1])
      }
      treedist(i1,j1):=dist[i,j][i1,j1]
     else
      dist[i,j][i1,j1]:=min{
       dist[i,j][i1-1,j1]+γ(T1[i1]→Λ)
       dist[i,j][i1,j1-1]+γ(Λ→T2[j1])
       dist[i,j][l(i1)-1,l(j1)-1]+treedist[i1,j1]
      }
end
```

It is easy to see that in the above algorithm all the terms, except $treedist[i_1,j_1]$, are available whenever needed. We now show that $treedist[i_1,j_1]$ is available whenever we use it. Our argument is similar to the one we used in the sequential case.

Notice that we compute all terms such that $(i_1-l(i))+(j_1-l(j))=k$ together. During that computation, all terms such that $(i_1-l(i))+(j_1-l(j))<k$ are available. So, when we need item $treedist[i_1,j_1]$, either $l(i_1)>l(i)$ or $l(j_1)>l(j)$. Let i_2 be the lowest ancestor of i_1 such that $i_2 \in LR\_keyroots(T_1)$. Let j_2 be the lowest ancestor of j_1 such that $j_2 \in LR\_keyroots(T_2)$. Since $l(i_1)=l(i_2)$ and $l(j_1)=l(j_2)$ we know either $l(i_2)>l(i)$ or $l(j_2)>l(j)$. Therefore, $(i_1-l(i_2))+(j_1-l(j_2))<(i_1-l(i))+(j_1-l(j)) = k$. Hence $treedist[i_1,j_1]$ was already computed in the computation of $dist[i_2,j_2][i_1,j_1]$ and put into the permanent tree distance array. This settles correctness.

Theorem 3: The Parallel Distance Algorithm has time complexity $O(|T_1|+|T_2|)$.

Proof: By simple analysis of the for loop. □

2.3.4. From Trees to Strings

Strings are an important special case of trees. This algorithm is a generalization of the natural dynamic programming algorithms on strings in two senses: time complexity and algorithmic style.

First we consider the time complexity. Since a string has only one leaf, applying our algorithms to strings yields a time complexity of $O(|T_1| \times |T_2|)$. This is the same as that of the best available algorithm for the general problem of string distance.

Second we consider the algorithm itself. For a string S, $LR\_keyroots(S) = root$. So the main loop will only have one iteration. In the dynamic programming subroutine, since $l(i) = 1$, we will never come to the case $l(i) \neq l(i_1)$ or $l(j) \neq l(j_1)$. So if we change i to $|T_1|$, j to $|T_2|$, $l(i)$ to 1, $l(j)$ to 1, delete the main loop and delete the case where $l(i) \neq l(i_1)$ or $l(j) \neq l(j_1)$, we will have exactly the string distance algorithm.

CHAPTER 3

The General Technique Applied to Other Problems

Many problems in strings can be solved with dynamic programming. Similarly, our algorithm not only applies to tree distance but also provides a way to do dynamic programming for a variety of tree problems with the same time complexity. In this section we show how to apply this general paradigm to approximate tree matching and constraint tree editing.

3.1. Algorithm Template

Here is the general form of the algorithm (assuming a left-to-right postorder traversal):

preprocessing (find postordering, LR_keyroots(), and l()).

main loop

for i' := 1 to $|LR\_keyroots(T_1)|$

for j' := 1 to $|LR\_keyroots(T_2)|$

$i = LR\_keyroots(T_1)[i']$;

$j = LR\_keyroots(T_2)[j']$;

compute $Tree\_D(i,j)$;

```
subroutine for Tree_D (i,j)
 empty_initialization
 for i1 := l(i) to i
    left_initialization
 for j1 := l(j) to j
    right_initialization
 for i1 := l(i) to i
    for j1 := l(j) to j
       if l(i1)=l(i) and l(j1)=l(j) then
          general_if_computation
          Tree_D (i1,j1) = Forest_D (T1[l(i)..i1],T2[l(j)..j1]);
       else
          general_else_computation
```

3.2. Approximate Tree Matching

We first consider approximate string matching [S-80, U-83, U-85, LV-86]. We will then give two natural generalizations of approximate string matching to approximate tree matching. This will also be a generalization of the exact tree matching algorithm as found in Hoffman and O'Donnell [HO-82].

The approximate string matching problem is the following. Given two strings *STEXT* and *SPAT*, the problem is to compute, for each i,

$SD[i,SPAT] = \min_j \{D(STEXT[j..i],SPAT)\}$, where $1 \le j \le i+1$ and D is the string distance metric. In the other word, the problem is to compute, for each i, the minimum number of

editing operations between the "pattern" string $SPAT[1..|PAT|]$ and the "text string" $STEXT[1..i]$ where any prefix can be removed from $STEXT[1..i]$. (Intuitively, the algorithm finds the "occurrence" in *TEXT* that most closely matches *PAT*.)

To extend this problem to trees, we must generalize the notion of removing a prefix. For us, a prefix will mean a collection of subtrees.

We first define two operations at a node.

Removing at node $T[i]$ means removing the subtree rooted at $T[i]$. In other word delete $T[l(i)..i]$.

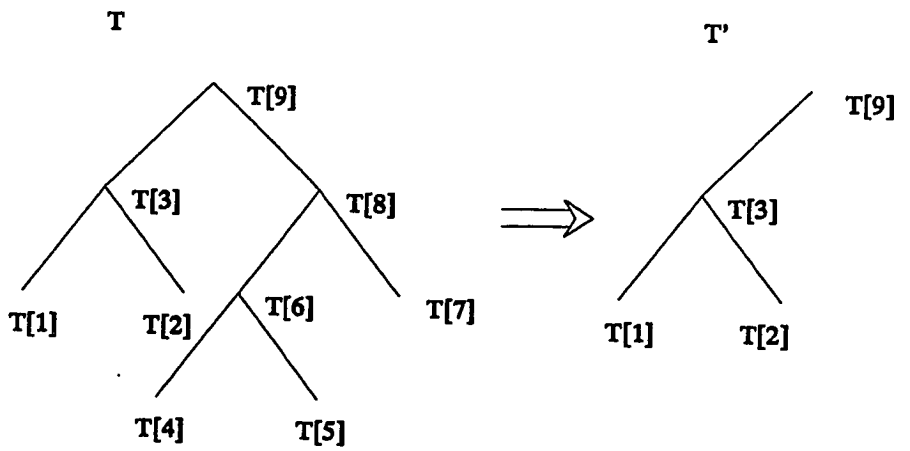


Figure 3.1 Remove subtree rooted at $T[8]$

Pruning at node $T[i]$ means removing all the descendants of $T[i]$. In other words, delete $T[l(i)..i-1]$. (Thus, a pruning never eliminates the entire tree.)

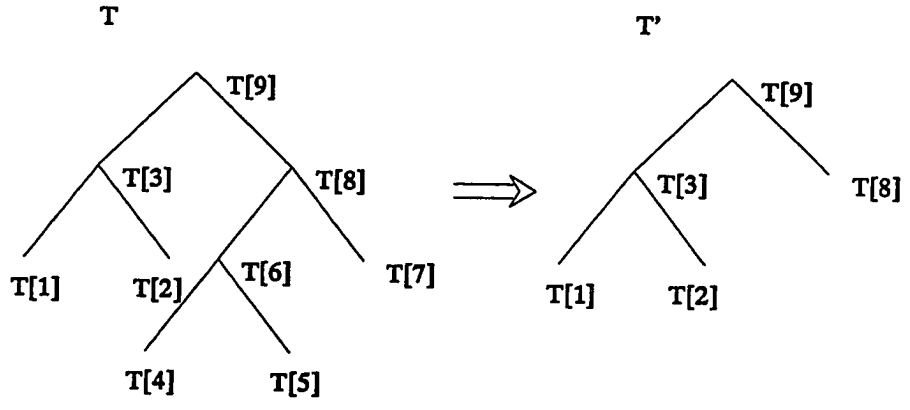


Figure 3.2 Pruning at $T[8]$ -- remove all its proper descendants

Assume an ordering for tree T. Define a subtree set $S(T)$ as follows: $S(T)$ is a set of numbers satisfying

1) $i \in S(T)$ implies that $1 \le i \le |T|$

2) $i,j \in S(T)$ implies that neither is an ancestor of the other.

Define $R(T,S(T))$ to be the tree T with removing at all nodes in $S(T)$.

Define $P(T,S(T))$ to be the tree T with pruning at all nodes in $S(T)$.

Now we can give the definition of approximate tree matching. Given tree *TEXT* and *PAT*, for each i, we want to calculate

$$DR(TEXT[l(i)..i,PAT) = \min_S \{ (treedist(R(T[l(i)..i],S(T[l(i)..i])),PAT) \}.$$

$$DP(TEXT[l(i)..i],PAT) = \min_{S} \{ (treedist(P(T[l(i)..i],S(T[l(i)..i])),PAT) \}.$$

The minimum here is over all possible subtree sets $S(T[l(i)..i])$. We consider each generalization in turn.

3.2.1. Remove any Number of Subtrees from TEXT Tree

The problem is as follows: Given trees T_1 and T_2, we want to know what is the minimum distance between $T_1[l(i)..i]$ and T_2 when zero or more subtrees can be removed from $T_1[l(i)..i]$.

Let $F\_DR(T_1[l(i)..i_1],T_2[l(j)..j_1])$ denote the minimum distance between forest $T_1[l(i)..i_1]$ and $T_2[l(j)..j_1]$ with zero or more subtrees removed from $T_1[l(i)..i_1]$. Let $T\_DR(i,j)$ denote the minimum distance between tree $T_1[l(i)..i]$ and $T_2[l(j)..j]$ with zero or more subtrees removed from $T_1[l(i)..i]$. We write the algorithm in the form suggested by the algorithm template.

Algorithm Subtree Removal

empty_initialization:
$F\_DR(\varnothing,\varnothing)=0$

left_initialization:
$F\_DR(T_1[l(i)..i_1],\varnothing)=0$

right_initialization:
$F\_DR(\varnothing,T_2[l(j)..j_1])=F\_DR(\varnothing,T_2[l(j)..j_1-1])+\gamma(\Lambda\rightarrow T_2[j_1])$

general_if_computation
/* applies if $l(i_1)=l(i)$ and $l(j_1)=l(j)$*/
$F\_DR(T_1[l(i)..i_1],T_2[l(j)..j_1])=$ min{
$F\_DR(\varnothing,T_2[l(j)..j_1])$,
$F\_DR(T_1[l(i)..i_1-1],T_2[l(j)..j_1])+\gamma(T_1[i_1]\rightarrow\Lambda)$,
$F\_DR(T_1[l(i)..i_1],T_2[l(j)..j_1-1])+\gamma(\Lambda\rightarrow T_2[j_1])$,
$F\_DR(T_1[l(i)..i_1-1],T_2[l(j)..j_1-1])+\gamma(T_1[i_1]\rightarrow T_2[j_1])$ }
/*put the derived treedist in the permanent array, as specified by template */

general_else_computation
$F\_DR(T_1[l(i)..i_1],T_2[l(j)..j_1])=$ min{
$F\_DR(T_1[l(i)..l(i_1)-1],T_2[l(j)..j_1])$,
$F\_DR(T_1[l(i)..i_1-1],T_2[l(j)..j_1])+\gamma(T_1[i_1]\rightarrow\Lambda)$,
$F\_DR(T_1[l(i)..i_1],T_2[l(j)..j_1-1])+\gamma(\Lambda\rightarrow T_2[j_1])$,
$F\_DR(T_1[l(i)..l(i_1)-1],T_2[l(j)..l(j_1)-1])+T\_DR(i_1,j_1)$ }

Lemma 3.1: Algorithm Subtree Removal is correct.

Proof: First we show that the initialization is correct. The empty-initialization and the right_initialization is the same as in the tree distance algorithm. The left-initialization $F\_DR(T_1[l(i)..i_1],\varnothing)=0$ is correct, because we can remove all of $T_1[l(i)..i_1]$.

For the general term $F\_DR(T_1[l(i)..i_1],T_2[l(j)..j_1])$, we ask first whether the subtree $T_1[l(i_1)..i_1]$ is removed or not. If it is removed, then the distance should be $F\_DR(T_1[l(i)..l(i_1)-1],T_2[l(j)..j_1])$. Otherwise, consider the mapping between $T_1[l(i)..i_1]$ and $T_2[l(j)..j_1]$ after we perform an optimal removal of subtrees of $T_1[l(i)..i_1]$. Now we have the same three cases as in Lemma 2.4. Hence the general expression should be the minimum of these four terms:

$F\_DR(T_1[l(i)..i_1],T_2[l(j)..j_1])=$ min{
$F\_DR(T_1[l(i)..l(i_1)-1],T_2[l(j)..j_1])$,
$F\_DR(T_1[l(i)..i_1-1],T_2[l(j)..j_1])+\gamma(T_1[i_1]\rightarrow\Lambda)$,
$F\_DR(T_1[l(i)..i_1],T_2[l(j)..j_1-1])+\gamma(\Lambda\rightarrow T_2[j_1])$,
$F\_DR(T_1[l(i)..l(i_1)-1],T_2[l(j)..l(j_1)-1])$
$+F\_DR(T_1[l(i_1)..i_1-1],T_2[l(j_1)..j_1-1])+\gamma(T_1[i_1]\rightarrow T_2[j_1])$ }

As in lemma 2.5, this specializes to the general_if_computation and the general_else_computation given in the algorithm. □

3.2.2. Prune at any Number of Nodes from the TEXT Tree

Given trees T_1 and T_2, we want to know what is the minimum distance between $T_1[l(i)..i]$ and T_2 when there have been zero or more prunings at nodes of $T_1[l(i)..i]$.

Let $F\_DP(T_1[l(i)..i_1],T_2[l(j)..j_1])$ denote the minimum distance between forest $T_1[l(i)..i_1]$ and $T_2[l(j)..j_1]$ with zero or more pruning from $T_1[l(i)..i_1]$. Let $T\_DP(i,j)$ denote the minimum distance between tree $T_1[l(i)..i]$ and $T_2[l(j)..j]$ with zero or more pruning from $T_1[l(i)..i]$. The following initialization and general term computation steps will give us an algorithm to solve our problem.

Algorithm Prunings

empty_initialization:
$F\_DP(\varnothing,\varnothing)=0$

left_initialization:
$F\_DP(T_1[l(i)..i_1],\varnothing) = F\_DP(T_1[l(i)..l(i_1)-1],\varnothing)+\gamma(T_1[i_1]\rightarrow\Lambda)$

right_initialization:
$F\_DP(\varnothing,T_2[l(j)..j_1]) = F\_DP(\varnothing,T_2[l(j)..j_1-1])+\gamma(\Lambda\rightarrow T_2[j_1])$

general_if_computation
/* applies if $l(i_1)=l(i)$ and $l(j_1)=l(j)$*/
$F\_DP(T_1[l(i)..i_1],T_2[l(j)..j_1])=$ min{
$F\_DP(\varnothing,T_2[l(j)..j_1-1])+\gamma(T_1[i_1]\rightarrow T_2[j_1])$,
$F\_DP(T_1[l(i)..i_1-1],T_2[l(j)..j_1])+\gamma(T_1[i_1]\rightarrow\Lambda)$,
$F\_DP(T_1[l(i)..i_1],T_2[l(j)..j_1-1])+\gamma(\Lambda\rightarrow T_2[j_1])$,
$F\_DP(T_1[l(i)..i_1-1],T_2[l(j)..j_1-1])+\gamma(T_1[i_1]\rightarrow T_2[j_1])$ }
/*put the derived treedist in the permanent array, as specified by template */

general_else_computation
$F\_DP(T_1[l(i)..i_1],T_2[l(j)..j_1])=$ min{
$F\_DP(T_1[l(i)..l(i_1)-1],T_2[l(j)..j_1])+\gamma(T_1[i_1]\rightarrow\Lambda)$,
$F\_DP(T_1[l(i)..i_1-1],T_2[l(j)..j_1])+\gamma(T_1[i_1]\rightarrow\Lambda)$,
$F\_DP(T_1[l(i)..i_1],T_2[l(j)..j_1-1])+\gamma(\Lambda\rightarrow T_2[j_1])$,
$F\_DP(T_1[l(i)..l(i_1)-1],T_2[l(j)..l(j_1)-1])+T\_DP(i_1,j_1)$ }

Lemma 3.2: Algorithm Prunings is correct.

Proof: First we show that the initialization is correct. The empty-initialization and the right_initialization is the same as in the tree distance algorithm. For left-initialization, the best we can do for tree $T_1[l(i_1)..i_1]$ is to prune at $T_1[i_1]$. Therefore $F\_DP(T_1[l(i)..i_1],\varnothing)=F\_DP(T_1[l(i)..l(i_1)-1],\varnothing)+\gamma(T_1[i_1]\rightarrow\Lambda)$ Hence the left_initialization is correct.

For the general term $F\_DP(T_1[l(i)..i_1],T_2[l(j)..j_1])$, we have the following similar three cases.

1) $T_1[i_1]$ is not touched by a line of M.

1.a (with pruning) $F\_DP(T_1[l(i)..i_1-1],T_2[l(j)..j_1])+\gamma(T_1[i_1]\rightarrow\Lambda)$

1.b (without pruning) $F\_DP(T_1[l(i)..l(i_1)-1],T_2[l(j)..j_1])+\gamma(T_1[i_1]\rightarrow\Lambda)$

2) $T_2[j_1]$ is not touched by a line of M. Since we only prune from T_1, there is only one case here.

$F\_DP(T_1[l(i)..i_1],T_2[l(j)..j_1-1])+\gamma(\Lambda\rightarrow T_1[i_1])$

3) both $T_1[i_1]$ and $T_2[j_1]$ are touched by lines of M.

3.a (with pruning)

$F\_DP(T_1[l(i)..l(i_1)-1],T_2[l(j)..l(j_1)-1])$ $+F\_DP(T_1[l(i_1)..i_1-1],T_2[l(j_1)..j_1-1])$ $+\gamma(T_1[i_1]\rightarrow T_2[j_1])$

3.b (without pruning)

$F\_DP(T_1[l(i)..l(i_1)-1],T_2[l(j)..l(j_1)-1]) + F\_DP(\varnothing,T_2[l(j_1)..j_1-1])+\gamma(T_1[i_1]\rightarrow T_2[j_1])$

If $l(i)=l(i_1)$ and $l(j)=l(j_1)$, consider cases 1.b and 3.b. Case 1.b becomes $F\_DP(\varnothing,T_2[l(j)..j_1])+\gamma(T_1[i_1]\rightarrow\Lambda)$. Case 3.b becomes $F\_DP(\varnothing,T_2[l(j)..j_1-1])+\gamma(T_1[i_1]\rightarrow T_2[j_1])$. Now from the right_initialization we know that $F\_DP(\varnothing,T_2[l(j)..j_1])+\gamma(T_1[i_1]\rightarrow\Lambda) \geq F\_DP(\varnothing,T_2[l(j)..j_1-1]) +\gamma(\Lambda\rightarrow T_2[j_1])+\gamma(T_1[i_1]\rightarrow\Lambda)$ $\geq F\_DP(\varnothing,T_2[l(j)..j_1-1])+\gamma(T_1[i_1]\rightarrow T_2[j_1])$. So the distance given by case 1.b $\geq$ the distance from 3.b. The proposed general_if_computation is therefore correct where the first term handles two cases.

If $l(i)\neq l(i_1)$ or $l(j)\neq l(j_1)$, consider case 3. As in lemma 6, 3.a and 3.b can be replaced by $F\_DP(T_1[l(i)..l(i_1)-1],T_2[l(j)..l(j_1)-1])+T\_DP(i_1,j_1)$. The proposed general_else_computation is therefore correct. Hence algorithm pruning is correct. □

3.3. Constraint Tree Editing

3.3.1. Constraint String Editing

Consider first the problem of constraint string editing.

Let X is a string of length N, Y is a string of length M. And a set of constraint τ. The constraint we consider here is following form: transform X to Y with n insertion; transform X to Y with m deletion; and transform X to Y with k exchange of label.

There is a unique set τ_I which is the insert constraint that present τ. Let T be the maximum element in τ_I.

Note that τ could also be represented by the delete constraint τ_D or modify constraint τ_M, though in this paper we only consider insert constraint representation.

We have following simple formulas.

$$D_I(0,0,0) = 0$$
$$D_I(0,j,0) = \infty$$
$$D_I(i,0,0) = D_I(i-1,0,0) + \gamma(X[i] \to \Lambda)$$

$$D_I(i,j,0) = \min \begin{cases} D_I(i-1,j,0) + \gamma(X[i] \to \Lambda), \\ D_I(i-1,j-1,0) + \gamma(X[i] \to Y[j]) \end{cases} \text{where } 1 \le i \le N \text{ and } 1 \le j \le M$$

$$D_I(0,0,I) = \infty$$
$$D_I(i,0,I) = \infty$$
$$D_I(0,j,I) = D_I(0,j-1,I-1) + \gamma(\Lambda \to Y[j])$$

$$D_I(i,j,I) = \min \begin{cases} D_I(i-1,j,I) + \gamma(X[i] \to \Lambda), \\ D_I(i,j-1,I-1) + \gamma(\Lambda \to Y[j]), \\ D_I(i-1,j-1,I) + \gamma(X[i] \to Y[j]) \end{cases} \text{where } 1 \le I \le T,\ 1 \le i \le N \text{ and } 1 \le j \le M$$

$$D_D(0,0,0) = 0$$
$$D_D(i,0,0) = \infty$$
$$D_D(0,j,0) = D_D(0,j-1,0) + \gamma(\Lambda \to X[i])$$

$$D_D(i,j,0) = \min \begin{cases} D_D(i,j-1,0) + \gamma(\Lambda \to X[i]), \\ D_D(i-1,j-1,0) + \gamma(X[i] \to Y[j]) \end{cases} \text{where } 1 \le i \le N \text{ and } 1 \le j \le M$$

$$D_D(0,0,D) = \infty$$
$$D_D(0,j,D) = \infty$$
$$D_D(i,0,D) = D_D(i-1,0,D-1) + \gamma(X[i] \to \Lambda)$$

$$D_D(i,j,D) = \min \begin{cases} D_D(i-1,j,D-1) + \gamma(X[i] \to \Lambda), \\ D_D(i,j-1,D) + \gamma(\Lambda \to Y[j]), \\ D_D(i-1,j-1,D) + \gamma(X[i] \to Y[j]) \end{cases}$$
where $1 \le D \le N - M + T$, $1 \le i \le N$ and $1 \le j \le M$

From above formula, it is clear that we can compute from $I=0$ to $I=T$, and find the minimum $D_I(N,M,k)$ such that $k \in \tau$. So the time will be $O(TNM)$. Space will be $O(NM)$ since for computing array $D_I(i,j,I)$ we only need array $D_I(i,j,I-1)$.

We can do better if T is small by cutting the corner of two dimension array for each I. This is based on following observation.

1) For any $0 \le I \le T$, $D_I(i,j,I)$ such that $i-j<-T$ will not be useful. This is because when $i-j<-T$ then, $j-i>T$, so $D_I(i,j,I)$ will be undefined.

2) For any $0 \le I \le T$, $D_I(i,j,I)$ such that $i-j>N-M+T$ will not be useful. This is because when $i-j>N-M+T$ then, $M-j-(N-i)>T$, so though $D_I(i,j,I)$ will be defined, it is not useful for the computation of $D_I(N,M,I)$ for any $I \le T$.

Therefore for each I, we can only compute portion $-T \le i-j \le N-M+T$ diagonals. Hence the time will be $O(T(N-M+2T)min(N,M))$. and space will be $O((N-M+2T)min(N,M))$.

By a careful analysis, we can see that for each I, we only need to compute diagonals $-I, -I+1, \cdots 0, \cdots N-M+T-I$. Hence the time will be $O(T(N-M+T)min(N,M))$. But we omit this details.

Note that in practical sense, only $N-M \le 2T$ is interested. Because if $N-M>2T$ then T insertion will correspond to $N-M+T$ deletion. So total insertion or deletion will greater or equal to $3T$.

Now we have following, In case $N-M \le 2T$, time complexity will be $O(T^2 min(N,M))$ and space will be $O(Tmin(N,M))$. In case $N-M>2T$, time complexity will be $O(T(N-M)min(N,M)$ and space will be $O((N-M)M)$. Or in both case space will be $O(NM)$.

If $N-M \leq 2T$ and $T \ll min(N,M)$ then time will be $O(min(N,M))$ space will be $O(min(N,M))$.

Compare above with the best previous result in [O-87]. In case $N-M \leq 2T$ time in [O-87] is $O(NM+T^2 min(N,M))$. In case $N-M>2T$ it has the same time bound as ours. (the $O(N^2+T^2N)$ time bound given in [O-87] is ture only for $N=M$ case.)

3.3.2. Constraint Tree Editing

Now we come to the constraint tree editing problem. Let T_1 be tree of size N and T_2 be tree of size M. Let τ be the set of constraint. Like in case of string, there is a unique set of insert constraint τ_I that represents τ. We now have following result. We can have following formula.

Formula zero insertion

empty_initialization:
$D_f(0,0,0) = 0$

left_initialization:
$D_f(i,0,0) = D_f(i-1,0,0)+\gamma(X[i]\rightarrow\Lambda)$

right_initialization:
$D_f(0,j,0) = \infty$

general_if_computation:
(if $l(i)=i$ and $l(j)=j$)

$$D_f(i,j,0) = \min \begin{cases} D_f(i-1,j,0)+\gamma(X[i]\rightarrow\Lambda), \\ D_f(i-1,j-1,0)+\gamma(X[i]\rightarrow Y[j]) \end{cases} \text{where } 1\leq i\leq N \text{ and } 1\leq j\leq M$$

general_else_computation:
(if $l(i)\neq i$ or $l(j)\neq j$)

$$D_f(i,j,0) = \min \begin{cases} D_f(i-1,j,0)+\gamma(X[i]\rightarrow\Lambda), \\ D_f(l(i)-1,l(j)-1,0)+D_t(i,j,0) \end{cases} \text{where } 1\leq i\leq N \text{ and } 1\leq j\leq M$$

Lemma 3.3 Formula zero insertion is correct.

Proof:

First it is easy to see that the empty initialization and left initialization is correct. For the right initialization, one can see that it is impossible to use zero insertion to transform an empty tree to a nonempty forest.

Consider now the general_if_computation. Consider the best mapping that use zero insertion. There are two case:

1) $T_1[i]$ is not in the mapping. So $D_f(i,j,0) = D_f(i-1,j,0)+\gamma(X[i]\to\Lambda)$.

2) $T_1[i]$ and $T_2[j]$ are both in the mapping. So $D_f(i,j,0) = D_f(i-1,j-1,0)+\gamma(X[i]\to Y[j])$.

Hence general_if_computation is correct.

We can similarly prove the general_else_computation. □

Formula positive insertion

empty_initialization:
$D_f(0,0,I) = \infty$

left_initialization:
$D_f(i,0,I) = \infty$

right_initialization:
$D_f(0,j,I) = D_f(0,j-1,I-1)+\gamma(\Lambda\to Y[j])$

general_if_computation:
(if $l(i)=i$ and $l(j)=j$)

$$D_f(i,j,I) = \min\begin{cases} D_f(i-1,j,I)+\gamma(X[i]\to\Lambda), \\ D_f(i,j-1,I-1)+\gamma(\Lambda\to Y[j]), \\ D_f(i-1,j-1,I)+\gamma(X[i]\to Y[j]) \end{cases} \quad \textit{where } 1\le I\le T,\ 1\le i\le N \text{ and } 1\le j\le M$$

general_else_computation:
(if $l(i)\neq i$ *or* $l(j)\neq j$)

$$D_f(i,j,I) = \min\begin{cases} D_f(i-1,j,I)+\gamma(X[i]\to\Lambda), \\ D_f(i,j-1,I-1)+\gamma(\Lambda\to Y[j]), \\ \min_k(D_f(l(i)-1,l(j)-1,I-k)+D_t(i,j,k)) \end{cases}$$

where $1\le I\le T$, $1\le i\le N$ and $1\le j\le M$

Lemma 3.4 Formula positive insertion is correct.

Proof:

First it is easy to see that the initialization are correct.

Consider now the general_if_computation. Consider the best mapping that use exactly $I \geq 0$ insertions. There are three case:

Case 1

$T_1[i]$ is not in the mapping. So $D_f(i,j,I) = D_f(i-1,j,I) + \gamma(X[i] \rightarrow \Lambda)$.

Case 2

$T_2[j]$ is not in the mapping. So $D_f(i,j,I) = D_f(i,j-1,I-1) + \gamma(\Lambda \rightarrow Y[j])$.

Case 3

$T_1[i]$ and $T_2[j]$ are both in the mapping. So $D_f(i,j,I) = D_f(i-1,j-1,I) + \gamma(X[i] \rightarrow Y[j])$.

Hence general_if_computation is correct.

Consider now the general_else_computation. Consider the best mapping that use exactly $I \geq 0$ insertion. There are three case:

Case 1 and Case 2

Exactly the same as Case 1 and Case 2 in general_if_computation.

Case 3

Since $T_1[i]$ and $T_2[j]$ are both in the mapping. From the condition for mapping, we know that $T_1[i]$ must map to $T_2[j]$. Hence the subtree rooted at $T_1[i]$ must map to subtree rooted at $T_2[j]$. But we do not know how many insertion are present in transform subtree rooted at $T_1[i]$ to subtree rooted at $T_2[j]$ with the best mapping. So we need to consider all the possibilities. Hence

$$D_f(i,j,I) = \min_k \left\{ D_f(l(i)-1, l(j)-1, I-k) + D_t(i,j,k) \right\}$$

Hence general_else_computation is correct. □

From above formula, it is clear that we can compute from $I=0$ to $I=T$, and find the minimum $D_t(N,M,k)$ such that $k \in \tau$. For each I, the time bound is $O(INM\,min(depth(T_1),leaves(T_1))\,min(depth(T_2),leaves(T_2)))$. So time complexity is $O(T^2NM\,min(depth(T_1),leaves(T_1))\,min(depth(T_2),leaves(T_2)))$ The space is $O(TNM)$.

We can do better if T is small by cutting the corner like in the case of string. This is based on following observation.

1) Consider $D_f(1..i,1..j,I)$ for $0 \le I \le T$, The same argument like in string works. So we do not need element in $-T \le i-j \le N-M+T$.

2) Same reason that we do not need any $D_t(i,j,I)$ such that $-T \le i-j \le N-M+T$ or $-T \le l(i)-l(j) \le N-M+T$.

3) In the computation of $D_t(i,j,I)$ such that $-T \le l(i)-l(j) \le N-M+T$ and $-T \le i-j \le N-M+T$, we only need diagonals from $-T-(l(i)-l(j))$ to $N-M+T-(l(i)-l(j))$. This is because other diagonals will not fit in the $-T, N-M+T$ diagonals of main array.

So we have following result. For each $D_t(i,j,I)$ time bound is $O(I(N-M+2T)min(size(i),size(j))$. For each I computation for $D_t(N,M,I)$ will have following time bound.

$$\sum_{i=1}^{N} I(N-M+2T)^2 size(i) \le I(N-M+2T)^2 NL_1$$

$$\sum_{j=1}^{M} I(N-M+2T)^2 size(j) \le I(N-M+2T)^2 ML_2$$

So total is $O(T^2(N-M+2T)^2\,min(NL_1,ML_2))$. if $N-M \le 2T$ $O(T^4\,min(NL_1,ML_2))$. if $N-M>2T$ $O(T^2(N-M)^2\,min(NL_1,ML_2))$. Space is $O(NMT)$ or $O(T^2\,min(N,M))$.

CHAPTER 4

NP-completeness for the Editing Distance between Unordered Labeled Trees

4.1. Definitions

Unordered labeled trees are trees whose nodes are labeled and in which the left-to-right order among siblings is not significant.

The definitions are similar to those for ordered labeled trees in Chapter 2. Changes will be highlighted in bold face.

4.1.1. Editing Operations and Editing Distance between Unordered Labeled Trees

Let us consider three kind operations. Changing a node n means changing the label on n. Deleting a node n means making the children of n become the children of the parent of n and then removing n. Inserting is the complement of deleting. This means that inserting n as the child of n' will make n the parent of a **subset** (as opposed to a consecutive subsequence) of the current children of n'. Figures 1, 2, and 3 illustrate these editing operations.

1. **Relabel: To change one node label to another.**

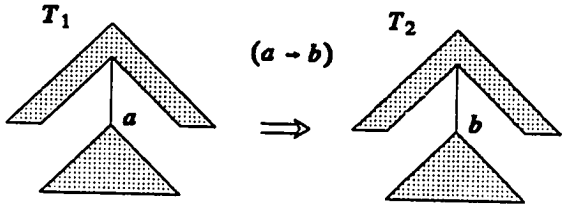


Figure 4.1 Relabeling

2. Delete: To delete a node.
(All children of the deleted node b become children of the parent a.)

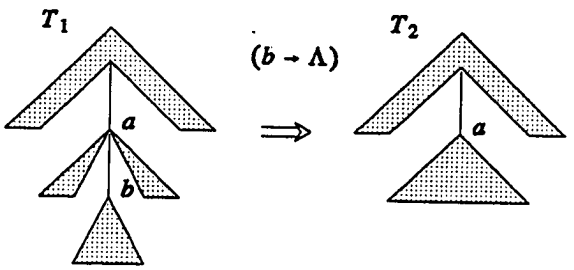


Figure 4.2 Deletion

3. Insert: To insert a node.
(A subset of the children of a become the children of b.)

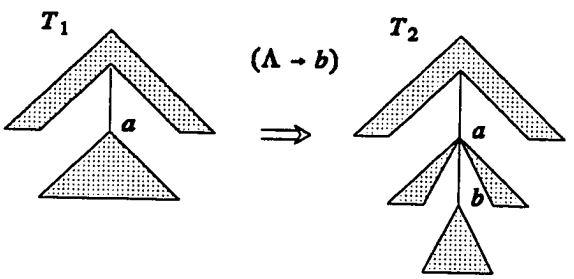


Figure 4.3 Insertion

we represent an edit operation as a pair $(a,b) \neq (\Lambda,\Lambda)$, sometimes written as $a \to b$, where a is either Λ or a label of a node in tree T_1 and b is either Λ or a label of a node in tree T_2. We call $a \to b$ a change operation if $a \neq \Lambda$ and $b \neq \Lambda$; a delete operation if $b = \Lambda$; and an insert operation if $a = \Lambda$.

Let S be a sequence $s_1, \ldots, s_k$ of edit operations. An S-derivation from A to B is a sequence of trees $A_0, \ldots A_k$ such that $A = A_0$, $B = A_k$, and $A_{i-1} \to A_i$ via s_i for $1 \leq i \leq k$.

Let γ be a cost function which assigns to each edit operation $a \to b$ a nonnegative real number $\gamma(a \to b)$.

We constrain γ to be a distance metric. That is,

i) $\gamma(a \to b) \geq 0$; $\gamma(a \to a) = 0$;

ii) $\gamma(a \to b) = \gamma(b \to a)$; and

iii) $\gamma(a \to c) \leq \gamma(a \to b) + \gamma(b \to c)$.

We extend γ to the editing operations sequence S by letting $\gamma(S) = \sum_{i=1}^{i=|S|} \gamma(s_i)$. Formally the distance between T_1 and T_2 is defined as:

$$\delta(T_1,T_2) = \min_S \{\gamma(S) \mid S \text{ is an edit operation sequence taking } T_1 \text{ to } T_2\}.$$

δ is also a distance metric according to the definition of γ.

4.1.2. Mapping

Let T_1 and T_2 be two unordered labeled trees with N_1 and N_2 nodes respectively. Suppose that we have an ordering for tree T, then $T[i]$ means the ith node of tree T in the given ordering.

The edit operations give rise to a mapping which is a graphical specification of what edit operations apply to each node in the two unordered labeled trees.

Formally we define a triple (M,T_1,T_2) to be a mapping from T_1 to T_2, where M is any set of pair of integers (i,j) satisfying:

(1) $1 \leq i \leq N_1$, $1 \leq j \leq N_2$;
(2) For any pair of (i_1,j_1) and (i_2,j_2) in M,
   (a) $i_1 = i_2$ iff $j_1 = j_2$ (one-to-one)
   (b) $T_1[i_1]$ is an ancestor of $T_1[i_2]$ iff
   $T_2[j_1]$ is an ancestor of $T_2[j_2]$ (ancestor order preserved)

We will use M instead of (M,T_1,T_2) if there is no confusion. Let M be a mapping from T_1 to T_2. Let I and J be the sets of nodes in T_1 and T_2, not touched by any line in M. Then we can define the cost of M:

$$\gamma(M)=\sum_{(i,j)\in M}\gamma(T_1[i]\to T_2[j])+\sum_{i\in I}\gamma(T_1[i]\to\Lambda)+\sum_{j\in J}\gamma(\Lambda\to T_2[j])$$

Note the difference between the definition of mapping for the ordered labeled trees and the unordered labeled trees. For the unordered trees, there is no restriction of left and right.

Mappings can be composed. Let M_1 be a mapping from T_1 to T_2 and let M_2 be a mapping from T_2 to T_3. Define

$$M_1 \circ M_2 = \{(i,j) \mid \exists\ k \ \ s.t. \ \ (i,k) \in M_1 \text{ and } (k,j) \in M_2\}$$

Lemma 4.1:

(1) $M_1 \circ M_2$ is a mapping

(2) $\gamma(M_1 \circ M_2) \leq \gamma(M_1) + \gamma(M_2)$

Proof:

Exactly the same as the proof of Lemma 2.1. □

The relation between a mapping and a sequence of editing operations is as follows:

Lemma 4.2: Given S, a sequence $s_1, \ldots, s_k$ of edit operations from T_1 to T_2, there exists a mapping M from T_1 to T_2 such that $\gamma(M) \leq \gamma(S)$. Conversely, for any mapping M, there exists a sequence of editing operations such that $\gamma(S) = \gamma(M)$.

Proof:

Exactly the same as the proof of Lemma 2.2. □

Theorem 4.1: $\delta(T_1,T_2)=\min\{\gamma(M) \mid M \text{ is a mapping from } T_1 \text{ to } T_2\}$

Proof: Immediately from lemma 4.2. □

4.2. NP-completeness

We will reduce Exact Cover by 3-Sets to unordered labeled tree editing distance. This was suggested by Rick Statman of Carnegie Mellon.

Exact Cover by 3-Sets

INSTANCE: A finite set S with $|S|=3q$ and a collection T of 3-element subsets of S.

QUESTION: Does T contain an exact cover for S, that is, a subcollection $T' \subseteq T$ such that every element of S occurs in exactly one member of T'?

Given an instance of the exact 3-cover problem, let the set $S = \{ s_1, s_2, \ldots s_m \}$, where $m=3k$. Let $T = \{T_1, T_2, \ldots T_n\}$. Here each $T_i = \{t_{i_1}, t_{i_2}, t_{i_3}\}$, where $t_{i_j} \in S$. Without loss of generality, we assume that $n>k$.

We construct two trees as in Figure 4.4. Note that this construction can be done in polynomial time.

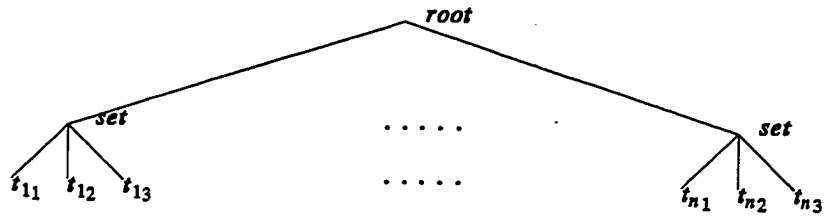
We want to show that there is an exact 3-cover iff $\delta(T_1,T_2)=3n-2k$. We first give several lemmas and then prove the theorem. Assume that T_1 and T_2 are trees as in Figure 4.4.

Lemma 4.3: Let M be a mapping between T_1 and T_2, if there are $d \geq 0$ nodes of T_2 not in mapping M, then $\gamma(M)(T_1,T_2) \geq 3n-2k+d$.

Proof:

Assume that there are $d \geq 0$ nodes of T_2 not in mapping M. Since $|T_1|=4n+1$ and $|T_2|=4(n-k)+3k+1$, $|M|=|T_2|-d=4(n-k)+3k+1-d$. Since there are at most $3k+(n-k)+1$ nodes of two trees that can have the same labels, $\gamma(M)(T_1,T_2) \geq |M|-(3k+(n-k)+1)+|T_1|-|M|+|T_2|-|M|$

T_1



T_2

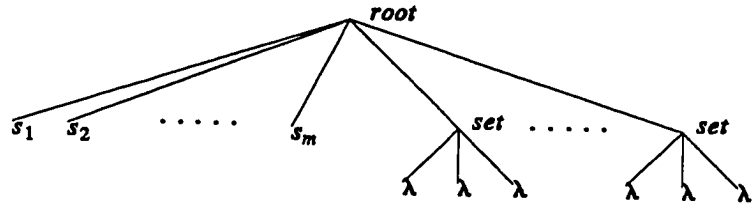


Figure 4.4 In T_2 there are $n-k$ nodes with label *set*, and $\lambda \notin S$

$= |T_1| - (3k + (n-k) + 1) + |T_2| - |M| = 4n + 1 - 2k - n - 1 + d = 3n - 2k + d.$ □

Lemma 4.4: $\delta(T_1, T_2) \geq 3(n-k) + k = 3n - 2k.$

Proof:

Let M be a mapping between T_1 and T_2, from Lemma 4.3, we know $\gamma(M)(T_1, T_2) \geq 3n - 2k$. Hence from Theorem 4.1 we know $\delta(T_1, T_2) \geq 3n - 2k$. □

Lemma 4.5: If there is an exact 3-cover, then $\delta(T_1, T_2) = 3n - 2k$.

Proof:

If there is an exact 3-cover, then there is a mapping M such that $\gamma(M)(T_1, T_2) = 3n - 2k$. From lemma 4.4 $\delta(T_1, T_2) \geq 3n - 2k$, therefore $\delta(T_1, T_2) = 3n - 2k$. □

Lemma 4.6: If $\delta(T_1,T_2)=3n-2k$, then there exists an exact 3-cover.

Proof:

If $\delta(T_1,T_2)=3n-2k$, from Lemma 4.3 we know that the best mapping M must include all nodes in T_2. It then must be the case that (*root*,*root*) is in M and $(n-k)$ nodes in T_2 with label *set* must be mapped to nodes in T_1 with label *set*. Otherwise M cannot include all the nodes in T_2. Now the nodes left in T_1 with label *set* cannot be in mapping M. Otherwise M cannot include all the nodes in T_2. Thus we can see that there is an exact 3-cover. □

Theorem 4.2: Computing the editing distance between unordered labeled trees is NP-complete.

Proof:

Given an instance of exact 3-cover, using polynomial time we can construct trees T_1 and T_2 as in Figure 4.4. From Lemma 4.5 and Lemma 4.6, there is an exact 3-cover iff $\delta(T_1,T_2)=3n-2k$. Hence determining the distance between unordered labeled trees is NP-complete. □

CHAPTER 5

Unit Cost Editing Distance between Trees

In this chapter we assume that all the cost be unit. This assumption allow us to develop several more efficient algorithms. We will present three algorithms in this chapter.

The first algorithm is an sequential algorithm. The second and third algorithms are parallel algorithms based on the application of suffix trees to the comparison problem. The cost of executing these algorithms is a monotonic increasing function of the distance between the two trees.

Results Let trees T_1 and T_2 have numbers of levels L_1 and L_2 respectively. Let k be the actual distance between T_1 and T_2. Let N be $min(|T_1|, |T_2|)$. The asymptotic running times (assuming a concurrent-read concurrent-write parallel random access machine for algorithm 2 and Algorithm 3) are:

Algorithm	Time	Processors
Algorithm 1	$k^2 N \times min(L_1, L_2)$	1
Algorithm 2 (parallel)	$k \times log(k) \times log(N)$	$k^2 \times N$
Algorithm 3 (parallel)	$(k^2 \times log(k)) + log(N)$	$k^2 \times N$

Algorithmic Significance We use the Ukkonen [U-83] technique of computing in waves along the center diagonals of the distance array. At the beginning of stage k, all distances up to $k-1$ have been computed. Stage k then computes in parallel all distances up to k. We use suffix trees, inspired by [LV86], to perform this computation fast. But, whereas Landau and Vishkin apply suffix trees to comparing strings we apply suffix trees to comparing trees.

That is, we map each of the two trees T_1 and T_2 to strings (each string is a traversal order where each node is associated with the number of its children), construct suffix trees from these strings, and then use the suffix trees to infer that portions of the T_1 are identical to portions of T_2. This leaves some subtle problems.

In the string case, if $S_1[i..i+h]=S_2[j..j+h]$, then the distance between $S_1[1..i-1]$ and $S_2[1..j-1]$ is the same as between $S_1[1..i+h]$ and $S_2[1..j+h]$. The main difficulty in the tree case is that preserving ancestor relationships in the mapping between trees prevents the analogous implication from holding. In addition, to compute the distance between two forests at stage k sometimes requires knowing whether two contained subtrees are distance k apart. We overcome these problems by exploiting the relationship between identical subforests and tree-to-tree mappings (section 5.2).

2 "Left-to-right postorder traversal notation -- the default"

Let $T[i]$ be the ith node in the tree according to the left-to-right postorder numbering (our default traversal order). $l(i)$ is the number of the leftmost leaf descendant of the subtree rooted at $T[i]$. When $T[i]$ is a leaf, $l(i)=i$. The number of children of $T[i]$ is denoted $nc(i)$. The parent of $T[i]$ is denoted $p(i)$. We define $p^0(i)=i$, $p^1(i)=p(i)$, $p^2(i)=p(p^1(i))$ and so on. Let $anc(i)=\{p^k(i) \mid 0 \le k \le depth(i)\}$.

$T[i..j]$ is the ordered subforest of T induced by the nodes numbered i to j inclusive (figure 5.1). If $i>j$, then $T[i..j] = \emptyset$. $T[1..i]$ will be referred to as *forest(i)*, when the tree referred to is clear. $T[l(i)..i]$ will be referred to as *tree* (i). *size(i)* is the number of nodes in *tree* (i). If $T[i]$ and $T[i']$ are nodes, then *lca(i,i')* is the post-order number of the least common ancestor of those two nodes. If $T[i]$ is an ancestor of $T[i']$ then the *proper child of* $T[i]$ *with respect to* $T[i']$ is the child of $T[i]$ on the path between $T[i']$ and $T[i]$.

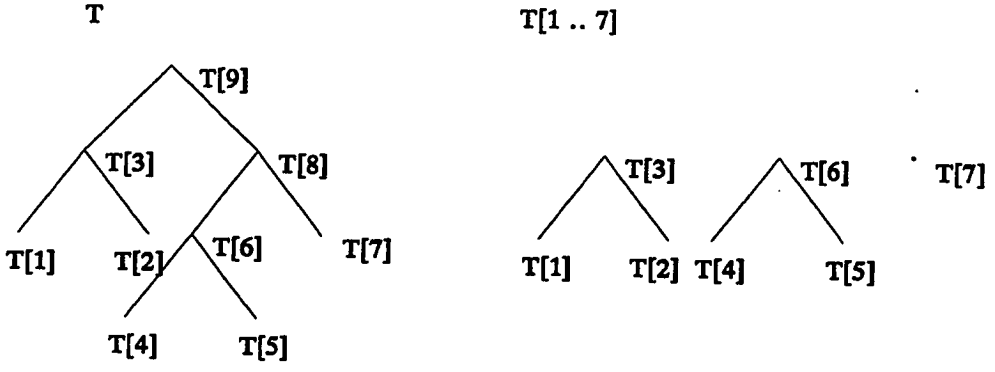


Figure 5.1 Left-to-right postorder numbering

The distance between $T_1[i'..i]$ and $T_2[j'..j]$ is denoted dist($T_1[i'..i]$, $T_2[j'..j]$) or *dist*$(i'..i, j'..j)$ if the context is clear. We use a more abbreviated notation for certain special cases. The distance between $T_1[1..i]$ and $T_2[1..j]$ is sometimes denoted *forestdist(i,j)*. The distance between the subtree rooted at i and the subtree rooted at j is sometimes denoted *treedist(i,j)*.

5.1. A Simple Algorithm

The following idea lead to improvement to the basic algorithm proposed in Chapter 2.

Deciding whether two trees differ by at most k requires less work than determining what the distance between two trees is (e.g. there is no need to compute *treedist*(i,j) where $|i-j|>k$).

Even if we don't know the distance, we can use the following approach

for j:=0 to $\lceil \log_2(|T_1|+|T_2|)\rceil$

if T_1 and T_2 are within distance 2^j

then print the distance and exit

end for

5.1.1. Relevance -- Computing for Short Distances First

Definition: Let the *treedistance triple* $(tree_1(i),tree_2(j),k)$ be shorthand for the question "Is the distance between $tree_1(i)$ and $tree_2(j)$ less than or equal to k and if so what is it?" for non-negative integer k.

Definition: For all i, j, if $k \le k'$, then we say that $(tree_1(i),tree_2(j),k')$ *covers* $(tree_1(i),tree_2(j),k)$.

Intuitively, answering $(tree_1(i),tree_2(j),k')$ gives us the answer to $(tree_1(i),tree_2(j),k)$.

We will restrict the set of treedistances calculated to those that are relevant. Intuitively, $(tree_1(i'),tree_2(j'),k')$ is relevant to $(tree_1(i),tree_2(j),k)$ if a mapping from $tree_1(i')$ to $tree_2(j')$ with distance k' could be a submapping of a mapping from $tree_1(i)$ to $tree_2(j)$ with distance k. Formally,

Definition: $(tree_1(i'),tree_2(j'),k')$ is *relevant* to $(tree_1(i),tree_2(j),k)$ if

1) $l(i) \le i' \le i$ and $l(j) \le j' \le j$.

2) $|(i'-l(i))-(j'-l(j))| \le k$.

3) $k' \le k - |(l(i')-l(i))-(l(j')-l(j))|$.

The first condition says that $tree_1(i')$ is a subtree of $tree_1(i)$a and similarly for $tree_2(j')$. If the second condition were false, then $dist(l(i)..i',l(j)..j')$ would necessarily exceed k. The difference in the third condition is the maximum distance of any mapping between $tree_1(i')$

and $tree_2(j')$ that would be a submapping of a mapping between $tree_1(i)$ and $tree_2(j)$ with distance k.

Definition: $(tree_1(i''),tree_2(j''),k'')$ is *transitively relevant* to $(tree_1(i),tree_2(j),k)$ if either $(tree_1(i''),tree_2(j''),k'')$ is relevant to $(tree_1(i),tree_2(j),k)$ or if $(tree_1(i''),tree_2(j''),k'')$ is relevant to $(tree_1(i'),tree_2(j'),k')$ and $(tree_1(i'),tree_2(j'),k')$ is transitively relevant $(tree_1(i),tree_2(j),k)$.

Lemma 5.1 (Relevance): If $(tree_1(i'),tree_2(j'),k')$ is relevant to $(tree_1(i),tree_2(j),k)$ and $(tree_1(i''),tree_2(j''),k'')$ is relevant to $(tree_1(i'),tree_2(j'),k')$, then $(tree_1(i''),tree_2(j''),k'')$ is relevant to $(tree_1(i),tree_2(j),k)$.

Proof: Condition 1 is obvious.

Condition 2. We want to show that $|(i''-l(i))-(j''-l(j))|\leq k$.

Let us manipulate the left hand side.

$|(i''-l(i))-(j''-l(j))|=$

$|(l(i')-l(i'))+(l(j')-l(j'))$

$+(i''-l(i))-(j''-l(j))|\leq$

$|(i''-l(i'))-(j''-l(j'))| + |(l(i')-l(i))-(l(j')-l(j))|$, by the triangle inequality.

But this expression is bounded from above by $k'+(k-k')=k$.

Condition 3. By definition, $k'\leq k- |(l(i')-l(i))-(l(j')-l(j))|$.

Also, $k''\leq k'- |(l(i'')-l(i'))-(l(j'')-l(j'))|$.

By the triangle inequality, $|(l(i'')-l(i))-(l(j'')-l(j)| \leq |(l(i')-l(i))-(l(j')-l(j))|+ |(l(i'')-l(i'))-(l(j'')-l(j'))|$.

So, $k''\leq$ (k $- |(l(i')-l(i))-(l(j')-l(j))|)- |(l(i'')-l(i'))-(l(j'')-l(j'))| \leq k-$

$|(l(i'')-l(i))-(l(j'')-l(j)|$. □

Corollary 1: If $(tree_1(i''),tree_2(j''),k'')$ is transitively relevant to $(tree_1(i),tree_2(j),k)$ then it is relevant to $(tree_1(i),tree_2(j),k)$. □

Definition: $allow(k,i,j)=k-|l(i)-l(j)|$.

Corollary 2: If $(tree_1(i'),tree_2(j'),h)$ is transitively relevant to (T_1,T_2,k), then $0 \leq h \leq allow(k,i',j')$ and $|i'-j'| \leq k$.

Proof: By corollary 1, transitive relevance is equivalent to relevance. The rest follows by definition of $allow(k,i,j)$ and relevance. □

Suppose that at some stage of the algorithm, we are only interested in (T_1,T_2,p). Lemma 5.1 and its corollaries permit us to restrict our attention to only relevant subtrees. This saves us space and processors.

5.1.2. A Simple Algorithm

Let us now consider the algorithm to solve the problem of (T_1,T_2,K), i.e. if the distance between T_1 and T_2 is within K, then return the distance, else fail.

In order to solve the problem of (T_1,T_2) we only need to solve $(tree_1(i),tree_2(j),K-|l(i)-l(j)|)$ such that $|i-j| \leq k$ and $|l(i)-l(j)| \leq K$. The reason is that this set of triples are relevant to (T_1,T_2,K) and covers all the relevant triples.

Hence from Lemma 5.1 we can observe following:

(1) We only need to compute $treedist(i,j)$ such that $|i-j| \leq K$ and $|l(i)-l(j) \leq K$.

(2) In the temporary array for compute *treedist*(i,j) we only need to consider center diagonals less than $K-|l(i)-l(j)|$.

The above observations lead to an algorithm which is better than the basic algorithm especially in the case where K is small.

Preprocessing

(To compute $l()$)

Main loop (to compute *treedist*(T_1,T_2,K))

for $i=1$ to $|T_1|$

for $j=max(i-k,1)$ to $min(|T_2|,i+k)$

if $|l(i)-l(j)|\leq K$

Compute *treedist*$(i,j,K-|l(i)-l(j)|)$;

We also use dynamic programming to compute *treedist*(i,j,k). The forestdist values computed and used here are put in a temporary array that is freed once the corresponding treedist is computed. The treedist values are put in the permanent treedist array.

The computation of *treedist*(i,j,k).

forestdist$(\emptyset,\emptyset)=0$;

for $i_1:=l(i)$ to $l(i)+k$

forestdist$(T_1[l(i)..i_1],\emptyset) = i_1-l(i)+1$

for $j_1:=l(j)$ to $l(j)+k$

forestdist$(\emptyset,T_2[l(j)..j_1]) = j_1-l(j)+1$

for $i_1:=l(i)$ to i

for $j_1 := max(i_1-k, l(j))$ to $min(j, j_1+k)$

if $l(i_1)=l(i)$ and $l(j_1)=l(j)$ then

$forestdist(T_1[l(i)..i_1], T_2[l(j)..j_1]) = \min\{$

$forestdist(T_1[l(i)..i_1-1], T_2[l(j)..j_1]) + 1,$

$forestdist(T_1[l(i)..i_1], T_2[l(j)..j_1-1]) + 1,$

$forestdist(T_1[l(i)..i_1-1], T_2[l(j)..j_1-1]) + \gamma(T_1[i_1] \to T_2[j_1])\}$

else

$forestdist(T_1[l(i)..i_1], T_2[l(j)..j_1]) = \min\{$

$forestdist(T_1[l(i)..i_1-1], T_2[l(j)..j_1]) + 1,$

$forestdist(T_1[l(i)..i_1], T_2[l(j)..j_1-1]) + 1,$

$forestdist(T_1[l(i)..l(i_1)-1], T_2[l(j)..l(j_1)-1]) + treedist(i_1, j_1))\}$

Note that we can combine the idea of keyroots to above algorithm, though we will not give the details here.

Let us consider the time complexity of this new algorithm.

Lemma 5.2 The time complexity of this simple algorithm is $O(K^2 min(T_1,T_2) min(L_1,L_2))$.

Proof:

For the computation of $treedist(i,j)$, the time will be bounded by $K \times min(size(i), size(j))$ since we only compute about $2\times(K-|l(i)-l(j)|)$ diagonals. For each $tree_1(i)$ in T_1, $treedist(i,l)$ will be computed for at most $2\times K+1$ $tree_2(l)$ in T_2 since at most $2\times K+1$ values of l can satisfy the condition $|l-l| \le k$. Hence the time will be at most $k^2 size(i)$ for each $tree_1(i)$.

The total time will be bounded by $K^2 \times T_1 \times L_1$. Symmetrically the total time will also be

bounded by $K^2 \times T_2 \times L_2$.

So we know that the time complexity will be bounded by following:

(1) $K^2 \times T_1 \times L_1$.

(2) $K^2 \times T_2 \times L_2$.

(3) $T_1 \times T_2 \times L_1 \times L_2$.

If $T_1 \leq T_2$ and $L_1 \leq L_2$, then from 1) we know that time complexity is $O(K^2 min(T_1,T_2) min(L_1,L_2))$.

If $T_1 \leq T_2$ and $L_2 \leq L_1$, then there are two cases. The first case is that $K \leq T_1$, since $T_2 \leq T_1 + K$, $T_2 \leq 2 \times T_1$. From 2) we know that time is $K^2 \times T_2 \times L_2$. So it will be $2 \times k^2 \times T_1 \times L_2$. Hence the time complexity is $O(K^2 min(T_1,T_2) min(L_1,L_2))$. The second case is that $K \geq T_1$, since $T_2 \leq T_1 + k$, $T_2 \leq 2 \times K$. From 3) we know that the time is $T_1 \times T_2 \times L_1 \times L_2$. So it will be less than $T_1 \times 2K \times K \times L_2$. Hence the time complexity is $O(K^2 min(T_1,T_2) min(L_1,L_2))$.

The other situation of $T_2 \leq T_1$ and $L_2 \leq L_1$ ($T_2 \leq T_1$ and $L_1 \leq L_2$) can be proved similarly.

Hence we prove that the time complexity of the simple algorithm is $O(K^2 min(T_1,T_2) min(L_1,L_2))$. □

5.2. Improving the Simple Algorithm

5.2.1. Strategies and Inspiration

5.2.1.1. Basic Strategies

The following idea lead to dramatic improvements to the simple algorithm proposed above.

If a portion of the two trees (or of two subtrees) is the same, then it should be possible to speed up the algorithm over this portion. (We will use suffix trees for this purpose.) This will be the difficult step to apply. For inspiration, we look at a modified form of the Landau-Vishkin string algorithm.

5.2.1.2. Inspiration: Landau-Vishkin Algorithm

In the following discussion, diagonal d corresponds to the the set of distances $\{stringdist(i,j) \mid i-j=d\}$. (The name diagonal comes from the distance array in the straightforward dynamic programming algorithm for string editing [U-83].)

The basic algorithm of [LV-86] can be modified to yield

for $p:=1$ to $|S_2|$ do

 for diagonals d between $-p$ and p inclusive pardo

  compute maximum row i in d such that $stringdist(i,i+d) \leq p$

  exit program when $stringdist(|S_1|,|S_2|)$ is computed

Here is the computation for a given diagonal at stage p.

(1) Find a row i in diagonal d with value p (consult diagonals $d-1$ and $d+1$ for this).

(2) Jump to $i+h$ if h is the maximum value such that $S_1[i..i+h]=S_2[i+d..i+d+h]$.

Both steps can be done in constant time, where step 2 uses a suffix tree. So the whole algorithm takes $O(k)$ time, where k is the actual distance between the two strings.

5.2.1.3. Problems in Applying this Approach to Trees

Problem 1: We would like to use suffix trees based on some traversal order, but a traversal order on labels alone is insufficient as figure 5.2a shows. On the other hand, it is well known [K-73] that any traversal (we use a left-to-right postorder traversal) in which each label is associated with the number of its children is sufficient to specify the tree. We will call that traversal SLR_T.

Problem 2: Identical traversals with children are not necessary. That is, $forestdist(i,j)=forestdist(i+h,j+h)$ is possible even though $SLR_{T_1}[i+1..i+h] \neq SLR_{T_2}[j+1..j+h]$. See figure 5.2b.

Problem 3: Identical traversals with children are not sufficient. That is, $forestdist(i,j)<forestdist(i+h,j+h)$ is possible even though $SLR_{T_1}[i+1..i+h]=SLR_{T_2}[j+1..j+h]$. See figure 5.2c.

So, what are these traversals good for? Well, if the single node labeled e in figure 5.2b or in 8c were replaced by a tree (or even forest) of size r, then in both cases $forestdist(3,2)=forestdist(3+r,2+r)$ and this would be discovered by establishing that $SLR_{T_1}[3+1..3+r]=SLR_{T_2}[2+1..2+r]$.

5.2.1.4. Lemmas to Achieve Second Idea

Here is a road map through the lemmas.

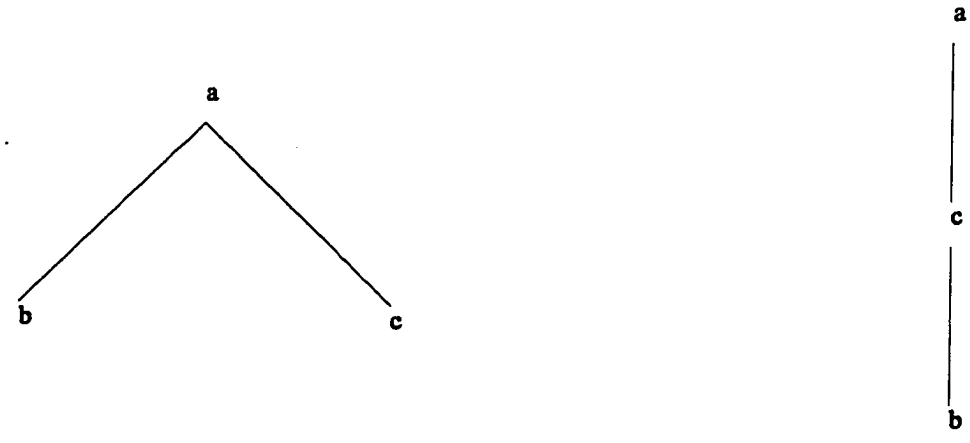
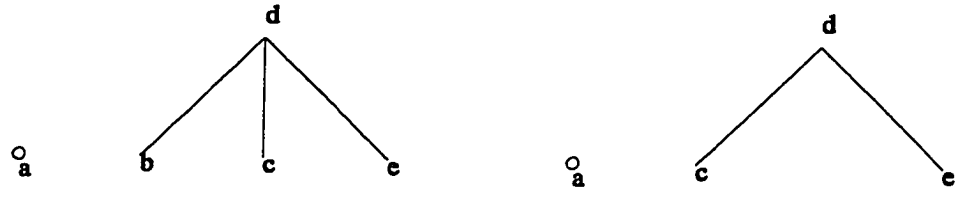
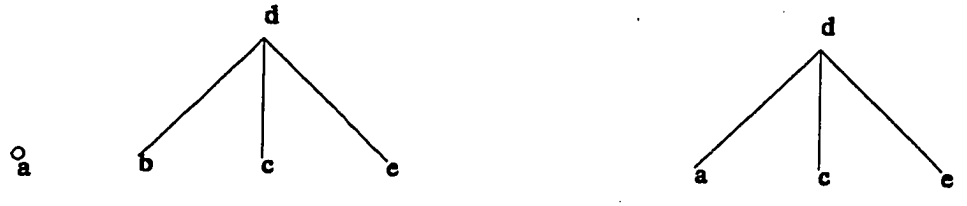


Figure 5.2a. Different Trees May have the Same Postorder Traversal



forestdist(3,2)=*forestdist*(5,4)

Figure 5.2b. Label with Number of Children Seems not Necessary



forestdist(3,2) ≠ *forestdist*(5,4)

Figure 5.2c. Label with Number of Children Seems not Sufficient

1. Lemma 5.3 shows that the dynamic programming array for the tree distance looks like the one for strings. In particular, the distance array of size $|T_1| \times |T_2|$ can be encoded into an array with only $O(k^2)$ elements if k is the distance between T_1 and T_2.

2. Lemmas 5.4 to 5.6 give assurance that using equal portions of two trees or forests in certain generic situations is not a mistake.

3. We then consider various traversal orderings of trees, where the nodes are represented by their label and the number of their children.

4. Lemmas 5.7 to 5.10 give conditions for using such encoded traversal orderings.

5. Finally, we define two predicates up and down, computable in constant time (assuming that certain data structures on individual trees have been created in a preprocessing step) that use the ideas of the previous points. Up and down are used in our algorithms.

5.2.2. Monotonicity

Lemma 5.3 (monotonicity): The distance values along the diagonal of any of the distance arrays are monotonically non-decreasing. That is, $forestdist(i-1,j-1) \leq forestdist(i,j)$, for $i,j \geq 1$.

Proof: By induction on $i+j$.

Base case: By definition of insert weight and delete weight, $forestdist(0,i) = forestdist(i,0) = i$, for any i. Therefore $forestdist(0,0) = 0$. So $forestdist(1,1) \geq forestdist(0,0)$.

Inductive step:

From our discussion of the basic step, there are three ways to handle $T_1[i]$ and $T_2[j]$ in the

computation of $forestdist(i,j)$:

$forestdist(i,j)=$

$$\min\begin{cases} forestdist(i-1,j)+1 \\ forestdist(i,j-1)+1 \\ dist(T_1[l(i)..i-1],T_2[l(j)..j-1])+forestdist((l(i)-1),(l(j)-1))+\gamma(T_1[i]\rightarrow T_2[j]) \end{cases}$$

The first and second terms are symmetric, so we address them by considering the case that $forestdist(i,j)=forestdist(i-1,j)+1$.

If $i=1$ then $forestdist(i,j) > forestdist(i-1,j)=forestdist(0,j)=j$ and $forestdist(i-1,j-1)=forestdist(0,j-1)=j-1$. So, $forestdist(i-1,j-1)\leq forestdist(i,j)$.

Otherwise by the distance calculation $forestdist(i-1,j-1)\leq forestdist(i-2,j-1)+1$. By induction hypothesis, $forestdist(i-2,j-1)+1\leq forestdist(i-1,j)+1$. So, by transitivity, $forestdist(i-1,j-1)\leq forestdist(i,j)$.

If the third term applies, observe that the sum $forestdist(l(i)..i-1,l(j)..j-1)+forestdist((l(i)-1),(l(j)-1)) \geq forestdist(i-1,j-1)$. Since $\gamma(T_1[i]\rightarrow T_2[j])$ is non-negative, we then have $forestdist(i-1,j-1)\leq forestdist(i,j)$. □

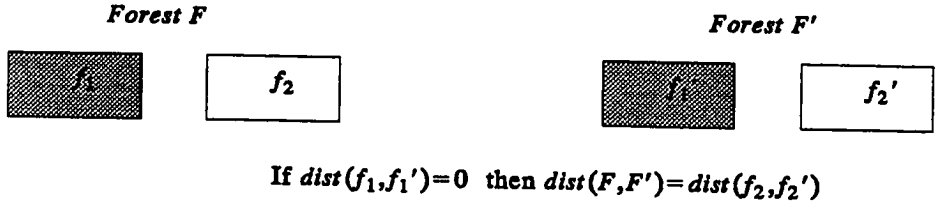
After some stage in the algorithm, we will have located the largest row values with distance value p in each diagonal. In the next stage, the algorithm finds the first row i in diagonal d with value $p+1$ and such that the neighboring positions in diagonals $d-1$ and $d+1$ have values $p+1$ or greater. That row in diagonal d corresponds to position (i,j) (where $j=i+d$). We then try to find the greatest q such that $forestdist(i,j)=forestdist(i+q,j+q)$.

By monotonicity, for each r, $1\leq r\leq q$, the values from diagonals $d-1$ and $d+1$ are irrelevant since $forestdist(i-1,j)\geq p+1$ and $forestdist(i,j-1)\geq p+1$. That means we only need to find a way to "continue" along diagonal d from position (i,j).

5.2.3. Proper Forests and Quarantined Subtrees

Definition: Given forest F consisting of the subtrees rooted at $t_1,\ldots,t_p$, we say that $F[l(t_i)..t_j]$ where $1 \le i \le j \le p$ is a *proper subforest* of F. Intuitively, a proper subforest of F is a consecutive subsequence of the trees of F.

Lemma 5.4 (one side proper subforest): Suppose $F[1..m]$ (F) and $F'[1..n]$ (F') are ordered forests, $F[1..x]$ (f_1) is a proper subforest of F and $F'[1..y]$ (f_1') is a proper subforest of F'.



If $dist(f_1,f_1')=0$ then $dist(F,F')=dist(f_2,f_2')$

If $dist(f_2,f_2')=0$ then $dist(F,F')=dist(f_1,f_1')$

If $dist(f_1,f_1')=0$ and $dist(f_3,f_3')=0$ then $dist(F,F')=dist(f_2,f_2')$

Figure 5.3 Proper Subforest Lemmas

(parenthetic labels correspond to those in Figure 5.3.)

(i) If $F[x+1..m]$ (f_2) and $F'[y+1..n]$ (f_2') are identical, then $dist(1..m,1..n)=dist(1..x,1..y)$.

(ii) If $F[1..x]$ and $F'[1..y]$ are identical, then $dist(1..m,1..n)=dist(x+1..m,y+1..n)$.

Proof: Let $left(F)=F[1..x]$ and $left(F')=F'[1..y]$. Let $right(F)=F[x+1..m]$ and $right(F')=F'[y+1..n]$.

(1) Since $right(F)$ and $right(F')$ are identical, $m-x=n-y$. From lemma 5.3 (monotonic), $dist(1..m,1..n)\geq dist(1..x,1..y)$.

(2) We construct a particular mapping between F and F' as follows: We map $left(F)$ to $left(F')$ using the best mapping between them. We map $right(F)$ to $right(F')$. Since $right(F)$ (respectively, $right(F')$) is a proper subforest of F (respectively, F'), this gives a mapping.

In the above mapping, $dist(1..m,1..n)\leq dist(1..x,1..y)+dist(x+1..m,y+1..n)$. Since $right(F)$ and $right(F')$ are identical, $dist(x+1..m,y+1..n)=0$. Hence, $dist(1..m,1..n)\leq dist(1..x,1..y)$.

Hence, $dist(1..m,1..n)=dist(1..x,1..y)$.

(ii) This proof is symmetric to (i). Just consider the left-right mirror images of F and F'. □

Lemma 5.5 (two sided proper subforest): Suppose $F[1..m]$ (F) and $F'[1..n]$ (F') are ordered forests, $F[1..x]$ (f_1) is a proper subforest of F, $F[x+1..y-1]$ (f_2) is a proper subforest of F, $F'[1..x]$ (f_1') is a proper subforest of F'. and $F'[x+1..z-1]$ (f_2') is a proper subforest of F'. If $F[1..x]$ is identical to $F'[1..x]$ and $F[y..m]$ (f_2) is identical to $F'[z..n]$ (f_2'). then $dist(1..m,1..n)=dist(x+1..y-1,x+1..z-1)$.

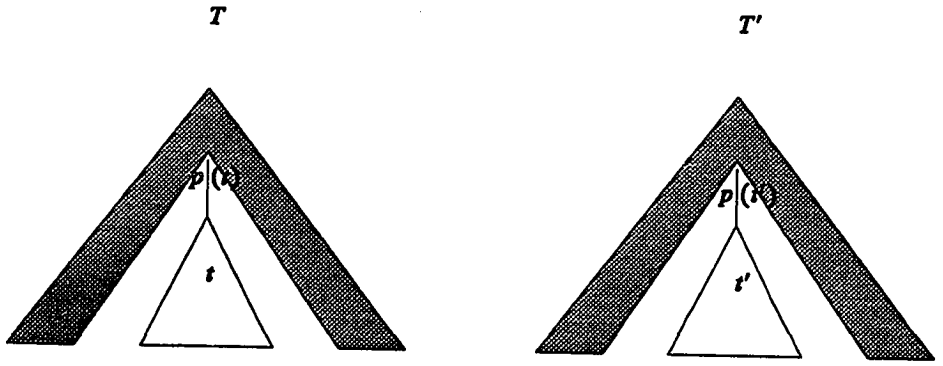
Proof: By (i) in lemma 5.4, $dist(1..m,1..n)=dist(1..y-1,1..z-1)$. By (ii) in lemma 5.4, $dist(1..m,1..n)=dist(1..y-1,1..z-1)=dist(x+1..y-1,x+1..z-1)$. □

We now propose the analogous property for trees.

Definition: Suppose $T_1[1..m]$ and $T_2[1..n]$ are ordered trees, $T_1[l(i)..i]$ ($tree_1(i)$) is a subtree of T_1 and $T_2[l(j)..j]$ ($tree_2(j)$) is a subtree of T_2. We say that *the only difference between* T_1 *and* T_2 *is between* $tree_1(i)$ *and* $tree_2(j)$ if replacing both $tree_1(i)$ and $tree_2(j)$ by a single node with the same label makes T_1 and T_2 identical.

Lemma 5.6 (Quarantined Subtree): If the only difference between T_1 and T_2 is between $tree_1(i)$ and $tree_2(j)$ then $dist(1..m,1..n)=dist(l(i)..i,l(j)..j)$. That is, $treedist(m,n)=treedist(i,j)$. (Figure 5.4.)

Proof: If $i=m$ and $j=n$, then the lemma is trivial. From the condition of the lemma, it is



If shaded part of the two trees are the same, then $dist(T,T')=dist(t,t')$

Figure 5.4 Quarantined Subtree Lemma

clear that $i=m$ and $j\neq n$ (or $i\neq m$ and $j=n$) is not possible. So we consider the case $i\neq m$ and $j\neq n$.

(1) Since only $tree_1(i)$ and $tree_2(j)$ differ, $m-i=n-j$. Hence from lemma 5.3 (monotonicity), we know $dist(1..m, 1..n)\geq dist(1..i, 1..j)$. Since $T_1[1..l(i)-1]$ and $T_2[1..l(j)-1]$ are identical and $T_1[l(i)..i]$ is a proper subforest of $T_1[1..i]$ and $T_2[l(j)..j]$ is a proper subforest of $T_2[1..j]$, from lemma 5.5, we know $forestdist(i,j)=dist(1..i, 1..j)=dist(l(i)..i,l(j)..j)$ $=treedist(i,j)$. Hence $forestdist(m,n)=dist(1..m, 1..n)\geq dist(l(i)..i,l(j)..j)=treedist(i,j)$.

(2) Recall that replacing both $T_1[l(i)..i]$ and $T_2[l(j)..j]$ by a single node with same label, causes T_1 and T_2 to be identical. So, we know that the following is a mapping for $T_1[1..m]$ and $T_2[1..n]$. Use the best mapping to map $T_1[l(i)..i]$ to $T_2[l(j)..j]$. Map the identical parts of T_1 and T_2 to each other. By construction, this does not violate the conditions of the mappings. Hence $forestdist(m,n)=dist(1..m, 1..n)\leq dist(l(i)..i,l(j)..j)=treedist(i,j)$.

Combining (1) and (2) gives the conclusion. □

5.2.4. Traversal Orderings and Continuation

For any tree, we consider four traversal orderings: left-to-right and right-to-left postorder and preorder. We also consider the suffix trees corresponding to these orders such that each element consists of a label of each node in order and the number of children of that node. (For tree T, these are called SLR_T and SRL_T for the postorder left-to-right and right-to-left traversals and $SLRP_T$ and $SRLP_T$ for the corresponding preorder traversals.) By default, we use the left-to-right post-order number as the identifier of each tree node. This ensures that our previous notation continues to make sense.

Given tree node i, then

$lr(i)=i$ is the left-to-right postorder number of node i.

$rl(i)$ is the right-to-left postorder number of node i.

$lrp(i)$ is the left-to-right preorder number of node i.

$rlp(i)$ is the right-to-left preorder number of node i.

To describe consecutive subsequences of nodes according to some traversal, we use the notation $T[traversaltype(number1..number2)]$, where the traversal type is $LR(left-to-right\ postorder)$, $RL(right-to-left\ postorder)$, $LRP(left-to-right\ preorder)$, or $RLP(right-to-left\ preorder)$. $Number1$ and $number2$ are numbers in the given traversal type. As a special case, we use $T[traversaltype(number)]$ to replace $T[traversaltype(number..number)]$.

Example: In figure 5.1, the nodes are numbered based on the left-to-right traversal order, however $T[LRP(3..7)]$ is the induced forest on the third through seventh nodes in the left-to-right preorder traversal. So, the forest consists of the nodes $T[1]$, $T[2]$, $T[8]$, $T[6]$, and $T[4]$. On the other hand $T[LRP(lrp(4)..lrp(7))]$ is the induced forest on the seventh ($lrp(4) = 7$) through the fifth ($lrp(8) = 5$) nodes in the left-to-right preorder traversal. This is empty. (Note that $lrp(4) = 7$, because node 4 in the left-to-right postorder traversal is the seventh node in the left-to-right preorder traversal.) End of example.

One notational convenience: $T[LR(lr(i)..lr(j))]$ is the induced forest in the left-to-right postorder traversal with numbers between i and j. That is, it is the induced forest on $T[i]$, $T[i+1]$, ..., $T[j]$. Since this ordering is the default, we sometimes write this as $T[i..j]$.

Fact: The following four representations uniquely represent a tree (see [K-73] vol. 1, p. 350).

1) The string derived from the left-to-right postorder of the tree such that element i of the

string contains both the label and the number of children of the node whose left-to-right postorder number is i. We use $SLR_T[1..n]$ to represent this string.

2) The analogous string derived from the right-to-left post order of the tree. We use $SRL_T[1..n]$ to represent this string. In this expression $SRL_T[i]$ is the node whose right-to-left postorder number is i.

3) The analogous string derived from the left-to-right preorder of the tree. We use $SLRP_T[1..n]$ to represent this string.

4) The analogous string derived from the right-to-left preorder of the tree. We use $SRLP_T[1..n]$ to represent this string.

Lemma 5.7 (post-order): Proposition Post-order: If $SLR_{T_1}[i..i+q]=SLR_{T_2}[j..j+q]$, then $T_1[i..i+q]$ and $T_2[j..j+q]$ are identical.

Proof: If $q=0$, the lemma is clearly true. If $nc(i+q'+1)=nc(j+q'+1)=0$, then $T_1[i+q'+1]$ and $T_2[j+q'+1]$ are leaves, so the conclusion is obvious. (Recall that $nc(i)$ is the number of children of i.) By post-order traversal, $T_1[i+q'+1]$ is the parent of the rightmost $nc(i+q'+1)$ trees in $T_1[i..i+q']$ (or all the trees in that ordered forest if there aren't that many in the forest). Similarly, for $T_2[j+q'+1]$. By induction hypothesis the trees from the two ordered forests must be identical. Conclusion follows. □

Fact (postpre correspondence): If tree T has n nodes, then left-to-right postorder node i is the same as right-to-left preorder node $n+1-i$. Therefore $T[LR(i..i+d)]=T[LRP(n+1-(i+d)..n+1-i)]$. A similar relationship holds between the right-to-left postorder and left-to-right preorder.

Lemma 5.8 (Pre-order): If in the pre-order traversal (left-to-right or right-to-left),

$SLRP_{T_1}[i..i+d]=SLRP_{T_2}[j..j+d]$ then the induced subgraphs $T_1[LRP(i..i+d)]=T_2[LRP(j..j+d)]$.

If $SLRP_{T_1}[i]\neq SLRP_{T_2}[j]$, then $tree_1(LRP(i))\neq tree_2(LRP(j))$.

Proof: By the postpre correspondence fact and lemma 5.7 (postorder). □

Lemma 5.9 (Must Include): Suppose $forestdist(i,j)=k$, any mapping between $T_1[1..i]$ and $T_2[1..j]$ with cost k must include (i,j) and $forestdist(i+q,j+q)=k$. Then

1. For s, $0\leq s\leq q$, $(i+s,j+s)$ must be in the mapping between $forest(i+q)$ and $forest(j+q)$ and $T_1[i+s]=T_2[j+s]$.

2. For any s such that $p^s(i) \leq i+q$, $(p^s(i),p^s(j))$ is in the mapping between $forest(i+q)$ and $forest(j+q)$ with cost k.

3. For any $q'\leq q$ if $T_1[i+q']$ is not an ancestor of $T_1[i]$, then $T_2[j+q']$ is not an ancestor of $T_2[j]$ and $nc(i+q')=nc(j+q')$.

4. If h, $h\leq q$, is the greatest value such that $T_1[i+h]$ is an ancestor of $T_1[i]$, then $T_1[i+h+1..i+q]$ is identical to $T_2[j+h+1..j+q]$.

Proof:

1. By the supposition, $forestdist(i-1,j)\geq k$ and $forestdist(i,j-1)\geq k$ (otherwise (i,j) would not have to be in the best mapping between $forest(i)$ and $forest(j)$). Therefore the lemma holds for $s = 0$.

Suppose the lemma holds up to some value r. By lemma 5.3 (monotonicity), $forestdist(i+r-1,j+r)\geq k$ and $forestdist(i+r,j+r-1)\geq k$. So, $(i+r+1,j+r+1)$ must be in the best mapping between $forest(i+r+1)$ and $forest(j+r+1)$ yielding distance k. If we

remove $(i+r+1,j+r+1)$ from that mapping we have a mapping between *forest*$(i+r)$ and *forest*$(j+r)$. The distance for that mapping is still k. By induction, for all s, $0 \leq s \leq r$, $(i+s,j+s)$ must be in this mapping. If $T_1[i+s] \neq T_2[j+s]$, then *forestdist*$(i+s,j+s) \geq k+1$.

Remark: Before proving 2 and 3, we first note that for any $q' \leq q$, $T_1[i+q']$ is an ancestor of $T_1[i]$ iff $T_2[j+q']$ is an ancestor of $T_2[j]$. This holds since from 1 we know that (i,j) and $(i+q',j+q')$ are in the best mapping so the conclusion follows from the ancestor condition on mappings.

Proof of 2: Suppose 2 is not true. Let t be the smallest value such that $p^t(\mathrm{i}) \leq i+q$ and $(p^t(\mathrm{i}), p^t(\mathrm{j}))$ is not in the best mapping. From the remark, we know that there is a $t'>t$ such that $(p^t(\mathrm{i}), p^{t'}(\mathrm{j}))$ is in the best mapping. Symmetrically, there must be a $t''>t$ such that $(p^{t''}(\mathrm{i}), p^t(\mathrm{j}))$ is in the best mapping. This violates the ancestor conditions on mappings.

Proof of 3: From the remark, we know that if $T_1[i+q']$ is not an ancestor of $T_1[i]$ then $T_2[j+q']$ is not an ancestor of $T_2[j]$. Suppose 3 is not true, let t be the smallest value, $t \leq q$, such that $T_1[i+t]$ is not an ancestor of $T_2[j]$, $T_2[j+t]$ is not an ancestor of $T_2[j]$, and $nc(i+t) \neq nc(j+t)$. Let $h<t$ be the greatest value such that $T_1[i+h]$ is an ancestor of $T_1[i]$. By definition of t and h, we know that $SLR_{T_1}[i+h+1..i+t-1]=SLR_{T_2}[j+h+1..j+t-1]$. By lemma 5.7 (postorder) we know that $T_1[i+h+1..i+t-1]$ is identical to $T_2[j+h+1..j+t-1]$. Since $(i+t,j+t)$ is in the best mapping and since neither $T_1[i+t]$ is an ancestor of $T_1[i]$ nor $T_2[j+t]$ is an ancestor of $T_2[j]$, $nc(i+t) \neq nc(j+t)$ implies that a child of one of them, say $T_1[i+t]$, maps to a non-child of the other. This violates the ancestor condition on mappings.

Proof of 4: By 1, 3 and lemma 5.7 (postorder). □

Lemma 5.9 show what $forestdist(i,j)=forestdist(i+q,j+q)$ implies. Lemma 5.10 shows how to obtain that condition.

Lemma 5.10 (continuation condition): Suppose $forestdist(i,j)=p$, any mapping between $T_1[1..i]$ and $T_2[1..j]$ with cost p must include (i,j), and $forestdist(i+q-1,j+q-1)=p$. Then $forestdist(i+q,j+q)=p$ if

1. $T_1[i+q]=T_2[j+q]$; and

2 a. $T_1[i+q]\in anc(T_1[i])$ and $T_2[j+q]\in anc(T_2[j])$ and $treedist(i+q,j+q) = p-forestdist(l(i+q)-1,l(j+q)-1)$.

or

2 b. $T_1[i+q]\notin anc(T_1[i])$ and $T_2[j+q]\notin anc(T_2[j])$ and $nc(i+q)=nc(j+q)$.

Proof: There are two possibilities. If case 1 and 2a hold, then $p=treedist(i+q,j+q)+forestdist(l(i+q)-1,l(j+q)-1)$. So, $forestdist(i+q,j+q)=p$ by definition of distance.

If 1 and 2b hold, then by lemma 5.9 (Must Include), point 4, (using the definition of h from that point), $T_1[i+h+1..i+q-1]$ is identical to $T_2[j+h+1..j+q-1]$. Therefore, we can use the distance p mapping from $forest(i+q-1)$ to $forest(j+q-1)$. By 2b, adding $(i+q,j+q)$ will preserve the ancestor condition on mappings. By 1, the addition won't increase the cost. □

If $T_1[i+q]\notin anc(T_1[i])$ and $T_2[j+q]\in anc(T_2[j])$ or $T_1[i+q]\in anc(T_1[i])$ and $T_2[j+q]\notin anc(T_2[j])$, then $(i+q,j+q)$ and (i,q) would still have to be in the mapping by the lemma 5.9 (Must Include) point 1, but this violates the ancestor condition on mappings. Contradiction. □

5.2.5. Up and Down

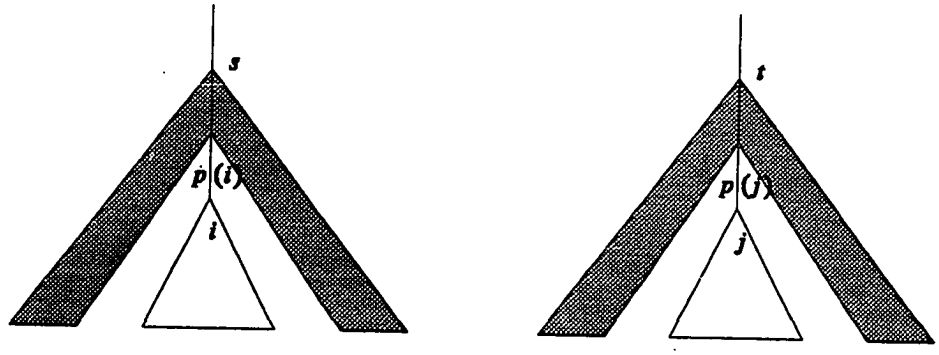
Definition: Given trees T_1 and T_2 and a pair of subtrees ($tree_1(i)$, $tree_2(j)$). Define $up\,(i,j)$ to be a pair of subtrees rooted at (s,t) satisfying the following (figure 5.5):

1) $tree_1(i)$ is a subtree of $tree_1(s)$ and $tree_2(j)$ is a subtree of $tree_2(t)$.

2) ($tree_1(s)$, $tree_2(t)$) is the largest subtree pair (equivalently s and t are the greatest ancestors of i and j) such that the only difference between them is between $tree_1(i)$ and $tree_2(j)$. That is, if $tree_1(i)$ and $tree_2(j)$ are both replaced by a node with the same label, then $tree_1(s)$ would be identical to $tree_2(t)$.

Notice that $up\,(i,j)$ is unique, because $level(s)-level(i)=level(t)-level(j)$, where $level(x)$ is the level of node x in the given tree.

1) Find r such that $SLR_{T_1}[i+1..i+r-1]=SLR_{T_2}[j+1..j+r-1]$ and



$up\,(i,j)=(s,t)$ shaded part of the two trees are the same

Figure 5.5 $up\,(i,j)$

$SLR_{T_1}[i+r] \neq SLR_{T_2}[j+r]$.

2) Find q such that $SRL_{T_1}[rl(i)+1..rl(i)+q-1] = SRL_{T_2}[rl(j)+1..rl(j)+q-1]$ and $SRL_{T_1}[rl(i)+q] \neq SRL_{T_2}[rl(j)+q]$.

3) Find the least common ancestor sr of $T_1[LR(lr(i))]$ $(= T_1[i])$ and $T_1[LR(lr(i+r))]$. Find the least common ancestor sl of $T_1[RL(rl(i))]$ $(= T_1[i])$ and $T_1[RL(rl(i)+q)]$.

4) Find the least common ancestor tr of $T_2[LR(lr(j))](=T_2[j])$ and $T_2[LR(lr(j+r)]$. Find the least common ancestor tl of $T_2[RL(rl(j))]$ $(=T_2[j])$ and $T_2[RL(rl(j)+q)]$.

5) Let s be the proper child of the lower of sl and sr with respect to i. Let t be the proper child of the lower of tl and tr with respect to j.

6) $up(i,j)=(s,t)$.

Lemma 5.11 (up): This algorithm correctly computes up.

Proof: By lemma 5.7 (postorder), point 1 implies that the induced subgraphs $T_1[LR(lr(i)+1..lr(i)+r-1)]$ and $T_2[LR(lr(j)+1..lr(j)+r-1)]$ are identical. However, since $SLR_{T_1}[i+r] \neq SLR_{T_2}[j+r]$, (sr,tr) cannot be $up(i,j)$, since there must be some difference between $tree_1(sr)$ and $tree_2(tr)$ besides the differences between $tree_1(i)$ and $tree_2(j)$. Now, let sr' be the proper child of sr with respect to i and tr' be the proper child of tr with respect to j. $sr' \leq i+r-1$ and $tr' \leq j+r-1$, so $T_1[i+1..sr']$ is identical to $T_2[j+1..tr']$. The same argument goes for the right-to-left postorder traversal. Define sl' and tl' similarly to sr' and tr'. Since s is just the lower of sr' and sl' and t is the lower of tr' and tl', $T_1[LR(lr(i)+1..lr(s))]$ is the same as $T_2[LR(lr(j)+1..lr(t))]$ and $T_1[RL(rl(i)+1..rl(s))]$ is the same as $T_2[RL(rl(j)+1..rl(t))]$. □

The function down is symmetric to up. The extra two arguments in its definition result from the fact that there is a unique root, but there are many leaves in a given tree.

Definition: Given trees T_1 and T_2 and subtree pair ($tree_1(i_1)$, $tree_2(j_1)$) which contains ($tree_1(i)$, $tree_2(j)$) respectively as subtrees, $down(i_1,j_1,i,j)$ is a pair of subtrees ($tree_1(y)$, $tree_2(z)$) satisfying the following (figure 5.6):

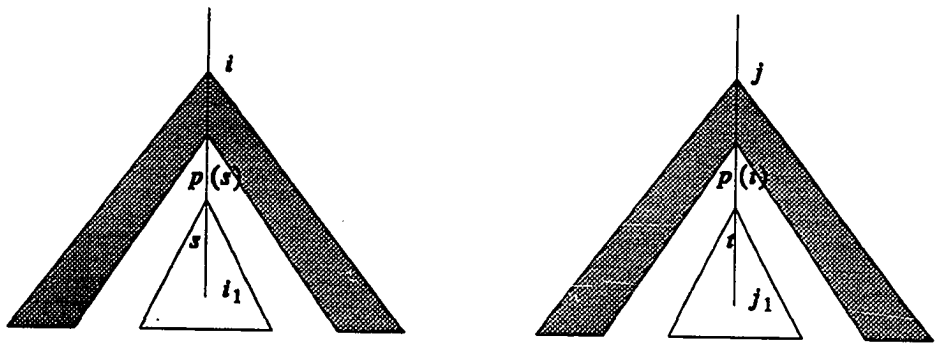
1) $tree_1(i)$ is a subtree of $tree_1(y)$ which is a subtree of $tree_1(i_1)$.
$tree_2(j)$ is a subtree of $tree_2(z)$ which is a subtree of $tree_2(j_1)$.

2) The pair (y,z) are the lowest ancestors of (i,j) such that ($tree_1(y)$, $tree_2(z)$) satisfies condition 1 and the only difference between $tree_1(i_1)$ and $tree_2(j_1)$ is between $tree_1(y)$ and $tree_2(z)$.

Just as for up, $down(i_1,j_1,i,j)$ is unique since $level(i_1)-level(y)=level(j_1)-level(z)$.

Here is how to find $down(i_1,j_1,i,j)$.

1) Find q such that $SLRP_{T_1}[lrp(i_1)..lrp(i_1)+q]=SLRP_{T_2}[lrp(j_1)..lrp(j_1)+q]$ and $SLRP_{T_1}[lrp(i_1)+q+1]\neq SLRP_{T_2}[lrp(j_1)+q+1]$. Let $q' := min(q+1, lrp(i) - lrp(i_1), lrp(j)$



$down(i,j,i_1,j_1)=(s,t)$ shaded part of the two trees are the same

Figure 5.6 $down(i,j,i_1,j_1)$

$- lrp(j_1))$.

2) Find r such that $SRLP_{T_1}[rlp(i_1)..rlp(i_1)+r] = SRLP_{T_2}[rlp(j_1)..rlp(j_1)+r]$ and $SRLP_{T_1}[rlp(i_1)+r+1] \neq SRLP_{T_2}[rlp(j_1)+r+1]$. Let $r' := min(r+1, rlp(i) - rlp(i_1), rlp(j) - rlp(j_1))$.

3) Let $sl :=$ the least common ancestor of $T_1[LR(lr(i))] (= T1[i])$ and $T_1[LRP(lrp(i_1)+q')]$. Let $sr :=$ the least common ancestor of $T_1[i]$ and $T_1[RLP(rlp(i_1)+r')]$.

4) Let $tr :=$ the least common ancestor of $T_2[j]$ and $T_2[LRP(lrp(j_1)+q')]$. Let $tl :=$ the least common ancestor of $T_2[j]$ and $T_2[RLP(rlp(j_1)+r')]$.

5) Let y be the highest of sl and sr. Let z be the highest of tl and tr.

6) $down(i_1, j_1, i, j) = (y, z)$.

Lemma 5.12 (*down*): The algorithm correctly computes down.

Proof of down: After step 1, we know that $T_1[LRP(lrp(i_1)..lrp(i_1)+q')] = T_2[LRP(lrp(j_1)..lrp(j_1)+q')]$. After step 2, we know that $T_1[RLP(rlp(i_1)..rlp(i_1)+r')] = T_2[RLP(rlp(j_1)..rlp(j_1)+r')]$.

Since $lrp(sl) \leq lrp(i_1)+q'+1$, we know that $T_1[LRP(lrp(i_1)..lrp(sl)-1)]$ is identical to $T_2[LRP(lrp(j_1)..lrp(j_1)+(lrp(sl)-(1+lrp(i_1))))]$. Similarly, we know that $T_1[RLP(rlp(i_1)..rlp(sr)-1)]$ is identical to $T_2[RLP(rlp(j_1)..rlp(j_1)+(rlp(sr)-(1+rlp(i_1))))]$.

Notice that $T_1[LRP(lrp(i_1)..lrp(i_1)+q')]$ maps to a portion of T_2 all of whose nodes are either on the path from j to j_1 or to the left of that path. Also, $lrp(sl)-1 \leq lrp(i_1)+q'$. So, $T_1[LRP(lrp(i_1)..lrp(sl)-1)]$ maps to nodes that are either on the path from j to j1 or to the left of that path in T_2. Similarly, $T_1[RLP(rlp(i_1)..rlp(sr)-1)]$ maps to nodes that are either on the path from j to j_1 or to the right of that path in T_2.

Since s is the least common ancestor of sl and sr, there must be two nodes tl' and tr' such

that

$T_1[LRP(lrp(i_1)..lrp(s)-1)]$ is identical to $T_2[LRP(lrp(j_1)..lrp(tl')-1)]$ and $T_1[RLP(rlp(i_1)..rlp(s)-1)]$ is identical to $T_2[RLP(rlp(j_1)..rlp(tr')-1)]$ (Remark: this implies that $level(i_1)-level(s)= level(j_1)-level(tl') =level(j_1)-level(tr')$.)

Claim: $T_2[tl']=T_2[tr']$.

Proof: Suppose not. Let x be the least common ancestor of tl' and tr' in tree T_2. By construction tl' is on or to the left of the path from j to j_1 and tr' is on or to the right, so x must be on the path from j to j_1. Because of the remark before the claim, $level(tl')=level(tr')$ so tl' and tr' must not be related by the ancestor relationship. This implies that x must be distinct from both of them.

Therefore $T_2[x]$ maps to a node in $T_1[LRP(irp(i_1)..lrp(s)-1)]$ and in $T_1[RLP(rlp(i_1)..rlp(s)-1)]$. The node must be the same node $T_1[y]$. (Reason: $level(j_1)-leve.(xt)=level(i_1)-level(y)$.) Let xl' be the proper child of x with respect to tl' and xr' be the proper child of x with respect to tr'. Let y' be the proper child of y with respect to i. Then, in the left-to-right preorder traversal y' maps to xl' and in the right-to-left preorder traversal, y' maps to xr'. There have to be the same number of left siblings of xl' as of y' and the same number of right siblings of xr' and y', therefore x and y would have different numbers of children. □

end of proof of claim.

Now, tl' (= tr') must be an ancestor (perhaps improper) of t, by the definition of t. By a symmetric construction, we can define sl' and sr' in tree T_1. It must be that $sl' = sr'$ and that sl' is an ancestor of s. By the remark before the claim, $level(i_1)-level(s)=level(j_1)-level(tl') \leq level(j_1)-level(t)=level(i_1)-level(sl')\leq level(i_1)$

$-level(s)$. So, $s=sl'$ and $t=tl'$. Conclusion follows. □

5.2.6. Preprocessing

Lemma 5.13 With $O(n)$ sequential preprocessing time and $O(logn)$ parallel time (using an optimal speedup algorithm), we can construct the following from a set of parent-child and sibling edges:

0. All traversal orderings, the number of children of each node and $l(i)$ [TV85].

1. A suffix tree for a string [LSV87]. In our case the string will be two postorder and two preorder traversals (left-to-right and right-to-left in each case) with the numbers of children associated with each node.

2. A data structure for constant-time computation of the least common ancestor of any two nodes in a tree [SV88].

3. A data structure for constant-time computation of the proper child of any node n with respect to descendant m (by a similar method to [SV88]). □

Corollary: Up and down can be computed in constant time using these suffix trees and data structures. □

5.3. Algorithms

5.3.1. Encoding of Distance Array

In these algorithms, we don't maintain $forestdist(i,j)$ for all i, j. Instead, we encode these values using the f-array. $f(d,p)$ is the row number r such that the intersection between diag-

onal d and row $r+1$ holds a number that is greater than p, but row r holds a number that is less than or equal to p in the distance array. Diagonal d in the distance array is the set of values *forestdist*(i,j) such that $i-j=d$. (Every different subtree pair that is evaluated has an analogous construct, called its g-array, but for notational simplicity, we will only discuss the f-array concerned with the main tree pair, T_1 and T_2.)

Suppose we want to determine *forestdist*(i,j). We proceed by binary search as follows:

Let $d=i-j$. Perform a binary search in row d to find the smallest r such that $i \leq f(d,r)$. Then *forestdist*$(i,j)=f(d,r)$.

Since the f-array can never have more than k columns when evaluating (T_1,T_2,k), the time to access this array is $O(\log k)$ time.

5.3.2. One Stage of the Algorithm

The two algorithms we are about to introduce execute in parallel to evaluate (T_1,T_2,p) for some distance value p. These are optimal speedup parallel algorithms, which is why we don't present their sequential counterparts.

Both algorithms consist of stages. At each stage k, the algorithm determines for each diagonal d, the maximum i such that *forestdist*$(i,i+d) \leq k$. We describe here the three parts that make up the kth stage along one diagonal d. See figure 5.7.

Part I finds an i and $j=i+d$ such that *forestdist*$(i,j)=k$ and (i,j) must be in any mapping where this is true. If no such i exists then stage k is over for this diagonal. Part II determines the maximum ancestors $T_1[i+q]$ (of $T_1[i]$) and $T_2[j+q]$ (of $T_2[j]$) such that *forestdist*$(i+q,j+q)=k$. Part III then determines the maximum h such that *forestdist*$(i+h,j+h)=k$, using a left-to-right postorder suffix tree. The algorithms only

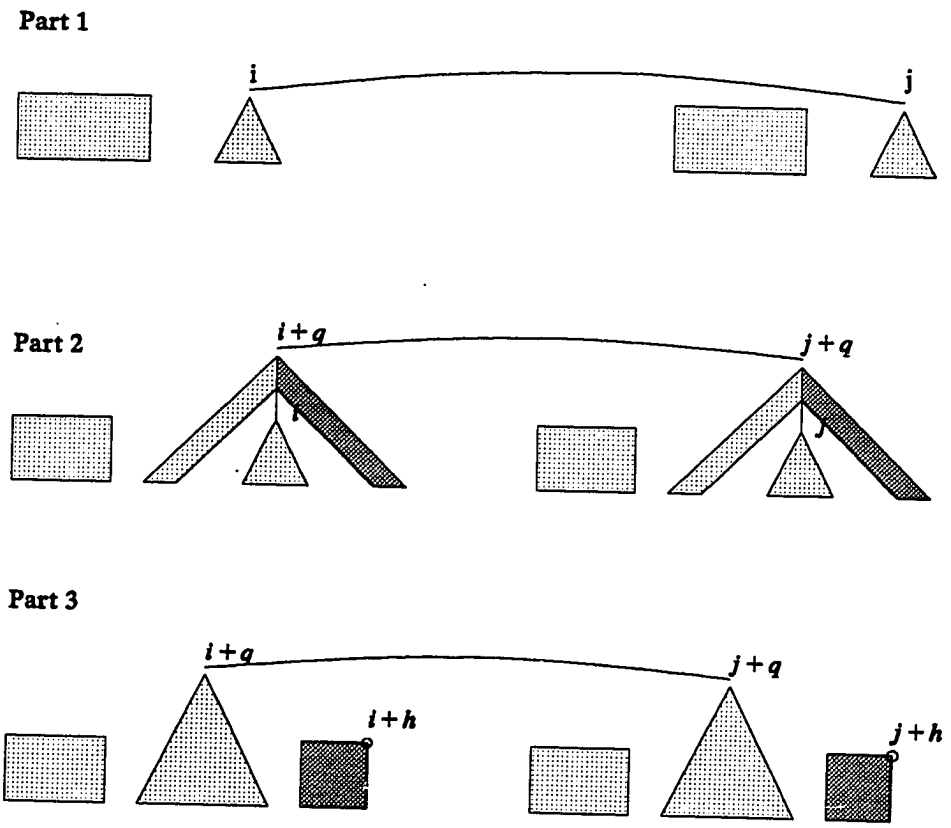


Figure 5.7 Three Parts to Basic Jump

differ in their implementation of part II.

When this stage begins $f(d',k-1)$ is known for $-(k-1)\leq d'\leq k-1$. Also, for any tree pair i, j such that $|i-j|\leq p$ (the relevant ones), we know the analogous quantity $g(i,j,d,h)$. That is, the maximum row value r such that $dist(l(i)..r,l(j)..r+d-(l(i)-l(j)))\leq h$, where $h=min(k-1,allow(p,i,j))$. For convenience, we only consider the step in computing the jump in the main distance array, i.e. the temporary array constructed when computing the distance between all of T_1 and all of T_2. Before presenting the algorithms, we explain the check if construct and the unknown test.

Remark 1. The algorithm makes use of statements of the form

check if $treedist(x,y)=k-forestdist(l(x)-1,l(y)-1)$.

In these cases, we will know that $forestdist(l(x)-1,l(y)-1)>0$. Since the algorithm is in stage k, either $treedist(x,y)$ will be unknown (and therefore $\geq k$) rendering the expression false or $treedist(x,y)$ will be known and the expression can be evaluated directly.

Remark 2. The algorithm also makes use of statements of the form

If $forestdist(l(t+1)-1,l(t+1+d)-1)$ is unknown so far,

This implies that $forestdist(l(t+1)-1,l(t+1+d)-1)\geq k$. If, in addition, $forestdist(t,t+d)=k-1$, then $(l(t+1)-1,l(t+1+d)-1)$ is not in diagonal d by lemma 5.3 (monotonicity). (Otherwise, $forestdist(l(t+1)-1,l(t+1+d)-1)\leq k$.) So, $t-l(t+1)\neq t+d-l(t+1+d)$. Hence, $treedist(t+1,t+d+1)>0$. Therefore, $treedist(t+1,t+d+1)+forestdist(l(t+1)-1,l(t+1+d)-1)>k$.

For a given k and each diagonal d, do the following:

Part I. Find (i,j) in diagonal d such that $forestdist(i,j)=k$ and (i,j) must be in the best mapping between $forest(i)$ and $forest(j)$. (Figure 5.8.)

step1. From the three diagonals, find the initial value.

$t:=max(f(d,k-1),f(d+1,k-1)+1,f(d-1,k-1))$

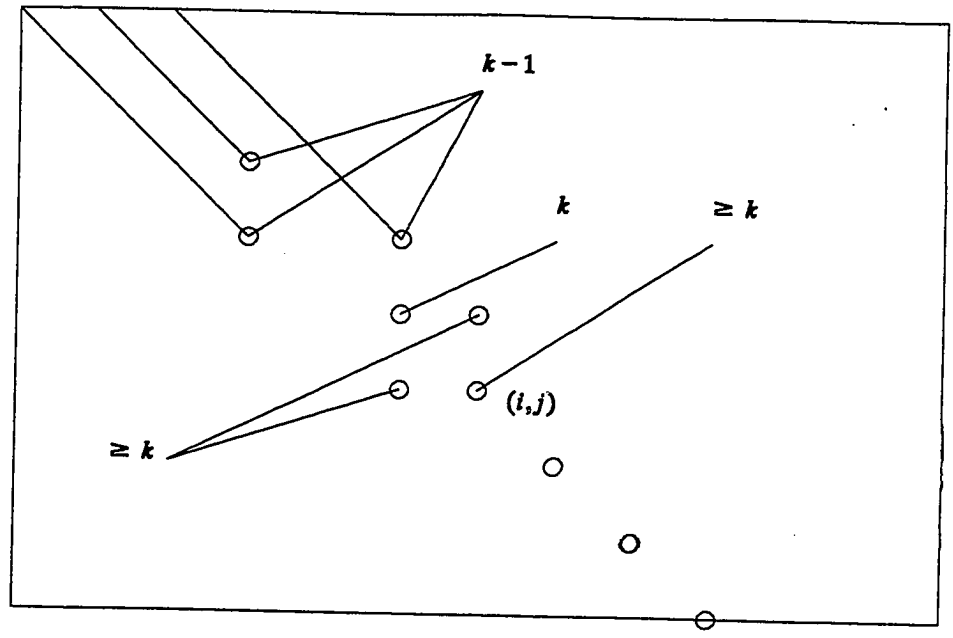
step2. We know that $forestdist(t,t+d)\leq k$. Check if $(t+1,t+d+1)$ is in the best mapping and $forestdist(t+1,t+d+1)=k$.

2a. if $forestdist(l(t+1)-1,l(t+1+d)-1)$ is unknown, $f(d,k)=t$

2b. if $forestdist(l(t+1)-1,l(t+1+d)-1)>0$,

check if $treedist(t+1,t+1+d)=k-forestdist(l(t+1)-1,l(t+1+d)-1)$.

if not, then $f(d,k)=t$ so exit parts I, II and III



forestdist$(i,j)=k$ iff the best mapping is of the following form

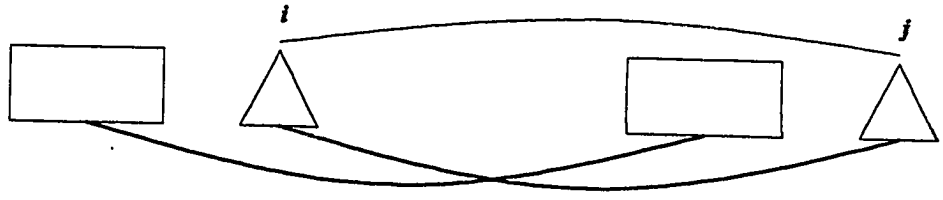


Figure 5.8 Part One of Three Basic Parts to Jump

if so, then *forestdist*$(t+1,t+1+d)=k$ and $(t+1,t+1+d)$ must be in best mapping

2c. if *forestdist*$(l(t+1)-1,l(t+1+d)-1)=0$,

then *forestdist*$(l(t+1)..t,l(t+1+d)..t+d)=$*forestdist*$(t,t+d)$ by lemma 5.5 (proper subforest).

Therefore *forestdist*$(t+1,t+d+1)=$*treedist*$(t+1,t+d+1)$.

if $forestdist(t,t+d)=k-1$ or $forestdist(t,t+d)=k$ and $T_1[t+1]=T_2[t+1+d]$
then $forestdist(t+1,t+1+d)=k$ and $(t+1,t+1+d)$ must be in best mapping
else $f(d,k)=t$ so exit parts I, II and III

Part II. Continue from $(t+1,t+1+d)$, provided $forestdist(t+1,t+d+1)=k$. (For notational convenience, $i=t+1$ and the $j=t+d+1$.) In this step, we find the largest r such that $p^r(i)$ and $p^r(j)$ are in diagonal d and $forestdist(p^r(i),p^r(j))=k$. Note that $forestdist(p^r(i),p^r(j))=k$ if and only if $forestdist(l(p^r(i))-1,l(p^r(j))-1))+treedist(p^r(i),p^r(j))=k$ by lemma 5.9 (Must Include). There are two methods for this part: binary search or bottom up search.

Method one. Binary search. The difficult ideas of this method are in the procedure down-probe. The outer loop does a binary search among the ancestors of (i,j) in diagonal $i-j$ and calls down-probe to see if a given pair of such ancestors (i_1,j_1) has the property that $forestdist(i_1,j_1)=k$. If so, then the outer loop restarts the search from (i_1,j_1).

Outer loop for one diagonal at stage k

$high_1 := |T_1|$

$high_2 := |T_2|$

(In the calculation of the distances between subtrees of T_1 and T_2, $high_1$ and $high_2$ are set to the roots of those subtrees.)

Let $mindiff := \min(high_1-i, high_2-j)$

loop

$x := i+mindiff/2$, rounding up.

$i_1 := lca.(x,i)$;

$j_1 := j+(i_1-i)$;

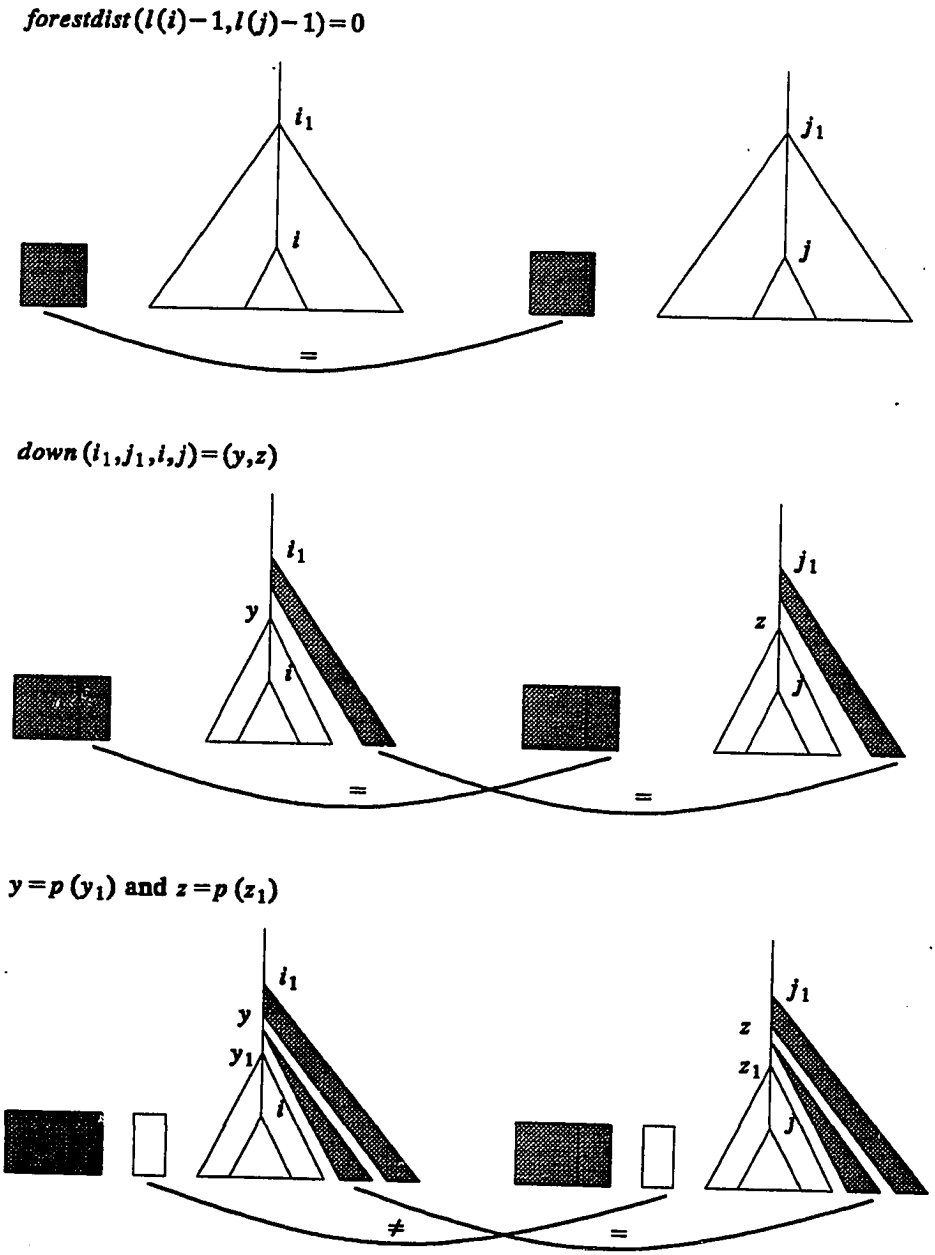


Figure 5.9 Hard case for Binary Search Approach

(So, (i_1,j_1) is in the same diagonal as (i,j).)

if j_1 is not an ancestor of j

then $forestdist(i_1,j_1)>k$ by lemma 5.9 (Must Include) and ancestor condition on mappings

else use the $Down-Probe(i,j,i_1,j_1)$ algorithm to determine whether $forestdist(i_1,j_1)=k$

end if

if $forestdist(i_1,j_1)=k$

then $i:=i_1; j:=j_1;$

else $high_1:=x$

$mindiff:=high_1-i;$

end if

exit if $high_1=i$ or $high_1=i+1$ through two iterations in this loop

end loop

Each time through the loop we halve the difference $high_1-i$. So, at most, we require $\log N$ calls to down-probe for any diagonal at this part.

Function $down-probe(i,j,i_1,j_1)$

Given (i_1,j_1) in diagonal d such that i_1 is ancestor of i, j_1 is ancestor of j and $level(i_1)-level(i)=level(j_1)-level(j)$, this function determines whether $forestdist(i_1,j_1)=k$.

Recall that (i_1,j_1) must be in any mapping such that $forestdist(i_1,j_1)=k$, since (i,j) must be in any such mapping and by lemma 5.9 (Must Include). Therefore $forestdist(i_1,j_1)=$ $forestdist(l(i_1)-1,l(j_1)-1)+treedist(i_1,j_1)$.

case 1. If $forestdist(l(i_1)-1,l(j_1)-1)$ is unknown, then $forestdist(i_1,j_1)>k$. (See remark 2.)

case 2. If $forestdist(l(i_1)-1,l(j_1)-1)>0$ then

check if $treedist(i_1,j_1)=k-forestdist(l(i_1)-1,l(j_1)-1)$.

if not, then $forestdist(i_1,j_1)>k$

if so, then $forestdist(i_1,j_1)=k$

case 3. If $forestdist(l(i_1)-1,l(j_1)-1)=0$ then find $down(i_1,j_1,i,j)=(y,z)$

3a. if $y=i$ and $z=j$

then $forestdist(y,z)=k$ and $forestdist(i_1,j_1)=k$

3b. if $y\neq i$ and $z\neq j$

find the proper child y_1 of y and the proper child z_1 of z.

use the left-to-right post-order suffix tree to see if

$SLR_{T_1}[y_1+1..y-1]=SLR_{T_2}[z_1+1..z-1]$ and $T_1[y]=T_2[z]$.

1. If not, then $forestdist(y,z)>k$ so $forestdist(i_1,j_1)>k$.

(By definition of down, (i_1,j_1) and (y,z) are in the same diagonal. So, by lemma 5.3 (monotonicity), $forestdist(y,z)>k$ implies that $forestdist(i_1,j_1)>k$.)

2. Suppose so. That is, $SLR_{T_1}[y_1+1..y-1]=SLR_{T_2}[z_1+1..z-1]$ and $T_1[y]=T2[z]$.

if $forestdist(l(y_1)-1,l(z_1)-1)$ is unknown

then $forestdist(y_1,z_1)>k$ so $forestdist(i_1,j_1)>k$ by the monotonicity lemma.

else (we know that $forestdist(l(y_1)-1,l(z_1)-1)>0$ by definition of down.)

check if $treedist(y_1,z_1)=k-forestdist(l(y_1)-1,l(z_1)-1)$.

if not then $forestdist(y_1,z_1)>k$, so $forestdist(i_1,j_1)>k$

if so, then $forestdist(y_1,z_1)=k$ by definition of distance

so, $forestdist(y,z)=k$ by the left-to-right postorder traversal

so, $forestdist(i_1,j_1)=k$ by definition of down and the quarantined subtree lemma.

The hard case is the very last one (depicted in figure 5.9). In that case,

0) $forestdist(l(i_1)-1,l(j_1)-1)=0$, $forestdist(i,j)=k$ and (i,j) must be in the mapping with

cost k from $forest(i)$ to $forest(j)$.

1) $down(i_1,j_1,i,j)=(y,z)$.

2) y_1 is the proper child of y with respect to i.

3) z_1 is the proper child of z with respect to j.

4) $T_1[y_1+1..y]=T_2[z_1+1..z]$.

then either $forestdist(y_1,z_1)>k$ or $forestdist(i_1,j_1)=k$.

Proof: By condition 4, $y-y_1=z-z_1$. So, by condition 1, (y,z) and (y_1,z_1) are in diagonal d. By lemma 5.9 (Must Include), $forestdist(y_1,z_1)=k$ implies that $forestdist(y_1,z_1)=treedist(y_1,z_1)+forestdist(l(y_1)-1,l(z_1)-1)=k$. If this sum is greater than k, then $forestdist(y_1,z_1)>k$.

Otherwise, we can assume $forestdist(y_1,z_1)=k$. From 4 and 1, we know that there is a non-zero distance between $T_1[l(y)..l(y_1)-1]$ and $T_2[l(z)..l(z_1)-1]$. By 4, (y_1,z_1) is in the same diagonal as (y,z), which in turn is in the same diagonal as (i_1,j_1) by condition 1. Now, $forestdist(l(y)-1,l(z)-1)=0$ by conditions 0 and 1. So, by lemma 5.5 (proper subforest), $forestdist(y,z)=treedist(y,z)$. This implies, by the quarantined subtree lemma, that $forestdist(i_1,j_1)=treedist(y,z)$.

By conditions 0 and 4, $treedist(y,z)=treedist(y_1,z_1)+dist(l(y)..l(y_1)-1,l(z)..l(z_1)-1)$ $=treedist(y_1,z_1)+forestdist(l(y_1)-1,l(z_1)-1)$ by lemma 5.5 (proper subforest). This is equal to $forestdist(y_1,z_1)$ since (y_1,z_1) must be in the mapping by lemma 5.9 (Must Include). So, if $k=forestdist(y_1,z_1)$ then $treedist(y,z)=forestdist(i_1,j_1)=k$ as well. □

Method two. Repeated hops. Given (i_1,j_1) in diagonal d such that i_1 is ancestor of i, j_1 is ancestor of j and $level(i_1)-level(i)=level(j_1)-level(j)$, we want to determine whether $forestdist(i_1,j_1)=k$. This is a bottom-up approach in which the loop may be repeated k times for a given diagonal at stage k as we show in lemma 5.15.

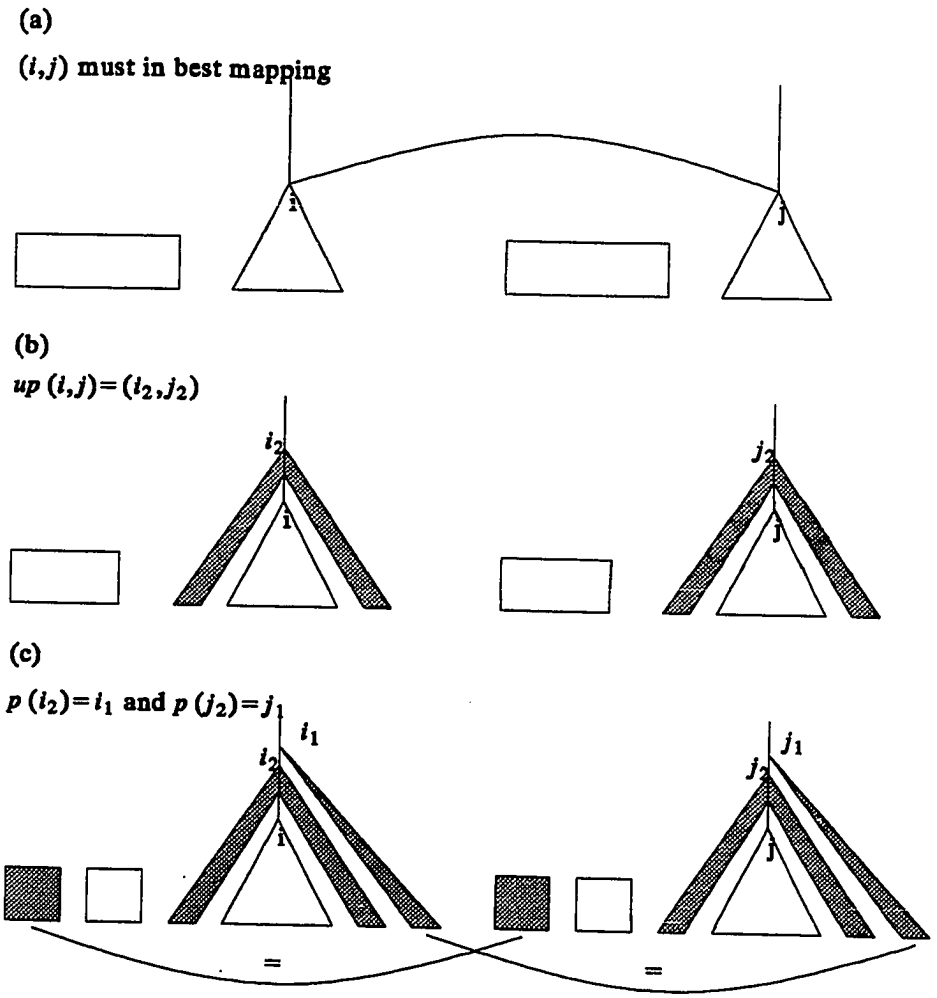


Figure 5.10 Hard Case for Bottom-up Approach

As in method one, by lemma 5.9 (Must Include), (i_1,j_1) must be in any mapping such that $forestdist(i_1,j_1)=k$. So, $forestdist(i_1,j_1)= forestdist(l(i_1)-1,l(j_1)-1)+treedist(i_1,j_1)$.

When we say $return(x,y)$ in this method, we mean that (x,y) are the ancestors of (i,j) that Part II should return, so it implies an exit of the loop.

loop

find $up(i,j)=(i_2,j_2)$. from the lemma 5.5 (proper subforest) and lemma 5.10 (continuation condition), we know that $forestdist(i_2,j_2)=k$.

Let $i_1=p(i_2)$ and $j_1=p(j_2)$.

Use the left-to-right post-order suffix tree to see if

$SLR_{T_1}[i_2+1..i_1-1]=SLR_{T_2}[j_2+1..j_1-1]$ and $T_1[i_1]=T2[j_1]$.

1. If not, then $forestdist(i_1,j_1)>k$ so $return(i_2,j_2)$

2. If so, check

2a. If $forestdist(l(i_1)-1,l(j_1)-1)$ is unknown, then

$forestdist(i_1,j_1)>k$ so $return(i_2,j_2)$.

2b. If $forestdist(l(i_1)-1,l(j_1)-1)>0$.

check if $treedist(i_1,j_1)=k-forestdist(l(i_1)-1,l(j_1)-1)$.

if not, then $forestdist(i_1,j_1)>k$ so $return(i_2,j_2)$

if so, then $forestdist(i_1,j_1)=k$, $i:=i_1$, $j:=j_1$

2c. If $forestdist(l(i_1)-1,l(j_1)-1)=0$ then

$forestdist(i_1,j_1)=k$, $i:=i_1$, $j:=j_1$.

end loop

The last case (2c) is the hardest.

Lemma 5.14: If

0. $up(i,j)=(i_2,j_2)$, $i_1=p(i_2)$, $j_1=p(j_2)$.

1. $forestdist(i,j)=k$ and (i,j) must be in the mapping with cost k from $forest(i)$ to $forest(j)$.

2. $SLR_{T_1}[i_2+1..i_1-1]=SLR_{T_2}[j_2+1..j_1-1]$ and $T_1[i_1]=T2[j_1]$.

3. $forestdist(l(i_1)-1,l(j_1)-1)=0$.

then $forestdist(i_1,j_1)=k$ (see figure 5.10).

Proof: By lemma 5.6 (quarantined subtree) and condition 0, $treedist(i_2,j_2)=treedist(i,j)$. By

condition 1, $k = forestdist(i,j) = treedist(i,j) + forestdist(l(i)-1, l(j)-1)$. By lemma 5.5 (proper subforest) and condition 0, $forestdist(l(i)-1, l(j)-1) = forestdist(l(i_2)-1, l(j_2)-1)$. Putting this together, $k = treedist(i_2,j_2) + forestdist(l(i_2)-1, l(j_2)-1)$. So, $forestdist(i_2,j_2) = k$.

By condition 3 and lemma 5.5 (proper subforest), $forestdist(l(i_2)-1, l(j_2)-1) = dist(l(i_1)..l(i_2)-1, l(j_1)..l(j_2)-1)$. By condition 2, $treedist(i_1,j_1) = treedist(i_2,j_2) + dist(l(i_1)..l(i_2)-1, l(j_1)..l(j_2)-1)$. This sums to k by the argument of the last paragraph. By condition 3 and the lemma 5.5 (proper subforest), this implies that $forestdist(i_1,j_1) = k$. □

Part III. At this point we have found the largest r such that $p^r(i)$ and $p^r(j)$ are in diagonal d and $forestdist(p^r(i), p^r(j)) = k$. Now, we use a left-to-right suffix tree to find the $f(d,k)$.

Let $i = p^r(i)$ and $j = p^r(j)$. Let h be such that $SLR_{T_1}[i..i+h] = SLR_{T_2}[j..j+h]$ and $SLR_{T_1}[i+h] \neq SLR_{T_2}[j+h]$. So, $f(d,k) = i+h$.

Parts I and III take constant time. So, the only question is how long part II takes. In the case of the first method, the $\log_2 N$ probes is shown with the algorithm for the outer loop. Part II of the second method is still at issue.

Lemma 5.15: The loop of part II of the second method will be repeated at most k times.

Proof: We will show that $treedist(i,j)$ will be increased by at least one in the course of each execution in which $forestdist(i_1,j_1) = k$. Since $up(i,j) = (i_2,j_2)$, we know that $treedist(i_2,j_2) = treedist(i,j)$ by the lemma 5.6 (quarantined subtree). Since $i_1 = p(i_2)$ and $j_1 = p(i_2)$, and $SLR_{T_1}[i_2+1..i_1-1] = SLR_{T_2}[j_2+1..j_1-1]$ and $T_1[i_1] = T_2[j_1]$, we know that $dist(l(i_1)..l(i_2)-1, l(j_1)..l(j_2)-1) > 0$. By lemma 5.9 (Must Include), (i_2,j_2) and (i_1,j_1) must both be in the mapping between $forest(i_1)$ and $forest(j_1)$, therefore $T_1[l(i_1)..l(i_2)-1]$ is mapped to $T_2[l(j_1)..l(j_2)-1]$. So, $treedist(i_1,j_1) \geq dist(l(i_1)..l(i_2)-1, l(j_1)..l(j_2)-1)$

$+ treedist(i_2,j_2) > treedist(i,j)$ □

5.4. Overall Resource Analysis of Algorithms 2 and 3

The algorithm has the following structure:

for $s := 0$ to $\log_2 N$

if $(T_1,T_2,2^s)$ is answered affirmatively then exit and report distance

end for

So, at any given stage in this algorithm, we are asking a question of the form (T_1,T_2,p), where p is the number of differences.

Proposition Resource: Suppose a (T_1,T_2,p) requires $T(p)$ resources (time, space, processors) to answer, where $T(p)/p$ is a nondecreasing function. Then if the final distance is k, then the algorithm requires $O(T(2k))$ resources.

Proof: Suppose $2^{c-1} < k \leq 2^c$. Then the total time for this algorithm is $\sum_{s=0}^{c} T(2^s) \leq \sum_{s=0}^{c} (1/2^s)T(2^c)$, since $T(2^s) \leq 2^s \times T(2^c)/2^c$. This sum is bounded from above by $2 \times T(2^c)$. This shows that the algorithm requires $O(T(2k))$ resources. □

Fact: For any of the complexity measures we derive $T(2k) = cT(k)$ (though the constant c differs). So, we simply address the problem of computing (T_1,T_2,k).

Time complexity The two algorithms can be characterized as follows for the problem of computing (T_1,T_2,k). The binary search algorithm requires at most $O(log\ N)$ probes. Each one requires $O(\log k)$ time to determine forest distances from the f-array. For each diagonal this must be done at most k times. So, the total time is $k \log k \log N$.

The repeated hop algorithm of method two requires at most k probes each time the distance

is increased by lemma 5.15. This happens k times along the diagonal, so the total time is $k^2 \log k$. Thus, method two should be used if $k \leq \log N$.

For purposes of processor and space complexity, we must count the tree distances that are relevant. For a given k, we only need to compute $(tree_1(i), tree_2(j), allow(k,i,j))$ such that $allow(k,i,j) \geq 0$ and such that $|i-j| \leq k$.

Here is why. When computing $(tree_1(i), tree_2(j), h)$, the algorithm only calls $(tree_1(i'), trees2(j'), h')$ if the latter is relevant to the former. Therefore the only treedistance triples ever called are transitively relevant to (T_1, T_2, k). Hence the only triples called are of the form $(tree_1(i), tree_2(j), h)$ where $h < allow(k,i,j)$, $allow(k,i,j) \geq 0$ and such that $|i-j| \leq k$. Since $(tree_1(i), tree_2(j), h)$ is covered by $(tree_1(i), tree_2(j), allow(k,i,j))$, the above is enough.

Processor complexity: There are $O(k \times N)$ relevant trees distances. In each tree only $2k+1$ diagonals (at most) are of interest. Each diagonal needs a processor. So $O(k^2 N)$ processors are enough. (This assumes that whenever we want a processor, we just use it and more processors can be added at will. In the real world of few processors, a run time queue simulates this situation.)

Space complexity: Each tree needs space k^2. There are $k \times N$ trees. So $O(k^3 N)$ space is enough. For the constrcution of suffix trees, we need $O(N^{1+r})$ space for any $r > 0$. Hence the space complexity is $O(k^3 N + N^{1+r})$ for any $r > 0$.

5.5. Conclusion

We have presented some fast algorithms for comparing two trees. The algorithms use several suffix trees and introduce hopping and binary search techniques as methods for speeding up dynamic programming.

CHAPTER 6

Applications

In chapters 2, 3, 4 and 5 algorithms and theoretical results were presented in detail. Algorithms for computing the tree distance not only have theoretical interest but also have important applications. In this chapter two examples of applications involving tree distance are presented. The first example is the problem to compare RNA secondary structures in Molecular Biology. The second example is an application to use tree comparison to compare shapes that might arise in image processing.

6.1. Comparison of Multiple RNA Secondary Structures

6.1.1. Introduction

The objective of comparing many RNA secondary structures is to determine how structural similarity is related to functional similarity. This approach is analogous to the philosophy behind sequence similarity programs [NW-70], [S-85], [S-80], [N-83], [WL-83] and [F-84] where portions of sequence that are similar behave in a similar way in a functional sense. Recent biological experiments [ALKBO-87], [BSSBWD-87], [DD-87] and [TGTGGGSDUTBG-88] have indicated several instances where similar structural motifs appear to have similar functionality. This concept of determining similarity may also be useful for structure prediction of phylogenetically related sequences.

The method discussed in this section has the ability to compare structural motifs by abstracting RNA secondary structures to trees and then using the algorithms discussed in

chapter 2 to compare them. We will show that the abstraction to trees can incorporate varying amounts of detail.

6.1.2. RNA Molecule [S-83]

A molecule of ribonucleic acid (RNA) is made up of a long chain of subunits -- ribonucleotides -- linked together. Each ribonucleotide contains one of four possible bases, abbreviated A, C, G, and U, and it is the sequence of bases in the chain that distinguishes one type of RNA from another. This base sequence is called the primary structure of the RNA molecule.

Under natural conditions, a ribonucleotide chain will twist and bend, and the bases will form bonds with one another in a complicated pattern, so that the molecule forms a coiled and looped conformation. Both the conformation and the pattern of bonding are called the secondary structure, but this chapter deals only with the pattern bonding. The base-to-base interactions that form the RNA secondary structure are of two kinds, namely, hydrogen bonding between an A and a U, and hydrogen boding between a C and a G. Figure 6.1 depicts the secondary structure of a transfer RNA molecule. Note this section is from [S-83] page 93.

6.1.3. RNA Secondary Structure and Trees

If one examines a typical secondary structure such as the one depicted in Fig 6.1, it becomes apparent that such a structure may be represented as a tree. It is also easy to see that such a tree is an ordered tree.

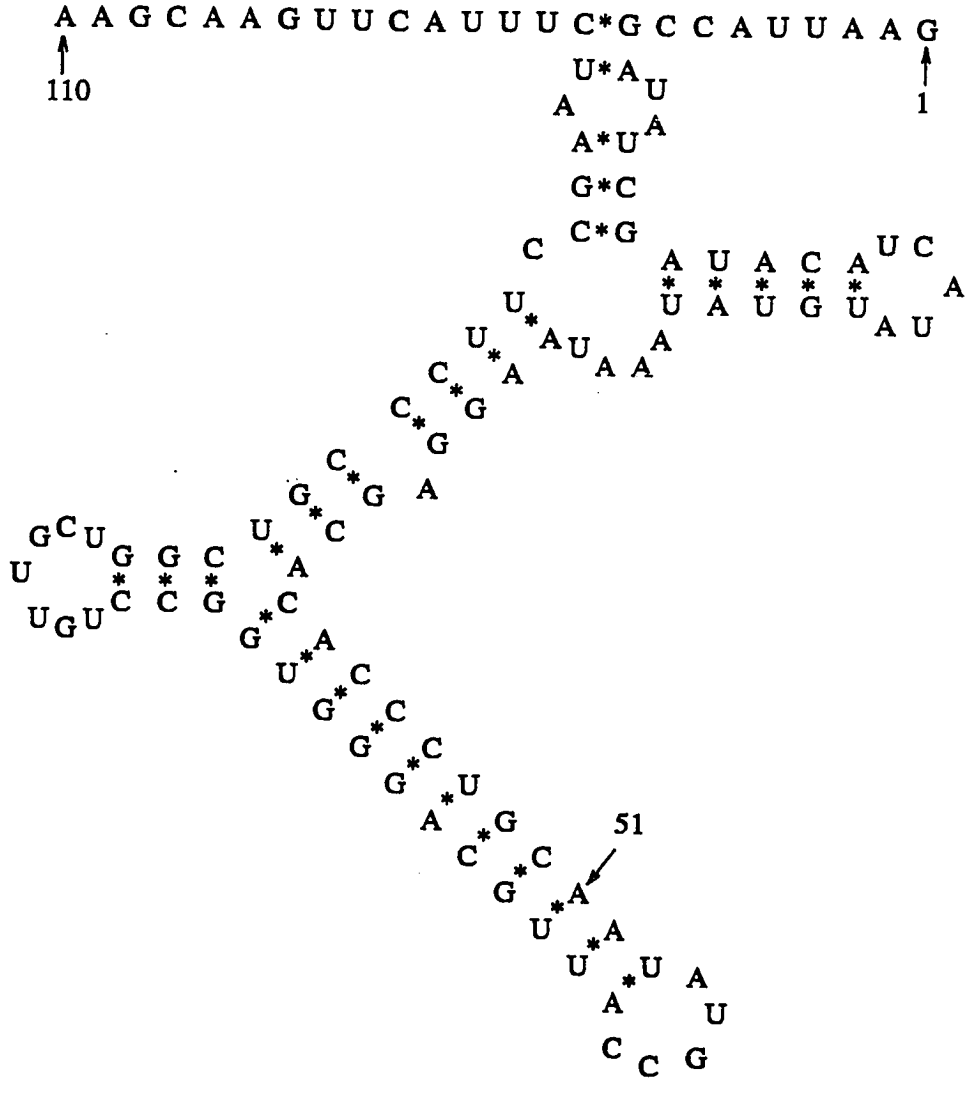


Figure 6.1 The secondary structure of a transfer RNA.

One may represent the helical stems and loops as nodes in a tree as shown in Fig 6.2, where hairpins are represented by H, internal loops are represented by I, bulge loops are represented by B, multiple loops are presented by M, helical stem regions are represented by R and N is a special node indicating the root of the tree.

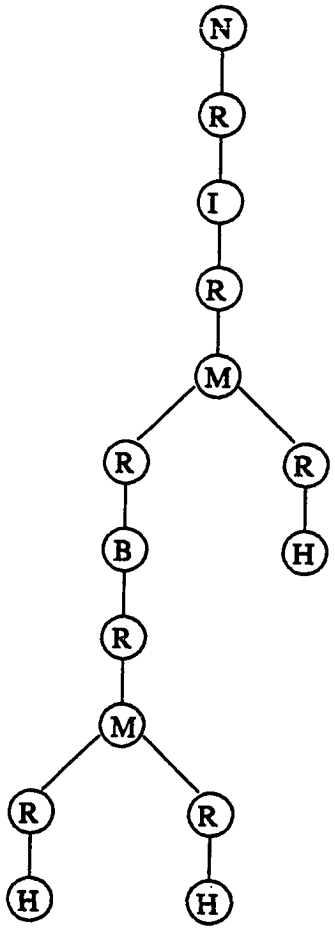


Figure 6.2 Tree representation of secondary structure of RNA in Figure 6.1.

Note that the only information in each node is the node type. One may include the size of the helical stems and loops as additional information in nodes as shown in Fig. 6.3.

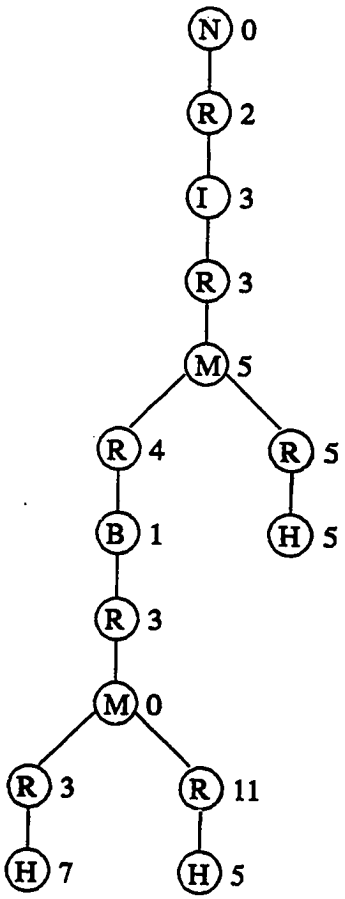


Figure 6.3 Tree representation with size information.

Yet at another level of detail, one may use the component size instead of the total size in each node (see Fig 6.4). The component size is an n-tuple where n is the number of edges incident on a node. An element of this tuple represents the number of unattached nucleotides between a given pair of consecutive edges. For example, in figure 6.4, the uppermost node with label M has a 4 between its left-child and right-child edges. That represents the length of the sequence UAAA of unattached nucleotides in the uppermost multiple-branch of figure 6.1.

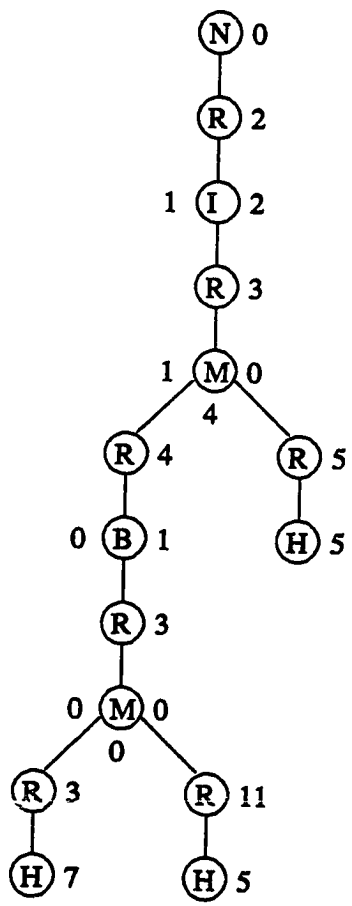


Figure 6.4 Tree representation with component size information.

Another refinement is to include the sequence data and break up stems into stacked pairs. This is shown in Fig 6.5 (only part of the tree is shown). This representation has the advantage that one can also include energy information for each node very easily.

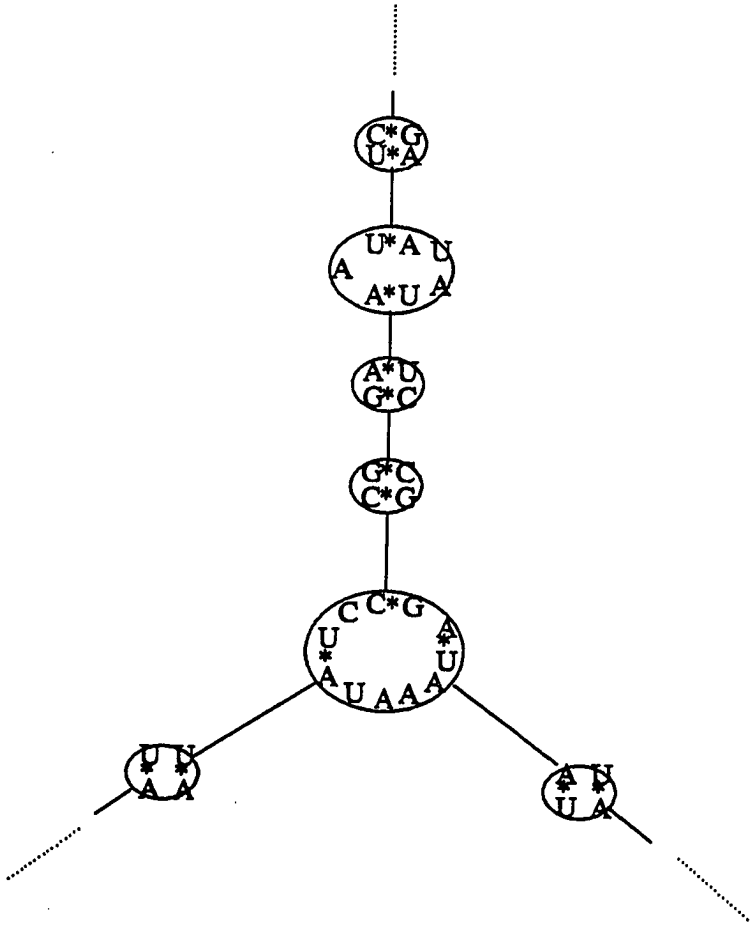


Figure 6.5 Another refinement of tree representation.

6.1.4. Comparing Multiple RNA Secondary Structures using Tree Comparisons

6.1.4.1. Pairwise Comparison using Tree Distance Algorithm

In order to compare multiple RNA secondary structures, we first use the tree distance algorithm presented in chapter 2 to compute the pairwise distance between RNA secondary structures. This is possible because RNA secondary structures can be represented as ordered trees by performing the traversal from the 3′ to 5′ end.

From chapter 2 it is clear that the only thing we have to deal with is how to determine the cost function used by the tree distance algorithm. The cost function varies depending upon the amount of information that is included within the nodes of the tree. The following symmetric table indicates the current heuristic costs (costs of node types) that are used when comparing structures.

	N	I	B	H	M	R	NULL
N	0	∞	∞	∞	∞	∞	∞
I		0	3	8	8	∞	5
B			0	8	8	∞	5
H				0	8	∞	100
M					0	∞	75
R						0	5
NULL							0

The simplest one just compares morphology in which case the only node types that are being compared are N, M, I, B, H and R. (These "type costs" may not all be unity for mismatches as shown in the above table.) When dealing with total size information, the cost function is the type cost plus the absolute difference in size of the loops and helical stems. For example, when changing M to N, add the cost for the mismatch to the cost of the difference in the number of nucleotides making up M and N.

A further refinement breaks the total size cost into its component loop parts. The cost is then determined by adding the sum of the component-by-component absolute differences to the type cost. (If the number of component parts being compared are not the same, the best sum of the differences is chosen although this may not be the ideal way.)

The final refinement is to break helical stems to stacked pairs and to incorporate sequence data. Comparison of component sequences within the loops may be handled by a standard string homology algorithms. Also energy information can be used to help to determine the cost. The cost function of this level is currently under investigation.

6.1.4.2. Clustering Algorithm

Once all the pairwise structure distances are computed the next step is to cluster together those structures that are most similar. This is accomplished by a clustering algorithm due to Hong and Tan [HT-89]. The basics of this algorithm is described in the next paragraphs, but we will not give any details here.

The output of the clustering algorithm is a taxonomy tree where each leaf contains the name of a tree representing a single RNA secondary structure. Each internal node contains a metric measure of the distance between the structures contained in the subtrees.

Intuitively, suppose the value of an internal node v in the cluster tree is k. Then for any pair of RNA structures at the leaves of the subtree rooted at v there is a path among the RNA structures at the leaves of the subtree rooted at v such that the distance between consecutive elements of that path is k or less.

Formally, internal node v has value k, if for every a and b in the subtree rooted at v, the following function is less than or equal to k. (Here, the a_i's are all from the subtree rooted at v and d is the pairwise distance as computed by our tree comparison algorithms.)

$$D(a, b) = \min_{a_0, \cdots a_n} \{ \max_{0 \le i < n} \{ d(a_i, a_{i+1}) \mid a_0 = a, a_n = b \} \}$$

6.1.5. Discussion

This subsection is largely taken from [SZ-89]. The different cost functions presented above were used to compare 197 structures which are the same as in [S-88]. [S-88] used sequence comparison, rather than tree comparison. The results show differences between tree and string comparison for all metrics. However, the differences are largest with the most detailed metrics (i.e. the ones which incorporated the most information in the trees).

The actual time required was basically the same as using string comparison algorithms. This is because the time complexity of our tree distance algorithm is in average the same as string algorithm for the general RNA secondary structures.

A comparison was also run with 100 tRNA structures chosen randomly from the sequence data base. These sequences are thought to have "clover-leaf"-like structures ([SK-76] and [W-84]). For the 47 clover leaf structures, 33 are mophologically identical and of course our tree comparison algorithm discovers this. The other 14 clover leaf structures are somewhat different.

One might fear that the more detailed cost functions, such as the one that incorporates size, might destroy this morphology-induced clustering. However, our experiments showed that this did not happen for this data. We compared these results with that of pure sequence comparison using the GENALIGN [SM-86] program. The 47 clover-leaf structures were scattered throughout the 100 multiple sequence alignments. This indicates that tree comparison algorithms are useful for this application.

Further experiments are required with the cost function to make it reflect the true costs of changing from one conformation to anther. Such a cost function should probably include a measure of the energy required to change such conformation. A weighted cost matrix should

also be included at the level of sequence comparisons so that, for example, differences due to compensatory base changes have a different cost than those due to nucleotide changes.

6.2. Application in Computer Vision

6.2.1. Represent Curves by Ordered Labeled Trees

In this section we will consider how to represent one dimensional curves by ordered labeled trees and then show how this can be used in some computer vision applications. The goal is to compare two one dimensional curves in a projection, rotation, and translation-independent fashion. It will be based on concavities and convexities. Levels of decreasing detail will correspond to blurrings of the curve. (A *blurring* consists of averaging a small collection, say 4, of neighboring pixels and then using a threshold of, say 0.4, as input to a thinning algorithm to find a curve.) This comparison will proceed in several steps. (The method was suggested by Bob Hummel.)

In the first step, we take each curve and build a corresponding tree as follows. Count the number N of concavities and convexities in the figure and let the number of blurrings be $\log_2 N$. (We seek a reasonably bushy tree.) Form all of the blurrings. Let level 1 of the tree correspond to the curve itself and level $\log_2 N$ correspond to the most blurred version of the tree. We call the blurring associated with level i the "i blurring."

The nodes at level $\log_2 N$ using the following method (that we call *labeling*).

> *Labeling:* The curve is divided into concave and convex portions. Associated with each portion will be a node labeled concave or convex (as appropriate) and perhaps including other information (e.g. length, degree of concavity/convexity, and so on).

Given the nodes at level i+1, the nodes at level i will be formed as follows. Each node at level i+1 corresponds to a curve (some portion of the larger curve). Find the portion of the i blurring corresponds to the curve. Form the nodes according to the labeling method.

Therefore this sequence of blurrings can be characterized by a tree, where each level corresponds to a blurring, each node is labeled as either concave or convex as well as some other information, and the parent of a node n is the node whose corresponding portion includes the portion represented by node n. (See Figure 6.6, Figure 6.7, Figure 6.8 and Figure 6.9.)

We will then use a tree comparison algorithm in chapter 2 to compare such trees.

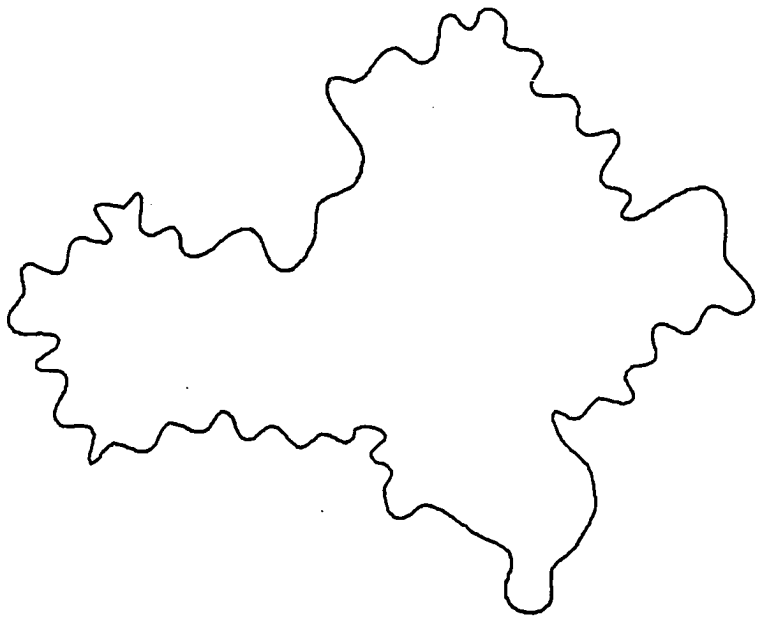


Figure 6.6 A closed curve.

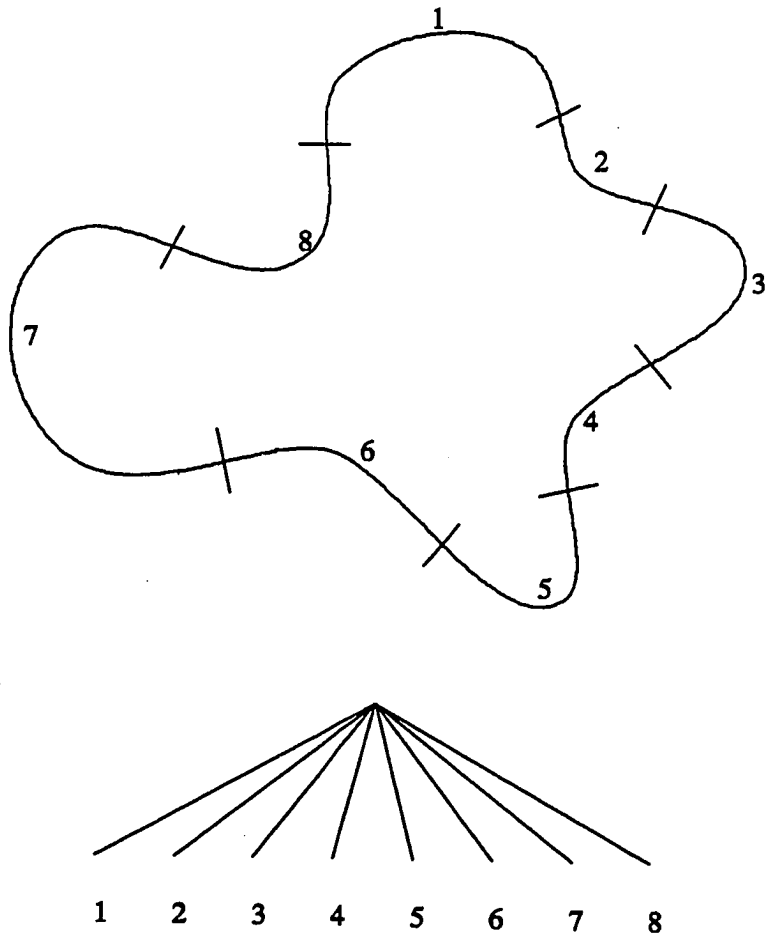


Figure 6.7 Blurring 2 and the corresponding tree.

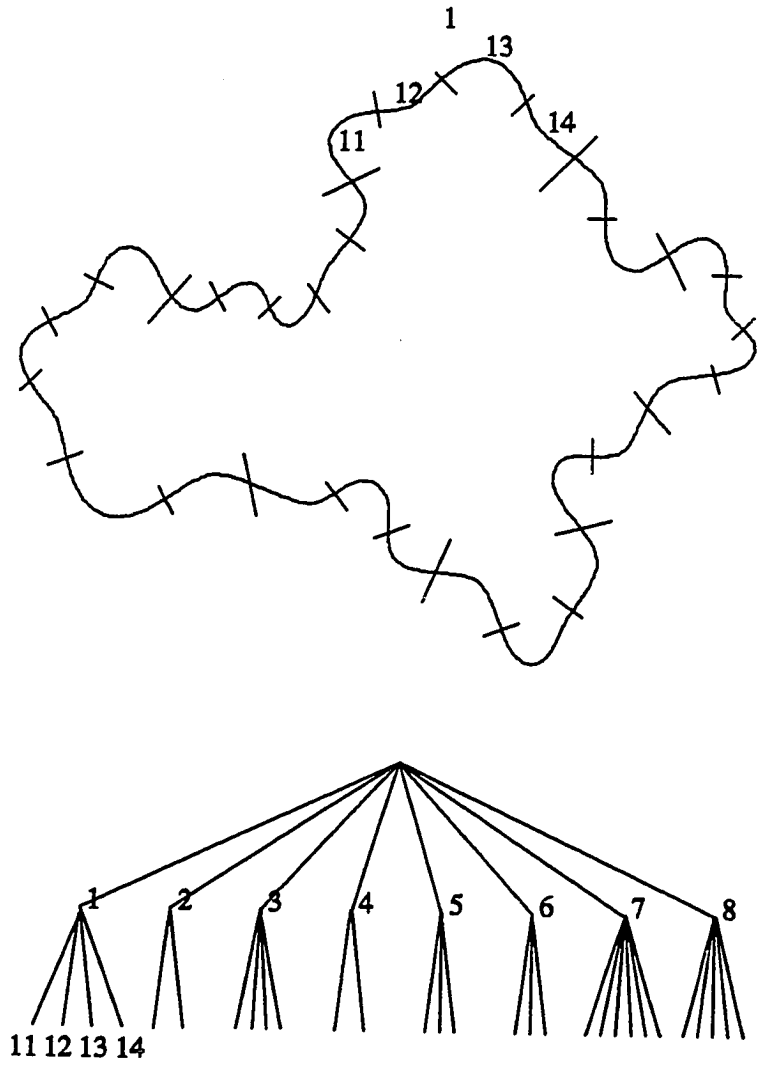


Figure 6.8 Blurring 1 and 2 and the corresponding tree.

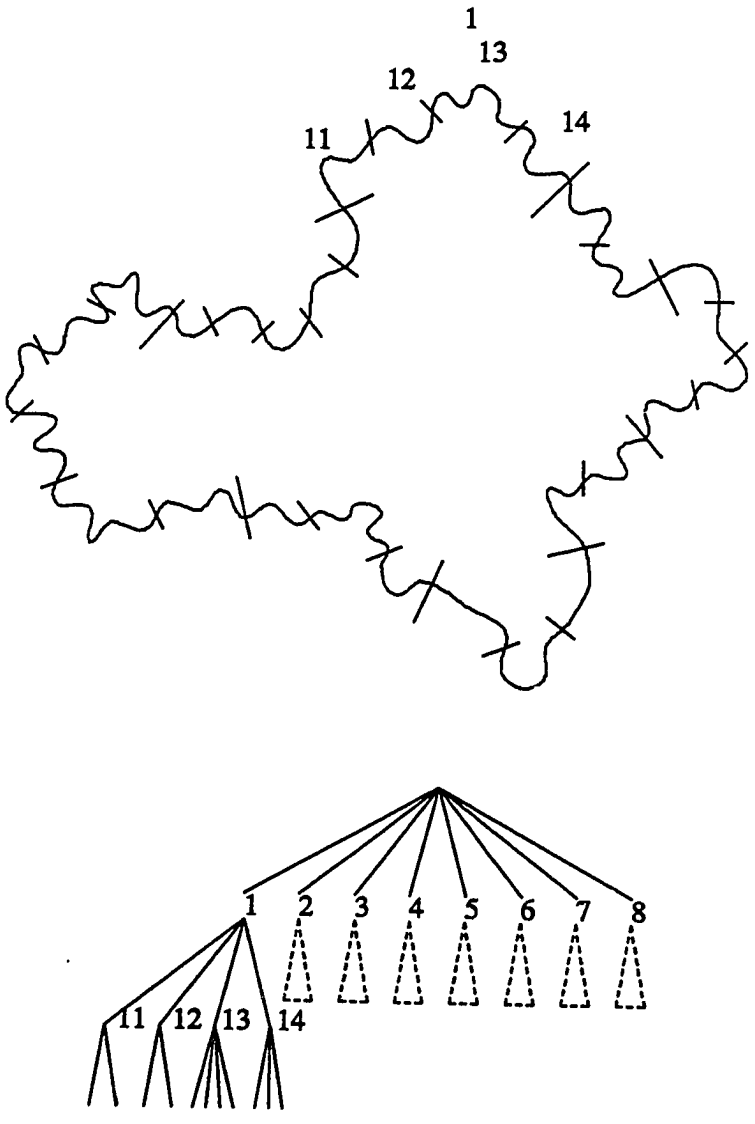


Figure 6.9 Tree representation with blurring 0 (no blurring), 1 and 2.

6.2.2. Applications

One application is waveform correlation ([EF-76] and [CL-85]). In this case we are not so interested in the distance measure. Instead the best mapping of two waveform become more important. Because the best mapping give us more information about how these two waveforms are related.

Another application is comparison of shapes. This is useful, for example, in model based vision. If we can represent our model objects to be model trees then we can use the tree comparison algorithm to determine which class a new object belongs to. Note that the curves for shapes are closed curves. Therefore each shape can be represented by several trees produced by the rotation of the children of the root of the tree (see Figure 6.10).

Hence when comparing shapes we will use our tree distance algorithm several times to compute the distance of one set of trees representing one shape to the set of trees representing the other shape. The distance measure between two shapes will be defined by the minimum of the above computed distances between the two sets of trees. It is easy to see that this distance measure will be a projection, rotation, and translation independent measure.

The critical problems for vision are the weights to assign to each editing operation. This must be determined by the nature of the problem at hand and may also be determined by experiments.

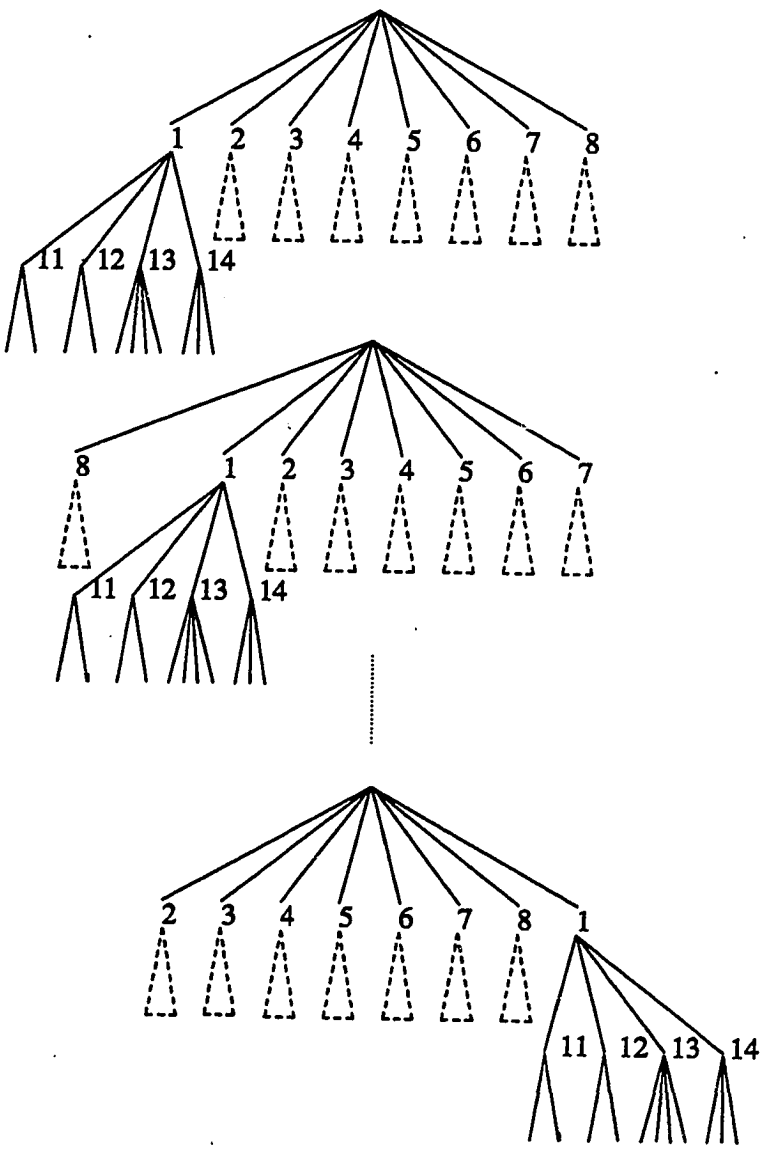


Figure 6.10 Trees to represent closed curve in Figure 6.6. The children of the root are rotated to yield the different trees.

CHAPTER 7

A tree Comparison Toolkit

We have implemented some of our tree editing algorithms into a tree comparison toolkit. The need for such an toolkit is apparent since there are many potential applications. The purpose of this toolkit is to provide a general tool that makes the tree comparison algorithms easy to use by various application users.

7.1. Architectural Review

7.1.1. The Organization of Tree Toolkit

The heart of the toolkit is composed of a module for tree comparison algorithms and a module for a program generator. We provide an simple language for users to define their format of tree node. This user defined node format is then fed into the module of program generator to produce I/O programs as well as default cost function programs for this specific node format. Those program can then be complied and linked with the tree comparison module to produce a customized tree comparator (see figure 7.1).

The only thing left to the user is the cost function. Even though we provide some default cost function, we can not provide a cost function that satisfies every application. If one wants to have a specific cost function, one must write his own cost function procedure. In our program generator, we provide some procedure frames where users can write their own cost functions.

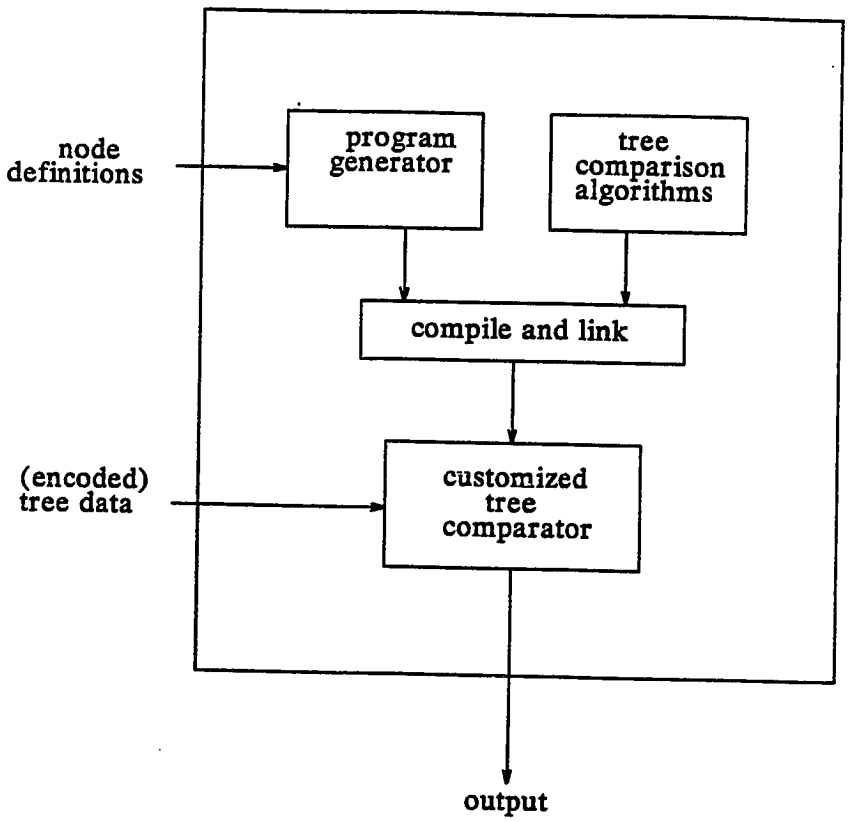


Figure 7.1 Tree toolkit organization

7.1.2. Node Format

As mentioned previously, nodes can be associated with a variety of information. This information can be grouped into the form of a tuple of fields of basic data types, i.e. integers, reals, characters, strings. Each field can either have a single value of a type, a fixed-sized array of that type, or an arbitrary-length sequence of that type. Formal rules governing the definitions of the node format are given, in a BNF description, below. Key-words are in italics.

Format	=> { seq_node_format }
seq_node_format	=> node_format [node_format]*
node_format	=> id { seq_field }
seq_field	=> [field;]*
field	=> char_field | string_field | int_field
char_field	=> *char* seq_var
string_field	=> *string* seq_var
int_field	=> *int* seq_var
seq_var	=> var | var , seq_var
var	=> id | id[] | id[intg]
id	=> letter [letter | digit | _]*
intg	=> [1-9] [digit]*
letter	=> [_a-zA-Z]
digit	=> [0-9]

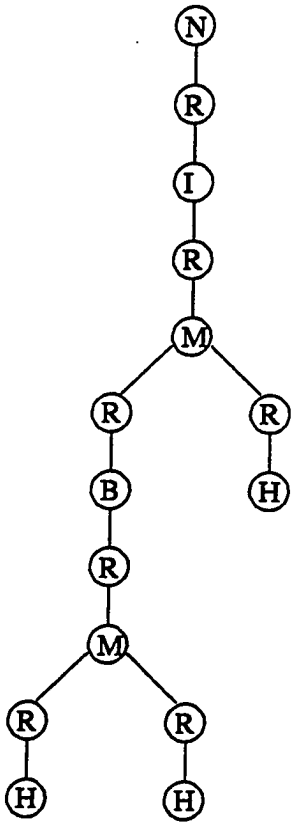


Figure 7.2 Each node has a label only.

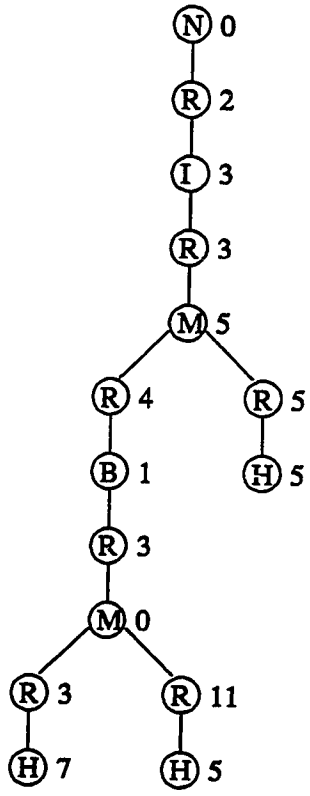


Figure 7.3 Each node has a label and a integer.

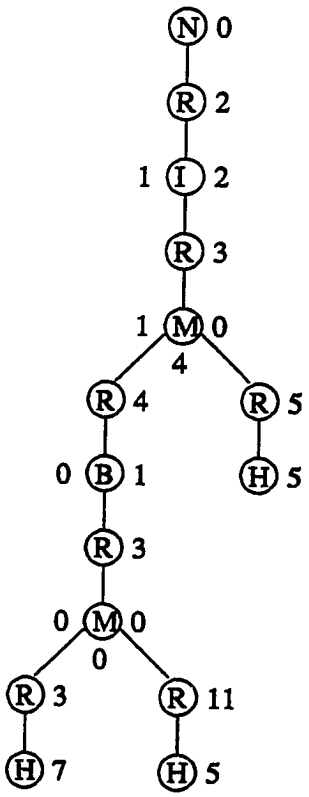


Figure 7.4 Each node has a label and a sequence of integers.

To illustrate the use of this grammar, consider that we want to define three different node formats. Node_L1 will correspond to the nodes in the tree of figure 6.2: each node has a label only. Node_L2 will correspond to the nodes in the tree of figure 6.3: each node has a label and size. Node_L3 will correspond to the nodes in the tree of figure 6.4: each node has a label and a sequence of sizes. (The figures are reproduced as figures 7.2, 7.3, and 7.4.) These three node types could be defined by the following Format.

{

```
Node_L1
   {
   }
Node_L2
   {
   int size;
   }
Node_L3
   {
   int csize[];
   }
}
```

So, Node_L1, Node_L2 and Node_L3 are node id's, 'size' is the name of the integer and 'csize' will be the name of a sequence of integers. Notice that there is no provision in the grammar to specify the label of any given node. The reason is that the label is an implicit field.

After the user has typed these descriptions, he invokes the *codegen* procedure (more details on the interaction follow in section 7.2) which generates the following C language structure definitions.

```
typedef struct
{
   char *label;
} Node_L1;

typedef struct
{
   char *label;
   int size;
} Node_L2;

typedef struct
{
   char *label;
   int *csize;
   int csize_count_;
} Node_L3;
```

Besides producing these structure definitions, *codegen* will produce I/O programs (to read the data according to this format) as well as default cost functions for the trees of these

kinds.

7.1.3. Tree Encoding

Having described the format of nodes, we now turn to the description of specific trees.

In principle, trees could be encoded in many ways, provided that the encoding is unique and consistent, and the decoding process incurs little overhead. The encoding scheme we adopt is a preorder traversal of a tree, with children and their parent being separated by parentheses, followed by the node information. That is, the parenthesized expression specifies the labels of the nodes. Following that expression, describe the values that each node takes in the other fields in preorder according to the Format given. (By, "in preorder" we mean that the set of field-value pairs corresponding to the ith node in the parenthesized expression should be the ith set of field-value pairs listed. Within each set of field-value pairs, the fields should be listed in the order specified by the Format.)

As examples, consider the trees in figure 7.2, 7.3 and 7.4.

We encode the tree in figure 7.2 as shown below.

(N(R(I(R(M(R(B(R(M(R(H))(R(H))))))(R(H)))))))

We encode the tree in figure 7.3 as shown below.

(N(R(I(R(M(R(B(R(M(R(H))(R(H))))))(R(H)))))))
size 0; size 2; size 3; size 3; size 5;
size 4; size 1; size 3; size 0; size 3;
size 7; size 11; size 5; size 5; size 5;

We encode the tree in figure 7.4 as shown below.

(N(R(I(R(M(R(B(R(M(R(H))(R(H))))))(R(H)))))))
csize 0; csize 2; csize 1 2; csize 3; csize 1 4 0;
csize 4; csize 0 1; csize 3; csize 0 0 0; csize 3;
csize 7; csize 11; csize 5; csize 5; csize 5;

For each node the order of the field is exactly the same as in the definition. Each field is headed by the field name as in the definition. Each field is ended by a ',' except the last field is ended by a ';'. (No commas are shown here, because each node has only one field for this example.) For a field of an array of data with dimension k, the number of data elements must be exactly k, separated by blanks, and ended by ',' or ';' as appropriate. For a field of a sequence of data, data will be separated by blanks and ended by ',' or ';' as appropriate.

For the input data file we embed the tree encoding in the following format.

<tree; treename
Tree Representation
tree encoding
>end of treename

.....

<tree; treename
Tree Representation
tree encoding
>end of treename

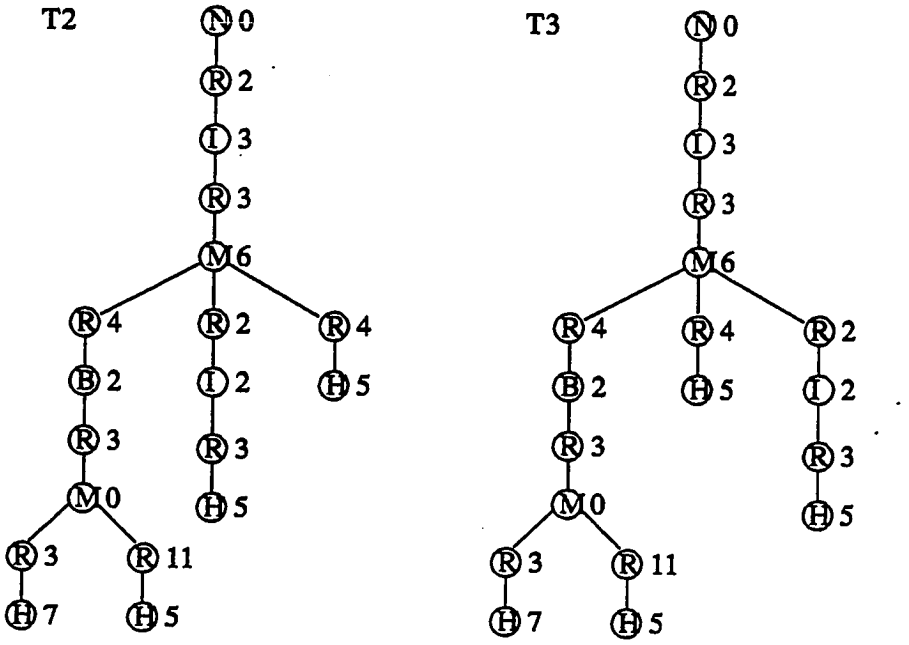


Figure 7.5 Another two trees with a label and a integer in each node.

For example, here are three trees corresponding to figures 7.3 and 7.5 in this format. We are calling tree in figure 7.3 "T1" and trees in figure 7.5 "T2" and "T3".

<tree; T1
Tree Representation
(N(R(I(R(M(R(B(R(M(R(H))(R(H))))))(R(H)))))))
size 0; size 2; size 3; size 3; size 5;
size 4; size 1; size 3; size 0; size 3;
size 7; size 11; size 5; size 5; size 5;
>end of T1
<tree; T2
Tree Representation
(N(R(I(R(M(R(B(R(M(R(H))(R(H))))))(R(I(R(H))))(R(H)))))))
size 0; size 2; size 3; size 3; size 6;
size 4; size 2; size 3; size 0; size 3;
size 7; size 11; size 5; size 2; size 2;
size 3; size 5; size 4; size 5;

```
>end of T2
<tree; T3
Tree Representation
(N(R(I(R(M(R(B(R(M(R(H))(R(H))))))(R(H))(R(I(R(H)))))))))
size 0; size 2; size 3; size 3; size 6;
size 4; size 2; size 3; size 0; size 3;
size 7; size 11; size 5; size 4; size 5;
size 2; size 2; size 3; size 5;
>end of T3
```

7.1.4. Cost Function

The code generator module (codegen) will provide default cost functions for the Format given. In fact for each kind of node it will provide one cost function that assumes that any edit operation has cost 1 (called *unit*) and another cost function that assumes that insert and delete have a constant cost and relabel has another constant cost (called *constant*). These must satisfy the triangle inequality (e.g. the relabeling cost must be less than the sum of the insert and delete costs).

In addition, each node definition two other user-specifiable cost functions: *user_def1* and *user_def2*. Initially, user_def1 and user_def2 are two empty procedure frames in file *node.c*. (Why two functions, you may ask? Users may want to experiment with different cost functions as applied to the same data. The toolkit allows users to experiment with two different cost functions in different comparison runs. Of course, to try yet other cost functions, users may just recompile with the new cost functions.)

7.2. Tree Toolkit Commands

7.2.1. Command Summary

Essentially, commands offered by the toolkit can be categorized into three classes.

class 1	help, load, print_tree, quit, !unix
class 2	cut_tree, prune_tree, subtree, subtree_limited, tree
class 3	taxonomy

Class 1 contains utilities and I/O commands. Class 2 consists a set of commands that deal with distance between trees and subtrees. The taxonomy command in class 3 implements Hong's taxonomy method [2]. (See chapter 6, biology section for a brief description of that method.)

7.2.2. Detailed Descriptions

- cut_tree [-c]: Tree to tree distance with ≥ 0 subtrees removed from one of the trees. (If there are two trees, T1 and T2, then cut_tree will compute the distance between T1 and T2' where T2' is T2 with some subtrees removed and then the distance between T1' and T2 where T1' is T1 with some subtrees removed.) The -c option gives the best mapping and best cut as well as the distance.

- help: Gives description of the commands.

- load: Load in trees from a set of trees. format: fully parenthesized in preorder followed by node information as described in section 7.1.3.

- print_tree: print trees in format used by algorithm, postorder with $l(i)$ values. (The node $l(i)$ is the leftmost descendant of the ith node in the postorder traversal of a tree. See chapter 2.) This is used for debugging.

- prune_tree [-p]: Tree to tree distance with ≥ 0 prunings of one of the trees. (Chapter 3 discusses pruning. Pruning at a node means removing all the descendants of that node.)

The -p option gives the best mapping and best pruning as well as the distance.

- quit: Leave the program
- subtree [-p]: Subtree to subtree distance in an array format. The -p option gives one largest (based on sum of size of trees in pair) subtree pair having each distance values from 0 to tree-to-tree distance This is done for every pair of trees that is loaded.
- subtree_limited: Subtree to subtree distance in a parenthesized format up to some chosen distance k.
- taxonomize: Given an array of tree-to-tree distances, produce Hong taxonomy.
- tree [-m]: Computes the tree to tree distance. The -m option gives the best mapping.
- !unix command: creates a shell and executes the given unix command. e.g. !ls

7.3. How to Use the Toolkit

This section illustrates by a complete example how to use a customized distance comparator in a UNIX BSD system.

For the time being, most commands are executed in a batch fashion -- that is, the input is written in a file and the output will also be printed out in a file. We intend to enhance the kit to support both interactive and batch environments in the near future.

7.3.1. Construct Your Own Tree Tool

Suppose you have the following node definition as in file *nodedef*:

```
{
Node_L1
   {
   }
Node_L2
   {
   int size;
   }
Node_L3
   {
   int csize[];
   }
}
```

Then your can type:

codegen <nodedef

This will produce two files: *node.h* and *node.c*.

If you want to write your own cost function you can find procedures in file *node.c* with the names of *Node_L1_user_def1* and *Node_L1_user_def2*, *Node_L2_user_def1* and *Node_L2_user_def2*, and *Node_L3_user_def1* and *Node_L3_user_def2*.

For example, Node_L2_user_def1 is defined initially as follows:

```
int
Node_L2_user_def1(ND1, ND2)
Node_L2 *ND1, *ND2;
{
   ;
}
```

You can now write your own cost function. For example, the following rather special purpose cost function (used later in this section) works as follows:

If inserting then the cost is 5 plus the size of the inserted node.

Else if deleting then the cost is 5 plus the size of the deleted. node.

Else if the labels are the same, then the cost is the difference in the sizes.

Else if one of the labels is 'I' and the other is 'B', then the difference is 3 plus the difference in sizes.

Else if either is a 'R' node or a 'N' node, then the cost to relabel is as big as deleting the first tree's node and inserting the second tree's node. (You can see that by checking the addition).

Finally, any other relabeling costs 8 plus the difference in the node sizes.

```
int
Node_L2_user_def1(ND1, ND2)
Node_L2 *ND1, *ND2;
{
   if (ND1 == NULL)
      return(5 + ND2->size);
   else if (ND2 == NULL)
        return(5 + ND1->size);
   else if ((*(ND1->label) == 'N') && (*(ND2->label) == 'N'))
      return(0);
   else if (*(ND1->label) == *(ND2->label))
        return(abs(ND1->size - ND2->size));
   else if ((*(ND1->label) == 'I') && (*(ND2->label) == 'B'))
      return(3 + abs(ND1->size - ND2->size));
   else if ((*(ND1->label) == 'B') && (*(ND2->label) == 'I'))
        return(3 + abs(ND1->size - ND2->size));
   else if ((*(ND1->label) == 'R') || (*(ND2->label) == 'R'))
      return(10 + ND1->size + ND2-> size);
   else if ((*(ND1->label) == 'N') || (*(ND2->label) == 'N'))
        return(10 + ND1->size + ND2-> size);
   else
      return(8 + abs(ND1->size - ND2->size));
}
```

Now to compile *node.c* and then link it to *treecom.o*, type:

cc -c node.c

cc -o tree node.o treecom.o

This will produce a file called *tree*. This is your own tree toolkit.

7.3.2. Running Calculations

At your usual operating system prompt, type:

tree

followed by a carriage return. On your screen, you will see the prompt:

@

Now you are ready to give the commands listed in the previous section. Here we shall show how to find the distance between two trees.

7.3.3. An Example

Suppose that you have file *data* with following three trees (see figure 7.3 and figure 7.5).

```
<tree; T1
Tree Representation
(N(R(I(R(M(R(B(R(M(R(H))(R(H))))))(R(H)))))))
size 0; size 2; size 3; size 3; size 5;
size 4; size 1; size 3; size 0; size 3;
size 7; size 11; size 5; size 5; size 5;
>end of T1
<tree; T2
Tree Representation
(N(R(I(R(M(R(B(R(M(R(H))(R(H))))))(R(I(R(H))))(R(H)))))))
size 0; size 2; size 3; size 3; size 6;
size 4; size 2; size 3; size 0; size 3;
size 7; size 11; size 5; size 2; size 2;
size 3; size 5; size 4; size 5;
>end of T2
<tree; T3
Tree Representation
(N(R(I(R(M(R(B(R(M(R(H))(R(H))))))(R(H))(R(I(R(H)))))))))
size 0; size 2; size 3; size 3; size 6;
size 4; size 2; size 3; size 0; size 3;
size 7; size 11; size 5; size 4; size 5;
size 2; size 2; size 3; size 5;
>end of T3
```

To run these, you first need to load the trees by typing:

@ load

The program will respond:

Please tell me the file name:

As the trees are stored in file 'data', you can type:

data

The system responds:

0 T1
1 T2
2 T3
Please tell me the tree names you are interested in:

Here the program has printed out a complete list of names of trees stored in file 'data', and the system is waiting for your favorite trees. You can continue by typing:

T1
T2
T3
.

As shown, you have asked for three trees to be compared. You can also typing *all* in this case. In general, there may be many trees in the data file and more than two may be requested. It is necessary to list the trees in the same order they appear in the data file. The period is used to terminate the list of trees. (If you want all the trees then just type *all* without a period.)

The comparator will then want to get node format information for each tree. It responds:

Please tell me the input tree format:
Node_L1
Node_L2
Node_L3

Here the system shows a complete list of node format names you defined before in *nodedef*. Any two trees that you wish to compare must have the same format in order to be comparable at all. Since the trees in file 'data' have the Node_L2 structure, you can go on by typing:

Node_L2

On the screen, you will again see the prompt:

@

Up to this point you have defined the trees of interest, including their names, structures, and information associated with nodes. Notice that the node structure you chose here (viz. Node_L2) should be consistent with that specified in the trees you have chosen previously (viz. T1, T2, and T3). Notice also that all the trees to be compared will have the same node structures.

7.3.4. Getting Results

In the previous section, you have provided the toolkit with all the necessary information concerning your favorite trees. You can now select any of the commands in class 2 to get various distance values between trees.

For example, to find the distance between the two trees, you continue by typing:

@tree

The system responds:

Please tell me the cost function:
unit
constant
user_def1
user_def2

Suppose that you decide to choose the constant cost function, you can type:

constant

The system will respond:

Please tell me the Insert or Delete Cost:

You can key in an appropriate number, say 3:

3

The system will then request another cost:

Please tell me the Relabeling Cost:

You may type:

2

Because data are processed in batch, the comparator will put the result in some file. It will ask:

Please tell me the output file name:

You can assign the result into an appropriate file by typing, say

outputfile

On the screen, you will then again see the prompt:

@

And your result is now stored in the file 'outputfile'. To get the output on the screen, you can create a shell and invoke the 'more' command. So you type:

@!more outputfile

Now the file is on the screen as follows:

3
T1
T2
T3
18

18 12

The number in the first line tells you the total number of trees in you current tree set. Following this number will be a sequence of tree names. After tree names is an array of tree to tree distances. The value of the (i,j)th position is the distance between tree i and tree j. We only show part of the array such that $1 < i \leq N$ and $1 \leq j < i$ since d(i,j)=d(j,i) and d(i,i)=0. In this example, the number 18 in the first line is the distance between *T2* and *T1*, the number 18 in the second line means the distance between *T3* and *T1*, and the number 12 in the second line means the distance between *T3* and *T2*.

After the file is shown on the screen, the prompt will appear again:

@

The system is now waiting for another command.

To find the distance between the three trees with cut, you continue by typing:

@cut_tree

The system responds:

Please tell me the cost function:
unit
constant
user_def1
user_def2

Suppose that you decide to choose *user_def1* cost, you can type:

user_def1

The system will ask:

Please tell me the output file name:

You can assign the result into an appropriate file by typing, say

outputfile1

On the screen, you will then again see the prompt:

@

And your result is now stored in the file 'outputfile1'. To get the result, you can type:

@!more outputfile1

Now the file is on the screen as follows:

```
3
T1
T2
T3
0  35  35
3   0  19
3  19   0
```

The numbers in the first line tells you the total number of trees in you current tree set. Following this number will be a sequence of tree names. After the tree names is an array of tree to tree distances. The value of (i,j)th position means the distance between tree i and tree j with zero or more cuts on tree i. (That is, the distance between tree i' and tree j, where tree i' is tree i with an optimal number of removals of subtrees.)

In this example, the first 35 in the first line is the tree distance between *T1* and *T2* with cuts on tree *T1* using user defined cost function *Node_L2_user_def1*. The second 35 in the first line is the tree distance between *T1* and *T3* with cuts on tree *T1* using user defined cost function *Node_L2_user_def1*. The number 3 in the second line is the tree distance between *T2* and *T1* with cuts on tree *T2*.

After the file is shown on the screen, the prompt will appear again:

@

The system is now waiting for another command. To conclude this session, you can type:

@quit

This will let you go back to the UNIX environment.

7.4. How to Install the Toolkit

The toolkit will be in a tape, written in *tar* format at 1600 bpi, includes following directories and files:

TREEtool/doc

Documentation including

1) manual (manual.tex) in latex format.

2) a brief description (help.txt) of toolkit commands.

TREEtool/treecomsrc

The source code for the tree comparison programs.

TREEtool/codegensrc

The source code for the program generator programs.

TREEtool/taxonomysrc

The source code for the taxonomy programs.

TREEtool/bin

Command files ***install*** and ***setuptree***.

To install the toolkit, proceed as follows:

- Step 1: Installation location

  Decide where the TREEtool directory is to be located. We assume below that *loc/TREEtool* is the full path name of that directory.

- Step 2: Installing the TREEtool directory

  Change your working directory to *loc*, mount the tape, and execute the command

  tar xv

- Step 3: Making the location of TREEtool directory known to system and toolkit

  Add full path (loc/TREEtool/bin) to variable PATH. Execute the command

  setenv TREE loc/TREEtool

  This can be most easily accomplished by adding above to your .login file.

- Step 4: Installing the toolkit

  Change your working directory to loc/TREEtool/bin. Execute the command

  install

- Step 5: Constructing your own tree comparator

  Change to a directory of your choice. Create your own tree node format definition file *nodedef*. Execute

  codegen <nodedef

  Maybe edit file *node.c* to create your own cost function. Execute

  setuptree

  Now in your working directory you will have file *tree*. This is your tree comparator.

CHAPTER 8

Conclusion and Future Work

8.1. Summary

In this thesis we have presented algorithms and applications of the editing distance between trees.

For the general cost function we have the following results for ordered trees.

[1] A dynamic programming algorithm that computes the distance between two trees in sequential time $O(|T_1| \times |T_2| \times \min(depth(T_1), leaves(T_1)) \times \min(depth(T_2), leaves(T_2)))$ and space $O(|T_1| \times |T_2|)$ compared with $O(|T_1| \times |T_2| \times (depth(T_1))^2 \times (depth(T_2))^2)$ for the best previous published algorithm. Our new algorithm is significantly simpler to understand and to implement. Further, our algorithm can be parallelized to give time $O(|T1| + |T_2|)$.

[2] We generalize the same technique to the problem of approximate tree matching. This is a generalization of the approximate string matching problem and the exact tree matching problem. We solve the approximate tree matching problem in the same time and space as in 1).

[3] We further generalize to the constrained tree editing problem. We also improved the previous best result for the constrained string editing problem.

For the unit cost function we have the following results. Let k be the actual distance between two trees. Let N be $min(|T_1|, |T_2|)$ and L be $min(depth(|T_1|), depth(|T_2|))$

[1] An $O(k^2 \times N \times L)$ sequential algorithm.

[2] An $O((k^2 \times log(k)) + log(N))$ time parallel algorithm with $k^2 \times$ N processors.

[3] An $O(k \times log(k) \times log(N)$ time parallel algorithm with $k^2 \times$ N processors.

The condition for [1] can be relaxed from the unit cost function to arbitrary integer cost functions that satisfy the triangle inequality. The condition for [2] and [3] can be relaxed from the unit cost function to any integer constant cost function (i.e. one which gives a fixed though possibly distinct values to relabelings, insertions, and deletions) that satisfies the triangle inequality.

For the unordered labeled trees, we show that the problem to determine the editing distance between two unordered labeled trees is NP-Complete. This seems to indicate that it will be very difficult to generalize editing distance to more complex structures.

One of our applications is to compare secondary structures of RNA molecules. We represent RNA secondary structures as trees and show how to use tree comparison to compare multiple RNA secondary structures. We describe another application to vision that uses tree comparison to compare shapes.

We also have implemented the algorithms in the form of a tree comparison toolkit. We provide a simple manual for other prospective users of the toolkit.

8.2. Future Work

(1) The problem of implementing these algorithms on special parallel architectures such as systolic arrays or meshes has been suggested to us by many researchers and is a promising direction.

(2) Find heuristics or algorithms for special cases of the problem of tree comparison on unordered trees, since the general problem is NP-complete. The problem is very important in

practice, e.g. for genetic disease tracking.

(3) Design PRAM parallel algorithms that can deal with general cost functions with time complexity in NC ([AALM-88] and [AKLMT-89]).

(4) An important problem in string comparison is: given a set of N strings, produce a new string whose maximum distance to any of the N strings is minimized. This is a challenging and potentially important problem for trees.

(5) The lower bound of the tree editing problem is still unknown. It would be interesting to find ([AHU-76], [H-78] and [WC-76]).

(6) We have described an application of tree comparison to computer vision. Further applications and experiments are a promising avenue of work.

REFERENCES

[AALM-88]

A. Apostolico, M.J. Atallah, L.L. Larmore, and H.S. McFaddin, "Efficient parallel algorithms for string editing and related problems" *Proc. 26th Annual Allerton Conf. on Communication, Control, and Computing*, Monticello, Illinois, pp.253-263. (Sept., 1988)

[AHU-74]

A.V. Aho, D.S. Hirschberg, and J.D. Ullman, "Bounds on the complexity of the longest common subsequence problem" *J. ACM*, 23, pp.1-12. (1974)

[AKLMT-89]

M.J. Atallah, S.R. Kosaraju, L.L. Larmore, G.L. Miller, and S.H.Teng, "Constructing trees in parallel" *Proc. 1989 ACM Symp. Parallel Algorithms and Architectures*, Santa Fe, New Mexico, pp.421-431. (June, 1989)

[ALKBO-87]

S. Altuvia, H. Locker-Giladi, S. Koby, O. Ben-Nun, and A.B. Oppenheim, "RNase III stimulates the translations of the cIII gene of bacteriophage lambda" *Proc. Natl. Acad. Sci. USA,*, 84, pp.1-5. (1987)

[BSSBWD-87]

B. Berkout, B.F. Schmidt, A. Strien, J. Boom, J. Westrenen, ans J. Duin, "Lysis gene of bacteriophage MS2 is activated by translation termination at the overlapping coat gene" *Proc. Natl. Acad. Sci. USA,*, 195, pp.517-524. (1987)

[CL-85]

Y.C. Cheng and S.Y. Lu, "Waveform correlation by tree matching" /fIIEEE Trans. on PAMI/fR, PAMI-7, no. 3, pp.299-305. (1985)

[DA-82]

N. Delihas and J. Anderson, "Generalized structures of 5s ribosomal RNA's" *Nucleic Acid Res.*, 10, p. 7323. (1982)

[DD-87]

I.C. Deckman and D.E. Draper, "S4-alpha mRNA translation regulation complex" *J. Mol. Biol.*, 196, pp.323-332. (1987)

[EF-76]

R.W. Ehrich and J.P. Foith, "Representation of random waveforms by relational trees" *IEEE Trans. Comput.*, C-25, pp.725-736. (1976)

[F-84]

J.W. Fickett, "Fast optimal alignment" *Nucleic Acids Res.*, 12, pp.175-180. (1984)

[H-78]

D.S. Hirschberg, "An information theoretic lower bound for the longest common subsequence problem" *Information Processing Letter*, 7, pp.40-41. (1978)

[HO-82]

C.M. Hoffmann and M.J. O'Donnell, "Pattern matching in trees" *J. ACM*, 29, no. 1, pp.68-95. (1982)

[HT-89]

J. Hong and X. Tan, "The taxonomy-based learning: algorithms and applications" *COINS Technical Report*, University of Massachusetts at Amherst. (July, 1989)

[K-83]

D.E. Knuth, "The Art of Computer Programming", vol. I *Addison-Wesley Publishing Company, Inc.*, Reading, Mass.

[L-79]

S.Y. Lu, "A tree-to-tree distance and its application to cluster analysis" *IEEE Trans. on PAMI*, PAMI-1, no. 2, pp.219-224. (1979)

[LSV-87]

G.M. Landau, B. Schieber, and U. Vishkin, "Parallel construction of a suffix tree" *Proc. 14th ICALP*, Lecture Notes in Computer Science 267, Springer-Verlag, pp.314-325. (1987)

[LV-86]

G.M. Landau and U. Vishkin, "Introducing efficient parallelism into approximate string matching" *Proc. 18th ACM Symposium on Theory of Computing,*, pp.220-230. (1986)

[N-83]

R. Nussinov, "An efficient code searching for sequence homology and DNA duplication" *Theor. Biol.*, 100, pp.319-328. (1983)

[NW-70]

S.B. Needleman and C.D. Wunsch, "A general method applicable to the search of similarities in the amino acid sequence of two proteins" *J. Mol. Biol.*, 48, pp.444-453. (1970)

[O-87]

B.J. Oommen "Recognition of noisy subsequences using constrained edit distances"

IEEE PAMI, PAMI-9, no. 5, pp.676-685 (1987)

[S-80]

P. H. Sellers, "The theory and computation of evolutionary distances" *J. of Algorithms*, 1, pp.359-373. (1980)

[S-83]

"D. Sankoff, "Time warps, string edits, and macromolecules: the theory and practice of sequence comparison" *Addison-Wesley Publishing Company, Inc.*, (1983)

[S-85]

"D. Sankoff, "Simultaneous solution of the RNA folding, alignment and protsequence problems" *SIAM J. Appl. Math.*, 45, pp.810-825. (1985)

[S-88]

B. A. Shapiro, "An algorithm for comparing multiple RNA secondary structures" *Comput. Applic. Biosci.*, 4, pp.387-393. (1988)

[SK-76]

J. L. Sussman and S. H. Kim, "Three dimensional structure of a transfer RNA in two crystal forms" *Science*, 192, p. 853. (1976)

[SM-86]

E. Sobel and H. Martinez, "A multiple sequence alignment program" *Nucleic Acids Res.*, 14, pp.363-374. (1986)

[SV-88]

B. Schieber and U. Vishkin, "Parallel computation of lowest common ancestor in trees" NYU Computer Science Technical Report, (1988)

[SZ-89]

"Comparing multiple RNA secondary structures using tree comparisons" Submitted for publication.

[T-79]

Kuo-Chung Tai, "The tree-to-tree correction problem" *J. ACM*, 26, pp.422-433. (1979)

[TGTGGGSDUTBG-88]

C. Tuerk, P. Gauss, C. Thermes, D.R. Groebe, M. Gayle, N. Guild, G. Stormo, Y. D'Aubenton-Carafa, O.C. Uhlenbeck, I.Jr. Tinoco, E.N. Brody, and L. Gold, "CUUCGG hairpins: extraordinarily stable RNA secondary structures associated with various biochemical processes" *Proc. Natl. Acad. Sci. USA*, 85, pp.1364-1368. (1988)

[TV-85]

R. E. Tarjan and U. Vishkin, "An efficient parallel biconnectivity algorithm" *SIAM J. Computing*, vol. 14, no. 4, (Nov. 1985)

[U-83]

E. Ukkonen, "On approximate string matching" *Proc. Int. Conf. Found. Comp. Theory, Lecture Notes in Computer Science*, 158, Spring-Verlag, pp.487-495. (1983)

[U-85]

E. Ukkonen, "Finding approximate string matching" *J. of Algorithms*, 6, pp.132-137. (1985)

[W-84]

S. Wolfram, "Principles of nucleic acid structure" *Spring-Verlag*, New York, pp.331-332. (1984)

[WC-76]

C.K. Wang and A.K. Chandra, "Bounds for the string-editing problem" *J. ACM*, 23, pp.13-16. (1976)

[WF-74]

R. Wagner and M. Fisher, "The string-to-string correction problem" *J. ACM*, 21, pp.168-178. (1974)

[WL-83]

W.J. Wilbur and D. Lipman, "Rapid similarity searches of nucleic acid and protein data banks" *Proc. natl. Acad. Sci. USA*, 80, pp.726-730. (1983)

[Z-83]

KaiZhong Zhang, "An algorithm for computing similarity of trees" Technical Report of Mathematics Department of Peking University. (1983)

[ZS-87]

K. Zhang and D. Shasha, "On the editing distance between trees and related problems" Ultracomputer Note 122, NYU C.S TR 310, (Aug. 1987)

[ZS-88]

K. Zhang and D. Shasha, "Simple fast algorithms for the editing distance between trees and related problems" *SIAM J. Computing*, In press.