1

**PM-1 3½"x4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT**

PRECISIONSM RESOLUTION TARGETS

PIONEERS IN METHYLENE BLUE TESTING SINCE 1974

THIRD COUNTY FILMS & SUPPLIES, INC. BABY, USA
TEL: 812-426-7887 FAX: 812-426-7887 TLA: 5100000-000

National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Tree Editing Algorithms

by

William W.L. Lee, B. Sc.

**A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of
Master of Computer Science**

**School of Computer Science
Carleton University
Ottawa, Ontario**

April 10, 1992

© copyright

1992, William W. L. Lee

National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-75987-9

Canada

The undersigned recommend to the Faculty of Graduate Studies
and Research acceptance of the thesis

"Tree Editing Algorithms"

submitted by William W. L. Lee, B. Sc.
in partial fulfillment of the requirements for
the degree of Master of Computer Science

Thesis Supervisor

Director, School of Computer Science

Carleton University

6 May 1992

Date

Abstract

In this thesis we consider various string and tree editing problems. After reviewing the known algorithms to evaluate the existing string and tree similarity metrics, we define various new numeric similarity and dissimilarity measures between two trees. This is achieved by defining an abstract mathematical foundation for the similarity indices in terms of a set of inter-symbol measures and two abstract operators, $\oplus$ and $\otimes$. Apart from defining these measures, we also present schemes by which they can be efficiently computed.

All the existing tree editing algorithms consider the scenario in which the edit processes is absolutely unconstrained. Apart from the abstract formulation mentioned above, this thesis will also consider the Constrained Tree Editing problem, in which the system has to compute the optimal way to edit one tree into another subject to the constraint that the number of individual edit operations done is specified by a user-defined constraint.

Acknowledgements

I would like to express my heart-felt thanks to Professor John Oommen, my thesis advisor, for his continued support and guidance throughout the entire period while I was working on this thesis. This thesis would not have been possible without his tireless support, ingenious observations and enthusiasm. His inquisitive mind, imagination, and willingness to explore new ideas were a constant source of inspiration to me.

I would also like to express my sincere appreciation to Dr. Radha Valiveti for his assistance in critically proof-reading the thesis.

I will also take this opportunity to thank Robert Black and Shirley Lee for the trouble they took in helping me organize my thoughts and for proof-reading the "draft" versions of the thesis.

Finally, I would like to thank my biggest supporters, Lorraine Kong, Dai Lee, and Sig Lee. Without their encouragement none of this would have been possible.

Table of Contents

	Page
<i>Acceptance Sheet</i>	ii
<i>Abstract</i>	iii
<i>Acknowledgements</i>	iv
<i>Table of Contents</i>	v
<i>List of Figures</i>	viii
<i>List of Algorithms</i>	x
<i>List of Abbreviations</i>	xii
 I. Introduction	 1
1.1 Decision-Theoretic and Syntactic Approaches to Pattern Recognition	7
1.2 Syntactic Pattern Recognition System	11
1.3 Organization of the Thesis	13
 II. String Editing Problems	 15
2.1 Introduction	15
2.1.1 Notation and Elementary Edit Distances	16
2.1.2 Distance between Strings	17
2.1.3 The set $G_{X,Y}$	17
2.2 The String-to-String Correction Problem	19
2.2.1 The Optimal Algorithm for Infinite Alphabet	19
2.2.2 Operation of Interchanging Character Positions	23
2.2.3 A Linear Space Algorithm	27
2.2.4 Faster Algorithms for Finite Alphabets	28
2.2.5 The Constrained String Editing Problem	29
2.3 Problems involving Subsequences and Noisy Subsequences	32
2.3.1 An Optimal Algorithm and a Linear Space Algorithm	33
2.3.2 Time Bounds of the LCS Problem and Faster Algorithms	35
2.3.3 Recognition of Noisy Subsequence	39
2.4 Problems on Supersequences	42
2.5 Relationship between Similarity Measures of Two Strings	45
2.6 Conclusions	48

III. The Tree Editing Problems	49
3.1 Introduction	49
3.2 Definitions and Notations	50
3.2.1 Edit Operations and Distance between Trees	52
3.2.2 Mappings between Trees	55
3.3 Insertions and Deletions Restricted to Leaves	57
3.4 A Less Restricted Edit Algorithm	64
3.4.1 An Iterative Algorithm for the Less Restricted Tree Editing Problem	65
3.4.2 Tai's Algorithm Rendered Recursive	81
3.5 Faster Algorithm of the Tree Editing Problem	83
3.6 Conclusions	90
 IV. Numeric Similarity and Dissimilarity Measures Between Two Trees	 91
4.1 Introduction	91
4.2 Notations and Definitions	93
4.2.1 Sub-Trees and Super-Trees	93
4.3 An Abstract Measure Involving Two Trees	93
4.3.1 The Abstract Structure : τ	96
4.3.2 The Set of Elementary Measures : ω	97
4.3.3 Mappings, Extended Mappings and the Set $\Gamma_{(T_1, T_2)}$	97
4.3.4 The Abstract Measure : $\Omega(T_1, T_2)$	99
4.4 Numeric Measures For $\Omega(T_1, T_2)$	99
4.4.1 The Distance between Two Trees	99
4.4.2 The Size of the Largest Common Sub-Tree	100
4.4.3 The Size of the Smallest Common Super-Tree	103
4.4.4 Probability of Erroneous Trees	106
4.4.5 Likelihood of Erroneous Trees	108
4.5 Computation of $\Omega(T_1, T_2)$	112
4.5.1 Recursive Properties of $\Omega(T_1, T_2)$	112
4.5.2 A Dynamic Programming Algorithm to Compute $\Omega(T_1, T_2)$	124
4.5.3 Remarks	127

4.6	Time and Space Complexities	129
4.7	Conclusions	130
V.	Constrained Tree Editing	132
5.1	Introduction	132
5.2	Notations and Definitions	133
5.3	Edit Constraints	133
5.3.1	Permissible and Feasible Edit Operations	133
5.3.2	Specification of Edit Constraints	135
5.3.3	Differences between the Constrained String and Tree Editing Problem Specification	136
5.4	Constrained Tree Editing	137
5.4.1	The Constrained Editing Algorithm	138
5.5	Time and Space Complexities	147
5.6	Conclusions	148
VI.	Summary and Open Problems	150
6.1	Open Problems	152
6.1.1	Non-numeric Similarity Measures between Two Trees	152
6.1.2	Probabilistic Correction of Trees	152
	References	159

List of Figures

Figure No.	Description	Page
Figure 1.1	Example of transforming an input pattern into a form suitable for machine manipulation.	3
Figure 1.2	Sampling of the values of a continuous function to yield a feature vector.	3
Figure 1.3	Description of the use of primitive elements in pattern recognition.	4
Figure 1.4	String representation of equilateral triangles.	5
Figure 1.5	Example of the tree representation of an image which has been subject to smoothing and boundary tracking operation.	6
Figure 1.6	The tree representation of a simple scene.	7
Figure 1.7	A simple decision function for two pattern classes.	8
Figure 1.8	Example of decomposing the submedian chromosome into predefined primitives.	10
Figure 1.9	Block diagram of a syntactic pattern recognition system.	11
Figure 2.1	An example of a Trace.	19
Figure 2.2	The trellis to compute $D(X, Y)$.	22
Figure 2.3	Example trace of interchanging of symbols.	25
Figure 3.1	Examples of preorder and postorder sequences.	52
Figure 3.2	An example of the insertion of a node.	53

Figure 3.3	An example of the deletion of a node.	53
Figure 3.4	An example of the substitution of a node by another.	54
Figure 3.5	An example of a mapping between two postorder numbered trees.	56
Figure 3.6	Notations used in Selkow's Algorithm.	58
Figure 3.7	Example illustrating the decomposition of a mapping.	60
Figure 3.8	Trees used to justify the correctness of a transformation.	64
Figure 3.9	Evaluate $E[s,u,i:t,v,j]$ when $s \leq u < i$ and $t \leq v < j$.	73
Figure 3.10	Evaluate $E[s,u,i:t,v,j]$ when $u = i$ or $v = j$.	75
Figure 3.11	Evaluate $E[s,u,i:t,v,j]$ when $u = i$ and $v = j$.	76
Figure 3.12	An example of a tree, a postorder forest, a subtree, and the associated notations.	85
Figure 4.1	Examples of Sub-Trees, $LCsT$ and $SLCsT$.	94
Figure 4.2	Examples of Super-Trees, $SCST$ and $SSCST$.	95
Figure 4.3	Examples of an extended mapping.	98
Figure 4.4	Trees used to distinguish the difference between $LLCS$ and $SLCsT$.	102
Figure 4.5	Examples of the linear relationship between $SSCST(T_1, T_2)$ and $SLCsT(T_1, T_2)$.	105
Figure 4.6	Pictorial description of Case (3) of Theorem 4.8.	118
Figure 6.1	Pictorial description of the "Postorder Sequence" approach to Modelling the Channel generating the noisy trees.	154

List of Algorithms

	Page
Chapter II : String Editing Problems	
ALGORITHM Wagner and Fischer	21
ALGORITHM Lowerance and Wagner	26
ALGORITHM Hirschberg	27
ALGORITHM Hirschberg_LCS_Verison	34
ALGORITHM Hunt and Szymanski	36
ALGORITHM Hunt and Szymanski Enhanced_Verison	37
ALGORITHM Subsequence_Testing	39
ALGORITHM Recognize_Noisy_Subsequences	41
ALGORITHM Compute_the_Set_of_Supersequences	44
ALGORITHM Generalized_Operator	46
Chapter III : Tree Editing Problems	
ALGORITHM Selkow	63
ALGORITHM Tai	78
PROCEDURE Compute_E	79
PROCEDURE Compute_Min_M	80
PROCEDURE Compute_H	80
ALGORITHM Noetzel and Selkow	83
ALGORITHM Zhang and Shasha	88
PROCEDURE Compute_T_Wt	89
Chapter IV : Numeric Similarity and Dissimilarity Between Two Trees	
ALGORITHM Abstract_T_Measure	126
PROCEDURE Compute_Abstract_T_Meas	128

Chapter V : Constrained Tree Editing

ALGORITHM Constrained_Tree_Distance	145
PROCEDURE τ _Weights	143
PROCEDURE Compute_Const_T_Wt	144

List of Abbreviations

Abbreviation	Description
LCS	Longest Common Subsequence.
LLCS	Length of the Longest Common Subsequence.
SCS	Shortest Common Supersequence.
LSCS	Length of the Shortest Common Supersequence.
Anc (y)	Ancestor of node y.
Subst (x,y)	Substitution of node x by node y.
LCsT	Largest Common Sub-Tree.
SLCsT	Size of the Largest Common Sub-Tree.
SCST	Smallest Common Super-Tree.
SSCST	Size of the Smallest Common Super-Tree.

I. Introduction

The increased availability and reduced cost of computers has created many new avenues of research that were unknown decades ago. One such area is the study of *machine intelligence*, wherein machines behave in ways normally associated with human behaviour. The motivations for research in machine intelligence are simple. First, from an economic point of view, intelligent machines can yield increased productivity at a reduced cost. Additionally, the development of systems with autonomous decision-making capabilities can increase the security of potentially hazardous systems such as nuclear power plants. Finally, systems that "learn" can also be used to interact dynamically with the real world so as to enhance their performance as their learning proceeds.

Among the different tools used in the field of machine intelligence, *Pattern Recognition* techniques are some of the most important. The primary goal of a Pattern Recognition system is the classification of input data into identifiable classes. This is achieved by extracting various significant features (or attributes) of the data from a background of irrelevant detail, and using these features to "learn" the characteristics of the identifiable classes. These features are then used as the basis in the classification or testing process. Possible applications of pattern recognition include medical diagnosis, the identification of human fingerprints and remotely-sensed data, the analysis of biological macromolecules such as R.N.A. and proteins, and the detection and identification of strategic military targets.

The input for a pattern recognition system is usually obtained by taking "measurements" from the real world. For example, visual pattern recognition involves the identification of two-dimensional patterns in a gray-level array. In this setting, the specification of one or more pattern-classes is done either analytically or by the provision of

representative templates which model the individual classes. In most visual pattern recognition applications the latter is the methodology used in the specification phase.

In pattern recognition problems, a *pattern* is generally a structural description of an object, while a *pattern class* is a symbolic representation for a set of patterns which share a large number of common properties. The essential goal of the pattern recognition system is to assign the various incoming patterns to their respective classes. For example, consider the problem of designing a machine to recognize monetary notes of various denominations. In this case there are five pattern classes, one for each of the four different denominations (two, five, ten, and twenty) and a class to represent the action of rejecting the processed bill.

In order to generate patterns in a form acceptable to a machine, the sensed data is typically represented in the form of a *pattern vector* :

$$\mathbf{X} = \begin{bmatrix} x_1 \\ x_2 \\ \cdot \\ \cdot \\ x_n \end{bmatrix}$$

where n is the number of measurements. For example, consider the case when the measurement grid illustrated in Figure 1.1 is of size m by m . If the measurements which are taken are the "bit-values" of the grid array, we have the dimensionality of the feature vector to be $n = m^2$, and each component x_i of the vector $\mathbf{X}$ assumes the value 1 or 0, depending on the state of the i^{th} cell in the grid. The feature vector in this particular case is :

$$\mathbf{X} = [000000110001010011000000]^T$$

Figure 1.1 : Example of transforming an input pattern into a form suitable for machine manipulation. (a) Input character. (b) Measuring grid. (c) A resulting binary pattern.

To explain this further, consider the values of a sample stock shown in Figure 1.2. If this function is sampled at discrete points $t_1, t_2, \dots, t_n$, a pattern vector may be formed by assigning $x_1 = f(t_1)$, $x_2 = f(t_2)$, and so on, up to $x_n = f(t_n)$.

Figure 1.2 : Sampling of the values of a continuous function to yield a feature vector.

As opposed to the above two examples, in some applications, the characteristics of the components of interest in the individual patterns are best described by resorting to structural relationships. Consider the pattern shown in Figure 1.3(a). This structure could easily be encoded in the form of a pattern vector by using a grid such as the one illustrated in Figure 1.1. However, a more efficient representation (in terms of the amount of data required to coded the image) would be obtained if we define the primitive elements to be four symbols, namely a, b, c, and d as shown in Figure 1.3(b). Subsequently, the pattern itself is represented using these primitive symbols in the form shown in Figure 1.3(c). Notice that this representation is a one-dimensional symbolic representation, and thus, in this case, the string $X = \text{"abadabbccdd"}$ completely characterizes the original structure. Note that such a representation implicitly assumes that, in the coding process, the primitives a, b, c, and d are connected in a "head-to-tail" manner as shown in Figure 1.3(b).

Figure 1.3 : Description of the use of primitive elements in pattern recognition. (a) The Input pattern. (b) The Primitive elements defined. (c) The Pattern coded in terms of the individual primitives to yield the string representation "abadabbccdd".

Another example of a *string representation* of a pattern is shown in Figure 1.4. By assuming that the lengths of the primitives *a*, *b*, and *c* are equal (as shown in Figure 1.4(a)), equilateral triangles of various sizes can be represented by strings of the form $a^n b^n c^n$, where *n* is the number of primitives of each type.

Figure 1.4 : String representation of equilateral triangles. (From [GT78])

Apart from the advantages of representing "two-dimensional" geometric patterns using strings, various pattern recognition problems, which do not incorporate any preprocessing to encode the structural information, naturally involve string representations. For example, such an application could involve the automatic correction (and recognition) of misspelled words, substrings and subsequences ([SK83], [KO83b], [KO84], [KO87]).

String representations are suitable for describing objects whose structure is based on a relatively simple concept of neighbourhood. However, when the object or scene considered is more complicated, a more powerful representation must be used. One such representation involves *trees*.

We observe that any hierarchical ordering scheme naturally leads to a tree structure. For example, consider the image shown in Figure 1.5(a). Let us define the entire image area by the symbol '\$'. The tree representation obtained by using the relationship "inside" is shown in Figure 1.5(b) which indicates that the image can be partitioned into regions a and b. Furthermore, region a contains regions c and d, and finally, region b contains region e which in turns contains regions f and g.

(a)

(b)

Figure 1.5 : Example of the tree representation of an image which has been subject to smoothing and boundary tracking operations (From [GT78]).

Another example of a tree representation of a picture (or scene) is shown in Figure 1.6, in which the tree representation is obtained by using the relationship "composed of".

Figure 1.6 : The tree representation of a simple scene. (From [GT78]).

1.1 Decision-Theoretic and Syntactic Approaches to Pattern Recognition

Many different techniques have been used to solve pattern recognition problems. Generally speaking, these techniques can be grouped into two general approaches: the *decision-theoretic (or statistical) approach* and the *syntactic (or structural) approach*. In the decision-theoretic approach, the system processes the features that are extracted from the patterns, and assigns the pattern to be a member of one of a pre-defined set of classes. This assignment is typically achieved by partitioning the feature space. This partitioning process, as shown in Figure 1.7, is usually based on the utilization of decision functions

([GT78], [Fu82]) that classify the feature vectors of the form described earlier. For example, consider the feature space shown in Figure 1.7, which contains two pattern classes denoted by ω_1 and ω_2 . Since these two classes are linearly separable, if $X=[x_1, x_2]^T$, we can define a linear function $D(X)$ which achieves the classification as below :

$$D(X) = w_1x_1 + w_2x_2 + w_3 = 0.$$

Note that any pattern X from class ω_1 yields a positive value for $D(X)$, and that the patterns from class ω_2 yield negative values for $D(X)$. Here, $D(X)$ is called the *discriminant function*, and can be used as to implement the classification decision by using the following rule :

Conclude that : $X \in \omega_1$ if $D(X) > 0$, and,
 $X \in \omega_2$ if $D(X) < 0$.

Figure 1.7 : A simple decision function for two pattern classes. (From [GT78])

When there are more than two pattern classes, one approach is to establish M decision functions $\{D_1(X), D_2(X), \dots, D_M(X)\}$ which partition the space as below :

Conclude that $X \in \omega_i$ iff

$$D_i(X) > D_j(X) \quad j \neq i, j = 1, 2, \dots, M.$$

In the discriminant approach to pattern recognition, the aim is to determine the coefficients w_{ij} for each class, so that the foregoing decision rule will hold for as many patterns as possible. The learning phase of the classifier essentially deals with algorithms for computing these coefficients by estimating the various parameters of the distribution of X in each class.

A lot of literature is available in the area of statistical pattern recognition. However, as explained earlier, there are numerous applications when the structure of a pattern also plays an important role in the classification process. In these applications, the decision-theoretic approach is generally unsuitable because it lacks the formalism required for handling "symbolic" pattern structures and their relationships.

The syntactic approach to pattern recognition has received increased attention during the past decade because it possesses the "structure-handling" capability lacked by the decision-theoretic approach. Tools used in syntactic pattern recognition are often based on concepts from formal language theory which decompose patterns into subpatterns and each subpattern is again described in terms of even simpler subpatterns, and so on. Evidently, for this approach to be advantageous, the simplest subpatterns selected, called the *pattern primitives*, should be much easier to recognize than the patterns themselves. This scenario is depicted in Figure 1.8 [GT78]. Figure 1.8(b) shows such a decomposition of a submedian chromosome in terms of the primitives defined in Figure 1.8(a). By tracking the chromosome's boundary in a clockwise direction, it can be represented by the string "abcbabdbabcbabdb".

(a)

(b) String representation : "abcbaabdbabcbabdb".

Figure 1.8 : Example of decomposing the submedian chromosome into predefined primitives. (a) The primitives used. (b) The coded submedian chromosome (From [GT78]).

After each primitive within the pattern is identified, the recognition process is then accomplished by comparing the string representation of the pattern with the corresponding representation of representative elements of the individual classes. This can be done by computing a distance metric or by performing a syntax analysis (parsing) of the *sentence*.

In this way we can determine whether the given pattern is syntactically correct with respect to a specified *grammar* which, in turn, is used to represent one class of patterns.

1.2 Syntactic Pattern Recognition System

A syntactic pattern recognition system consists of two major components -- the *Analyzer* and the *Recognizer*. The analysis part consists of selecting the primitives and inferring the patterns which are representative of a particular class. The recognition phase consists of preprocessing the obtained patterns, recognizing the primitives and finally comparing this pattern with the individual representative patterns. A simple block diagram of the system is shown in Figure 1.9 below.

Figure 1.9 : Block diagram of a syntactic pattern recognition system.

Typically, the function of preprocessing includes : (i) pattern encoding and approximation, and (ii) filtering, restoration, and enhancement. An input pattern is first coded into some convenient form for further processing. Subsequently, the techniques of filtering, restoration, and/or enhancement are used to increase the signal-to-noise ratio, to

restore the degradation, and/or improve the quality of the encoded patterns. Each preprocessed pattern is then represented by a string, tree or graph, and is segmented or decomposed into subpatterns and pattern primitives based on the prespecified syntactic operations. The decision as to whether or not the pattern is syntactically correct (with respect to a specified grammar) is then made by the syntax analyzer.

Perhaps the simplest form of recognition is that which utilizes *template matching*. Based on a selected matching or similarity criterion, the input pattern is classified in the same class as the prototype pattern that is the "best" match to the input.

When the patterns are noisy and distorted, the following two possibilities could happen. The first possibility is that a noisy or distorted pattern is not accepted (recognized) by any pattern grammar. The second possibility is that patterns belonging to two or more different classes may appear to be the same. At least three approaches have been proposed to solve these problems :

- (i) the use of approximations in the original patterns,
- (ii) the use of the transformational grammars [KC72], and,
- (iii) the use of similarity and error-correcting parsing.

The first approach proposes to reduce the effect of noise and distortion by using approximation at the preprocessing and primitive extraction stages. The second approach attempts to define the relation between noisy patterns and their corresponding noise-free patterns by a transformational grammar. When a noisy or distorted pattern cannot be accepted by any pattern grammar, a similarity criterion or an error-correcting parsing procedure can be used to achieve the recognition. To accomplish this, various *similarity* measures between two strings have been defined in the literature, and thus the definition and computation of this measure is a fundamental part of designing the syntax analyzer.

Note that these fundamental concepts are still valid if the pattern structures were represented by a tree instead of a string. In this case, the recognition process is carried out

by considering tree similarities as opposed to string similarities. Thus, in the case when the patterns are noisy and distorted, the similarity measures between two trees will be used instead of the measures which compare strings. This is essentially the premise of this thesis.

1.3 Organization of the Thesis

In the last section we saw that in any syntactic pattern recognition system it is often necessary to describe the recognition process in terms of the comparison between strings or trees. There exist many papers related to the string editing (string correction) problem. Although this problem, the string editing problem (the one-dimensional version of the tree editing (tree correction) problem), has been well studied, the existing literature reports only one similarity measure relating trees. This measure is indeed the "distance" between two trees which is calculated on a metric analogous to the one defined by Levenshtein [Le67] and involves the way by which one tree can be edited into another using various elementary edit operations. More precisely, the first algorithm which calculated the distance between two trees was presented by Tai [Ta79] in 1979. Subsequently, most of the research in this area has been directed towards designing faster algorithms. As a result, this index for measuring similarity (dissimilarity) between trees has become the only one.

To extend the reported results in the area of tree editing, we first study many of the representative algorithms concerning the string editing problem. This will be done in Chapter II. Also, algorithms that study other measures relating the strings in question such as those involving their Longest Common Subsequence, and their Shortest Common Supersequence will be presented in the same chapter.

Due to various applications of the tree editing problem, there are two specific special cases encountered in tree editing. In the first, insertions and deletions are permitted only at the leaves, and in the second, these operations are permitted at any internal node (except the root) within the tree. In Chapter III, existing algorithms for each of these cases

will be considered. Also, the analysis of the time and space complexities of these algorithms will also be discussed.

In Chapter IV, various new numeric similarity and dissimilarity measures between two trees will be defined. Indeed, this will be achieved by presenting a mathematical foundation for the numerical similarity indices involving the two trees in question. This abstract measure is presented in terms of a set of elementary inter-symbol measures and two abstract operators $\oplus$ and $\otimes$. By appropriately choosing the concrete values for these two operators and for the elementary inter-symbol measures, a variety of numeric similarity and dissimilarity indices can be studied using the the same formulation. Apart from defining these measures, Chapter IV also discusses how they can be computed in an efficient manner.

Unlike the string editing problems, all existing tree editing algorithms consider the editing of one tree, say T_1 , and transforming it to another tree T_2 , with the edit processes being absolutely unconstrained. In Chapter V, we will consider a more complicated tree editing problem, namely, the Constrained Tree Editing problem. In this problem, the system has to compute the optimal way to edit T_1 into T_2 subject to the constraint that the number of individual edit operations done is specified by a user-defined constraint. This problem is studied in Chapter V, and a general fast algorithm which solves this problem is derived.

Chapter VI concludes the thesis with a summary of results and a discussion of some open problems.

II. String Editing Problems

2.1 Introduction

A common and interesting problem arising in syntactic pattern recognition ([GK82], [KO83a], [Oo87]), the comparison of biological macromolecules ([BSSU74], [WL83]), and in geological stratigraphic sequence comparison [SW80] is the computation of the *distance* or *degree of dissimilarity* between two strings. The earliest attempt to define the similarity between pairs of character strings can be dated back to 1931 [Al67]. Since then, various metrics for measuring the distance between strings have been proposed. In particular, let $X=x_1x_2..x_M$ and $Y=y_1y_2..y_M$ be two strings of length M , where x_i and y_i are the symbols at position i in the strings X and Y . Then three well-known metrics can be expressed as follows.

(i) The *Euclidean Distance* : $\sqrt{\sum_{i=1}^M (\text{Ord}[x_i] - \text{Ord}[y_i])^2}$,

(ii) The *City Block Distance* : $\sum_{i=1}^M |\text{Ord}[x_i] - \text{Ord}[y_i]|$, and,

(iii) The *Hamming Distance*, which measures the number of positions in which the symbols x_i and y_i are different.

Note that in the first two cases, whenever x_i and y_i are compared, their relative inter-symbol distance is expressed in terms of the difference between the ordinal values of the symbols themselves. In the case of Hamming distance, this relative inter-symbol distance is unity if $x_i \neq y_i$, and is zero whenever $x_i = y_i$.

In 1966, Levenshtein [Le66] introduced a new concept of a distance metric $D(X, Y)$ in terms of the smallest number of substitutions, insertions, and deletions required to transform the string X into the string Y . Subsequently, it has become the most widely used metric in scientific research in this field ([Ha88], [HD80], [Hi75], [KO82], [KO83a], [KO83b], [KO84], [LW75], [MP80], [My86], [Oo86], [Oo87], [SK83], [Ser74], [Uk85a], [Uk85b], [WF74], [WC76], [WMM91], [WMMM90]). In addition, related problems such as the determination of the length of the longest (maximal) common subsequence and the set of subsequences, or the set of smallest (minimal) common supersequences of two strings and their lengths, and the recognition of patterns (in substrings and subsequences) have also been studied extensively.

In this chapter, we give a brief review of the existing algorithms related to the string editing problem.

2.1.1 Notation and Elementary Edit Distances

The symbol $\mathcal{A}$ will be used to represent the alphabet, whether it be finite or infinite. $\mathcal{A}^*$ will be the set of all possible strings over $\mathcal{A}$. Let λ be the null symbol ($\lambda \notin \mathcal{A}$). Also, $\tilde{\mathcal{A}} = \mathcal{A} \cup \{\lambda\}$ is referred to as the Appended Alphabet.

A string $X \in \mathcal{A}^*$ of the form $X = x_1 x_2 \dots x_M$, where each $x_i \in \mathcal{A}$, is said to have length $|X| = M$. The prefix of the string X of length i ($i < M$) will be denoted by $X_i = x_1 \dots x_i$, and a substring of X consisting of characters from position i through j will be denoted by $X_{ij} = x_i \dots x_j$, where $1 \leq i < j \leq M$. Throughout this chapter, uppercase letters represent strings, and lowercase letters represent symbols of the alphabet under consideration. In most cases, we will use X and Y to represent the strings under consideration. Without loss of generality we shall assume that $|X| = M$, and that $|Y| = N$ with $M \leq N$.

An edit operation is a pair $(a, b) \neq (\lambda, \lambda)$, with $a, b \in \tilde{\mathcal{A}}$, (also written as $a \rightarrow b$) and is interpreted as the transformation of one occurrence of the symbol a to the symbol b .

Thus, for $a, b \in \mathcal{A}$, (a, b) represents the substitution operation, (a, λ) represents the deletion operation, and (λ, b) the insertion operation.

Let $d(\cdot, \cdot)$ be an arbitrary cost function which assigns to each edit operation $a \rightarrow b$, a non-negative real number $d(a, b)$. Then $d(a, b)$, $d(a, \lambda)$, and $d(\lambda, b)$ will represent the cost of the edit operations of substitution, deletion, and insertion respectively. Without loss of generality, we assume that

$$d(a, b) + d(b, c) \geq d(a, c).$$

This is essentially the triangle inequality, specifying that no sequence of multiple-step transformations can achieve a lower cost than a single operation.

2.1.2 Distance between Strings

Let S be a sequence $S_1, \dots, S_k$ of edit operations. We say that S takes X to Y if there is a sequence of strings $A^{(0)}, A^{(1)}, \dots, A^{(k)}$, where $A^{(0)} = X$ and $A^{(k)} = Y$, such that S_i transforms $A^{(i-1)}$ to $A^{(i)}$. In such a case, we write $X \xrightarrow{S} Y$ to represent such a sequence. We define the edit distance $D(X, Y)$ from string X to string Y as the minimum cost of all sequences of edit operations which transform X into Y . Formally, we define $D(X, Y)$ as :

$$D(X, Y) = \text{Min} \left\{ \sum_{(\alpha, \beta) \in S} d(\alpha, \beta) \mid X \xrightarrow{S} Y \right\}.$$

2.1.3 The set $G_{X,Y}$

Let Z be any element in $\tilde{\mathcal{A}}^*$, the set of strings over $\tilde{\mathcal{A}}$. The *compression operator* $C(\cdot)$ is a mapping from $\tilde{\mathcal{A}}^*$ to $\mathcal{A}^*$: $C(Z)$ is Z with all occurrences of the symbol λ removed from Z . Note that $C(\cdot)$ preserves the order of the non- λ symbols in Z . For example, if $Z = "d\lambda o\lambda o\lambda r"$, $C(Z) = "door"$.

For every pair (X, Y) , with $X, Y \in \mathcal{A}^*$, the finite set $G_{X,Y}$ is defined by means of the compression operator $C(\cdot)$, as a subset of $\tilde{\mathcal{A}}^* \times \tilde{\mathcal{A}}^*$:

$$G_{X,Y} = \{ (X', Y') \mid (X', Y') \in \tilde{\mathcal{A}}^* \times \tilde{\mathcal{A}}^* \} \text{ obeys}$$

- (i) $C(X') = X, C(Y') = Y,$
- (ii) $|X'| = |Y'|, \quad \text{and,}$
- (iii) $(x_i', y_i') \neq (\lambda, \lambda) \text{ for all } 1 \leq i \leq |X'|.$

The pair $(X', Y') \in G_{X,Y}$ corresponds to one way of editing X into Y , using the edit operations of substitution, deletion, and insertion. The edit operations themselves are specified for all $1 \leq i \leq |X'|$ by (x_i', y_i') , which represents the transformation of x_i' to y_i' . Hence, $G_{X,Y}$ is an exhaustive enumeration of the set of all the ways by which X can be edited to Y without destroying the order of occurrence of the symbols in X and Y . Note that the transformation of a symbol $a \in \mathcal{A}$ into itself is also considered as an operation in an arbitrary pair $(X', Y') \in G_{X,Y}$.

EXAMPLE 2.1

Let $X = "p"$ and $Y = "my"$, then

$$G_{X,Y} = \{(p\lambda, my), (\lambda p, my), (p\lambda\lambda, \lambda my), (\lambda p\lambda, m\lambda y), (\lambda\lambda p, my\lambda)\}.$$

In particular, the pair $(p\lambda, my)$ represents the edit operations of replacing the "p" by an "m" and inserting the "y".

Since the *distance* between X and Y is the minimum of the sum of the edit distances associated with the edit operations required to transform X to Y , this distance can be written as

$$D(X, Y) = \min_{((X', Y') \in G_{X,Y})} \left\{ \sum_{i=1}^{|X'|} d(x_i', y_i') \right\}$$

Since the transformation of a symbol $a \in \mathcal{A}$ to itself is considered as a substitution operation, an application in which this is not treated as an operation must specify $d(a, a)=0$.

2.2 The String-to-String Correction Problem

Many algorithms have been proposed for finding the edit distance $D(X, Y)$ between strings X and Y (see, for example, [LW75], [MP80], [Oo86], [Ser74] and [WF74]). Wagner and Fischer [WF74] presented an algorithm for evaluating this distance, and this algorithm was shown by Wong and Chandra [WC76] to be the optimal one for the infinite-alphabet case. The edit operations considered by Wagner and Fischer are the "standard" ones, namely, the substitution, insertion, and deletion operations. The execution time of their algorithm is proportional to the product of the lengths of the two strings involved. The heart of their algorithm will be presented in the next subsection.

2.2.1 The Optimal Algorithm for Infinite Alphabet

In order to compute the distance between two strings X and Y , a structure called a *Trace* is defined to describe how S transforms X into Y . This structure ignores the order in which the operations are done and is minimal in that it avoids redundancy. Let X and Y be the strings "abxcabdbdcab" and "cbyccaadcefafe" respectively. One possible trace involved in transforming X to Y is presented in Figure 2.1.

Figure 2.1 : An example of a trace.

In this figure, the line which connects character position i of X to position j of Y indicates that y_j is derived from x_i . Note that the transformation $x_i \rightarrow y_j$ is considered an operation even if $x_i = y_j$. Any character positions of X not connected by any lines will be deleted by S . Similarly, any positions of Y not connected by any lines will be inserted.

Formally, a Trace from X to Y is a triple (T, X, Y) , where T is any set of ordered pairs of integers (i, j) satisfying the following conditions :

- (i) $1 \leq i \leq |X|, \quad 1 \leq j \leq |Y|.$
- (ii) for any two distinct pairs (i_1, j_1) and (i_2, j_2) in T ,
 - (a) $i_1 \neq i_2$ and $j_1 \neq j_2.$
 - (b) $i_1 < i_2$ if and only if $j_1 < j_2.$

Condition (i) ensures that our lines actually connect character positions of the respective strings. Condition (ii)a ensures that each character position of the strings is connected by at most one line, and condition (ii)b ensures that no lines in T cross each other. We will not distinguish between the triple (T, X, Y) and the set of pairs in T . The cost of the trace T transforming X to Y is defined as follows :

$$\text{cost}(T) = \sum_{(i,j) \in T} d(x_i, y_j) + \sum_{i \in I'} d(x_i, \lambda) + \sum_{j \in J'} d(\lambda, y_j),$$

where I' and J' are the sets of positions in X and Y respectively, not connected by any lines in T . Thus, the cost of 'f' is just the cost of the edit sequence S taking X to Y . If we define:

$$D(X, Y) = \text{Min} \left\{ \text{cost}(T) \mid T \text{ is a trace from } X \text{ to } Y \right\}, \text{ and,}$$

$$H(i, j) = D(X_i, Y_j),$$

where $H(i, j)$ is the minimum edit distance of transformation X_i , the prefix of string X of length i , to Y_j , the prefix of string Y of length j , then $D(X, Y)$ is exactly $H(M, N)$ if $|X|=M$ and $|Y|=N$. Using this notation, $H(i, j)$ can be shown to obey the following recursive properties.

THEOREM 2.1

$$\begin{aligned}
H(0, 0) &= 0; \\
H(i, 0) &= \sum_{r=1}^i d(x_r, \lambda); & 1 \leq i \leq |X| \\
H(0, j) &= \sum_{r=1}^j d(\lambda, y_r); & 1 \leq j \leq |Y| \\
H(i, j) &= \text{Min} \begin{cases} H(i-1, j-1) + d(x_i, y_j) \\ H(i-1, j) + d(x_i, \lambda) \\ H(i, j-1) + d(\lambda, y_j) \end{cases} & 1 \leq i \leq |X|, 1 \leq j \leq |Y|
\end{aligned}$$

PROOF : See proofs of Theorems 2 and 3 in [WF74].

Theorem 2.1 leads to the following dynamic programming algorithm.

ALGORITHM Wagner and Fischer¹

INPUT : The strings X, Y, of lengths M and N respectively, and the set of elementary edit distances defined by d(...).

OUTPUT : The edit distance D(X, Y).

BEGIN

 H[0][0] := 0 ;

 FOR i := 1 to M DO

 H[i][0] := H[i-1][0] + d(x_i, λ) ;

 ENDFOR

 FOR j := 1 to N DO

 H[0][j] := H[0][j-1] + d(λ, y_j) ;

 ENDFOR

 FOR i := 1 to M DO

 FOR j := 1 to N DO

 H[i][j] = Min $\begin{cases} H[i-1][j-1] + d(x_i, y_j) \\ H[i-1][j] + d(x_i, \lambda) \\ H[i][j-1] + d(\lambda, y_j) \end{cases}$;

 ENDFOR

 ENDFOR

 D(X, Y) = H[M][N] ;

END ALGORITHM Wagner and Fischer.

¹Throughout this thesis, in all the *algorithms and procedures* which are formally written, the arrays will be specified using the notation that the array element H(i,j) is given by H[i][j].

The algorithm is best understood by means of a structure called a *Trellis* (see Figure 2.2) which is a cycle-free graph whose nodes correspond to the prefixes of the strings, (X_i, Y_j) , and whose edges correspond to the elementary edit operations. These edges are labeled with the cost of performing the elementary edit operations.

Figure 2.2 : The trellis to compute $D(X, Y)$.

In this trellis, there is a directed edge to the node (X_i, Y_j) from each of the nodes (X_{i-1}, Y_j) , (X_i, Y_{j-1}) and (X_{i-1}, Y_{j-1}) which are referred to as the nodes *adjacent* to (X_i, Y_j) . Note that horizontal edge from (X_i, Y_{j-1}) to (X_i, Y_j) represents the insertion of y_j , and the associated cost is $d(\lambda, y_j)$. The vertical edge from (X_{i-1}, Y_j) to (X_i, Y_j) represents the

deletion of x_i with the associated cost $d(x_i, \lambda)$, and the diagonal edge from (X_{i-1}, Y_{j-1}) to (X_i, Y_j) represents the substitution of x_i by y_j with the associated cost $d(x_i, y_j)$.

We now briefly explain the operation of Algorithm Wagner and Fischer.

The initialization step makes $H(0, 0) = 0$. The horizontal and vertical axes are then traced using the first two FOR-loops. Subsequently, the rest of the trellis is traced using the nested loops, and in each stage the computation uses only the quantities which have already been computed.

It is easy to see that the time complexity is $O(MN)$. Indeed, Algorithm Wagner and Fischer performs exactly $4MN + M + N + 1$ assignment operations. If the actual trace T yielding the minimal cost has to be computed, one can backtrack through the trellis starting from node (X_M, Y_N) and trace the optimal path to (X_0, Y_0) using the distance $H(i, j)$ stored in each node.

2.2.2 Operation of Interchanging Character Positions

In 1975, Lowerance and Wagner [LW75] extended the set of allowable edit operations to include the operation of interchanging the positions of two adjacent characters. In terms of the underlying traces this allows some lines in a trace to cross each other. In order to compute the cost of the trace recursively, they proved the following three theorems in which W_C , W_S , W_I and W_D represent the cost of substitution (changing), interchanging (swapping) positions, insertion, and deletion of a symbol, respectively. Also, a line in the trace is said to be balanced if the two symbols which it connects are identical.

THEOREM 2.2

If $2W_S \geq W_I + W_D$, then there is at least one minimal trace T from X to Y in which no line of T crosses more than one other line.

PROOF : See proof of Theorem 3 in [LW75].

THEOREM 2.3

If $2W_S \geq W_I + W_D$, then at least one minimal cost trace exists such that every line of T crosses at most one other line and that every line of T which crosses another line is balanced.

PROOF : See the proof of Theorem 4 in [LW75].

THEOREM 2.4

If $2W_S \geq W_I + W_D$, there is at least one minimal trace satisfying the above two theorems and which, if (u_1, u_2) and (v_1, v_2) are lines in T which cross with $u_1 < v_1$, then there is no integer i in the range $u_1 < i < v_1$ satisfying $x_{u_1} = x_i$. Similarly, there is also no integer j in the range $v_2 < j < u_2$ which satisfies $y_{v_2} = y_j$.

PROOF : See proof of Theorem 5 in [LW75].

We now concentrate on the computation of $D(X, Y)$ in the context of these expanded set of edit operations. Let $H(i, j)$ be the cost of the trace transforming string X_i to string Y_j . Then the above theorems lead to the following strategy for computing $H(i, j)$.

- i) If x_i is not connected by a line of T , then

$$H(i, j) = H(i-1, j) + W_D.$$

- ii) If y_j is not connected by a line of T , then

$$H(i, j) = H(i, j-1) + W_I.$$

- iii) If x_i and y_j are connected by the same line of T , then

$$H(i, j) = H(i-1, j-1) + W_C.$$

- iv) If x_i and y_j are connected by different lines of T , let (h, j) and (i, k) be the lines. Then,

$$H(i, j) = H(h-1, k-1) + (i-h-1) \cdot W_D + W_S + (j-k-1) \cdot W_I.$$

The logic which proves (i) - (iii) above is quite straightforward. To show (iv) we consider the trace shown in the Figure 2.3.

Figure 2.3 : Example trace of interchanging of symbols.

By Theorem 2.3, we know that $a_h = b_j$ and $a_i = b_k$. Furthermore, by Theorem 2.4, we know that the integer h must be the largest integer less than or equal to the integer i such that $a_h = b_j$. Similarly, the integer k must be the largest integer less than or equal to the integer j such that $a_i = b_k$. Thus, the cost of the trace $H(i, j)$ is equal to the sum of the following quantities :

- (a) the cost of the subtrace $H(h-1, k-1)$,
- (b) the cost of the deletion of $(i-h-1)$ elements in position $(h+1)$ through $(i-1)$ in string X ,
- (c) the cost of the insertion of $(j-k-1)$ elements in position $(k+1)$ through $(j-1)$ in string Y , and,
- (d) the cost of interchanging the elements in position h and i in string X (or interchanging the elements in position k and j in string Y).

Thus,

$$H(i, j) = H(h-1, k-1) + (i-h-1) \cdot W_D + W_S + (j-k-1) \cdot W_I.$$

The algorithm which computes the string edit distance based on the above arguments is formally given below.

ALGORITHM Lowerance and Wagner

INPUT : The strings X, Y , of length M and N respectively, and the set of elementary edit distances defined by $d(\dots)$.

OUTPUT : The edit distance $D(X, Y)$.

ASSUMPTION : Each character is represent by an integer. We use $X[i]$ and $Y[j]$ for the integers representing x_i and y_j respectively. We assume that the elements of $X[i]$ and $Y[j]$ are from the set $\{1, 2, \dots, C\}$.

BEGIN

FOR $i := 0$ to M DO
 $H[i][0] := i * W_D$;
 $H[i][-1] := \infty$;

ENDFOR

FOR $j := 1$ to N DO
 $H[0][j] := j * W_I$;
 $H[-1][j] := \infty$;

ENDFOR

FOR $d := 1$ to C DO /* $DA[c]$ holds the largest $h \leq i-1$ such that
 $DA[d] := 0$; $X[h] = c$, for every character c . */

ENDFOR

FOR $i := 1$ to M DO

DB := 0; /* DB holds the largest $k \leq j-1$ such that $Y[k] = X[i]$ */

FOR $j := 1$ to N DO

h := $DA[B[j]]$; /* position in string X for splitting the trace. */

k := DB; /* position in string Y for splitting the trace. */

IF $X[i] = Y[j]$ THEN

d := 0;

ELSE

d := W_C ;

ENDIF

$$H[i][j] := \text{Min} \begin{cases} H[i-1][j-1] + d \\ H[i][j-1] + W_I \\ H[i-1][j] + W_D \\ H[i_1-1][j_1-1] + (i-h-1)*W_D + W_S + (j-k-1)*W_I \end{cases};$$

ENDFOR

$DA[A[i]] := i$; /* update the value of $DA[]$. */

ENDFOR

END ALGORITHM Lowerance and Wagner.

The running time of the Algorithm Lowerance and Wagner is $O(MN)$. If the actual sequence of edit operations is needed, it can be derived by backtracking through the array $H(\dots)$, and tracing through the choices made by the algorithm in the forward pass.

2.2.3 A Linear Space Algorithm

Both the algorithms mentioned above require quadratic time and space (measured in terms of the size of the input strings). In 1975, Hirschberg [Hi75] presented an algorithm similar to Algorithm Wagner and Fischer, which solved this problem in quadratic time and *linear* space. We briefly explain this algorithm below.

To compute the entries in the trellis, the Algorithm Wagner and Fischer [WF74] maintains a two-dimensional array. However, since the value of the entry $H(i, j)$ depends only on the values of $H(i-1, j)$, $H(i-1, j-1)$ and $H(i, j-1)$ for any i and j , it is sufficient to maintain only two rows of the array. The values of the elements corresponding to the nodes of the trellis are evaluated row by row using the previously computed rows to yield the pertinent values for the present rows. Thus, this improved algorithm requires only $O(N)$ space. This algorithm is presented as Algorithm Hirschberg below.

ALGORITHM Hirschberg

INPUT : The strings X, Y, of lengths M and N respectively, and the set of elementary edit distances defined by $d(\dots)$.

OUTPUT : The edit distance $D(X, Y)$.

BEGIN

FOR j := 0 to N DO

$$K[1][j] := 0;$$

```
/* initialize */
```

ENDFOR

FOR i := 1 to M DO

FOR j := 0 to N DO

$$K[0][j] := K[1][j];$$

```
/* copy the previous row */
```

ENDFOR

FOR j := 1 to N DO

$$K[1][j] := \text{Min} \begin{cases} K[1][j-1] + d(\lambda, y_j) \\ K[0][j] + d(x_i, \lambda) \\ K[0][j-1] + d(x_i, y_j) \end{cases} ; \quad /* \text{ update the new row } */$$

ENDFOR

ENDFOR
$$D(X, Y) = K[1][N];$$

END ALGORITHM Hirschberg

We shall later see how this same algorithm can be used to compute the *Longest Common Subsequence (LCS)* between X and Y .

2.2.4 Faster Algorithms for Finite Alphabets

To reduce the running time (as opposed to the space required by the classical Wagner and Fischer's algorithm), Masek and Paterson [MP80] presented a faster algorithm in 1980 which imposed certain constraints on the alphabet size and on the cost of individual edit operations. The main idea motivating their strategy is based on the "Four Russians Algorithm" used to compute the transitive closure of a directed graph [ADKF70]. This algorithm (i.e. the "Four Russians Algorithm") is based on the classical divide-and-conquer paradigm. Thus, whereas the transitive closure of a directed graph with n nodes can be easily computed with an $n \times n$ matrix using $O(N^2)$ row operations, the new algorithm requires only $O(\frac{N^2}{\log N})$ row operations.

Masek and Paterson's algorithm to compute the Wagner and Fischer's [WF74] edit matrix involves the following steps. Initially, all possible $(m+1) \times (m+1)$ submatrices which can occur in the original matrix are computed for a suitably chosen parameter m . Subsequently these submatrices are combined to construct the overall matrix, from which the edit cost is evaluated. In [MP80] Masek and Paterson present an algorithm to compute the edit distance between two strings with a running time of $O(\frac{MN}{\log N})$. However, to achieve this time bound, the algorithm requires that the following two constraints are satisfied :

- (i) the size of the alphabet must be finite.
- (ii) the domain for the cost function must be discrete, implying that the cost associated with every edit operation must be an integral multiple of a single positive real number.

With regard to the space complexity of the algorithm, it requires $O(\frac{MN}{\log N})$ space to both compute and store the desired results.

Although it was shown by Wong and Chandra [WC76] that $O(MN)$ is the lower-bound of the time complexity of the string editing algorithm when the alphabet size is *infinite*, one can conclude from the above discussion that faster algorithms can be developed when the size of the alphabet is restricted. Ukkonen [Uk85b], and Myers [My86] presented algorithms with a worst-case time complexity of $O(ND)$, where D is the edit distance between the strings X and Y . These algorithms are efficient when X and Y are similar. Myers further showed how to refine his result so to require $O(N \log N + D^2)$ time in the worst case using suffix trees. Hadlock [Ha88], and Wu *et. al.* [WMMM90] improved the running time marginally to yield an $O(NP)$ algorithm, where P is the number of deletions in the shortest edit script consisting only of insertions and deletions. These two algorithms are faster than the reported $O(ND)$ algorithms when $|X|$ is much smaller than $|Y|$. Recently, Wu *et. al.* [WMM91] presented an algorithm with a worst-case running time $O(N + \frac{ND}{\log N})$, which runs in "almost" linear time when X and Y are similar.

Although it is interesting to investigate these "restricted" algorithms, we omit the details of these algorithms and consider the more general case where the size of the alphabet is unknown *a priori*. This is the scenario encountered in the comparison of files, in which entire lines are considered as elements of the alphabet, and the file itself is considered as a string of lines.

2.2.5 The Constrained String Editing Problem

All of the algorithms mentioned above consider the edit process to be absolutely unconstrained with regard to the number of edit operations that are used. Sankoff [Sa72] pioneered the study of constrained string editing involving a specialized constraint which can be applied to the comparison of amino acid sequences. Oommen [Oo86] studied the

constrained string editing problem in its generality and presented the first known algorithm to compute the constrained distance between two strings X and Y subject to enforcing constraints on the number and type of edit operations which can be used in the edit process. We will now present the main idea motivating Oommen's algorithm.

Suppose we are editing a prefix of X with $|X| = M$, into a prefix of Y with $|Y| = N$, using exactly i insertions, e deletions, and s substitutions. This corresponds to editing $X_r = X_{e+s}$, the prefix of X , into $Y_q = Y_{i+s}$, the prefix of Y . The magnitudes of these variables, i , e , and s , are bounded by :

$$\begin{aligned} \text{Max } \{0, N-M\} &\leq i \leq q \leq N; \\ 0 &\leq e \leq r \leq M; \\ 0 &\leq s \leq \text{Min } \{N, M\}. \end{aligned}$$

Values of triples (i, e, s) which satisfy these constraints are termed *feasible values* of the variables. Let

$$\begin{aligned} H_i &= \{ j \mid \text{Max } \{0, N-M\} \leq j \leq N \}; \\ H_e &= \{ j \mid 0 \leq j \leq M \}; \\ H_s &= \{ j \mid 0 \leq j \leq \text{Min}\{M, N\} \}. \end{aligned}$$

These three sets, H_i , H_e , and H_s are called the set of *permissible values* of i , e , and s respectively. Hence, to edit X_r to Y_q , the set of feasible triples is given by :

$$\{ (i, r-q+i, q-i) \mid \text{Max } \{0, q-r\} \leq i \leq q \}.$$

The above is true because the number of insertions which can occur can take any value from $\text{Max } \{0, q-r\}$ to q , and furthermore, for every value of i in this range, the facts that $(i+s) = q$ and $(e+s) = r$ imply that $s = (q-i)$ and $e = (r-s) = (r-q+i)$.

Apart from defining the set of feasible triples in terms of the number of insertions, Oommen showed that any arbitrary edit constraint, τ , can be expressed as a unique subset

of H_i . The reason for this has been derived in [Oo86], but we shall omit the proof of this claim here because the analogous (generalized) claim for trees will be proved in Chapter V. However, an example will help clarify the concept.

EXAMPLE 2.2 :

Let $X = \text{"boy"}$ and $Y = \text{"hi"}$, and the constraint be :

"Edit X to Y using at least 1 insertion, at most 1 substitution, and exactly 2 deletion operations."

Then it can be shown that every edit sequence which achieves this must perform *exactly* one insertion operation. Thus,

$$\tau = \{1\}. \quad ***$$

Let $D_\tau(X, Y)$ represent the edit distance subject to the constraint τ , and let $W(i, e, s)$ be edit distance subject to the constraint that exactly i insertions, e deletions, and s substitutions are performed in the process of editing. We now state a theorem due to Oommen [Oo86].

THEOREM 2.5

$$W(i, e, s) = \text{Min} \begin{cases} W(i-1, e, s) + d(\lambda, y_{i+s}) \\ W(i, e-1, s) + d(x_{e+s}, \lambda) \\ W(i, e, s-1) + d(x_{e+s}, y_{i+s}) \end{cases}$$

$$\text{and } D_\tau(X, Y) = \text{Min} \{ W(i, M-N+i, N-i) \mid i \in \tau \}.$$

Proof : See proofs of Theorems 3 and 4 in [Oo86].

Theorem 2.5 leads to an interesting dynamic programming algorithm for computing the constrained string editing distance. It was also shown in [Oo86] that the time and space complexities of the algorithm were both equal to $O(MN * \text{Min}\{M, N\})$. Again, by

backtracking through the array of constrained distances, the best edit sequence to achieve this distance can be obtained by tracing the path traversed in the forward pass when the array $W(\dots)$ is constructed.

An improved version of this algorithm was later presented by Oommen [Oo87]. In order to reduce the space complexity of the algorithm, four two-dimensional arrays (instead of a single three-dimensional array) were used to keep track of the values of the three-dimensional trellis. The space complexity of the improved algorithm can be shown to be $O(MN)$. Oommen also showed that the time complexity of the improved version could be reduced to $O(N^2 + t^2N)$ when M and N are the length of the strings X and Y respectively with $M \leq N$, and t is the largest element in the set τ . Thus, if the set τ contains only one element T and $T \ll N$, the time complexity of the algorithm is $O(N^2)$.

The constrained string editing problem for the finite alphabet case remains unsolved.

We shall later consider the application of this algorithm in correcting noisy subsequences. Also, in Chapter V, we shall consider the problem of editing trees subject to arbitrary constraints specified as in this section.

2.3 Problems involving Subsequences and Noisy Subsequences

A string $Z \in \mathcal{A}^*$ is a *subsequence* of a string X if it can be obtained from X by deleting zero or more symbols (not necessarily contiguous) from X . Equivalently, $Z = z_1 z_2 \dots z_p$ is a *subsequence* of string $X = x_1 x_2 \dots x_M$ if and only if there is a strictly monotonically increasing mapping F satisfying

$$F : \{1, 2, \dots, p\} \rightarrow \{1, 2, \dots, M\} \text{ such that } f(i) = k \text{ only if } z_i = x_k.$$

A string Z is a *common subsequence* of two strings X and Y if it is a subsequence of both X and Y ; it is a *Longest Common Subsequence (LCS)* of two strings X and Y if and only if it is a common subsequence of both and has the largest

length among all the common subsequences. The length of the LCS is referred to as the ***Length of the Longest Common Subsequence (LLCS)***. For example, "fh" and "ft" are both LCSs of "fight" and "father", and the LLCS of these strings is 2.

Applications of LCS can be found in data processing and in the study of genetic evolution. In data processing, the LCS is often used to measure the dissimilarity between two files of data. Thus, for example, as described in Section 2.2.4, if we consider a file to be a string in which each line of the file is treated as a single symbol, the LCS of two files identifies positions of the files that are identical. In the study of genetic evolution, the LCS is used to measure the correlation between two long molecules such as proteins and RNA molecules ([SK83]). Due to the varied applications of LCS, many different algorithms to compute the LCS have been proposed by different researchers during the last few decades ([AHD76], [AG87], [Hi75], [Hi77], [Hi78], [HS77], [KO83b], [KT82], [Ma78], [Mu80], [Oo87], [Uk85a], [WF74], [WMMM90]). We shall now briefly review some of these algorithms in the following subsections.

2.3.1 An Optimal Algorithm and a Linear Space Algorithm

The Wagner and Fischer's algorithm (see Section 2.2.1) and the algorithm due to Hirschberg can both be used to find the LCS of two strings. Furthermore, Aho *et. al.* [AHD76] proved that the algorithm due to Wagner and Fischer [WF74] is optimal for the infinite-alphabet case of the LCS problem just as it is optimal for the string editing problem under identical settings. The applicability of Wagner and Fischer's algorithm to solve the LCS problem will now be clarified.

If P is the length of the LCS of strings X and Y , then it is clear from the definition of a trace (refer to Section 2.2.1) that P is also the maximum number of pairs (i, j) in any trace from X to Y for which $x_i = y_j$. Let T be such a trace and let $d(a, b) = 0$ if $a = b$ and $d(a, b) = 2$ if $a \neq b$. Furthermore, let $d(a, \lambda)$ and $d(\lambda, a)$ be equal to 1. Then

$$\text{cost}(T) = |X| + |Y| - 2P.$$

Since T is the least cost trace from X to Y , $\text{cost}(T) = D(X, Y)$. Hence,

$$P = \frac{|X| + |Y| - D(X, Y)}{2},$$

which is indeed the length of the LCS between X and Y .

Clearly this quantity can be computed in time $O(MN)$ and space $O(MN)$. Furthermore, any single LCS can also be found easily from T by backtracking through the array of edit distances maintained by the algorithm.

As opposed to the above, a "different" algorithm can be devised to calculate the length of the LCS by marginally modifying Algorithm Wagner and Fischer. This is seen by studying the following algorithm due to Hirschberg.

ALGORITHM Hirschberg LCS_Version

INPUT : The strings X, Y .

OUTPUT : The length of their longest common sequence, $\text{LCS}(X, Y)$.

BEGIN

FOR $i = 1$ to M DO

FOR $j = 0$ to N DO

$K[0][j] = K[1][j]$; /* maintain the previous calculated row */

ENDFOR

FOR $j = 1$ to N DO /* start to calculate the new row */

IF $x_i = y_j$ THEN

$K[1][j] = K[0][j-1] + 1$;

ELSE

$K[1][j] = \text{Max} \{K[1][j-1], K[0][j]\}$; /* insertion or deletion */

ENDIF

ENDFOR

ENDFOR

$\text{LCS} := K[1][N]$;

END ALGORITHM Hirschberg_LCS_Version.

The space required for the above algorithm is linear since two one-dimensional arrays have been used instead of a single two-dimensional array to represent the trellis. It is also clear that this algorithm requires $O(MN)$ time. An algorithm to recover an actual LCS itself after the forward pass has also been proposed by Hirschberg [Hi75]. In the interest of brevity we omit the details of this algorithm and refer the reader to [Hi75].

2.3.2 Time Bounds for the LCS Problem and Faster Algorithms

Over the years, many researchers have attempted to derive faster algorithms for the LCS problem. In 1976, Aho *et. al.* [AHD76] derived some lower-bound results and showed that if an algorithm was restricted to making "equal-unequal" comparisons of symbols in the various positions of the two strings, then $O(MN)$ such comparisons would be required to solve the LCS problem for the unrestricted alphabet size and $O(NS)$ such comparisons would be required for the case when the alphabet size is restricted to S , where $|X| = M \leq N = |Y|$. Hirschberg [Hi78], on the other hand, showed that if the type of comparison was restricted to "less than - equal to - greater than" operations, then only $O(N \log N)$ comparisons would be required in the unrestricted alphabet case when $M \leq N$.

In 1977, an algorithm which achieved this lower time bound for most of the inputs, but which required $O(N^2 \log N)$ time in the worst case, was presented by Hunt and Szymanski [HS77]. We present below the main idea motivating their algorithm.

Let $T_{i,k}$ be the smallest j such that the prefix of strings X and Y of length i and j (where $A_{1,i} = A_i$ and $B_{1,j} = B_j$) contains a common subsequence of length k . Thus, for example, if $X = \text{"abcbdd"}$, $Y = \text{"badbab"}$, we have :

$$T_{5,1} = 1, \quad T_{5,2} = 3, \quad T_{5,3} = 6 \quad \text{with} \quad T_{5,4} \text{ being undefined.}$$

It is easy to see that $T_{i,k-1} < T_{i+1,k} \leq T_{i,k}$ since if X_i and $Y_{T_{i,k}}$ have a common subsequence of length k , so do X_{i+1} and $Y_{T_{i,k}}$, making $T_{i+1,k} \leq T_{i,k}$. Also, if X_{i+1} and $Y_{T_{i+1,k}}$ have a common subsequence of length k , then deleting the last element from each

of these sequences removes at most one element from this common subsequence. Thus, X_i and $Y_{T_{i+1,k}-1}$ have common subsequence of length $k-1$. Accordingly, $T_{i,k-1} \leq T_{i+1,k} - 1$ and $T_{i,k-1} \leq T_{i+1,k}$.

We can therefore compute $T_{i+1,k}$ from $T_{i,k-1}$ and $T_{i,k}$ by using the condition that the value of $T_{i+1,k}$ will be equal to the smallest value of j such that $x_{i+1}=y_j$ with the additional constraint that $T_{i,k-1} < j \leq T_{i,k}$ if such a value exists. Otherwise, it will equal $T_{i,k}$. This leads us to the algorithm presented below.

ALGORITHM Hunt and Szymanski

INPUT : The strings X, Y , with lengths both equal to N .

OUTPUT : The length of their longest common subsequence, $LCS(X, Y)$.

BEGIN

$T[0] := 0$; /* initialize $T[k]$ */

 FOR $i = 1$ to N DO

$T[i] = N+1$;

 ENDFOR

 FOR $i = 1$ to N DO

 FOR $j = N$ downto 1 DO

 IF $x_i = y_j$ THEN /* $T[k] = T_{i-1,k}$ at the start of the iteration on i */

 Find k such that $T[k-1] < j \leq T[k]$;

$T[k] := j$; /* $T[k] = T_{i,k}$ at the end of the iteration on i */

 ENDIF

 ENDFOR

 ENDFOR

 Print largest k such that $T[k] \neq N+1$;

END ALGORITHM Hunt and Szymanski.

Observe that the direction of traversing the j -loop is reversed (j is decremented). This is mandatory, because, if for some i , a_i matches several different elements of Y in a given T -interval (say $y_{j_1}, \dots, y_{j_m}$ with $T[k-1] = T_{i-1,k-1} < j_1 < \dots < j_m \leq T_{i-1,k} = T[k]$), then $T_{i,k} = j_1$. Thus, $T[k]$ should be updated to this final value. Reversing the traversal of

the j -loop will cause $T[k]$ to take on successively smaller values until it is assigned the desired value of j_1 .

Since the array T is monotonically increasing, we can use a binary search to implement the "find" operation in $O(\log N)$ time, and thus Algorithm Hunt and Szymanski requires $O(N^2 \log N)$ time. If a list of the corresponding j positions (such that $x_i = y_j$) for each position i can be obtained by preprocessing, the time complexity of the algorithm can be improved to $O((R+N) \log N)$, where R is the total number of ordered pairs of the positions at which the two strings match. This *enhanced* technique is described formally below.

ALGORITHM Hunt and Szymanski Enhanced Version

INPUT : The strings X and Y , with lengths both equal to N .

OUTPUT : The length of their longest common sequence, $LCS(X, Y)$.

PREPROCESSING

```
FOR i = 1 to N DO
    set matchlist [i] = (j1, j2, ..., jp)
    such that j1 > j2 > ... > jp and xi = yjq for 1 ≤ q ≤ p ;
```

```
ENDFOR
```

BEGIN /* Main Algorithm */

```
T[0] = 0 ; /* initialize */
```

```
FOR i = 1 to N DO
```

```
    T[i] = N+1 ;
```

```
ENDFOR
```

```
FOR i = 1 to N DO /* main body */
```

```
    FOR j ∈ matchlist [i] DO
```

```
        Find k such that T[k-1] < j ≤ T[k] ;
```

```
        If j < T[k] THEN
```

```
            T[k] := j ;
```

```
        ENDIF
```

```
    ENDFOR
```

```
ENDFOR
```

```
Print largest k such that T[k] ≠ N+1 ;
```

END ALGORITHM Hunt and Szymanski Enhanced Version.

The preprocessing step requires $O(N \log N)$ time and $O(N)$ space, since we can implement it by sorting each string while keeping track of each element's original position. We can then merge the sorted sequences and create the required "matchlist". The initialization step requires $O(N)$ time and space for the initialization of the T-array. For the main body of the algorithm, the two outer loops should be considered as a single loop over all pairs (i, j) such that $x_i = y_j$. This requires $O(R \log N)$ time and $O(R)$ space. The final step clearly requires $O(N)$ time and $O(N)$ space. Thus, this "enhanced" algorithm requires $O((R+N) \log N)$ time and $O(R+N)$ space.

It was shown by Hunt and Szymanski [HS77] that their algorithm can be implemented to have a running time of $O(R \log \log N + N \log N)$ for the infinite alphabet case, and $O((N+R) \log \log N)$ when the alphabet is fixed and finite. Details of this implementation can be found in [HS77].

Although Aho *et. al.* [AHO76] suggested the existence of linear time algorithms for the LCS problem when the alphabet size is fixed, algorithms published prior to their paper generally required a running time which, in their worst cases, are marginally less than $O(MN)$. However, the algorithm due to Hunt and Szymanski [HS77] runs in $O((R+N) \log N)$ time, where R is the total number of ordered pairs of positions at which the two strings match. Apostolico and Guerra [AG87] and Mukhopadhyay [Mu80] have also presented algorithms with running time complexities smaller than $O(MN)$. Hirschberg [Hi77], on the other hand, presented two different algorithms which require $O(PN + N \log N)$ and $O(P(M+1-P) \log N)$ time, respectively, where P is the length of the LCS of X and Y . The former runs in almost linear time when P is small, and the latter runs in time of $O(M \log N)$ time when P is large. One of the best algorithms reported is due to Nakatsu *et. al.* [NKY82] which has a time complexity of $O(N(M-P))$. It thus runs in almost linear time when P is close to M . The details of these algorithms are omitted since this is not the primary focus of this thesis.

2.3.3 Recognition of Noisy Subsequence

All the algorithms discussed in the previous subsections are for the determination of the LCS of two strings and hence they can be said to solve the "*The Exact Subsequence Determination Problem*". A closely related problem is described below.

Given a finite dictionary $\mathcal{H}$, $\mathcal{H} \subseteq \mathcal{A}^*$, and a string $Z \in \mathcal{A}^*$ (not necessarily in $\mathcal{H}$), compute the set of strings $\mathcal{S}$, $\mathcal{S} \subseteq \mathcal{H}$, such that Z is a subsequence of each string $X_i \in \mathcal{S}$.

A straightforward solution to this problem can be easily given. We can verify that $X \in \mathcal{H}$ is in $\mathcal{S}$ by merely checking if Z is a subsequence of X by using the property defined in Section 2.3. Thus, for a given pairs of strings X and Z , a linear time algorithm for testing if Z is a subsequence of X can be described as below.

ALGORITHM Subsequence Testing

INPUT : Two strings X and Z .

OUTPUT : True if Z is a subsequence of X , False otherwise.

BEGIN

```

    j = 1 ; k = 1 ;
    WHILE (k ≤ |X|) and (j ≤ |Z|) DO
        IF  $x_k \neq z_j$  THEN
            k := k + 1 ;
        ELSE
            k := k + 1 ;      j = j + 1 ;
        ENDIF
    ENDWHILE
    IF j = |Z| + 1 THEN
        RETURN (True) ;
    ELSE
        RETURN (False) ;
    ENDIF

```

END ALGORITHM Subsequence_Testing.

Proof of the correctness of Algorithm Subsequence Testing is as follows : If Z is a subsequence of X , by definition, the mapping F exists such that there exist integers

$k_1 < k_2 < \dots < k_p$, with $p = |Z|$ such that $x_{k_1} = z_1, x_{k_2} = z_2, \dots, x_{k_p} = z_p$. The value of j in the algorithm will then be increased by unity exactly p times and thus the algorithm would return the value "True". Conversely, the value "True" returned by the algorithm implies that there are exactly $p = |Z|$ different positions in X , say $k_1, k_2, \dots, k_p$, such that $x_{k_1} = z_1, \dots, x_{k_p} = z_p$ and thus X contains a subsequence which is Z . Notice that this algorithm has an $O(|X|)$ time complexity implying that the algorithm is optimal.

We shall now consider a more general version of the above problem, namely the scenario when the string Z is permitted to be noisy. If Z is a noisy version of a subsequence of some string $X \in \mathcal{H}$ containing insertion, deletion and substitution errors, then the problem is far more complicated since it may be the case that no string $X \in \mathcal{H}$ contains a subsequence which is equal to Z . Furthermore, since every string X has exactly $2^{|X|}$ possible subsequences, Z could be a noisy version of any one of them. The only algorithms presented in the literature to solve this problem are the original algorithm due to Oommen [Oo87] and its improved version [OF91]. These papers compute the best estimate of the string X using the constrained edit distance described in Section 2.2.5. The latter distance is utilized as described below.

Let Z be the noisy version of a subsequence of an unknown string $X^* \in \mathcal{H}$ where the noise is assumed to be induced due to transmission over a noisy channel. Since Z can be any arbitrary subsequence of any X , and since the words of the dictionary can be of different lengths, it is meaningless to compare Z to every X in $\mathcal{H}$ using the general string-to-string edit distance. Thus, before we can compare Z to the individual words of $\mathcal{H}$, we must be able to use additional information that helps us characterize the properties of the noisy channel.

Since the number of insertions introduced in any transmission is unknown, it is reasonable to compare $X \in \mathcal{H}$ and Z subject to the constraint that the number of insertions that actually took place is its best estimate. This estimate, in the absence of any other information, is the expected value of the number of insertions per transmission. Let L be

the expected number of insertions introduced by the channel. If L is a feasible value for editing X to Z , we set $\tau = \{L\}$. On the other hand, if L is not a feasible value for the number of insertions, we set $\tau = \{i\}$, where i is the closest feasible value to L . The best estimate of X^* in $\mathcal{H}$ is now obtained by computing $D_\tau(X, Z)$ for every $X \in \mathcal{H}$. This algorithm is presented below.

ALGORITHM Recognize_Noisy_Subsequences

INPUT : $\mathcal{H}$, L , and Y , where $\mathcal{H}$ is the finite dictionary, L is the expected number of insertions that can take place per transmission, and Y is the noisy version of a subsequence of an unknown $X^* \in \mathcal{H}$.

OUTPUT : The estimate X^+ of X^* .

BEGIN

FOR every string $X \in \mathcal{H}$ DO

IF L is a feasible value THEN

$\tau := \{L\}$;

ELSE

$\tau := \{i \mid i \text{ is the closest feasible integer to } L.\}$;

ENDIF

Compute $D_\tau(X, Y)$ using Theorem 2.5 as shown in [Oo86] ;

ENDFOR

X^+ is the string minimizing $D_\tau(X, Y)$;

END ALGORITHM Recognize_Noisy_Subsequences.

The experimental results shown in [Oo87] demonstrate the accuracy of this scheme. As an example, the algorithm was tested for a thousand noisy strings of lengths ranging between 40 to 60 which contained 26.55 errors per string on the average. Of these, 995 were correctly recognized, representing an accuracy of 99.5 percent. In a more rigorous test, the algorithm was tested for 500 noisy strings which contain an average number of 44.72 errors per string. In this case 477 were correctly recognized, representing an accuracy of 95.4 percent.

2.4 Problems on Supersequences

A similar but dual problem of the LCS problem is the one of finding the Shortest Common Supersequence (SCS) of two strings. A string $Z \in \mathcal{A}^*$ is a *supersequence* of $X \in \mathcal{A}^*$ if it can be obtained from X by inserting zero or more symbols into it. It is a *common supersequence* of two strings X and Y if it is a supersequence of both X and Y , and it is a *shortest common supersequence (SCS)* of two strings X and Y if and only if it is a common supersequence of both, and is the shortest of all such common supersequences. The length is referred to as the *Length of the Shortest Common Supersequence (LSCS)*. For example, "abbab" is a SCS of strings "abbb" and "bab" and the LSCS of these strings is 5.

The complexity of the SCS problem for multiple strings was analyzed by Maier [Ma78]. As of date, little work has been done on the SCS problem ([KO83b], [Ma78] and [SK83]). To the best of our knowledge, only one algorithm has been reported to compute both the length of the SCS and the Set of Common Supersequences for the two input strings. The basis for this algorithm proposed by Kashyap and Oommen [KO83b] is summarized in the following Theorem.

THEOREM 2.6

Algorithm Wagner and Fischer can be used to compute the length of SCS of two strings if we use the following cost function for calculating the cost of the edit operations :

$$\begin{aligned} d(a, b) &= 1 && \text{if } a = b; a, b, \in \mathcal{A} ; \\ d(a, b) &= \infty && \text{if } a \neq b; a, b \in \mathcal{A} ; \\ d(a, \lambda) &= d(\lambda, a) = 1 && \text{for all } a \in \mathcal{A} ; \end{aligned}$$

Proof : See proof of Theorem 6 in [KO83b].

Thus, we can compute the length of the SCS for two strings of length M and N in $O(MN)$ time and linear space.

It was also shown in [KO83b] that the LLCS, the Length of the Longest Common Subsequence, and the LSCS, the Length of the Shortest Common Supersequence, of X and Y are related as in equation (2.1) :

$$\text{LSCS}(X, Y) = |X| + |Y| - \text{LLCS}(X, Y). \quad (2.1)$$

Thus any of the algorithms used to compute the LLCS can also be used to compute the LSCS. Furthermore, the actual SCS itself can be obtained by backtracking through the trellis used to compute the lengths of the supersequences. Alternatively (and more powerfully), the Set of Common Supersequences can be obtained using the method presented by Kashyap and Oommen [KO83b] which is described below.

Let " $\cup$ " be the set union operator, and " $\cdot$ " be the set concatenation operator where if Q_1 and Q_2 are subsets of $\mathcal{A}^*$ then $Q_1 \cdot Q_2 = \{XY \mid X \in Q_1, Y \in Q_2\}$, and XY is the string concatenation of X and Y . Furthermore, let $D_s(X, Y)$ represent the set of all Common Supersequences of X and Y . Also, let us define :

$$\begin{aligned} d_s(a, b) &= \{a\} & \text{if } a = b; a, b, \in \mathcal{A}; \\ d_s(a, b) &= \phi & \text{if } a \neq b; a, b \in \mathcal{A}; \\ d_s(a, \lambda) &= d_s(\lambda, a) = \{a\} & \text{for all } a \in \mathcal{A}; \end{aligned} \quad (2.2)$$

where ϕ is the identity of the union operator " $\cup$ ". In this case, the following algorithm can be used to compute the set of all supersequences whose lengths are less than or equal to the sum of $|X|$ and $|Y|$.

The set of all SCS of the strings can be obtained by extracting from $D_s(X_M, Y_N)$ the strings with the smallest length.

ALGORITHM Compute the Set of Supersequences

INPUT : Two strings X and Y , and the elementary edit measures $d_s(\dots)$ defined by equation (2.1) above.

OUTPUT : $D_s[X][Y]$, the set of common supersequences X and Y whose lengths are less than or equal to $|X| + |Y|$.

BEGIN

$D_s[\mu][\mu] = \{\mu\}$, the null string ;

FOR $i := 1$ to M **DO**

$D_s[X_i][\mu] := D_s[X_{i-1}][\mu] \cdot d_s(x_i, \lambda)$;

ENDFOR

FOR $j := 1$ to N **DO**

$D_s[\mu][Y_j] := D_s[\mu][Y_{j-1}] \cdot d_s(\lambda, y_j)$;

ENDFOR

FOR $i := 1$ to M **DO**

FOR $j := 1$ to N **DO**

$$D_s[X_i][Y_j] := \bigcup \left\{ \begin{array}{l} D_s[X_{i-1}][Y_{j-1}] \cdot d_s(x_i, y_j) \\ D_s[X_{i-1}][Y_j] \cdot d_s(x_i, \lambda) \\ D_s[X_i][Y_{j-1}] \cdot d_s(\lambda, y_j) \end{array} \right\} ;$$

ENDFOR

ENDFOR

END ALGORITHM Compute_the_Set_of_Supersequences.

2.5 Relationship between Similarity Measures of Two Strings

As we can see from the above sections, finding the lengths of the LCS and the SCS of two strings and finding the actual LCS and SCS, are all similar problems. Kashyap and Oommen [KO83b] showed that all these measures could be computed by using a *single generic algorithm*, by merely choosing the operators and elementary similarity measures for the different problems. The details of this algorithm are presented below.

In order to relate the different measures together, Kashyap and Oommen defined an abstract structure $\tau = (T, \oplus, \otimes, \theta, I)$ where $(T, \oplus, \theta)$ is a commutative monoid with θ as the identity of the operator $\oplus$, $(T, \otimes, I)$ is a commutative monoid with I as the identity of the operator $\otimes$, and where $\otimes$ distributes over $\oplus$ from both sides. The iterative use of $\oplus$ and $\otimes$ will be denoted by $\bigoplus$ and $\bigotimes$ respectively.

They also defined an abstract measure $\Omega(X, Y)$ between $X, Y \in \mathcal{A}^*$ induced by the system $(\mathcal{A}, \tau, \omega)$ where $\mathcal{A}$ is the alphabet, τ is as defined above, and ω is a generalized cost function $\omega(.,.)$ whose parameters are the symbols of the alphabet $\mathcal{A}$. This measure $\Omega(.,.)$ is defined as follows :

$$\Omega(X, Y) = \begin{cases} I; \text{ the identity for } \otimes. & \text{if } X = Y = \mu; \text{ the null string} \\ \sum_{((X', Y') \in G_{X, Y})} \left\{ \prod_{i=1}^{|X'|} \omega(x_i', y_i') \right\} & \text{otherwise.} \end{cases}$$

where $G_{X, Y}$ is the set $\{(X', Y')\}$ defined in Section 2.1.3.

Although $\Omega(X, Y)$ is defined explicitly in terms of the set $G_{X, Y}$, it has been shown in [KO83b] that it can be computed recursively without evaluating the terms for every element of the set individually. The following theorem forms the basis of the computation.

THEOREM 2.7

Let $X = x_1..x_M$ and $Y = y_1..y_N$, where both X and Y are not null, then,

$$\Omega(X_M, \mu) = \Omega(X_{M-1}, \mu) \otimes \omega(x_M, \lambda)$$

$$\Omega(\mu, Y_N) = \Omega(\mu, Y_{N-1}) \otimes \omega(\lambda, y_N)$$

$$\Omega(X_M, Y_N)$$

$$= \{ \Omega(X_{M-1}, Y_{N-1}) \otimes \omega(x_M, y_N) \} \otimes \{ \Omega(X_M, Y_{N-1}) \otimes \omega(\lambda, y_N) \} \otimes \{ \Omega(X_{M-1}, Y_N) \otimes \omega(x_M, \lambda) \}$$

Proof : See proof of Theorem 9 in [KO83b].

Theorem 2.7 leads to the following algorithm for computing $\Omega(X, Y)$. Note that running time of this algorithm involves $O(|X| \cdot |Y|)$ operations of $\otimes$ and $\oplus$. This is of the same order of complexity as the optimal algorithm shown by Wagner and Fischer for the infinite alphabet case.

ALGORITHM Generalized Operator

INPUT : The strings X, Y , the set of elementary edit distances defined by $\omega(\dots)$, and procedures which execute the two operators $\otimes$ and $\oplus$.

OUTPUT : The abstract measure $\Omega(X, Y)$.

BEGIN

$\Omega[\mu][\mu] = 1$;

FOR $i := 1$ **to** M **DO**

$\Omega[X_i][\mu] := \Omega[X_{i-1}][\mu] \otimes \omega(x_i, \lambda)$;

ENDFOR

FOR $j := 1$ **to** N **DO**

$\Omega[\mu][Y_j] := \Omega[\mu][Y_{j-1}] \otimes \omega(\lambda, y_j)$;

ENDFOR

FOR $i := 1$ **to** M **DO**

FOR $j := 1$ **to** N **DO**

$\Omega[X_i][Y_j] := \{ \Omega[X_{i-1}][Y_{j-1}] \otimes \omega(x_i, y_j) \} \oplus \{ \Omega[X_{i-1}][Y_j] \otimes \omega(x_i, \lambda) \}$
 $\oplus \{ \Omega[X_i][Y_{j-1}] \otimes \omega(\lambda, y_j) \}$;

ENDFOR

ENDFOR

$\Omega(X, Y) := \Omega[X_M][Y_N]$;

END ALGORITHM Generalized_Operator.

If we assign τ to be as follows :

$$\tau := (Z_p', \text{MIN}, +, \infty, 0),$$

where Z_p' is the set consisting of nonnegative integers and let $\omega(\dots)$ be as defined below :

$$\omega(a, b) = 1 \quad \text{if } a = b; a, b \in \mathcal{A} ;$$

$$\omega(a, b) = \infty \quad \text{if } a \neq b; a, b \in \mathcal{A} ;$$

$$\omega(a, \lambda) = \omega(\lambda, a) = 1 \quad \text{for all } a \in \mathcal{A} ;$$

then $\Omega(X, Y)$ will yield the length of the SCS of X and Y .

Alternatively, if we assign τ to be :

$$\tau := (Z_p', \text{MAX}, +, 0, 0),$$

and $\omega(\dots)$ as defined below :

$$\omega(a, b) = 1 \quad \text{if } a = b \neq \lambda; a, b \in \tilde{\mathcal{A}} ;$$

$$\omega(a, b) = 0 \quad \text{otherwise ;}$$

then, $\Omega(X, Y)$ will be the length of LCS of X and Y .

We have summarized numerous choices for the operators $\oplus$ and $\ominus$ in Table 2.1 given below. The corresponding assignment for the function $\omega(\dots)$ is also tabulated. In each case, the quantity which is represented by $\Omega(X, Y)$ has also been recorded. Note that in the table :

- (i) R_p' is the set consisting of nonnegative real numbers and ∞ ,
- (ii) T_A is the power set of $\mathcal{A}^*$, the set of strings over $\mathcal{A}$,
- (iii) $\cup$ is the set union operator,
- (iv) $\cdot$ is the set concatenation operator,
- (v) ϕ is the identity of $\cup$, the union operator, and,
- (vi) $\{\mu\}$, the null string, is the identity of $\cdot$, the set concatenation operator.

Problem	$\tau=(T, \oplus, \ominus, \theta, l)$	$\omega(\dots)$
Length of LCS	$(Z_p', \text{MAX}, +, 0, 0)$	$\omega(a,b) = 1$ if $a = b \neq \lambda$; $\omega(a,b) = 0$ otherwise.
Length of SCS	$(Z_p', \text{MIN}, +, \infty, 0)$	$\omega(a,b) = 1$ if $a = b$; $\omega(a,b) = \infty$ if $a \neq b$; $\omega(a,\lambda) = 1$ and $\omega(\lambda,a) = 1$.
Set of Common Subsequence*	$(T_A, \cup, \cdot, \phi, \{\mu\})$	$\omega(a,b) = \{a\}$ if $a = b \neq \lambda$; $\omega(a,b) = \{\mu\}$ otherwise.
Set of Common Supersequence*	$(T_A, \cup, \cdot, \phi, \{\mu\})$	$\omega(a,b) = \{a\}$ if $a = b$; $\omega(a,b) = \phi$ if $a \neq b$; $\omega(a,\lambda) = \{a\}$ $\omega(\lambda,a) = \{a\}$.
Set of Shuffles	$(T_A, \cup, \cdot, \phi, \{\mu\})$	$\omega(a,b) = \phi$ for all $a, b \in \mathcal{A}$; $\omega(a,\lambda) = \omega(\lambda,a) = \{a\}$.
Probability of erroneous strings	$([0,1], +, *, 0, 1)$	$\omega(a,b) = p(b a)$

* The set of SCS (or LCS) can be obtained from the resulting set by always retaining only the shortest (or longest) elements of the set.

Table 2.1 : Summary of the use of generalized operators and generalized edit measures.

2.6 Conclusions

We have briefly reviewed the string editing problem and several related problems. Various algorithms for solving these problems have been presented. The best algorithm for finding the edit distance between X and Y for an infinite alphabet is the one due to Wagner and Fischer [WF74] which runs in $O(MN)$ time where $|X| = M$ and $|Y| = N$. Additionally, $O(ND)$, $O(NP)$, and $O(N + \frac{ND}{\log N})$ time algorithms have been presented by various authors ([Uk85b], [My86], [Ha88], [WMMM90], [WMM91]) for the finite alphabet case, where D is the edit distance between the strings X and Y , and P is the number of deletions in the shortest edit script consisting only of insertions and deletions. For the LCS problem, although the $O(MN)$ time algorithm [WF74] was shown to be optimal when the size of the alphabet is infinite, $O((R+N) \log N)$ time algorithms have been presented by [HS77], [AG87], [Mu80] for the finite alphabet case, where R is the total number of ordered pairs of the positions at which the two strings match. As of date, the fastest solutions to this problem are the ones due to Hirschberg [Hi77] and Nakatsu *et. al.* [NKY82]. Finally, in this chapter we discussed the generalization of these techniques introduced by Kashyap and Oommen [KO83b]. In [KO83b], these authors showed that by using only one common mathematical framework, all the similarity measures discussed in this chapter could be computed in $O(MN)$ time and linear space.

Among the different multi-dimensional generalization of strings, trees are considered to be one of the most important in computer science. In the chapters to follow, we shall study how the string editing (and related) problems can be extended to trees.

III. The Tree Editing Problem

3.1 Introduction

Trees, graphs, and webs are typically considered to be multi-dimensional generalization of strings. Among these different structures, trees are considered to be among the most important "non-linear" structures in computer science; consequently we generalize the concepts of string editing to tree editing in this chapter.

The tree editing problem has been studied since 1976. It is similar to the string editing problem and it concerns the determination of the distance between two trees, as measured by the minimum cost sequence of edit operations. Typically, the edit sequence includes the substitution, insertion, and deletion of nodes needed to transform one tree into the other. Applications of the tree-to-tree editing problem can be found in the theory of amino-acid sequence comparison ([GK82], [SK83]), the syntactic pattern recognition of human fingerprints and remote sensing ([Fu68], [KC72], [GT78]), and in the parsing of sentences from a stochastic grammar ([Fu82], [GT78]).

Unlike the string editing problem which has been well-developed, only few results have been published concerning the tree editing problem. In 1977, Selkow [Se77] presented a tree editing algorithm in which insertions and deletions were only restricted to the leaves. This algorithm is a straightforward generalization of the string editing algorithm due to Sankoff [Sa72], and is presented in Section 3.3. In 1979, Tai [Ta79] presented another algorithm in which insertions and deletions could take place at any node within the tree, except the root. The algorithm due to Lu [Lu79], on the other hand, was related to the problem of hierarchical clustering, and therefore, the trees under consideration have unlabeled nonterminal nodes and labelled terminal nodes. Thus, the algorithm due to Lu did not solve this problem for trees of more than two levels. The best known algorithm (in

terms of both the running time and space storage space requirements) for solving the tree editing problem is the one due to Zhang and Shasha [ZS89]. This algorithm will be discussed in greater detail in Section 3.5.

The tree editing problem has several domains of applications. Depending on where this problem is encountered, one could consider either the most general problem or one of its restricted versions. The most general case corresponds to one in which insertions and deletions can take place anywhere in the tree. A restricted version of this problem allows the insertions and deletions to apply to the leaf nodes only. In this chapter, we shall study both of these cases. We will demonstrate that a careful analysis of edit operations on ordered trees leads to algorithms for determining the minimum cost of editing one tree to another. The analysis of the time and space complexities of each algorithm are also included.

3.2 Definitions and Notations

A tree is a directed graph with no cycles which satisfies the following properties :

- (i) There is only one vertex, namely the root, which has no incident edges.
- (ii) All other vertices, except the root, have exactly one incoming edge.
- (iii) There is a unique path from the root to each vertex.

Thus, a tree has a distinguished vertex called the root, and the remaining vertices can be partitioned into $m \geq 0$ disjoint sets, each of which is itself a tree.

Let x, y be two distinct nodes in a tree T . If there is an edge from x to y , then x is called the parent of y and is denoted by $f(y)$, and y is called a child of x . Natural extensions to this notation can be defined as follow :

$$f^0(y) = y, \quad f^1(y) = f(y), \quad f^2(y) = f(f^1(y)), \quad \text{and so on .}$$

If there is a path from x to y , then x is an **ancestor** of y , and y is a **descendant** of x . We use

$$\text{Anc}(y) = \{ f^k(y) \mid 0 \leq k \leq \text{Depth}(y) \}$$

to represent the set of ancestors of y , where $\text{Depth}(y)$ is the length of the path from the root of the tree to y . A vertex with no descendants is called a **leaf**. Also, a vertex x together with all its descendants is called a **subtree** of a tree and x is called the root of this subtree.

An **ordered tree** is a tree in which the sons of each vertex are ordered. We assume that they are ordered from left to right. All trees discussed in this chapter are rooted, ordered and labeled.

Many tree operations require the traversal of the tree. The **preorder traversal** of a tree is determined by applying the following recursive steps :

1. Visit the node.
2. Visit each of the subtrees from left to right in the preorder sequence.

The sequence of nodes resulting from the preorder traversal is called the **preorder sequence** of the tree. An example of a preorder sequence is shown in Figure 3.1(a). Similarly, the **postorder traversal** of a tree is defined by the following recursive steps :

1. Visit each of the subtrees from left to right in the postorder sequence.
2. Visit the node.

As in the above, the sequence of nodes generated by the postorder traversal is called the **postorder sequence** of the tree. An example of a postorder sequence is shown in Figure 3.1(b).

Figure 3.1 : Examples of preorder and postorder sequences.

With regard to notation, if T is a tree, then $|T|$ represents the number of nodes of T ; $T[i]$ represents the node of T at position i in the chosen numbering sequence. Thus, for example, if we consider the tree of Figure 3.1, if we use the preorder numbering of nodes, $T[1]$, $T[3]$ and $T[7]$ are A , E , and D respectively.

3.2.1 Edit Operations and Distance between Trees

Let λ be the null node, distinct from μ , the null tree. An edit operation on a tree is either an insertion of a node, a deletion of a node, or a substitution of one node by another. In terms of notation, an edit operation is represented symbolically as $x \rightarrow y$ where x and y can either be a node from the trees under consideration, or λ , the null node. The case when $x = \lambda$ and $y \neq \lambda$ represents an insertion; $x \neq \lambda$ and $y = \lambda$ represents a deletion; and $x \neq \lambda$ and $y \neq \lambda$ represents a substitution. Note that the case in which $x = \lambda$ and $y = \lambda$ represents the null operation and is not needed. The formal definitions of each operation are described as follows :

- (i) An insertion of node x into tree T .

Node x will either be inserted as a son of some node u of T , or it will be inserted as the root of the tree T . In the latter case, it will take as its son the root of the original tree T . In the former case, it may either be inserted with no sons or take as sons any subsequence of the sons of u . Formally, for the former case, if u had sons $u_1, u_2, \dots, u_k$ prior to the

insert operation, then for some $0 \leq i \leq j \leq k$, node u in the resulting tree will have sons $u_1, \dots, u_i, x, u_{j+1}, \dots, u_k$. Node x will have no sons if $j = i+1$, or else have sons $u_{i+1}, \dots, u_{j-1}$. An example of an insertion operation is shown in Figure 3.2.

Figure 3.2 : An example of the insertion of a node.

(ii) A deletion of node y from a tree T .

Let us assume that node y has sons $y_1, y_2, \dots, y_k$ and node u , the father of y , has sons $u_1, u_2, \dots, u_j$ with $u_i = y$ (See Figure 3.3). Then node u in the resulting tree obtained after the deletion will have sons $u_1, u_2, \dots, u_{i-1}, y_1, y_2, \dots, y_k, u_{i+1}, \dots, u_j$. The deletion of the root is not allowed if the root has more than one son. An example of the deletion operation is shown in the Figure 3.3.

Figure 3.3 : An example of the deletion of a node.

(iii) Substitution of node x by node y in T .

In this case, node y in the resulting tree will have the same father and sons as node x in the original tree. An example of substitution is shown in Figure 3.4.

Figure 3.4 : An example of the substitution of a node by another.

Let $d(x,y) \geq 0$ be the cost of transforming a node x to another node y . If $x, y \neq \lambda$, $d(x,y)$ will represent the cost of substitution of node x by node y . Similarly, the cases when $x \neq \lambda, y = \lambda$ and $x = \lambda, y \neq \lambda$ correspond to the deletion and insertion of node x and y respectively. Without loss of generality, we assume that

$$d(x,y) \geq 0; d(x,x) = 0; \quad (3.1)$$

$$d(x,y) = d(y,x) \quad (3.2)$$

$$\text{and} \quad d(x,z) \leq d(x,y) + d(y,z) \quad (3.3)$$

where (3.3) ensures that no sequence of edit operations can achieve, at lower cost, the same effect as a single operation, and is thus a "triangular" inequality constraint.

Let S be a sequence $s_1, \dots, s_k$ of edit operations. We say that S takes T_1 to T_2 if there is a sequence of trees $\gamma_0, \dots, \gamma_k$ where $T_1 = \gamma_0, T_2 = \gamma_k$, such that s_i transforms γ_{i-1} to γ_i . In such a case, we symbolically represent the sequence by the notation : $T_1 \xrightarrow{S} T_2$. We extend $d(\dots)$ to the sequence S by assigning :

$$W(S) = \sum_{i=1}^{|S|} d(s_i).$$

With the introduction of $W(S)$, the distance between T_1 and T_2 can be defined as follows :

$$D(T_1, T_2) = \text{Min} \left\{ W(S) \mid T_1 \xrightarrow{S} T_2 \right\}.$$

It is easy to observe that, if the postorder notation is used to represent the tree, then

$$D(T_1, T_2) \leq d(T_1[|T_1|], T_2[|T_2|]) + \sum_{i=1}^{|T_1|-1} d(T_1[i], \lambda) + \sum_{j=1}^{|T_2|-1} d(\lambda, T_2[j]). \quad (3.4)$$

This is obtained by considering the following edit sequence : delete all nodes of T_1 except the root of T_1 ; substitute the root of T_1 by the root of T_2 ; finally insert back all nodes of T_2 except the root. Clearly the quantity on the R.H.S. of equation (3.4) is an upper bound for the value of $D(T_1, T_2)$.

3.2.2 Mappings between Trees

A Mapping is a description of how a sequence of edit operations transforms a tree T_1 into another tree T_2 . Informally, we can describe a mapping as follows :

- (i) Lines connecting $T_1[i]$ and $T_2[j]$ correspond to substituting $T_1[i]$ by $T_2[j]$.
- (ii) Nodes in T_1 not touched by any line are to be deleted.
- (iii) Nodes in T_2 not touched by any line are to be inserted.

Formally, a mapping is a triple (M, T_1, T_2) , where M is any set of pairs of integers (i, j) satisfying :

- (i) $1 \leq i \leq |T_1|, 1 \leq j \leq |T_2|$;
- (ii) For any pair of (i_1, j_1) and (i_2, j_2) in M ,
 - (a) $i_1 = i_2$ if and only if $j_1 = j_2$ (one-to-one).

- (b) $T_1[i_1]$ is to the left of $T_1[i_2]$ if and only if $T_2[j_1]$ is to the left of $T_2[j_2]$. This is referred to as the *Sibling Property*.
- (c) $T_1[i_1]$ is an ancestor of $T_1[i_2]$ if and only if $T_2[j_1]$ is an ancestor of $T_2[j_2]$. This is referred to as the *Ancestor Property*.

A pictorial representation of a mapping for a sample pair of trees is given in Figure 3.5. Whenever there is no ambiguity we will use M to represent the triple (M, T_1, T_2) , the mapping from T_1 to T_2 .

Let $\Phi(a, m; b, n)$ represent the class of mappings from a tree formed by the nodes $T_1[a].. T_1[m]$, to another tree formed by the nodes $T_2[b].. T_2[n]$. If $M \in \Phi(1, m; 1, n)$ and $(x, y) \in M$, then M represents an edit sequence in which $T_1[x]$ is substituted by $T_2[y]$.

Let I, J be the sets of nodes in T_1 and T_2 respectively, not touched by any lines in M . We can define the cost of M as follows :

$$\text{Cost}(M) = \sum_{(i,j) \in M} d(T_1[i], T_2[j]) + \sum_{i \in I} d(T_1[i], \lambda) + \sum_{j \in J} d(\lambda, T_2[j]). \quad (3.5)$$

Figure 3.5 : An example of a mapping between two postorder numbered trees.

Since mappings can be composed to yield new mappings ([Ta79], [ZS89]), the relationship between a mapping and a sequence of edit operations can now be specified.

Lemma 3.1

Given S , a sequence of edit operations that takes T_1 to T_2 , there exists a mapping M from T_1 to T_2 such that $\text{Cost}(M) \leq W(S)$. Conversely, for any mapping M , there exists a sequence of editing operations such that $W(S) = \text{Cost}(M)$.

PROOF : Same as proof of Lemma 2 in [ZS89].

Due to the above lemma, we obtain

$$D(T_1, T_2) = \text{Min} \left\{ \text{Cost}(M) \mid M \text{ is a mapping from } T_1 \text{ to } T_2. \right\}$$

In other words, the problem of searching for the minimal cost edit sequence has been reduced to a search for the minimal cost mapping.

We now consider the editing problem where insertions and deletions are restricted to leaves only.

3.3 Insertions and Deletions Restricted to Leaves

When insertions and deletions take place only at leaves, a node may be inserted only as a son of a leaf, and only nodes which are leaves can be deleted. An algorithm for solving the tree-to-tree editing problem under this restriction was presented by Selkow [Se77], and a careful analysis of this algorithm was made by Noetzel and Selkow [SK83]. We will now present this algorithm in detail. Note that in the algorithm, the preorder numbering is used to number the nodes in the tree.

Let T be a tree where the root has $m \geq 0$ sons. We denote the subtree² formed

²In the description of these algorithms, we will slightly deviate from our original notation. Indeed, rather than speak about the distance between the trees T_1 and T_2 we will be computing the distance between the trees T and T' . This is because, the individual subtrees themselves will now be having subscripts, and this

by these sons by $t_1, t_2, \dots, t_m$. Let $T_{1,i}$ denote the tree obtained from T by removing subtrees $t_{i+1}, t_{i+2}, \dots, t_m$. Note that $T = T_{1,m}$. Finally, let $i_1, i_2, \dots, i_m$ be the last node of the subtrees of the root, in the preorder sequence of T . Let T' be another tree with $n \geq 0$ sons at its root. We denote the subtree formed by these sons by $t'_1, \dots, t'_n$. As before, $T'_{1,j}$ denotes the tree obtained from T' by removing subtrees $t'_{j+1}, \dots, t'_n$; $j_1, \dots, j_n$ are the last nodes of the subtrees of the root, in the preorder sequence of T' . Figure 3.6 illustrates this notation.

Figure 3.6 : Notations used in Selkow's Algorithm. Note that in the figure the preorder numbering is being used.

Let M be a mapping from T to T' , i.e., $M \in \Phi(1, i_m; 1, j_n)$. If some node i in T is not deleted, then none of its ancestors can be a leaf in the edit sequence and hence cannot be deleted or inserted. This is due to the fact that insertion and deletion operations can only be applied to leaves. In other words, if $(i, j) \in M$ for some mapping M , then $(f(i), f(j))$ must also be in the same M and so must all pairs of ancestors of both i and j : $(f^2(i), f^2(j)), \dots, (1, 1)$. This leads to the following lemma stated in [SK83].

will make an already cumbersome notation quite unreadable.

Lemma 3.2

Let i be a node in the subtree t_p of the root of T and let j be a node in the subtree t'_q of the root of T' . If $(i,j) \in M$ for some mapping M , then, if any other node in subtree t_p or t'_q is touched by the same mapping, it must be linked to a node in subtree t'_q or t_p respectively.

PROOF :

Since $(i,j) \in M$, so do all pairs of ancestors of both i and j : $(f(i),f(j)), \dots, (1,1)$. Thus, M must link the root of subtree t_p to the root of subtree t'_q . Now assume that there exists another node, say h , in subtree t_p of the root of T which is linked to a node, say k in a different subtree, say t'_r , of the root of T' . Then since $(h,k) \in M$, so do all pairs of their ancestors, and hence, the root of subtree t_p is also linked to the root of subtree t'_r . This contradicts the definition of a mapping that each node can be touched by at most one line, and the lemma is proved. ***

Since the lines associated with a particular mapping do not cross, if subtree t_p is linked to subtree t'_q and subtree t_r is linked to subtree t'_s , it is easy to see that $p < r$ implies $q < s$. Hence, it is possible to decompose any mapping into submappings corresponding to the individual pairs of subtrees that are linked. An example of this is shown in Figure 3.7.

Now, suppose that $M \in \Phi(1, i_p : 1, j_q)$ is the minimum-cost mapping that links nodes of T and T' up to subtree t_p and subtree t'_q . In other words, M is a mapping between $T_{1,p}$ and $T'_{1,q}$. But since M represents a minimum cost mapping, by definition, M has the value $D(T_{1,p}, T'_{1,q})$. However, since M can be decomposed into submappings, we can decompose the computation of $D(T_{1,p}, T'_{1,q})$ in an analogous way as follow.

For any tree x with nodes $x_0, x_2, \dots, x_k$, $k \geq 0$, let

$$W_I(x) = \sum_{i=0}^k d(\lambda, x_i) \quad \text{and} \quad W_D(x) = \sum_{i=0}^k d(x_i, \lambda)$$

Preorder Sequence of Tree 1

Preorder Sequence of Tree 2

Figure 3.7 : Example illustrating the decomposition of a mapping.

be the costs of edit sequences which insert and delete all the nodes in the subtree x . Then the following three properties hold :

- (i) If M touches no nodes of a direct subtree t'_q of the root of T , then insertion edit operations must be used for inserting the entire subtree t'_q in T . In this case, the mapping can be decomposed into two sequences: a sequence which converts the tree $T_{1,p}$ to the tree $T_{1,q-1}$ and a sequence consisting of the insertion operations for the entire subtree t'_q of T . Consequently,

$$D(T_{1,p}, T_{1,q}) = D(T_{1,p}, T_{1,q-1}) + W_I(t'_q). \quad (3.6)$$

- (ii) If M touches no nodes of a direct subtree t_p of root in T , deletion operations must be used for deleting the entire subtree t_p in T , and the mapping can be decomposed into a sequence which converts the tree $T_{1,p-1}$ to the tree $T'_{1,q}$ and a sequence consisting of the operations which delete the whole subtree t_p of T . Thus, in this case,

$$D(T_{1,p}, T_{1,q}) = D(T_{1,p-1}, T_{1,q}) + W_D(t_p). \quad (3.7)$$

(iii) We now consider the final case when M touches both subtrees t_p and t'_q , in which case these subtrees must be linked to each other. The latter is true because if (p,y) and $(x,q) \in M$, then we have: $p < x$ iff $y < q$ and $p = x$ iff $y = q$; but since $p \geq x$ and $q \geq y$, it follows that $p = x$ and $q = y$. Thus, M may be partitioned into $M_1 \in \Phi(1, i_{p-1}; 1, j_{q-1})$ and $M_2 \in \Phi(i_{p-1}+1, i_p; j_{q-1}+1, j_q)$ such that M_1 is a mapping from $T_{1,p-1}$ to $T_{1,q-1}$, and M_2 is a mapping from t_p to t'_q , and the cost of M is the sum of their minimum-costs of M_1 and M_2 and hence we have :

$$D(T_{1,p}, T_{1,q}) = D(T_{1,p-1}, T_{1,q-1}) + D(t_p, t'_q). \quad (3.8)$$

Note that if neither subtree t_p in T nor subtree t'_q in T' are touched by any line in M , either equations (3.6) or (3.7) can be used.

Since these three cases exhaust all the possible mappings yielding $D(T_{1,p}, T_{1,q})$, from the definition of the distance between two trees, the minimum of these three cases (i.e. equations (3.6), (3.7), and (3.8)) yields the required result. Thus, we have :

$$D(T_{1,i}, T_{1,j}) = \text{Min} \begin{cases} D(T_{1,i-1}, T_{1,j-1}) + D(t_i, t'_j) \\ D(T_{1,i}, T_{1,j-1}) + W_I(t'_j) \\ D(T_{1,i-1}, T_{1,j}) + W_D(t_i) \end{cases} \quad (3.9)$$

This leads us to the following lemma.

Lemma 3.3

Let T be a tree with m subtrees $t_1, t_2, \dots, t_m$, and let $T_{1,i}$ denote the tree obtained from T by removing subtrees $t_{i+1}, t_{i+2}, \dots, t_m$. Similarly, let T' be another tree with n subtrees $t'_1, t'_2, \dots, t'_n$, and let $T'_{1,j}$ denote the tree obtained from T' by removing subtrees $t'_{j+1}, \dots, t'_n$. Then, the following four equalities hold :

- (i) $D(T_{1,0}, T_{1,0}) = d(T[1], T[1]),$
- (ii) $D(T_{1,0}, T_{1,j}) = d(T[1], T[1]) + \sum_{k=1}^j W_I(t'_k), \quad \text{for } 0 \leq j \leq n.$
- (iii) $D(T_{1,i}, T_{1,0}) = d(T[1], T[1]) + \sum_{k=1}^i W_D(t_k), \quad \text{for } 0 \leq i \leq m.$
- (iv) $D(T_{1,i}, T_{1,j}) = \text{Min} \begin{cases} D(T_{1,i-1}, T_{1,j-1}) + D(t_i, t'_j) \\ D(T_{1,i}, T_{1,j-1}) + W_I(t'_j) \\ D(T_{1,i-1}, T_{1,j}) + W_D(t_i) \end{cases}$
for $1 \leq i \leq m ; 1 \leq j \leq n.$

PROOF :

The first three equalities follow directly from the non-negativity of the costs of insertion and deletion operations. The fourth equality follows from the previous argument and equation (3.9). ***

The algorithm to compute the cost of the optimal editing sequence be derived directly from Lemma 3.3 and is presented as Algorithm Selkow.

In order to speed up the computation, two arrays, namely the arrays $\text{Ins}[]$ and the $\text{Del}[]$, are precomputed every time the procedure $\text{Local_Edit}(T, T')$ is called. The array $\text{Ins}[]$ contains the insertion cost of each subtree t'_k of T' , and the array $\text{Del}[]$ contains the deletion cost of each subtree t_k of T . With these two precomputed arrays, the value $W_I()$ and $W_D()$ can be evaluated in $O(1)$ time by merely looking up the elements stored in these arrays.

We shall now briefly discuss the analysis of Algorithm Selkow.

Let the *Signature* of a tree T be a vector $(a_0, \dots, a_d)$ such that a_i is the number of vertices of T at depth i . It is shown in [Se77] that in computing $D(T, T')$, the procedure Local_Edit is called once for each pair of vertices at the same depth. Therefore, if T and T'

ALGORITHM Selkow

INPUT : Two trees T and T'.
OUTPUT : The edit distance D(T, T').

PROCEDURE Local_Edit (T, T')

ASSUMPTION : Assume that T has subtrees $t_1, \dots, t_m$ and T' subtrees $t'_1, \dots, t'_n$.

NOTATION : $H[i][j]$ represents $D(T_{1,i}, T'_{1,j})$, the distance between the trees.

BEGIN

$H[0][0] = d(T[1], T'[1]);$

FOR j = 1 to n **DO**

$H[0][j] = H[0][j-1] + W_I(t'_j);$

ENDFOR

FOR i = 1 to m **DO**

$H[i][0] = H[i-1][0] + W_D(t_i);$

ENDFOR

FOR i = 1 to m **DO**

FOR j = 1 to n **DO**

$$H[i][j] = \text{Min} \begin{cases} H[i-1][j-1] + \text{Local_Edit}(t_i, t'_j) \\ H[i][j-1] + W_I(t'_j) \\ H[i-1][j] + W_D(t_i) \end{cases};$$

ENDFOR

ENDFOR

RETURN (H[m][n]);

END PROCEDURE Local_Edit;

BEGIN

$D(T, T') := \text{Local_Edit}(T, T');$

END ALGORITHM Selkow.

have Signatures $(a_0, \dots, a_{h_1})$ and $(b_0, \dots, b_{h_2})$, the procedure Local_Edit(T, T') requires :

$$O\left(\sum_{i=0}^{\text{Min}(h_1, h_2)} a_i \cdot b_i\right)$$

time. Furthermore, precomputation of the arrays Ins[] and Del[] (referred to above) for all subtrees requires :

$$O\left(\sum_{i=0}^{h_2} b_i\right) \quad \text{and} \quad O\left(\sum_{i=0}^{h_1} a_i\right)$$

time respectively. In other words, if M and N are the number of nodes of the two trees, and h_1 and h_2 are the depths of trees, the algorithm will require $O(M \cdot N \cdot \min\{h_1, h_2\})$ computation time.

Noetzel and Selkow [SK83] re-formalized this algorithm and presented a purely *recursive* version in 1983. We omit the details of the latter algorithm since the ideas used there are exactly the same as those used in Algorithm Selkow.

3.4 A Less Restricted Edit Algorithm

We now consider the problem of finding the minimum-cost edit sequence when the operations of insertion and deletion are allowed at any node, except the root, of a tree. Note that the substitution operation may take place anywhere within the tree including the root of the tree. Thus, in general, in this case, the minimum-cost mapping is not decomposable into analogous minimum-cost submappings. To clarify this, consider the trees shown in the Figure 3.8.

Figure 3.8 : Trees used to justify the correctness of a transformation. (From [Ta79])

For the sake of simplicity, assume that the cost of every substitution, deletion and insertion operation is unity. Then, it is clear that the set $M = \{(1, 1), (2, 3), (3, 4)\}$ represents the minimum-cost mapping from T to T' , and the cost of this edit sequence will then be 2. This represents the substitution of node B by node C, and the insertion of node B. But any minimum-cost submapping $M' \in \Phi(1, x:1, y)$, with $x = 2$ and $2 \leq y \leq 3$, will always include the element $(2, 2)$, which no minimum-cost mapping M can achieve by merely extending the corresponding minimum-cost submappings. In other words, if $M \in \Phi(1, m:1, n)$ is the minimum-cost mapping and is known to include an element (i, j) , then M may be partitioned into submappings $M_1 \in \Phi(1, i-1:1, j-1)$ and $M_2 \in \Phi(i, m:j, n)$. But the cost of M may not be assumed to be the sum of the minimum costs of submappings M_1 and M_2 since not every mapping $M_1 \in \Phi(1, i-1:1, j-1)$ may be extended to permit the inclusion of (i, j) .

Consider now the transformation shown in Figure 3.8. The mapping $M = \{(1, 1), (2, 2)\} \in \Phi(1, 2:1, 3)$ can not be extended by the addition of the pair $(3, 4)$, since the trees resulting from the extended mapping are not isomorphic and therefore the mapping is invalid.

Tai [Ta79] presented an iterative algorithm to solve this less restricted case of the tree editing problem. Based on the principles found in [Ta79], Noetzel and Selkow [SK83] developed a recursive scheme for computing the minimum edit distance between two trees. We will present these algorithms in detail in the following subsections.

3.4.1 An Iterative Algorithm for the Less Restricted Tree Editing Problem

In determining edit distance between trees, Tai [Ta79] used the preorder numbering to number the nodes in trees. Assuming that the nodes in a tree are numbered as per the preordering sequence and that the only edit operation permitted for the root is the substitution operation, the following Lemma holds :

Lemma 3.4

Let $T[x]$ and $T[y]$ be two nodes in a tree T such that $T[x]$ is an ancestor of $T[y]$.

Then,

- (i) $x < y$.
- (ii) $T[k]$ is a descendant of $T[x]$ for all k such that $x < k \leq y$.
- (iii) Let $T[u]$ be the father of $T[y]$. Then $T[u]$ is either $T[y-1]$ or an ancestor of $T[y-1]$, and $T[u]$ is on the path from $T[x]$ to $T[y-1]$.

PROOF :

(i) and (ii) follow directly from the definition of preorder sequence. To prove (iii) we can use the following argument. First of all, we know that $T[u]$ is the father $T[y]$. If $T[y]$ is the first son of $T[u]$ then $T[u] = T[y-1]$. Otherwise, $T[y-1]$ is in the subtree whose root is the son of $T[u]$ which is left of $T[y]$. Hence, $T[u]$ is an ancestor of $T[y-1]$. Thus, in either case, $T[u]$ will be on the path from $T[x]$ to $T[y-1]$. ***

Let $T[x..y]$ denote the portion of a tree T which consists of nodes $T[x], T[x+1], \dots, T[y]$ and their associated edges. We required that x is an ancestor of y , and so, by virtue of the preorder sequence, $x < y$. Let $H(i+1, j+1)$ be the *distance* from tree $T[1..i+1]$ to tree $T[1..j+1]$, i.e., $H(i+1, j+1) = D(T[1..i+1], T[1..j+1])$. Suppose $M \in \Phi(1, i+1; 1, j+1)$ is a minimum cost mapping from $T[1..i+1]$ to $T[1..j+1]$. Since $\text{Cost}(M) = H(i+1, j+1)$, at least one of the following three cases must hold :

- (i) If $T[i+1]$ is not touched by any line of the mapping, then,

$$\text{Cost}(M) = H(i, j+1) + d(T[i+1], \lambda). \quad (3.10)$$

This equality is true because the deletion operation must necessary be used for deleting node $T[i+1]$. Thus, the mapping can be decomposed into two sequences : a

sequence to convert the tree $T[1..i]$ to the tree $T[1..j+1]$, and a subsequence which corresponds to the deletion of node $T[i+1]$.

- (ii) If $T[j+1]$ is not touched by any line of the mapping, then,

$$\text{Cost}(M) = H(i+1, j) + d(\lambda, T[j+1]) \quad (3.11)$$

This equality is true because the insertion operation must be used for inserting node $T[j+1]$, and consequently, the mapping can be decomposed into two sequences : a sequence to convert tree $T[1..i+1]$ to tree $T[1..j]$, and a subsequence which corresponds to the insertion of node $T[j+1]$.

- (iii) Consider the case when $T[i+1]$ and $T[j+1]$ are touched by lines of the optimal mapping. Furthermore, let these be touched by lines $(i+1, q)$ and $(p, j+1)$. Thus, $i+1 = p$ and $j+1 = q$, since $i+1 \geq p$ and $j+1 \geq q$, and by the definition of mapping the relation $(i+1 = p \text{ iff } q = j+1)$ and $(i+1 > p \text{ iff } q > j+1)$ holds. Thus, if $T[i+1]$ and $T[j+1]$ are touched by lines of the mapping, they must both be touched by the same line $(i+1, j+1)$. However, if $M' = M - \{(i+1, j+1)\}$ is not a minimum-cost mapping from $T[1..i]$ to $T[1..j]$, then

$$\begin{aligned} \text{Cost}(M) &= \text{Cost}(M') + d(T[i+1], T[j+1]) \\ &> H(i, j) + d(T[i+1], T[j+1]) \end{aligned}$$

which implies that $H(i+1, j+1) > H(i, j) + d(T[i+1], T[j+1])$.

Note that in this case, the equality holds when $M' = M - \{(i+1, j+1)\}$ is a minimum-cost mapping from $T[1..i]$ to $T[1..j]$.

Let $\text{Min}_M(.,.)$ be a function defined as follows :

$$\text{Min}_M(i+1, j+1) = \text{Min} \left\{ \text{Cost}(M) \mid \begin{array}{l} M \text{ is a mapping from } T[1..i+1] \text{ to } \\ T[1..j+1] \text{ such that } (i+1, j+1) \in M. \end{array} \right\}$$

Then, in this case we have the following :

$$\begin{aligned} \text{Cost (M)} &= H(i+1, j+1) \\ &= \text{Min_M (i+1, j+1)}. \end{aligned} \quad (3.12)$$

Since these three cases exhaust all the possible mappings yielding $H(i+1, j+1)$, the definition of Min_M leads to the following lemma.

Lemma 3.5

$$H(i+1, j+1) = \text{Min} \begin{cases} H(i, j+1) + d(T[i+1], \lambda) \\ H(i+1, j) + d(\lambda, T[j+1]) \\ \text{Min_M (i+1, j+1)} \end{cases} \quad (3.13)$$

for all $1 \leq i < |T|$, $1 \leq j < |T|$.

PROOF :

We have already presented the three cases which exhaust all possible mappings that yield $H(i+1, j+1)$. The proof follows since $H(i+1, j+1)$ merely involves the minimum of these three costs, i.e., the minimum of the right hand sides of equations (3.10) to (3.12).

Based on the assumption that *the root of any tree remains unchanged*³, every mapping between trees $T[1..i+1]$ and $T'[1..j+1]$ must contain the pair (1, 1), and thus $H(1, 1)$ is assigned the value zero. Thus, we have :

$$H(1, 1) = 0; \quad (3.14)$$

³The original paper [Ta79] considers only the case when the roots of the trees are identical. This means that the root cannot be substituted, inserted or deleted. It is easy to extend the results for the case when the root can be substituted but not inserted or deleted by just increasing this substitution distance after the rest of the computations are done.

$$H(i, 1) = \sum_{k=2}^i d(T[k], \lambda) \quad \text{for } 1 < i \leq |T|. \quad (3.15)$$

$$H(1, j) = \sum_{k=2}^j d(\lambda, T'[k]) \quad \text{for } 1 < j \leq |T'|. \quad (3.16)$$

Equations (3.14), (3.15), and (3.16) together form the boundary values of $H(\dots)$ when either tree contains only one node. For computing the intermediate values (i.e. non-boundary values) of $H(\dots)$, Tai [Ta79] proved the following two Lemmas which assist in the calculation of $\text{Min}_M(i+1, j+1)$.

Lemma 3.6

Assume that $T[p]$ and $T'[q]$ are ancestors of $T[i+1]$ and $T'[j+1]$, respectively, and that $M \in \Phi(1, i+1; 1, j+1)$ is a mapping from $T[1..i+1]$ to $T'[1..j+1]$ such that (p, q) and $(i+1, j+1) \in M$. Let M_1 and M_2 be the subsets of M defined by :

$$M_1 = \left\{ (m, n) \mid (m, n) \in M, 1 \leq m \leq p \text{ and } 1 \leq n \leq q \right\},$$

$$\text{and } M_2 = \left\{ (m, n) \mid (m, n) \in M, p \leq m \leq i \text{ and } q \leq n \leq j \right\}.$$

Then, the following assertions hold :

- (i) Since M_1 can be viewed as a mapping from $T[1..p]$ to $T'[1..q]$, and M_2 can be viewed as a mapping from $T[p..i]$ to $T'[q..j]$, and furthermore, since

$$M = M_1 \cup M_2 \cup \{(i+1, j+1)\},$$

the Cost associated with M satisfies :

$$\text{Cost}(M) = \text{Cost}(M_1) + \text{Cost}(M_2) - d(T[p], T'[q]) + d(T[i+1], T'[j+1]).$$

(ii)

$$\text{Cost}(M) = \text{Min} \left\{ \text{Cost}(M') \mid \begin{array}{l} M' \text{ is a mapping from } T[1..i+1] \text{ to } T'[1..j+1] \\ \text{such that } (p, q) \text{ and } (i+1, j+1) \in M' \end{array} \right\}$$

if and only if

$$\text{Cost}(M_1) = \text{Min} \left\{ \text{Cost}(M_1') \mid \begin{array}{l} M_1' \text{ is a mapping from } T[1..p] \text{ to } \\ T'[1..q] \text{ such that } (p, q) \in M_1' \end{array} \right\}, \text{ and}$$

$$\text{Cost}(M_2) = \text{Min} \left\{ \text{Cost}(M_2') \mid \begin{array}{l} M_2' \text{ is a mapping from } T[p..i] \text{ to } \\ T'[q..j] \text{ such that } (p, q) \in M_2' \\ \text{and } M_2' \cup \{(i+1, j+1)\} \text{ is a mapping} \end{array} \right\}$$

Proof : See proof of Lemma 4.1 in [Ta79].

The claim of the result can be informally described as follows. A mapping from $T[1..i+1]$ to $T'[1..j+1]$ can be decomposed into submappings such that each submapping is a mapping from one portion of $T[1..i+1]$ to one portion of $T'[1..j+1]$. Moreover, for minimum cost mappings, each of the submappings must also be a minimum cost mapping.

Lemma 3.6 dictates how the computation must be executed if nodes p and q satisfying the premise of the lemma exist. We shall now show that such vertices do indeed exist, and that the premise of the Lemma is always satisfied.

Assume that $M \in \Phi(1, i+1; 1, j+1)$ is a mapping from $T[1..i+1]$ to $T'[1..j+1]$ with $(i+1, j+1) \in M$. Let $T[s_M]$ and $T'[t_M]$ be the latest ancestors of $T[i+1]$ and $T'[j+1]$, respectively, touched by the lines of M . Since it has been assumed that every mapping from $T[1..i+1]$ to $T'[1..j+1]$ contains $(1, 1)$, both $T[s_M]$ and $T'[t_M]$ must necessarily exist. Thus, by Lemma 3.4, the following properties hold :

- (i) $(s_M, t_M) \in M$ (since $(i+1, j+1) \in M$).
- (ii) The father of $T[i+1]$, namely, $T[f(i+1)]$ is on the path from $T[s_M]$ to $T[i]$, and analogously, the father of $T'[j+1]$, namely, $T'[f(j+1)]$ is on the path from $T'[t_M]$ to $T'[j]$.

Based on these two properties, Tai [Ta79] presented a lemma which shows that the quantity $\text{Min_M}(i+1, j+1)$ can be computed by using the values $\text{Min_M}(s, t)$, $1 \leq s \leq i$ and $1 \leq t \leq j$, and by incorporating them with a quantity $E(\dots, \dots)$ to be defined. The main result which uses Lemma 3.6 and the above properties is stated below.

Lemma 3.7

$$\begin{aligned}
 & \text{Min_M}(i+1, j+1) \\
 &= d(T[i+1], T[j+1]) \\
 &+ \min_{(s,t)} \left\{ \min \left\{ \text{Cost}(M_1) \mid \begin{array}{l} M_1 \text{ is a mapping from } T[1..s] \\ \text{to } T'[1..t] \text{ with } (s, t) \in M_1. \end{array} \right\} \right. \\
 &\quad \left. + \min \left\{ \text{Cost}(M_2) \mid \begin{array}{l} M_2 \text{ is a mapping from } T[s..i] \text{ to } T'[t..j] \\ \text{such that } (s, t) \in M_2 \text{ and conditions (i)} \\ \text{and (ii) are satisfied.} \end{array} \right\} \right. \\
 &\quad \left. - d(T[s], T[t]) \right\}. \tag{3.17}
 \end{aligned}$$

where :

- (i) no descendant of $T[s]$ on the path from $T[s]$ to $T[f(i+1)]$ is touched by any line of M_2 , and,
- (ii) no descendant of $T[t]$ on the path from $T[t]$ to $T[f(j+1)]$ is touched by any line of M_2 .

PROOF :

Following the proof of Lemma 4.2 in [Ta79], if M is a mapping from $T[1..i+1]$ to $T'[1..j+1]$ with $(i+1, j+1) \in M$, then M contains (s_M, t_M) as claimed above. By Lemma 3.6, M can be decomposed into submapping M_1 and M_2 such that M_1 is a mapping from $T[1..s_M]$ to $T'[1..t_M]$, M_2 is a mapping from $T[s_M..i]$ to $T'[t_M..j]$, and $M = M_1 \cup M_2 \cup \{(i+1, j+1)\}$. Thus, it follows that $\text{R.H.S} \leq \text{L.H.S}$.

Assume that $T[s]$ and $T[t]$ are ancestors of $T[i+1]$ and $T[j+1]$, respectively. Let M_1 be a mapping from $T[1..s]$ to $T'[1..t]$ with $(s, t) \in M_1$, and M_2 be a mapping from $T[s..i]$ to $T'[t..j]$ such that $(s, t) \in M_2$. Any descendant of $T[s]$ on the path from $T[s]$ to $T[f(i+1)]$ is not touched by any line of M_2 . Similarly, any descendant of $T[t]$ on the path from $T[t]$ to $T'[f(j+1)]$ is not touched by any line of M_2 . Consequently, $M = M_1 \cup M_2 \cup \{(i+1, j+1)\}$ is a mapping from $T[1..i+1]$ to $T'[1..j+1]$. Therefore $L.H.S \leq R.H.S$. This argument and the arguments of the previous paragraph prove the result. ***

Consider the indices s, u, i, t, v and j defined as follows :

$$1 \leq s \leq u \leq i \leq |T|,$$

$$\text{and,} \quad 1 \leq t \leq v \leq j \leq |T'|$$

such that $T[u]$ is on the path from $T[s]$ to $T[i]$, and $T'[v]$ is on the path from $T'[t]$ to $T'[j]$. The indices s, u, i, t, v and j satisfying these conditions will be referred to as the **Pertinent** indices.

For all Pertinent indices we define the quantity $E(s, u, i; t, v, j)$ to be as follows :

$$E(s, u, i; t, v, j) = \text{Min} \left\{ \text{Cost}(M) \mid M \text{ is a mapping from } T[s..i] \text{ to } T'[t..j] \text{ such that conditions (i) (ii) and (iii) are satisfied.} \right\}$$

where these conditions are :

- (i) $(s, t) \in M$,
- (ii) no descendant of $T[s]$ on the path from $T[s]$ to $T[u]$ is touched by any line of M , and,
- (iii) no descendant of $T'[t]$ on the path from $T'[t]$ to $T'[v]$ is touched by any line of M .

Then, from Lemma 3.7, we have the following result :

$$\begin{aligned} & \text{Min_M}(i+1, j+1) \\ &= d(T[i+1], T[j+1]) \\ &+ \min_{(s,t)} \left\{ \text{Min_M}(s, t) + E(s, f(i+1), i; t, f(j+1), j) - d(T[s], T[t]) \right\} \end{aligned} \quad (3.18)$$

where $T[s]$ and $T[t]$ are ancestors of $T[i+1]$ and $T[j+1]$, respectively.

We now show how the quantity $E[s, u, i; t, v, j]$ can be computed for all the Pertinent indices whenever $T[u]$ is on the path from $T[s]$ to $T[i]$, and $T[v]$ is on the path from $T[t]$ to $T[j]$. We shall derive the expression by considering two mutually exclusive and exhaustive cases as follows :

Case (A) : $s \leq u < i$ and $t \leq v < j$.

In this case, $T[u]$ has a son on the path from $T[u]$ to $T[i]$ and $T[v]$ has a son on the path from $T[v]$ to $T[j]$. Let us consider Figure 3.9 in which $T[x]$ is the son of $T[u]$ on the path from $T[u]$ to $T[i]$ and $T[y]$ is the son of $T[v]$ on the path from $T[v]$ to $T[j]$. The darker line in tree T , $T[s..i]$, indicates that any descendant of $T[s]$ on the path from $T[s]$ to $T[u]$ is not touched by any line of M . The darker line in $T[t..j]$ is defined analogously.

Figure 3.9 : Evaluate $E[s, u, i; t, v, j]$ when $s \leq u < i$ and $t \leq v < j$.

By the definition of a mapping, in this case, at least one of the following three subcases must hold :

(A.1) If $T[x]$ is not touched by a line of M , then

$$E(s, u, i : t, v, j) = E(s, x, i : t, v, j). \quad (3.19)$$

(A.2) If $T[y]$ is not touched by a line of M , then

$$E(s, u, i : t, v, j) = E(s, u, i : t, y, j). \quad (3.20)$$

(A.3) If $T[x]$ and $T[y]$ are touched by lines (x, q) and (p, y) of M , then the argument is more involved. Assume $p > x$. Then $i \geq p > x$, which means $T[p]$ is a descendant of $T[x]$. This implies that $T[y]$ is an descendant of $T[q]$, which in turn implies that $T[q]$ is on the path from $T[t]$ to $T[v]$. However, this contradicts the assumption that no descendant of $T[t]$ on the path from $T[t]$ to $T[v]$ is touched by a line in M . A similar contradiction also exists if we assume that $p < x$, and consequently, p must be identically equal to x . Similarly q must be identically equal to y . Thus, M can be decomposed into M_1 from $T[s..x-1]$ to $T[t..y-1]$ and M_2 from $T[x..i]$ to $T[y..j]$, implying that :

$$\text{Cost}(M) = \text{Cost}(M_1) + \text{Cost}(M_2).$$

Since $\text{Cost}(M) = E(s, u, i : t, v, j)$, it follows that

$$\text{Cost}(M_1) = E(s, u, x-1 : t, v, y-1), \quad \text{and} \quad \text{Cost}(M_2) = E(x, x, i : y, y, j),$$

and consequently,

$$E(s, u, i : t, v, j) = E(s, u, x-1 : t, v, y-1) + E(x, x, i : y, y, j). \quad (3.21)$$

Since these three subcases exhaust all the possible mappings for pertinent s, u, i, v, t , and j , satisfying $s \leq u < i$ and $t \leq v < j$, we take the minimum of the three costs given by

equations (3.19) to (3.21) to yield :

$$E(s,u,i:t,v,j) = \text{Min} \begin{cases} E(s,x,i:t,v,j) \\ E(s,u,i:t,y,j) \\ E(s,u,x-1:t,v,y-1) + E(x,x,i:y,y,j) \end{cases} \quad (3.22)$$

Case (B) : $u = i$ or $v = j$.

In this case, one or both of $T[x]$ and $T[y]$ do not exist. Three subcases given by B.1 to B.3 emerge :

(B.1) If $u = i$ and $v < j$, then there are two subcases to be considered :

Figure 3.10 : Evaluate $E[s,u,i:t,v,j]$ when $u = i$ or $v = j$.

(B.1.1) If $s = u = i$, then $T[s..i]$ contains exactly one node, $T[s]$. Since no descendant of $T[t]$ is touched by a line of M , node $T[j]$ must be inserted in every edit sequence associated with this mapping. From Lemma 3.1, $T[f(j)]$ is on the path from $T[t]$ to $T[j-1]$, and thus,

$$E(s,u,i:t,v,j) = E(s,u,i:t,f(j),j-1) + d(\lambda, T[j]). \quad (3.23)$$

(B.1.2) If $s < u = i$, then, by Lemma 3.1, $T[f(i)]$ is on the path from $T[s]$ to $T[i-1]$. Since no descendant of $T[s]$ on the path from $T[s]$ to $T[f(i)]$ is touched by a line of M , node $T[i]$ must be deleted in every edit sequence associated with this mapping. It follows that

$$E(s, u, i: t, v, j) = E(s, f(i), i-1: t, v, j) + d(T[i], \lambda). \quad (3.24)$$

(B.2) If $u < i$ and $v = j$, the symmetric scenario results (i.e., T and T' interchange roles), and leads to cases (B.2.1) and (B.2.2) parallel to cases (B.1.1) and (B.1.2) above. Thus, we have,

(B.2.1) If $t = v = j$, then

$$E(s, u, i: t, v, j) = E(s, f(i), i-1: t, v, j) + d(T[i], \lambda). \quad (3.25)$$

(B.2.2) If $t < v = j$, then

$$E(s, u, i: t, v, j) = E(s, u, i: t, f(j), j-1) + d(\lambda, T[j]). \quad (3.26)$$

(B.3) If $u = i$ and $v = j$, then there are four subcases to be considered. We consider each of them individually in (B.3.1) to (B.3.4) below.

Figure 3.11 : Evaluate $E[s, u, i: t, v, j]$ when $u = i$ and $v = j$.

(B.3.1) If $s = u = i$ and $t = v = j$, then M is a mapping between two single-node subtrees. By the definition of $E(s, u, i, t, v, j)$, $T[i]$ is connected to $T[j]$ and so we have :

$$E(s, u, i, t, v, j) = d(T[i], T[j]). \quad (3.27)$$

(B.3.2) If $s = u = i$ and $t < v = j$, then node $T[j]$ cannot be touched by any such mapping since (s, t) must be in every such mapping. Hence, we again obtain :

$$E(s, u, i, t, v, j) = E(s, u, i, t, f(j), j-1) + d(\lambda, T[j]). \quad (3.28)$$

(B.3.3) If $s < u = i$ and $t = v = j$ (the symmetric cases of (B.3.2)), then node $T[i]$ cannot be touched by any such mapping since (s, t) must be in every such mapping. Hence, we can conclude :

$$E(s, u, i, t, v, j) = E(s, f(i), i-1, t, v, j) + d(T[i], \lambda). \quad (3.29)$$

(B.3.4) If $s < u = i$ and $t < v = j$ then, neither $T[i]$ nor $T[j]$ is touched by a line of M . Therefore,

$$E[s, u, i, t, v, j] = \begin{cases} E(s, f(i), i-1, t, v, j) + d(T[i], \lambda) & \text{or} \\ E(s, u, i, t, f(j), j-1) + d(\lambda, T[j]) & \text{or} \\ E(s, f(i), i-1, t, f(j), j-1) + d(T[i], \lambda) + d(\lambda, T[j]). \end{cases} \quad (3.30)$$

Observe that, whereas eight subcases are considered (refer to the equations (3.23) to (3.30)), only four different equations are used to compute the value of $E[s, u, i, t, v, j]$ for the case when $u = i$ or $v = j$.

Based on the above results, an algorithm for computing the minimum edit distance between two trees subject to the constraint that the root remains unchanged is presented formally as Algorithm Tai [Ta79] below.

ALGORITHM Tai**INPUT :** Two trees T and T' .**OUTPUT :** The minimum edit distance $D(T, T')$.**ASSUMPTIONS :** The algorithm invokes three Procedures named Compute_E , Compute_Min_M and Compute_H . The parameters for the procedures, namely the arrays E , Min_M and H are assumed global.**BEGIN** Compute_E ; Compute_Min_M ; Compute_H ; Return ($H[|T|][|T'|]$);**End ALGORITHM Tai.**

As we can see the algorithm involves three central procedures. First of all, an array $E[\dots\dots\dots]$ is evaluated by using the results shown in equations (3.19) to (3.30) for all relevant s , u , i , t , v , and j . This is done in Procedure Compute_E . Subsequently, by using these computed values, the minimum cost submappings $\text{Min_M}(i,j)$ are computed for all i and j , where $1 \leq i \leq |T|$ $1 \leq j \leq |T'|$, and this is done in the procedure Compute_Min_M which utilizes the result shown in equation (3.18). Finally, by using these computed costs, the edit distances between the trees $T[1..i]$ and $T'[1..j]$ are computed for all values of i and j , where $1 \leq i \leq |T|$ and $1 \leq j \leq |T'|$ in the procedure Compute_H which utilizes the results shown in equation (3.13). The details of these three individual procedures are presented formally below.

PROCEDURE Compute E**OUTPUT :** The array E for all valid indices s, u, i, t, v, j , where

- (i) $1 \leq i \leq |T|, 1 \leq j \leq |T|$,
- (ii) $T[u]$ is on the path from $T[1]$ to $T[i]$
- (iii) $T[s]$ is on the path from $T[1]$ to $T[u]$
- (iv) $T[v]$ is on the path from $T[1]$ to $T[j]$, and,
- (v) $T[t]$ is on the path from $T[1]$ to $T[v]$.

ASSUMPTIONS : E is assumed global.**NOTATION :** In this procedure :

- (i) $T[x]$ will refer to the son of $T[u]$ on the path from $T[u]$ to $T[i]$, and,
- (ii) $T[y]$ will refer to the son of $T[v]$ on the path from $T[v]$ to $T[j]$.

BEGINFOR $i = 1, 2, \dots, |T|$ DOFOR $j = 1, 2, \dots, |T|$ DOFOR $u = i, f(i), f^2(i), \dots, 1$ DOFOR $s = u, f(u), f^2(u), \dots, 1$ DOFOR $v = j, f(j), f^2(j), \dots, 1$ DOFOR $t = v, f(v), f^2(v), \dots, 1$ DOIF $(s = u = i)$ and $(t = v = j)$ THEN $E[s][u][i][t][v][j] = d(T[i], T[j])$;ELSEIF $(s = u = i)$ or $(t < v = j)$ THEN $E[s][u][i][t][v][j] = E[s][u][i][t][f(j)][j-1] + d(\lambda, T[j])$;ELSEIF $(s < u = i)$ or $(t = v = j)$ THEN $E[s][u][i][t][v][j] = E[s][f(i)][i-1][t][v][j] + d(T[i], \lambda)$;

ELSE

 $E[s][u][i][t][v][j]$

$$= \text{Min} \begin{cases} E[s][x][i][t][v][j] \\ E[s][u][i][t][y][j] \\ E[s][u][x-1][t][v][y-1] + E[x][x][i][y][y][j] \end{cases} ;$$

ENDIF

ENDFOR

ENDFOR

ENDFOR

ENDFOR

ENDFOR

ENDFOR

END PROCEDURE Compute_E

PROCEDURE Compute Min_M**OUTPUT :** The array Min_M for all i, j , where $1 \leq i \leq |T|$ and $1 \leq j \leq |T|$.**ASSUMPTIONS :** Min_M and E are assumed global.**BEGIN** $\text{Min_M}[1][1] = 0;$ **FOR** $i = 2, 3, \dots, |T|$ **DO** **FOR** $j = 2, 3, \dots, |T|$ **DO** $\text{Min_M}[i][j] = \infty;$ **FOR** $s = f(i), f^2(i), \dots, 1$ **DO** **FOR** $t = f(j), f^2(j), \dots, 1$ **DO** $\text{temp} = \text{Min_M}[s][t] + E[s][f(i)][i-1][t][f(j)][j-1] - d(T[s], T[t]);$ $\text{Min_M}[i][j] = \text{Min}(\text{temp}, \text{Min_M}[i][j]);$ **ENDFOR** **ENDFOR** $\text{Min_M}[i][j] = \text{Min_M}[i][j] + d(T[i], T[j]);$ **ENDFOR** **ENDFOR****END PROCEDURE Compute_MIN_M****PROCEDURE Compute H****OUTPUT :** The array H for all i, j , where $1 \leq i \leq |T|$ and $1 \leq j \leq |T|$.**ASSUMPTIONS :** Min_M and H are assumed global.**BEGIN** $H[1][1] = 0;$ **FOR** $i = 2, 3, \dots, |T|$ **DO** $H[i][1] = H[i-1][1] + d(T[i], \lambda);$ **ENDFOR** **FOR** $j = 2, 3, \dots, |T|$ **DO** $H[1][j] = H[1][j-1] + d(\lambda, T[j]);$ **ENDFOR** **FOR** $i = 2, 3, \dots, |T|$ **DO** **FOR** $j = 2, 3, \dots, |T|$ **DO**

$$H[i][j] = \text{Min} \begin{cases} H[i][j-1] + d(\lambda, T[j]) \\ H[i-1][j] + d(T[i], \lambda) \\ \text{Min_M}[i][j] \end{cases};$$
 ENDFOR **ENDFOR****END PROCEDURE Compute_H**

The time and space complexities of Algorithm Tai can be easily determined. Let M and N be the numbers of nodes of T and T' respectively, and $\text{Depth}(T)$ and $\text{Depth}(T')$ denote the maximum depths of T and T' respectively, then

- (i) Procedure `Compute_E` takes $O(M \cdot N \cdot \text{Depth}(T)^2 \cdot \text{Depth}(T')^2)$ time and space,
- (ii) Procedure `Compute_Min_M` takes $O(M \cdot N \cdot \text{Depth}(T) \cdot \text{Depth}(T'))$ time and space, and,
- (iii) Procedure `Compute_H` takes $O(M \cdot N)$ time and space.

Thus, using the above procedures, the distance $D(T, T')$ can be computed in time $O(M \cdot N \cdot \text{Depth}(T)^2 \cdot \text{Depth}(T')^2)$.

The space complexity is the same as the time complexity.

3.4.2 Tai's Algorithm Rendered Recursive

Although Algorithm Tai seems to be a perfect solution for the tree editing problem in which the root cannot be deleted or inserted, it can be improved by carefully studying the definition of $E(s, u, i:t, v, j)$. Recall from Section 3.4.1 that

$$E(s, u, i:t, v, j) = \text{Min} \left\{ \text{Cost}(M) \mid M \text{ is a mapping from } T[s..i] \text{ to } T'[t..j] \text{ such that conditions (i) (ii) and (iii) are satisfied.} \right\}$$

where these conditions are :

- (i) $(s, t) \in M$,
- (ii) no descendant of $T[s]$ on the path from $T[s]$ to $T[u]$ is touched by any line of M , and,
- (iii) no descendant of $T'[t]$ on the path from $T'[t]$ to $T'[v]$ is touched by any line of M .

Based on this definition, Noetzel and Selkow [SK83] derived a recursive scheme using a structure called a *Restricted Mapping*. For any $s \leq u \leq i$ and $t \leq v \leq j$, let $\Psi(s, u, i; t, v, j)$ be the *class* of all mappings from $T[s..i]$ to $T'[t..j]$ that contain (s, t) but which do not touch any descendant of $T[s]$ on the path from $T[s]$ to $T[u]$, or any descendant of $T'[t]$ on the path from $T'[t]$ to $T'[v]$. Mappings of this class $\Psi(s, u, i; t, v, j)$ will be called *Restricted Mappings*.

If trees T and T' have m and n nodes, respectively, then, from the above definition, $\Psi(1, 1, m; 1, 1, n) = \Phi(1, m; 1, n)^4$, and hence,

$$\begin{aligned} E(1, 1, m; 1, 1, n) &= \text{Min} \left\{ \text{Cost}(M) \mid M \in \Psi(1, 1, m; 1, 1, n) \right\} \\ &= \text{Min} \left\{ \text{Cost}(M) \mid M \in \Phi(1, m; 1, n) \right\} \\ &= D(T, T') \end{aligned}$$

is the minimum-cost being sought.

In other words, to compute $D(T, T') = E(1, 1, m; 1, 1, n)$, the procedures `Compute_Min_M` and `Compute_H` in Algorithm Tai can be omitted and the procedure `Compute_E` can be rewritten as so as to yield the following Algorithm [SK83] which evaluates $E(s, u, i; t, v, j)$ using the recursive function `Eval_E`.

⁴Note that the original definition for the set of unrestricted mappings, $\Phi(a, m; b, n)$, as per the definition in Section 3.2.2 permits only four parameters. In this case, we are referring to the set of restricted mappings, and these are given by the set where each element is defined by six parameters, $\Psi(s, u, i; t, v, j)$.

ALGORITHM Noetzel and Selkow**INPUT :** Two trees T and T' .**OUTPUT :** The minimum edit distance $D(T, T')$.**ASSUMPTION :** The algorithm assumes that a procedure $f(\cdot)$ is available which preprocesses the trees and yields the values $f(x)$ for all x (i.e. the father of all nodes) in both trees.**NOTATION :** In this procedure :

- (i) $T[x]$ will refer to the son of $T[u]$ on the path from $T[u]$ to $T[i]$, and,
- (ii) $T'[y]$ will refer to the son of $T'[v]$ on the path from $T'[v]$ to $T'[j]$.

FUNCTION Eval_E (s,u,i:t,v,j)**BEGIN**If $(s = i)$ and $(t = j)$ THEN $\text{Eval_E} := d(T[i], T[j])$;ELSEIF $(u = i)$ or $(t = j)$ THEN $\text{Eval_E} := \text{Eval_E}(s, f(i), i-1:t, v, j) + d(T[i], \lambda)$;ELSEIF $(s = i)$ or $(v = j)$ THEN $\text{Eval_E} := \text{Eval_E}(s, u, i:t, f(j), j-1) + d(\lambda, T'[j])$;

ELSE

$$\text{Eval_E} := \text{Min} \begin{cases} \text{Eval_E}(s, x, i:t, v, j) \\ \text{Eval_E}(s, u, i:t, y, j) \\ \text{Eval_E}(s, u, x-1:t, v, y-1) + \text{Eval_E}(x, x, i:y, y, j) \end{cases} ;$$
ENDIF**END Function.****BEGIN** $D(T, T') = \text{Eval_E}(1, 1, m:1, 1, n)$;**END Algorithm Noetzel and Selkow.****3.5 Faster Algorithm of the Tree Editing Problem**

The $O(|T_1| \cdot |T_2| \cdot (\text{Depth}(T_1))^2 \cdot (\text{Depth}(T_2))^2)$ time and space complexities of Tai's algorithm are not desirable, especially in practical applications. For example, suppose that we are to compute the edit distance between two trees with 1,000 nodes each (which is not unusual when the similarity between two R.N.A. molecules is to be evaluated). Let us assume that the machine has a capability of executing five million instructions per second. Let us further assume that each integer requires 2 bytes of memory. Then, the time

2

of/de

2

PM-1 3½"x4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANL./ISO #2 EQUIVALENT

required by Algorithm Tai for computing the edit distance will be $O(10^{12})$ seconds, which is approximately 6,000 years. Similarly, the space required by the algorithm will be $O(10^{18})$ bytes, which is of the order of a quintillion bytes.

We shall now consider a faster algorithm due to Zhang and Shasha [ZS89]. This algorithm (unlike Tai's) is applicable for the general tree editing problem and is considerably faster. Indeed, if $\text{Span}(T)$ is defined as the $\text{Min}\{\text{Depth}(T), \text{Leaves}(T)\}$, the time and space complexities of this algorithm are :

$$\begin{aligned} \text{Time :} & \quad O(|T_1| * |T_2| * \text{Span}(T_1) * \text{Span}(T_2)), & \text{and,} \\ \text{Space :} & \quad O(|T_1| * |T_2|). \end{aligned}$$

The Zhang and Shasha algorithm, unlike the previously discussed algorithms, considers the distance between two ordered forests. Also, as opposed to the previous researchers, Zhang and Shasha used the *postorder numbering*, instead of preorder numbering, to number the nodes in the trees. With the postorder numbering, $T[1..i]$ will generally (and almost always) be a forest as shown in Figure 3.12. Note that the definition of a mapping for ordered forests is exactly the same as that for trees.

Let $T[i]$ be the i^{th} node in the tree according to the left-to-right postorder numbering, and $\delta(i)$ represent the postorder number of the leftmost leaf descendant of the subtree rooted at $T[i]$. Note that in the case when $T[i]$ is a leaf, $\delta(i) = i$.

Let $T[i..j]$ represents the postorder forest induced by nodes $T[i]$ to $T[j]$ inclusive, of tree T . $T[\delta(i)..i]$ will be referred to as $\text{Tree}(i)$. $\text{Size}(i)$ is the number of nodes in $\text{Tree}(i)$. Figure 3.12 clarifies these notations.

For convenience, we will use $F_Wt(T_1[i'..i], T_2[j'..j])$ to denote the distance between forest $T_1[i'..i]$ and forest $T_2[j'..j]$. The distance between the subtree rooted at i and the subtree rooted at j is denoted by $T_Wt(i, j)$.

When the postorder numbering of trees is used, the following results about the quantity F_Wt can be derived.

Figure 3.12 : An example of a tree, a postorder forest, a subtree, and the associated notations.

Lemma 3.8 : Let $i_1 \in \text{Anc}(i)$ and $j_1 \in \text{Anc}(j)$, then,

$$(i) \quad F_Wt(\phi, \phi) = 0. \quad (3.31)$$

$$(ii) \quad F_Wt(T_1[\delta(i_1)..i], \phi) = F_Wt(T_1[\delta(i_1)..i-1], \phi) + d(T_1[i], \lambda). \quad (3.32)$$

$$(iii) \quad F_Wt(\phi, T_2[\delta(j_1)..j]) = F_Wt(\phi, T_2[\delta(j_1)..j-1]) + d(\lambda, T_2[j]). \quad (3.33)$$

PROOF :

Case (i) requires no edit operation. In cases (ii) and (iii), the distances correspond to the cost of deleting and inserting the nodes in $T_1[\delta(i_1)..i]$ and $T_2[\delta(j_1)..j]$, respectively.

Note that equations (3.31), (3.32), and (3.33) govern the boundary values of $F_Wt(\dots)$. For the intermediate values of the $F_Wt(\dots)$, we have the following lemma :

Lemma 3.9 : Let $i_1 \in \text{Anc}(i)$ and $j_1 \in \text{Anc}(j)$, then,

$$\begin{aligned}
 & F_Wt(\delta(i_1)..i, \delta(j_1)..j) \\
 &= \text{Min} \begin{cases} F_Wt(\delta(i_1)..i-1, \delta(j_1)..j) + d(T_1[i], \lambda) \\ F_Wt(\delta(i_1)..i, \delta(j_1)..j-1) + d(\lambda, T_2[j]) \\ F_Wt(\delta(i_1)..i-1, \delta(j_1)..j-1) \\ \quad + F_Wt(\delta(i)..i-1, \delta(j)..j-1) + d(T_1[i], T_2[j]). \end{cases} \quad (3.33)
 \end{aligned}$$

PROOF :

This lemma is a special case of a more general result involving arbitrary similarity metrics comparing trees T_1 and T_2 . Rather than prove this special case result we shall prove the more general result in the next chapter. However, an alternate proof (utilizing techniques distinct from the ones which we have used in Chapter IV) which justifies merely this result is also given by Zhang and Shasha [ZS89]. ***

Lemma 3.9 suggests a dynamic programming algorithm to solve the tree editing problem. However, a slight modification to Lemma 3.9 will lead to an algorithm which takes less time and space. The property which relates the F_Wt and T_Wt indices, and which motivates this modification is given below.

Lemma 3.10 : Let $i_1 \in \text{Anc}(i)$ and $j_1 \in \text{Anc}(j)$, then,

If $\delta(i) = \delta(i_1)$ and $\delta(j) = \delta(j_1)$ then

$$\begin{aligned}
 & F_Wt(T_1[\delta(i_1)..i], T_2[\delta(j_1)..j]) \\
 &= \text{Min} \begin{cases} F_Wt(T_1[\delta(i_1)..i-1], T_2[\delta(j_1)..j]) + d(T_1[i], \lambda) \\ F_Wt(T_1[\delta(i_1)..i], T_2[\delta(j_1)..j-1]) + d(\lambda, T_2[j]) \\ F_Wt(T_1[\delta(i_1)..i-1], T_2[\delta(j_1)..j-1]) + d(T_1[i], T_2[j]) \end{cases}
 \end{aligned}$$

Otherwise,

$$F_Wt(T_1[\delta(i_1)..i], T_2[\delta(j_1)..j]) \\ = \text{Min} \begin{cases} F_Wt(T_1[\delta(i_1)..i-1], T_2[\delta(j_1)..j]) + d(T_1[i], \lambda) \\ F_Wt(T_1[\delta(i_1)..i], T_2[\delta(j_1)..j-1]) + d(\lambda, T_2[j]) \\ F_Wt(T_1[\delta(i_1)..i-1], T_2[\delta(j_1)..j-1]) + T_Wt(i, j). \end{cases}$$

PROOF :

By Lemma 3.9, if $\delta(i) = \delta(i_1)$ and $\delta(j) = \delta(j_1)$, the forests $T_1[\delta(i_1)..i-1]$ and $T_2[\delta(j_1)..j-1]$ will both be empty. Thus,

$$F_Wt(T_1[\delta(i_1)..i-1], T_2[\delta(j_1)..j-1]) = F_Wt(\phi, \phi)$$

which is equal to zero and the first part of the Lemma follows.

For the second part, since the edit distance is the cost of the minimal cost mapping, we know that:

$$F_Wt(T_1[\delta(i_1)..i], T_2[\delta(j_1)..j]) \\ \leq F_Wt(T_1[\delta(i_1)..i-1], T_2[\delta(j_1)..j-1]) + T_Wt(i, j). \quad (3.35)$$

This is true since the latter formula represents a particular mapping of $T_1[\delta(i_1)..i]$ to $T_2[\delta(j_1)..j]$ in which forest $T_1[\delta(i_1)..i-1]$ is transformed into the forest $T_2[\delta(j_1)..j-1]$ and subsequently the tree rooted at i is transformed into the tree rooted at j . Thus the second term in the above expression is a $T_Wt(.,.)$ and not an $F_Wt(.,.)$. For the same reason, we have :

$$T_Wt(i, j) \leq F_Wt(T_1[\delta(i_1)..i-1], T_2[\delta(j_1)..j-1]) + d(T_1[i], T_2[j]). \quad (3.36)$$

Lemma 3.9 and inequalities (3.35) and (3.36) justify the substitution of $T_Wt(i, j)$ for the corresponding F_Wt expressions, and the result follows. ***

To utilize the above results in a formal algorithm Zhang and Shasha introduced the following definition.

Definition : We define the set LR_keyroots of tree T as follows :

$$\text{LR_keyroots}(T) = \{k \mid \text{there exists no } k' > k \text{ such that } \delta(k) = \delta(k')\}.$$

Note that if k is in $\text{LR_keyroots}(T)$, then either k is the root or k has a left sibling. Also note that the function $\delta(\cdot)$ and the set $\text{LR_keyroots}()$ can be computed in linear time. We assume that the results are stored in arrays $\delta[]$ and $\text{LR_keyroots}[]$ respectively, and that the array $\text{LR_keyroots}[]$ is sorted in the increasing order of the postorder numbering.

Using these quantities, Algorithm Zhang and Shasha is presented formally below.

ALGORITHM Zhang and Shasha

INPUT : Trees T_1 and T_2 and the set of elementary edit distances.

OUTPUT : $T_Wt(i, j)$, $1 \leq i \leq |T_1|$, $1 \leq j \leq |T_2|$.

ASSUMPTION : The algorithm assumes that a procedure **Preprocess** (T_1, T_2) is available which preprocesses the trees and yields the $\delta[]$ and $\text{LR_keyroots}[]$ arrays for both trees. These quantities are assumed to be global. It also invokes the procedure **Compute_T_Wt()** presented later.

BEGIN

Preprocess (T_1, T_2) ;

FOR $i' = 1$ to $|\text{LR_keyroots}_1[]|$ DO

FOR $j' = 1$ to $|\text{LR_keyroots}_2[]|$ DO

$i = \text{LR_keyroots}_1[i']$;

$j = \text{LR_keyroots}_2[j']$;

Compute_T_Wt (i, j) ;

ENDFOR

ENDFOR

END ALGORITHM Zhang and Shasha.

PROCEDURE Compute_T_Wt

INPUT : Indexes i, j and the quantities assumed global in Algorithm Zhang and Shasha.

OUTPUT : $T_Wt[i_1][j_1]$, $\delta_1(i) \leq i_1 \leq i$, $\delta_2(j) \leq j_1 \leq j$.

BEGIN

$F_Wt[\phi][\phi] = 0;$

FOR $i_1 := \delta(i)$ **to** i **DO**

$F_Wt[T_1[\delta(i)..i_1]][\phi] := F_Wt[T_1[\delta(i)..i_1-1]][\phi] + d(T_1[i_1] \rightarrow \lambda);$

ENDFOR

FOR $j_1 := \delta(j)$ **to** j **DO**

$F_Wt[\phi][T_2[\delta(j)..j_1]] := F_Wt[\phi][T_2[\delta(j)..j_1-1]] + d(\lambda \rightarrow T_2[j_1]);$

ENDFOR

FOR $i_1 := \delta(i)$ **to** i **DO**

FOR $j_1 := \delta(j)$ **to** j **DO**

IF $\delta(i_1) = \delta(i)$ **and** $\delta(j_1) = \delta(j)$ **THEN**

$F_Wt[T_1[\delta(i)..i_1]][T_2[\delta(j)..j_1]]$

$= \text{Min} \begin{cases} F_Wt[T_1[\delta(i)..i_1-1]][T_2[\delta(j)..j_1]] + d(T_1[i_1] \rightarrow \lambda) \\ F_Wt[T_1[\delta(i)..i_1]][T_2[\delta(j)..j_1-1]] + d(\lambda \rightarrow T_2[j_1]) \\ F_Wt[T_1[\delta(i)..i_1-1]][T_2[\delta(j)..j_1-1]] + d(T_1[i_1] \rightarrow T_2[j_1]) \end{cases}$

$T_Wt[i_1][j_1] = F_Wt[T_1[\delta(i)..i_1], T_2[\delta(j)..j_1]]$

ELSE

$F_Wt[T_1[\delta(i)..i_1]][T_2[\delta(j)..j_1]]$

$= \text{Min} \begin{cases} F_Wt[T_1[\delta(i)..i_1-1]][T_2[\delta(j)..j_1]] + d(T_1[i_1] \rightarrow \lambda) \\ F_Wt[T_1[\delta(i)..i_1]][T_2[\delta(j)..j_1-1]] + d(\lambda \rightarrow T_2[j_1]) \\ F_Wt[T_1[\delta(i)..i_1-1]][T_2[\delta(j)..j_1-1]] + T_Wt[i_1][j_1] \end{cases}$

ENDIF

ENDFOR

ENDFOR

END PROCEDURE Compute_T_Wt

After the array $T_Wt[.,.]$ has been computed, the distance $D(T_1, T_2)$ between trees T_1 and T_2 is stored in the array $T_Wt[|T_1|, |T_2|]$.

It was shown in [ZS89] that if, for a tree, $\text{Span}(T)$ is defined as the $\text{Min}(\text{Depth}(T), \text{Leaves}(T))$, the time and space complexities of Algorithm Zhang and Shasha are :

Time : $O(|T_1| * |T_2| * \text{Span}(T_1) * \text{Span}(T_2))$

Space : $O(|T_1| * |T_2|)$.

The proof of this claim is omitted here, but the result can be seen to be true from the proof of the more general result given in Chapter IV as Theorem 4.10.

3.6 Conclusions

In this chapter we have presented a brief overview of the papers dealing with the tree editing problem. To the best of our knowledge, little work has been done in this area. The first algorithm which we discussed was due to Selkow [Se77] which dealt with editing trees in which the insertion and deletion operations were restricted because they could only be done at the leaves. The next algorithm presented was due to Tai [Ta79] which was a less-restricted algorithm. In the case of the latter, edit operations could be done at every node except the root, whose node value could only be substituted. A recursive version of Tai's algorithm [Ta79] due to Noetzel and Selkow [SK83] was also briefly discussed. Finally, the best known algorithm for evaluating the edit distance in the unrestricted case was also presented. This algorithm is due to Zhang and Shasha [ZS89].

In the next chapter, we generalize the technique introduced in this chapter to yield families of similarity and dissimilarity measures involving the trees under consideration.

All of the algorithms discussed in this chapter consider the editing of one tree and transforming it into another, when the edit process is **unconstrained** (except due to the limitation of the algorithm in question). In Chapter V, we will study a more complicated tree editing problem, namely, the **Constrained Tree Editing Problem**. In the case of this problem, the numbers and types of edit operations can be arbitrarily specified by the user. The techniques which we have introduced in this chapter will be foundational for understanding the algorithm introduced in Chapter V.

IV. Numeric Similarity and Dissimilarity Measures Between Two Trees

4.1 Introduction

For a given pair of strings, there are many metrics which quantify the similarity and dissimilarity between them. Their Generalized Levenshtein Distance (GLD), the Length of their Longest Common Subsequence (LLCS), the Length of their Shortest Common Supersequence (LSCS) (a list of references using these can be found in Section 2.2, Section 2.3, and Section 2.4, respectively), and their transmission-reception probabilities ([KO82], [KO83], [KO84]) are examples of these numeric indices. Others, like the set of their Longest Common Subsequences, the set of their Shortest Common Supersequences, and the set of their Shuffles are the examples of various nonnumeric quantities. Furthermore, as mentioned in Section 2.5, Kashyap and Oommen [KO83] presented a common mathematical framework by which *all* these quantities can be related. In their paper, they defined an abstract measure between two strings X and Y , denoted by $D(X, Y)$. The latter was defined in terms of two abstract operators, and a binary function, $d(\dots)$, whose arguments are essentially symbols of the alphabet $\mathcal{A}$ and the null symbol. Depending on various concrete operators used for the abstract operators and the specific function used for $d(\dots)$, all the numeric and nonnumeric quantities discussed above can be seen to be particular cases of $D(X, Y)$.

As opposed to the case of strings, the literature reports only *one* numeric dissimilarity measure relating two trees. This measure is indeed the “distance” between the trees as measured by the minimum cost sequence of edit operations, and was discussed in detail in the previous chapter. Typically, as mentioned earlier, the edit sequence considered includes the substitution, insertion, and deletion of nodes needed to transform one tree into

the other. In this chapter, we define a few more new numeric indices which measure the similarity between two trees. These include the Size of their Largest Common Sub-Tree (SLCsT), the Size of their Smallest Common Super-Tree (SSCST), and the transmission-receptive probabilities of the trees when the transmission is done across channels with a variety of properties. Indeed, more generally, we will define entire families of numeric measures involving the trees in question.

The central issue in this chapter is the definition and formulation of an abstract measure of comparison, $\Omega(T_1, T_2)$, between two trees T_1 and T_2 . This measure will be presented in terms of a set of elementary inter-symbol measures $\omega(\dots)$ and two abstract operators $\oplus$ and $\otimes$. By appropriately choosing the concrete values for these two operators and for $\omega(\dots)$, the SLCsT and the SSCST can be obtained as special cases. The quantity $\text{Prob}(T_2|T_1)$, the probability of receiving T_2 given that T_1 was transmitted across a channel causing independent substitution and deletion errors, can also be seen to be another special case of $\Omega(T_1, T_2)$. Finally, the same abstract measure can be used to quantify the probability of T_1 being the transmitted tree given that T_2 is the received tree containing independent substitution, insertion and deletion errors and that the transmitted tree is an element from a finite dictionary.

Apart from the definition and formulation of $\Omega(T_1, T_2)$, we shall also derive the recursive properties of the measure whence an iterative dynamic programming scheme to compute $\Omega(T_1, T_2)$ will be presented. Thus, the same algorithm with the appropriate choice of the operators and of $\omega(\dots)$ can be used to compute all of the above measures.

Our algorithm is similar to the earlier foundational work of Zhang and Shasha [ZS89] discussed in Section 3.5. However, we emphasize that our scheme is not merely a straightforward extension of [ZS89] in which the operators and the elementary distances are correspondingly generalized. The differences between our present scheme and the results in [ZS89] will be clarified in a subsequent section.

4.2 Notations and Definitions

Unless otherwise specified, the notation used in this chapter is similar to that used in Chapter III. The definitions of the edit operations and mappings are identical to the definitions shown in Section 3.2, and is omitted here.

4.2.1 Sub-Trees and Super-Trees

A tree $T_1 \in \mathcal{N}^*$ is a *Sub-Tree* of a tree $T_2 \in \mathcal{N}^*$ if it can be obtained from T_2 by deleting zero or more nodes from T_2 . It is a *Common Sub-Tree* of two trees T_2 and T_3 if it is a Sub-Tree of both T_2 and T_3 , and it is their *Largest Common Sub-Tree (LCsT)* if and only if it is a common sub-tree of both that is among the largest of the common sub-trees. The size of the LCsT is referred to as the *Size of their Largest Common Sub-Tree (SLCsT)*. Examples of these are shown in Example 4.1 (See Figure 4.1 on next page).

A similar but dual problem of the LCsT problem is the one of finding the Smallest Common Super-Tree (SCST) of two trees. A tree $T_1 \in \mathcal{N}^*$ is a *Super-Tree* of $T_2 \in \mathcal{N}^*$ if it can be obtained from T_2 by inserting zero or more symbols into it. It is a *Common Super-Tree* of two trees T_2 and T_3 if it is a super-tree of both T_2 and T_3 ; it is their *Smallest Common Super-Tree (SCST)* of two trees T_2 and T_3 if and only if it is a common super-tree of both that is as small as any other common super-trees. The size of the SCST is referred to as the *Size of their Smallest Common Super-Tree (SSCST)*. Examples of SCST and SSCST are shown in Example 4.2 (See Figure 4.2).

4.3 An Abstract Measure Involving Two Trees

We consider a functional $\Omega(T_1, T_2)$, between two trees T_1 and T_2 , $T_1, T_2 \in \mathcal{N}^*$, induced by a system $(\mathcal{N}, \tau, \omega)$, where $\mathcal{N}$ is the alphabet set under consideration, τ is an algebraic structure described below and ω is a binary function over $\tilde{\mathcal{N}} \times \tilde{\mathcal{N}}$. τ and ω are defined as follows.

EXAMPLE 4.1 :

Then

are examples of the Sub-Tree of T_2 , and is a LCST of T_2 and T_3 with
SLCST (T_2, T_3) = 4.**Figure 4.1 : Examples of Sub-Trees, LCsT and SLCsT.**

EXAMPLE 4.2 : **Figure 4.2 : Examples of Super-Trees, SCST and SSCST.**

4.3.1 The Abstract Structure τ

The structure τ is defined by the 5-tuple $(T, \oplus, \otimes, \theta, /)$, where :

- i) $(T, \oplus, \theta)$ is a commutative monoid [KO83], i.e., T is a finite or infinite set, $\oplus$ is an associative and commutative operator defined from $T \times T$ to T , and T is closed with respect to $\oplus$. Also, $\theta \in T$ is the identity for $\oplus$.
- ii) $(T, \otimes, /)$ is a monoid with $/ \in T$ as the identity for $\otimes$. Notice that $\otimes$ need not be commutative.
- iii) $\otimes$ distributes over $\oplus$ from both sides.

The iterative use of $\oplus$ and $\otimes$ are denoted by $\bigoplus$ and $\bigotimes$ respectively, defined as below for all $h_i \in T$.

$$\bigoplus_{i=1}^n h_i = h_1 \oplus h_2 \oplus \dots \oplus h_n$$

$$\bigotimes_{i=1}^n h_i = ((\dots(h_1 \otimes h_2) \otimes h_3) \dots \otimes h_n).$$

EXAMPLE 4.3 :

Consider the structure $\tau_1 = (R_p', \text{MIN}, +, \infty, 0)$ where :

- i) R_p' is the set consisting of the non-negative real numbers and ∞ .
- ii) MIN is the binary operator defined for all $p, q \in R_p'$ by :

$$\text{MIN}(p, q) = \begin{cases} p & \text{if } p \leq q; \\ q & \text{otherwise.} \end{cases}$$

and iii) $+$ represents arithmetic addition.

τ_1 is a well-defined structure since $(R_p', \text{MIN}, \infty)$ is a commutative monoid, $(R_p', +, 0)$ is a monoid, and $+$ distributes over MIN from both sides. ***

4.3.2 The Set of Elementary Measures : ω

$\omega(.,.)$ is a function whose arguments are a pair of nodes belonging to $\tilde{\mathcal{N}}$, the Appended Alphabet, and whose range is T , the set defined in τ . $\omega(\lambda, \lambda)$ is undefined since it is not needed. The elementary measure $\omega(x, y)$ can be interpreted as the measure associated with transforming "x" to "y", for $x, y \in \tilde{\mathcal{N}}$. In other words, $\omega(x, y)$ will represent the "cost" (weight or advantage) of substituting node x by node y if $x \neq \lambda \neq y$. Similarly, $x \neq \lambda, y = \lambda$ and $x = \lambda, y \neq \lambda$ will represent the cost of deletion and insertion of node x and y respectively.

4.3.3 Mappings, Extended Mappings and the set $\Gamma_{(T_1, T_2)}$

For every pair (T_1, T_2) , $T_1, T_2 \in \mathcal{T}^*$, the finite set $\Gamma_{(T_1, T_2)}$ is defined as follows. Let M be a mapping from T_1 to T_2 defined as in Section 3.2. Let I, J be sets of nodes in T_1 and T_2 , respectively, not touched by any lines in M . Then, with respect to M , we define the *Extended Mapping*, denoted by M' , between T_1 and T_2 as :

$$M' = \left\{ (T_1[x], T_2[y]) \mid (x, y) \in M \right\} \cup \left\{ (i, \lambda) \mid i \in I \right\} \cup \left\{ (\lambda, j) \mid j \in J \right\}.$$

In other words, every set M' corresponds to at least one way of transforming T_1 into T_2 , using these edit operations of substitution, deletion and insertion. Let

$$\Gamma_{(T_1, T_2)} = \left\{ M' \mid M' \text{ is an extended mapping between } T_1 \text{ and } T_2 \right\}.$$

Then $\Gamma_{(T_1, T_2)}$ is an exhaustive enumeration of the set of all the ways by which T_1 can be transformed to T_2 using the elementary edit operations. Note that in every case the sibling order and the ancestor order in T_1 and T_2 are maintained. An example of $\Gamma_{(T_1, T_2)}$ is shown in Example 4.4 (See Figure 4.3). Note that in particular, the pair M_8 in Example 4.4 represents the edit operations of replacing "a" by "b" and "d" by "e", and deleting "c".

EXAMPLE 4.4 :

Let

then, $\Gamma_{T_1, T_2} = \{M_i \mid 1 \leq i \leq 9\}$ where

Figure 4.3 : Example of an extended mapping

4.3.2 The Set of Elementary Measures : ω

$\omega(.,.)$ is a function whose arguments are a pair of nodes belonging to $\tilde{\mathcal{N}}$, the Appended Alphabet, and whose range is T , the set defined in τ . $\omega(\lambda, \lambda)$ is undefined since it is not needed. The elementary measure $\omega(x, y)$ can be interpreted as the measure associated with transforming "x" to "y", for $x, y \in \tilde{\mathcal{N}}$. In other words, $\omega(x, y)$ will represent the "cost" (weight or advantage) of substituting node x by node y if $x \neq \lambda \neq y$. Similarly, $x \neq \lambda, y = \lambda$ and $x = \lambda, y \neq \lambda$ will represent the cost of deletion and insertion of node x and y respectively.

4.3.3 Mappings, Extended Mappings and the set $\Gamma_{(T_1, T_2)}$

For every pair (T_1, T_2) , $T_1, T_2 \in \mathcal{T}^*$, the finite set $\Gamma_{(T_1, T_2)}$ is defined as follows. Let M be a mapping from T_1 to T_2 defined as in Section 3.2. Let I, J be sets of nodes in T_1 and T_2 , respectively, not touched by any lines in M . Then, with respect to M , we define the *Extended Mapping*, denoted by M' , between T_1 and T_2 as :

$$M' = \left\{ (T_1[x], T_2[y]) \mid (x, y) \in M \right\} \cup \left\{ (i, \lambda) \mid i \in I \right\} \cup \left\{ (\lambda, j) \mid j \in J \right\}.$$

In other words, every set M' corresponds to at least one way of transforming T_1 into T_2 , using these edit operations of substitution, deletion and insertion. Let

$$\Gamma_{(T_1, T_2)} = \left\{ M' \mid M' \text{ is an extended mapping between } T_1 \text{ and } T_2 \right\}.$$

Then $\Gamma_{(T_1, T_2)}$ is an exhaustive enumeration of the set of all the ways by which T_1 can be transformed to T_2 using the elementary edit operations. Note that in every case the sibling order and the ancestor order in T_1 and T_2 are maintained. An example of $\Gamma_{(T_1, T_2)}$ is shown in Example 4.4 (See Figure 4.3). Note that in particular, the pair M_8 in Example 4.4 represents the edit operations of replacing "a" by "b" and "d" by "e", and deleting "c".

EXAMPLE 4.4 :

Let

then, $\Gamma_{T_1, T_2} = \{M_i \mid 1 \leq i \leq 9\}$ where **Figure 4.3 : Example of an extended mapping**

4.3.4 The Abstract Measure : $\Omega(T_1, T_2)$

The abstract measure $\Omega(T_1, T_2)$ between T_1 and T_2 induced by the system $(\mathcal{N}, \tau, \omega)$, where $\tau = (T, \oplus, \otimes, \theta, I)$, is a map whose domain is $\mathcal{N}^* \times \mathcal{N}^*$ and whose range is T . Formally, we define

$$\Omega(T_1, T_2) = \begin{cases} I - \text{the identity for } \otimes & \text{if } T_1 = T_2 = \mu; \\ \bigoplus_{M' \in \Gamma_{(T_1, T_2)}} \left[\bigotimes_{(x_i, y_i) \in M'} \omega(x_i, y_i) \right] & \text{otherwise.} \end{cases} \quad (4.1)$$

In general, $\Omega(T_1, T_2)$ need not be equal to $\Omega(T_2, T_1)$. The equality holds only when the function $\omega(\dots)$ inducing it obeys the following conditions :

$$\omega(a, b) = \omega(b, a) \quad \text{and} \quad \omega(\lambda, a) = \omega(a, \lambda) \quad \text{for all } a, b \in \mathcal{N},$$

4.4 Numeric Measures For $\Omega(T_1, T_2)$

We shall now show that various numeric measures involving T_1 and T_2 are obtained as special cases of $\Omega(T_1, T_2)$ by appropriately choosing the concrete values for the abstract operators, and for the function $\omega(\dots)$.

4.4.1 The Distance between Two Trees

The distance between two trees T_1 and T_2 (introduced and studied in detail in Chapter III), denoted here by $\text{Dist}(T_1, T_2)$, is defined as the minimum of the sum of the elementary edit distances associated with the edit operations required to transform T_1 to T_2 ([Sc77], [Ta79], [Lu79] and [ZS89]). The elementary edit distances themselves are specified in terms of a map $\omega_1(\dots)$ (which is identical to the function $d(\dots)$ of Chapter III) from $\tilde{\mathcal{N}} \times \tilde{\mathcal{N}}$ to $\mathbb{R}_p$ obeying the following conditions :

$$\begin{aligned} \omega_1(x, y) &\geq 0; \quad \omega_1(x, x) = 0; \\ \omega_1(x, y) &= \omega_1(y, x); \\ \omega_1(x, z) &\leq \omega_1(x, y) + \omega_1(y, z). \end{aligned}$$

The symmetry condition (requiring that $\omega_1(x,y) = \omega_1(y,x)$) need not necessarily be satisfied. But if it is, in general, a greater value of the $\text{Dist}(T_1, T_2)$ indicates a greater dissimilarity between them.

THEOREM 4.1

The quantity $\text{Dist}(T_1, T_2)$ between T_1 and T_2 is exactly the measure between them induced by the system $(\mathcal{N}, \tau_1, \omega_1)$, where τ_1 is the structure defined in Example 4.3.

PROOF

The theorem is trivially true when $T_1 = T_2 = \mu$, since $\text{Dist}(\mu, \mu) = 0$. Consider the case when either T_1 or T_2 or both is not μ . From Section 4.3.3, we seen that the set of ways by which T_1 can be edited to T_2 is given by the set $\Gamma_{(T_1, T_2)}$. Any extended map $M' \in \Gamma_{(T_1, T_2)}$ represents the edit operation of transforming x_i to y_i for all $(x_i, y_i) \in M'$. Hence the sum of the edit distances corresponding to this extended mapping, M' , is given by the number $\sum_{(x_i, y_i) \in M'} \omega_1(x_i, y_i)$. By definition, the minimum of this quantity is the distance between T_1 and T_2 . Since $\Gamma_{(T_1, T_2)}$ is the set of all extended mappings,

$$\text{Dist}(T_1, T_2) = \min_{M' \in \Gamma_{(T_1, T_2)}} \left[\sum_{(x_i, y_i) \in M'} \omega_1(x_i, y_i) \right], \quad (4.2)$$

which is exactly the value of the measure between T_1 and T_2 induced by the system $(\mathcal{N}, \tau_1, \omega_1)$, giving the desired result. ***

4.4.2 The Size of the Largest Common Sub-Tree

We now show that by using the operators $\oplus$ and $\otimes$ to represent the $\text{MAX}(\dots)$ and the arithmetic addition operations, we can compute $\text{SLCsT}(T_1, T_2)$, the Size of the Largest Common Sub-Tree of T_1 and T_2 provided the values of $\omega(\dots)$ are appropriately defined.

By definition, $SLCsT(T_1, T_2) = 0$ if $T_1 = T_2 = \mu$. Consider the nontrivial case when either T_1 or T_2 or both are not μ . Let Z_p' be the set consisting of the non-negative integers and ∞ , and let $\omega_2(.,.)$ be a function defined from $\tilde{\mathcal{N}} \times \tilde{\mathcal{N}}$ to Z_p' by

$$\omega_2(a, b) = \begin{cases} 1 & \text{(if } a = b \neq \lambda, a, b \in \mathcal{N} \text{);} \\ 0 & \text{otherwise.} \end{cases}$$

Consider any extended mapping $M' \in \Gamma_{(T_1, T_2)}$. Corresponding to this extended mapping we can associate an integer W_2 where

$$W_2 = \sum_{(x_i, y_i) \in M'} \omega_2(x_i, y_i).$$

By the definition of $\omega_2(.,.)$, the value of W_2 is exactly the number of non- λ pairs of nodes (x_i, y_i) in M' with $x_i = y_i$, which will yield a common Sub-Tree between T_1 and T_2 . Note that this is true since the sibling and ancestor orders will be preserved by the extended mapping. Hence, W_2 is the size of a Common Sub-Tree. By the definition of $\Gamma_{(T_1, T_2)}$, the maximum value of W_2 is the value $SLCsT(T_1, T_2)$. Hence

$$SLCsT(T_1, T_2) = \text{Maximum}_{M' \in \Gamma_{(T_1, T_2)}} \left[\sum_{(x_i, y_i) \in M'} \omega_2(x_i, y_i) \right]. \quad (4.3)$$

This leads to the following theorem.

THEOREM 4.2

Let τ_2 be the abstract structure $(Z_p', \text{MAX}, +, 0, 0)$ where

- i) Z_p' is the set consisting of the non-negative integers and ∞ .
- ii) MAX is the commutative binary operator defined by

$$\text{MAX}(p, q) = \begin{cases} p & \text{if } p \geq q; \\ q & \text{otherwise.} \end{cases}$$

- iii) $+$ represents arithmetic addition.

Then the $SLCsT(T_1, T_2)$ is the measure induced by the system $(\mathcal{N}, \tau_2, \omega_2)$.

PROOF

The Theorem follows by applying (4.1) to the system $(\mathcal{N}, \tau_2, \omega_2)$ and comparing the result with (4.3). ***

At this juncture, it is appropriate to distinguish between the Length of the Longest Common Subsequence Problem and the Size of the Largest Common Sub-Tree Problem. Due to the necessity of preserving the Sibling and the Ancestor Properties of a tree, we can NOT compute the $SLCsT(T_1, T_2)$ between two trees by merely computing the LLCS using merely the string representations of the corresponding trees. As an example, consider the trees shown in the Figure 4.4. Since the postorder sequences of T_1 and T_2 are: "debca" and "bceda", respectively, an LLCS is "bca". However, no Common Sub-Tree of T_1 and T_2 may consist of both the nodes "b" and "c" simultaneously. In other words, trees obtained using the string representation of their LCSs may represent trees which cannot be their common Sub-Trees. More precisely, the sibling and the ancestor constraints of the "tree" will be totally ignored if we try to compute the $SLCsT$ using any LCS algorithm. This explains why the problem currently studied is more complex than the corresponding string-related problem.

Figure 4.4 : Trees used to distinguish the difference between LLCS and $SLCsT$.

4.4.3 The Size of the Smallest Common Super-Tree

As mentioned in Section 4.2.1, we shall refer to the Size of the Smallest Common Super-Tree between T_1 and T_2 as $\text{SSCST}(T_1, T_2)$. By definition, $\text{SSCST}(T_1, T_2) = 0$ if $T_1 = T_2 = \mu$. As before, consider the nontrivial case when either T_1 or T_2 or both are not μ . Let $a, b \in \mathcal{N}$, and let $\omega_3(\cdot, \cdot)$ be a function defined from $\tilde{\mathcal{N}} \times \tilde{\mathcal{N}}$ to $\mathbb{Z}_p'$ by

$$\omega_3(a, b) = \begin{cases} 1 & \text{if } a = b; \\ \infty & \text{if } a \neq b; \\ 1 & \text{if } a = \lambda \text{ or } b = \lambda. \end{cases}$$

Consider any extended mapping $M' \in \Gamma_{(T_1, T_2)}$. Corresponding to this extended mapping we can associate an integer W_3 where

$$W_3 = \sum_{(x_i, y_i) \in M'} \omega_3(x_i, y_i).$$

By the definition of $\omega_3(\cdot, \cdot)$, if $W_3 \neq \infty$, the value of W_3 is exactly the number of pairs of nodes (x_i, y_i) in M' with either $x_i = y_i \neq \lambda$, $x_i \neq \lambda$, or $y_i \neq \lambda$. In other words, if $W_3 \neq \infty$, W_3 is the sum of the values of $|M|$, $|I|$, and $|J|$ where M is the set of nodes from T_1 which will remain unchanged during the transformation and appear in T_2 (or vice versa), I is the set of nodes from T_1 that will be deleted when transforming T_1 into T_2 , and J is the set of nodes from T_2 that will be inserted in the process. If $W_3 \neq \infty$, we know that $|M| + |I| = |T_1|$ and $|M| + |J| = |T_2|$. Thus, a Super-Tree of T_1 , say S_1 , can be formed by inserting those nodes in J belonging to T_2 into T_1 . Similarly, a Super-Tree of T_2 , say S_2 , can be formed by inserting those nodes in I belonging to T_1 into T_2 . Since only substitutions of identical nodes are permitted, these will contribute to finite values whenever $x_i = y_i$ for all $(x_i, y_i) \in M$. Thus, the resultant tree will be a Common Super-Tree of both T_1 and T_2 . By the definition of $\Gamma_{(T_1, T_2)}$, the minimum value of W_3 is the value of $\text{SSCST}(T_1, T_2)$. Hence

$$\text{SSCST}(T_1, T_2) = \text{Minimum}_{M' \in \Gamma_{(T_1, T_2)}} \left[\sum_{(x_i, y_i) \in M'} \omega_3(x_i, y_i) \right]. \quad (4.4)$$

This leads to the following theorem.

THEOREM 4.3

Let τ_3 be the abstract structure $(Z_p', \text{MIN}, +, \infty, 0)$ where

- i) Z_p' is the set consisting of the non-negative integers and ∞ .
- ii) MIN is the commutative binary operator as defined in Example III.
- iii) $+$ represents arithmetic addition.

Then the $\text{SSCST}(T_1, T_2)$ is the measure induced by the system $(\mathcal{N}, \tau_3, \omega_3)$.

PROOF

The Theorem follows by applying (4.1) to the system $(\mathcal{N}, \tau_3, \omega_3)$ and comparing the result with (4.4). ***

Remark : Given the results of the above two theorems, we observe a marked similarity between the SLCsT problem and the SSCST problem. A closer examination reveals that if we are given T_1, T_2 and any one of their LCsTs , we can obtain at least one of their SCSTs . An SCST is generated by including in it every node not in the LCsT , and by including every node in the LCsT exactly once. Thus we obtain the following result relating the quantities $\text{SLCsT}(T_1, T_2)$ and $\text{SSCST}(T_1, T_2)$.

COROLLARY 4.1 :

Let $T_1, T_2 \in \mathcal{N}^*$, then

$$\text{SSCST}(T_1, T_2) = |T_1| + |T_2| - \text{SLCsT}(T_1, T_2).$$

PROOF

Consider the sets M, I , and J introduced in the proofs of the previous theorems. From the definition of the LCsT , a tree formed by including all nodes belonging to M will be a LCsT of T_1 and T_2 if the maximum $|M|$ over every $M' \in \Gamma_{(T_1, T_2)}$ is considered. Furthermore, a SCST can be form by including all elements in the sets I, J and M . Since the sets M, I , and J are disjoint, we have $|M| + |I| = |T_1|$ and $|M| + |J| = |T_2|$ and since $|M| =$

$SLCsT(T_1, T_2)$, we have :

$$SSCST(T_1, T_2) = |M| + |I| + |J| = |T_1| + |T_2| - |M|$$

whence the result follows. ***

An example which clarifies this corollary is shown in Example 4.5 below.

EXAMPLE 4.5 :

Let

then,

That is,

$$SSCST(T_1, T_2) = 8 = |T_1| + |T_2| - SLCsT(T_1, T_2).$$

Figure 4.5 : Example of the linear relationship between $SSCST(T_1, T_2)$ and $SLCsT(T_1, T_2)$.

4.4.4 Probability of Erroneous Trees

Consider the problem of transmitting a tree across a noisy channel. A pertinent question is that of evaluating the probability of receiving a tree T_2 given that a tree T_1 is transmitted⁵. In this section, we will show that this quantity can also be obtained as a special case of $\Omega(T_1, T_2)$ by appropriately choosing the concrete operators $\oplus$ and $\ominus$, and the specific function $\omega(\dots)$.

As a first attempt at solving the problem we consider the model analogous to the one used in [KO83] which causes only independent substitution and deletion errors. Let the elementary error probabilities (also called the "confusion" probabilities) be given by the function mapping $\tilde{\mathcal{N}} \times \tilde{\mathcal{N}}$ to $[0, 1]$ as below:

- (i) $p(b|a)$ is the conditional probability of receiving the node symbol "b" given that the node symbol "a" is transmitted.
- (ii) $p(\lambda|a)$ is the conditional probability of the channel deleting the transmitted node symbol "a".

We define $p(a|\lambda) = 0$ for $a \in \mathcal{N}$. Again, the quantity $p(\lambda|\lambda)$ is undefined since it is not needed.

Let M' be any arbitrary element of $\Gamma_{(T_1, T_2)}$ and let

$$W_4 = \prod_{(x_i, y_i) \in M'} p(y_i | x_i),$$

where Π represents *arithmetic* multiplication.

W_4 is the product of the probabilities associated with the extended mapping M' given that every y_i was caused by the corresponding x_i . Since $p(a|\lambda) = 0$ for all $a \in \mathcal{N}$, W_4 is equal to zero whenever the extended mapping M' represents the insertion of at least one node symbol in T_2 .

⁵Here we assume that the tree itself is transmitted as a two dimensional entity. The actual serialization of this transmission process is not considered here.

We define $\text{Prob}(T_2|T_1)$ as the *total* probability of receiving T_2 given that T_1 is transmitted. $\text{Prob}(T_2|T_1)$ is defined as unity if both T_1 and T_2 are μ . Otherwise, it is defined as the sum of W_4 over all extended mappings M' in $\Gamma_{(T_1, T_2)}$. That is,

$$\text{Prob}(T_2|T_1) = \sum_{M' \in \Gamma_{(T_1, T_2)}} \left[\prod_{(x_i, y_i) \in M'} p(y_i|x_i) \right]. \quad (4.5)$$

The quantity $\text{Prob}(T_2|T_1)$ has the following properties.

THEOREM 4.4

Let τ_4 be the abstract structure $([0, 1], +, *, 0, 1)$ where

- i) $[0, 1]$ is the set consisting of the non-negative real numbers between 0 and 1.
- ii) $+$ and $*$ represent arithmetic addition and multiplication respectively.
- iii) $*$ distributes over $+$ from both sides.

Then, $\text{Prob}(T_2|T_1)$, the *probability of receiving T_2 when T_1 is transmitted*, is exactly the measure between T_1 and T_2 induced by the system $(\mathcal{N}, \tau_4, \omega_4)$, where $\omega_4(\dots)$ is defined by :

$$\omega_4(a, b) = p(b|a) \quad \text{for all } a, b \in \tilde{\mathcal{N}}.$$

PROOF

The form of the expression obtained for $\text{Prob}(T_2|T_1)$ (See Equation (4.5)) can be seen to be the one which follows when (4.1) is applied to the system $(\mathcal{N}, \tau_4, \omega_4)$. We have yet to show that the quantity defined by (4.5) is not just computationally well-defined, but also a well-defined *probability* measure. We now show that $\text{Prob}(T_2|T_1)$ is a valid probability assignment for the probability of receiving T_2 given T_1 is transmitted, given that the channel only substitutes and deletes the node values.

Let $|T_1| = m$. Since $\text{Prob}(T_2|T_1) \geq 0$, all that has to be proved is that

$$\sum_{T_2 \in \tilde{\mathcal{N}}_m} \text{Prob}(T_2|T_1) = 1,$$

where $\tilde{\mathcal{N}}_m$ is the set of trees over $\mathcal{N}$ of size less than or equal to m .

Since the channel cannot cause insertion errors, $M' \in \Gamma_{(T_1, T_2)}$ will yield a non-zero contribution to $\text{Prob}(T_2|T_1)$ if and only if $|M'| = m$. Let the subset of elements of $\Gamma_{(T_1, T_2)}$ which have $|M'| = m$ be Γ_m . Then,

$$\text{Prob}(T_2|T_1) = \sum_{M' \in \Gamma_m} \left[\prod_{(x_i, y_i) \in M'} p(y_i|x_i) \right].$$

The sum on the R.H.S involves *independently* considering all the possible transformations of the individual symbols of T_1 . Thus, the addition and multiplication symbols can be interchanged yielding

$$\sum_{T_2 \in \tilde{\mathcal{N}}_m} \text{Prob}(T_2|T_1) = \prod_{i=1}^m \left[\sum_{v \in \tilde{\mathcal{N}}} p(v|x_i) \right].$$

which equals unity since $\sum_{v \in \tilde{\mathcal{N}}} p(v|x_i) = 1$ for every x_i , and the results follows. ***

4.4.5 Likelihood of Erroneous Trees

In the last section we have considered the problem of transmitting a tree across a noisy channel which causes **no** insertion error. However, the problem of evaluating the total probability of receiving a tree T_2 given that a tree T_1 is transmitted across a noisy channel which causes *all* the three kinds of edit errors (including an arbitrary number of insertions) is far from trivial (see [KO84] for the corresponding problem in the case of strings). This is because of the fact that if we do not restrict the number of possible insertion errors which can occur, the computation of the quantity $\sum_{T_2 \in \mathcal{N}^*} \text{Prob}(T_2|T_1)$,

which is encountered in the process of justifying the model, will involve an infinite sum which is almost certainly divergent. In order to solve the problem in a more reasonable way, we shall consider the *a posteriori* approach in which we intend to evaluate the

probability of a *received* tree T_2 having been caused by the transmission of a tree T_1 , where T_1 itself is a member of a finite dictionary of trees.

Consider a channel which causes independent substitution, deletion, and insertion errors. Let the elementary error probabilities be given by the function mapping $\tilde{\mathcal{N}} \times \tilde{\mathcal{N}}$ to $[0,1]$ as below:

- (i) $q(a \rightarrow b)$ is the conditional probability of receiving the node symbol "b" given that the node symbol "a" is transmitted.
- (ii) $q(a \rightarrow \lambda)$ is the conditional probability of the channel deleting the transmitted node symbol "a".
- (ii) $q(\lambda \rightarrow a)$ is the conditional probability of the channel inserting the symbol "a" during the transmission given that an insertion is taking place.

Since a transmitted symbol can either be transmitted correctly, deleted or transformed into another symbol, and since the inserted symbol can be any one of the symbol in $\mathcal{N}$, we have :

$$\sum_{v \in \mathcal{N}} q(\lambda \rightarrow v) = 1, \quad \text{and} \quad \sum_{v \in \tilde{\mathcal{N}}} q(a \rightarrow v) = 1 \quad \text{for all } a \in \mathcal{N}.$$

Observe again that the quantity $p(\lambda|\lambda)$ is undefined since it is not needed.

Let M' be any arbitrary element of $\Gamma_{(T_1, T_2)}$ and let

$$W_5 = \prod_{(x_i, y_i) \in M'} q(x_i \rightarrow y_i),$$

where Π represents *arithmetic* multiplication. W_5 is the product of the probabilities associated with the extended mapping M' given that every y_i was caused by the corresponding x_i .

Let us define $L(T_1 \rightarrow T_2)$ as the likelihood⁶ of transmitting T_1 given that T_2 is received. $L(T_1 \rightarrow T_2)$ is defined as unity if both T_1 and T_2 are μ . Otherwise, it is defined as the sum of W_5 over all extended mappings M' in $\Gamma_{(T_1, T_2)}$. That is,

$$L(T_1 \rightarrow T_2) = \sum_{M' \in \Gamma_{(T_1, T_2)}} \left[\prod_{(x_i, y_i) \in M'} q(x_i \rightarrow y_i) \right]. \quad (4.6)$$

The quantity $L(T_1 \rightarrow T_2)$ has the following properties.

THEOREM 4.5

Let τ_4 be the abstract structure as defined in Theorem 5.4. Then, the quantity $L(T_1 \rightarrow T_2)$, the likelihood of having T_1 being transmitted given T_2 is received, is exactly the measure between T_1 and T_2 induced by the system $(\mathcal{N}, \tau_4, \omega_5)$, where $\omega_5(\dots)$ is defined by :

$$\omega_5(a, b) = q(a \rightarrow b) \quad \text{for all } a, b \in \tilde{\mathcal{N}}.$$

PROOF

The theorem follows directly by applying (4.1) to the system $(\mathcal{N}, \tau_4, \omega_5)$ and comparing the result with (4.6). ***

Note that the *a priori* likelihoods $L(T_1 \rightarrow Q)$ do not constitute a valid probability assignment for the probability of *receiving* Q given that T_1 is transmitted. This is because we can express $L(T_1 \rightarrow Q)$ as the sum of $L_j(T_1 \rightarrow Q)$ for $(0 \leq j \leq |Q|)$, where $L_j(T_1 \rightarrow Q)$ is the likelihood of having transmitted T_1 given that Q is received and that j insertion errors have taken place. Hence,

⁶The quantity $L(T_1 \rightarrow T_2)$ is termed as a *likelihood* because we shall show that it is not a consistent probability measure. However, viewed from an *a posteriori* perspective the normalized quantities can be viewed as probabilities.

$$\sum_{Q \in \mathcal{N}^*} L(T_1 \rightarrow Q) = \sum_{Q \in \mathcal{N}^*} \left[\sum_{j=0}^{|Q|} L_j(T_1 \rightarrow Q) \right]$$

and since we are only dealing with positive quantities that are bounded by unity, the summations can be interchanged to yield :

$$\sum_{Q \in \mathcal{N}^*} L(T_1 \rightarrow Q) = \sum_{j=0}^{\infty} \left[\sum_{Q \in \mathcal{Q}^j} L_j(T_1 \rightarrow Q) \right]$$

where $\mathcal{Q}^j$ is the set of trees which can be received if exactly j insertions have taken place.

Thus,

$$\sum_{Q \in \mathcal{N}^*} L(T_1 \rightarrow Q) = \sum_{j=1}^{\infty} \left[\sum_{Q \in \mathcal{Q}^j} L_j(T_1 \rightarrow Q) \right] + \sum_{Q \in \mathcal{Q}^0} L_0(T_1 \rightarrow Q)$$

However, from the last section, we see that $\sum_{Q \in \mathcal{Q}^0} L_0(T_1 \rightarrow Q) = 1$. Hence, we have

$$\sum_{Q \in \mathcal{N}^*} L(T_1 \rightarrow Q) = \sum_{j=1}^{\infty} \left[\sum_{Q \in \mathcal{Q}^j} L_j(T_1 \rightarrow Q) \right] + 1 \geq 1.$$

However, by viewing the problem from an *a posteriori* perspective a valid probability measure can be well defined. To achieve this, let T_1 be known to be an element of a finite dictionary of trees, $\mathcal{H}$. Then, if Q is any received tree, we define :

$$P(T_i|Q) = \frac{L(T_i \rightarrow Q)}{\sum_{T_j \in \mathcal{H}} L(T_j \rightarrow Q)}, \quad \text{for all } T_i \in \mathcal{H}.$$

Next we show that the quantity $P(T_i|Q)$ is a valid probability assignment for the *a posteriori* probability of having transmitted $T_1 \in \mathcal{H}$ given that Q is received.

THEOREM 4.6

The quantity $P(T_i|Q)$ is a valid probability assignment for the *a posteriori* probability of having transmitted $T_1 \in \mathcal{H}$ given that Q is received.

PROOF

Assume $|\mathcal{H}| = k$. Since $P(T_i|Q) \geq 0$, we essentially have to prove that

$$\sum_{T_i \in \mathcal{H}} P(T_i|Q) = 1.$$

But,

$$\begin{aligned} \sum_{T_i \in \mathcal{H}} P(T_i|Q) &= \sum_{i=1}^k P(T_i|Q) \\ &= \sum_{i=1}^k \frac{L(T_i \rightarrow Q)}{\sum_{j=1}^k L(T_j \rightarrow Q)} \\ &= \frac{\sum_{i=1}^k L(T_i \rightarrow Q)}{\sum_{j=1}^k L(T_j \rightarrow Q)} \end{aligned}$$

which equals unity since the identical summations in the numerator and denominator are finite. ***

4.5 Computation Of $\Omega(T_1, T_2)$.**4.5.1 Recursive Properties of $\Omega(T_1, T_2)$**

Since we are working with the postorder numbering of the trees, $T_1[i'..i]$ and $T_2[j'..j]$ will generally (and almost often) be forests. Fortunately, the definitions of mapping and extended mapping for ordered forests are valid just as in the case of trees. For convenience, we use $\Gamma_{(F_1, F_2)}$ and $\Omega(F_1, F_2)$ to represent the set of all extended

mappings and the abstract measure between the forests $F_1=T_1[i'..i]$ and $F_2=T_2[j'..j]$, respectively. We also use $F_Meas(T_1[i'..i], T_2[j'..j])$ to denote the abstract measure between the forest $T_1[i'..i]$ and the forest $T_2[j'..j]$. The abstract measure between the *subtree* of T_1 rooted at i and the *subtree* of T_2 rooted at j is denoted by $T_Meas(i, j)$. Finally, the abstract measure over all extended mappings M' between the *subtree* of T_1 rooted at i and the *subtree* of T_2 rooted at j such that $(T_1[i], T_2[j])$ belongs to each M' is denoted $T_Meas_S(i, j)$. Note that this measure constrains the mapping to have $T_1[i]$ substituted by $T_2[j]$. We now prove the analytic properties of the above quantities.

THEOREM 4.7

Let $i_1 \in Anc(i)$ and $j_1 \in Anc(j)$, then,

- (i) $F_Meas(\mu, \mu) = I$, the identity for $\otimes$.
- (ii) $F_Meas(T_1[\delta(i_1)..i], \mu) = F_Meas(T_1[\delta(i_1)..i-1], \mu) \otimes \omega(T_1[i], \lambda)$.
- (iii) $F_Meas(\mu, T_2[\delta(j_1)..j]) = F_Meas(\mu, T_2[\delta(j_1)..j-1]) \otimes \omega(\lambda, T_2[j])$.

PROOF

For case (i), since the set $\Gamma_{(F_1, F_2)}$ is empty, by the definition of $\Omega(F_1, F_2)$,

$F_Meas(\mu, \mu) = I$, the identity for $\otimes$.

For case (ii), the set $\Gamma_{(F_1, F_2)}$ has only one element, namely $\{(T_1[\delta(i_1)], \lambda), (T_1[\delta(i_1)+1], \lambda), \dots, (T_1[i], \lambda)\}$. Hence, by (3.1),

$$F_Meas(T_1[\delta(i_1)..i], \mu) = \bigotimes_{j=\delta(i_1)}^i \omega(T_1[j], \lambda),$$

which by the associativity of $\otimes$ reduces to :

$$\bigotimes_{j=\delta(i_1)}^{i-1} \omega(T_1[j], \lambda) \otimes \omega(T_1[i], \lambda).$$

But $\bigotimes_{j=\delta(i_1)}^{i-1} \omega(T_1[j], \lambda) = F_Meas(T_1[\delta(i_1)..i-1], \mu)$. Hence,

$$F_Meas(T_1[\delta(i_1)..i], \mu) = F_Meas(T_1[\delta(i_1)..i-1], \mu) \otimes \omega(T_1[i], \lambda),$$

and the result follows.

In an exactly similar way (iii) can also be proved, giving the desired result. ***

The above equations form the boundary values of $F_Meas(...)$ where either tree (forest) is empty. For the intermediate values of the $F_Meas(...)$, we have the following result which is the central result involving the recursive property of F_Meas .

THEOREM 4.8

Let $i_1 \in Anc(i)$ and $j_1 \in Anc(j)$, then,

$$\begin{aligned} F_Meas(\delta(i_1)..i, \delta(j_1)..j) \\ = [F_Meas(\delta(i_1)..i-1, \delta(j_1)..j) \otimes \omega(T_1[i], \lambda)] \\ \otimes [F_Meas(\delta(i_1)..i, \delta(j_1)..j-1) \otimes \omega(\lambda, T_2[j])] \\ \otimes [F_Meas(\delta(i_1)..i-1, \delta(j_1)..j-1) \otimes F_Meas(\delta(i_1)..i-1, \delta(j_1)..j-1) \\ \otimes \omega(T_1[i], T_2[j])]. \end{aligned}$$

PROOF

We compute $F_Meas(\delta(i_1)..i, \delta(j_1)..j)$ for $\delta(i_1) \leq i \leq i_1$ and $\delta(j_1) \leq j \leq j_1$. The extended mapping, $M' \in \Gamma_{(F_1, F_2)}$, between the forest $F_1 = T_1[\delta(i_1)..i]$ and the forest $F_2 = T_2[\delta(j_1)..j]$ can be seen to include the nodes $T_1[i]$ and $T_2[j]$ in the following three ways:

- Case (1) : If $T_1[i]$ is not touched by any line in M' .
- Case (2) : If $T_2[j]$ is not touched by any line in M' .
- Case (3) : If $T_1[i]$ and $T_2[j]$ are touched by lines in M' .

Thus, if the individual terms evaluated for $F_Meas(\delta(i_1)..i, \delta(j_1)..j)$ for the above three

cases are given for $1 \leq k \leq 3$ by $F_Meas(\delta(i_1)..i, \delta(j_1)..j) \Big|_{Case(k)}$, then, using the associativity of $\oplus$, we can see that :

$$\begin{aligned} & F_Meas(\delta(i_1)..i, \delta(j_1)..j) \\ &= \bigoplus_{k=1}^3 F_Meas(\delta(i_1)..i, \delta(j_1)..j) \Big|_{Case(k)} \end{aligned} \quad (4.7)$$

We shall consider each of these individual cases separately.

Case (1). If $T_1[i]$ is not touched by any line in M' , then $T_1[i]$ is to be deleted, and thus $(T_1[i], \lambda) \in M'$. Let Γ_A be the maximal subset of $\Gamma_{(F_1, F_2)}$ such that each $M' \in \Gamma_A$ contains the element $(T_1[i], \lambda)$. Then, we have :

$$F_Meas(\delta(i_1)..i, \delta(j_1)..j) \Big|_{Case(1)} = \bigoplus_{M' \in \Gamma_A} \left[\bigotimes_{(x_i, y_i) \in M'} \omega(x_i, y_i) \right].$$

By the associativity of $\otimes$, we have :

$$\begin{aligned} & F_Meas(\delta(i_1)..i, \delta(j_1)..j) \Big|_{Case(1)} \\ &= \bigoplus_{M' \in \Gamma_A} \left[\bigotimes_{(x_i, y_i) \in M' - \{(T_1[i], \lambda)\}} \omega(x_i, y_i) \otimes \omega(T_1[i], \lambda) \right]. \end{aligned}$$

Using the distributivity of $\otimes$ over $\oplus$, the above expression reduces to

$$\bigoplus_{M' \in \Gamma_A} \left[\bigotimes_{(x_i, y_i) \in M' - \{(T_1[i], \lambda)\}} \omega(x_i, y_i) \right] \otimes \omega(T_1[i], \lambda).$$

However, if we consider the set $\Gamma_{(F_1', F_2)}$, where $F_1' = T_1[\delta(i_1)..i-1]$ and $F_2 = T_2[\delta(j_1)..j]$, we have :

$$F_Meas(\delta(i_1)..i-1, \delta(j_1)..j) = \sum_{M' \in \Gamma_{(F_1, F_2)}} \left[\prod_{(x_i, y_i) \in M'} \omega(x_i, y_i) \right].$$

By comparing the elements between the sets Γ_A and $\Gamma_{(F_1, F_2)}$, we find that for every element $m \in \Gamma_{(F_1, F_2)}$, there is a unique element $m \cup \{(T_1[i], \lambda)\} \in \Gamma_A$ and vice versa. Hence, we have

$$\begin{aligned} & \sum_{M' \in \Gamma_A} \left[\prod_{(x_i, y_i) \in M' - \{(T_1[i], \lambda)\}} \omega(x_i, y_i) \right] \\ &= \sum_{M' \in \Gamma_{(F_1, F_2)}} \left[\prod_{(x_i, y_i) \in M'} \omega(x_i, y_i) \right] \\ &= F_Meas(\delta(i_1)..i-1, \delta(j_1)..j), \end{aligned}$$

which implies that the contribution due to the pertinent elements in Case 1 yields :

$$\begin{aligned} & F_Meas(\delta(i_1)..i, \delta(j_1)..j) \Big|_{\text{Case(1)}} \\ &= [F_Meas(\delta(i_1)..i-1, \delta(j_1)..j) \otimes \omega(T_1[i], \lambda)]. \end{aligned} \quad (4.8)$$

Case (2). Consider the case when $T_2[j]$ is not touched by any line in M' . Clearly, then $T_2[j]$ is to be inserted, and thus $(\lambda, T_2[j]) \in M'$. Let Γ_B be the maximal subset of $\Gamma_{(F_1, F_2)}$ such that each $M' \in \Gamma_B$ contains the element $(\lambda, T_2[j])$. Then, we have :

$$F_Meas(\delta(i_1)..i, \delta(j_1)..j) \Big|_{\text{Case(2)}} = \sum_{M' \in \Gamma_B} \left[\prod_{(x_i, y_i) \in M'} \omega(x_i, y_i) \right].$$

By the associativity of $\otimes$, we have :

$$F_Meas(\delta(i_1)..i, \delta(j_1)..j) \Big|_{\text{Case(2)}} \\ = \bigoplus_{M' \in \Gamma_B} \left[\bigoplus_{(x_i, y_i) \in M' - \{(\lambda, T_2[j])\}} \omega(x_i, y_i) \oplus \omega(\lambda, T_2[j]) \right].$$

Again, using the distributivity of $\oplus$ over $\oplus$, the above expression reduces to

$$\bigoplus_{M' \in \Gamma_B} \left[\bigoplus_{(x_i, y_i) \in M' - \{(\lambda, T_2[j])\}} \omega(x_i, y_i) \right] \oplus \omega(\lambda, T_2[j]).$$

Now, if we consider the set $\Gamma_{(F_1, F_2')}$, where $F_1 = T_1[\delta(i_1)..i]$ and $F_2' = T_2[\delta(j_1)..j-1]$, we have :

$$F_Meas(\delta(i_1)..i, \delta(j_1)..j-1) = \bigoplus_{M' \in \Gamma_{(F_1, F_2')}} \left[\bigoplus_{(x_i, y_i) \in M'} \omega(x_i, y_i) \right].$$

By comparing the elements between the sets Γ_B and $\Gamma_{(F_1, F_2')}$, we observe that for every element $m \in \Gamma_{(F_1, F_2')}$, there is a unique element $m \cup \{(\lambda, T_2[j])\} \in \Gamma_B$ and vice versa. Hence,

$$\begin{aligned} \bigoplus_{M' \in \Gamma_B} \left[\bigoplus_{(x_i, y_i) \in M' - \{(\lambda, T_2[j])\}} \omega(x_i, y_i) \right] &= \bigoplus_{M' \in \Gamma_{(F_1, F_2')}} \left[\bigoplus_{(x_i, y_i) \in M'} \omega(x_i, y_i) \right] \\ &= F_Meas(\delta(i_1)..i, \delta(j_1)..j-1), \end{aligned}$$

which implies that the contribution due to the pertinent elements in Case 2 yields :

$$\begin{aligned} F_Meas(\delta(i_1)..i, \delta(j_1)..j) \Big|_{\text{Case(2)}} \\ = [F_Meas(\delta(i_1)..i, \delta(j_1)..j-1) \oplus \omega(\lambda, T_2[j])]. \end{aligned} \quad (4.9)$$

Case (3). Consider the final case when both $T_1[i]$ and $T_2[j]$ are touched by lines in M' . Let (i,k) and (h,j) be the respective lines, i.e. (i,k) and $(h,j) \in M'$. If $\delta(i_1) \leq h \leq \delta(i)-1$, then i is to the right of h and so k must be to the right of j by virtue of the sibling property of M' . But this is impossible in $T_2[\delta(j_1)..j]$ since j is the rightmost sibling in $T_2[\delta(j_1)..j]$. Similarly, if i is a proper ancestor of h , then k must be a proper ancestor of j by virtue of the ancestor property of M' . This is again impossible since $k \leq j$, and hence h must be equal to i . By symmetry, k must equal j , so $(i,j) \in M'$.

Now, by the ancestor property of M , any node in the subtree rooted at $T_1[i]$ can only be touched by a node in the subtree rooted at $T_2[j]$. This situation is depicted by Figure 4.6.

Figure 4.6 : Pictorial description of Case (3) of Theorem 4.8.

Let Γ_C be the maximal subset of $\Gamma_{(F_1, F_2)}$ such that each $M' \in \Gamma_C$ contains the element $(T_1[i], T_2[j])$. Then, by the associativity and distributivity of $\otimes$ and $\oplus$, we have :

$$\begin{aligned}
F_Meas(\delta(i_1)..i, \delta(j_1)..j) \Big|_{\text{Case(3)}} &= \bigoplus_{M' \in \Gamma_C} \left[\bigotimes_{(x_i, y_i) \in M'} \omega(x_i, y_i) \right] \\
&= \bigoplus_{M' \in \Gamma_C} \left[\bigotimes_{(x_i, y_i) \in M' - \{(T_1[i], T_2[j])\}} \omega(x_i, y_i) \otimes \omega(T_1[i], T_2[j]) \right] \\
&= \bigoplus_{M' \in \Gamma_C} \left[\bigotimes_{(x_i, y_i) \in M' - \{(T_1[i], T_2[j])\}} \omega(x_i, y_i) \right] \otimes \omega(T_1[i], T_2[j]). \quad (4.10)
\end{aligned}$$

We now invoke the ancestor property of M to further partition each individual forest into a subforest and a tree respectively. To achieve this we define Z to be the maximal subset of $M' - \{(T_1[i], T_2[j])\}$ such that $x_i \geq \delta(i)$ and $y_i \geq \delta(j)$ for all $(x_i, y_i) \in Z$. Note that all the remaining elements in the set $M' - \{(T_1[i], T_2[j])\} - Z$ satisfy the condition $x_i < \delta(i)$ and $y_i < \delta(j)$ due to the ancestor property of M . Then, using the associativity of $\otimes$, equation (4.10) further reduces to

$$\bigoplus_{M' \in \Gamma_C} \left[\bigotimes_{(x_i, y_i) \in M' - \{(T_1[i], T_2[j])\} - Z} \omega(x_i, y_i) \otimes \bigotimes_{(x_i, y_i) \in Z} \omega(x_i, y_i) \right] \otimes \omega(T_1[i], T_2[j]).$$

By using the distributivity of $\otimes$ and $\oplus$ the above expression reduces to :

$$\begin{aligned}
&\bigoplus_{M' \in \Gamma_C} \left[\bigotimes_{(x_i, y_i) \in M' - \{(T_1[i], T_2[j])\} - Z} \omega(x_i, y_i) \right] \otimes \bigoplus_{M' \in \Gamma_C} \left[\bigotimes_{(x_i, y_i) \in Z} \omega(x_i, y_i) \right] \\
&\quad \otimes \omega(T_1[i], T_2[j]),
\end{aligned}$$

Consider now the sets $\Gamma_{(F_1', F_2')}$ and $\Gamma_{(F_1'', F_2'')}$, where $F_1' = T_1[\delta(i_1).. \delta(i)-1]$, $F_2' = T_2[\delta(j_1).. \delta(j)-1]$, $F_1'' = T_1[\delta(i).. i-1]$, and $F_2'' = T_2[\delta(j).. j-1]$.

By definition we know that :

$$F_Meas(\delta(i_1)..i-1, \delta(j_1)..j-1) = \sum_{M' \in \Gamma_{(F_1', F_2')}} \left[\prod_{(x_i, y_i) \in M'} \omega(x_i, y_i) \right],$$

and

$$F_Meas(\delta(i)..i-1, \delta(j)..j-1) = \sum_{M' \in \Gamma_{(F_1'', F_2'')}} \left[\prod_{(x_i, y_i) \in M'} \omega(x_i, y_i) \right].$$

By comparing the elements between the sets Γ_C and $\Gamma_{(F_1', F_2')}$, and between the sets Γ_C and $\Gamma_{(F_1'', F_2'')}$, we observe that :

$$\sum_{M' \in \Gamma_C} \left[\prod_{(x_i, y_i) \in M' - \{(T_1[i], T_2[j])\} - Z} \omega(x_i, y_i) \right] = \sum_{M' \in \Gamma_{(F_1', F_2')}} \left[\prod_{(x_i, y_i) \in M'} \omega(x_i, y_i) \right],$$

and also that

$$\sum_{M' \in \Gamma_C} \left[\prod_{(x_i, y_i) \in Z} \omega(x_i, y_i) \right] = \sum_{M' \in \Gamma_{(F_1'', F_2'')}} \left[\prod_{(x_i, y_i) \in M'} \omega(x_i, y_i) \right].$$

This implies that

$$\begin{aligned} F_Meas(\delta(i_1)..i, \delta(j_1)..j) \Big|_{\text{Case(3)}} &= F_Meas(\delta(i_1)..i-1, \delta(j_1)..j-1) \otimes \\ &F_Meas(\delta(i)..i-1, \delta(j)..j-1) \otimes \\ &\omega(T_1[i], T_2[j]). \end{aligned} \quad (4.11)$$

We now formalize the principle motivating (4.7). Since both $x_i, y_i \in \tilde{\mathcal{N}}$, and since both cannot be λ simultaneously, every element in $\Gamma_{(F_1, F_2)}$ will be in only one of the three sets Γ_A , Γ_B , and Γ_C . Thus, these three sets partition $\Gamma_{(F_1, F_2)}$ and hence, are mutually exclusive and exhaust the possible ways for yielding $F_Meas(\delta(i_1)..i, \delta(j_1)..j)$. Using the

associativity of $\oplus$, we have

$$\begin{aligned}
 & F_Meas(\delta(i_1)..i, \delta(j_1)..j) \\
 &= \sum_{M' \in \Gamma_{(F_1, F_2)}} \left[\prod_{(x_i, y_i) \in M'} \omega(x_i, y_i) \right] \\
 &= \sum_{M' \in \Gamma_A} [P] \oplus \sum_{M' \in \Gamma_B} [P] \oplus \sum_{M' \in \Gamma_C} [P] \tag{4.12}
 \end{aligned}$$

where $P = \prod_{(x_i, y_i) \in M'} \omega(x_i, y_i)$.

Substituting (4.8), (4.9), and (4.11) in (4.12) proves the theorem. ***

It is easy to construct a recursive algorithm by using the above theorem. However, both the time and space complexities of the algorithm will be prohibitively large. Improvements in both the time and space complexities can be obtained by utilizing the following result involving the quantity $T_Meas_S(...)$ described earlier.

The main handicap with Theorem 4.8 is that when substitutions are involved, the quantity $F_Meas(T_1[\delta(i_1)..i], T_2[\delta(j_1)..j])$ evaluated between the forests $T_1[\delta(i_1)..i]$ and $T_2[\delta(j_1)..j]$ is computed using the F_Meas of the forests $T_1[\delta(i_1)..i-1]$ and $T_2[\delta(j_1)..j-1]$ and incorporating them with the F_Meas of the remaining forests $T_1[\delta(i)..i-1]$ and $T_2[\delta(j)..j-1]$. But if we observe that under certain conditions the removal of a sub-forest leaves us with an *entire tree*, the computation need not be so extensive. Thus, if $\delta(i) = \delta(i_1)$ and $\delta(j) = \delta(j_1)$ (which means that both i and i_1 , and j and j_1 span the same subtree), the sub-forests from $T_1[\delta(i_1)..i-1]$ and $T_2[\delta(j_1)..j-1]$ do not get included in the computation. On the other hand if this is not the case, the $F_Meas(T_1[\delta(i_1)..i], T_2[\delta(j_1)..j])$

can be considered as a combination of the $F_Meas(T_1[\delta(i_1)..i-1], T_2[\delta(j_1)..j-1])$ and the abstract tree measure between the trees rooted at i and j respectively. The latter abstract measures are computed in a straightforward way when $T_1[i], T_2[j]$ are deleted and inserted respectively. However, in the case when $T_1[i]$ is substituted for by $T_2[j]$ the latter abstract measure is the relevant term with the additional constraint that the last operation considered is a substitution. Indeed, this is exactly what we have referred to as $T_Meas_S(i, j)$. This result is presented in the following theorem.

THEOREM 4.9

Let $i_1 \in Anc(i)$ and $j_1 \in Anc(j)$, then,

If $\delta(i) = \delta(i_1)$ and $\delta(j) = \delta(j_1)$ then

$$\begin{aligned} F_Meas(T_1[\delta(i_1)..i], T_2[\delta(j_1)..j]) \\ = [F_Meas(T_1[\delta(i_1)..i-1], T_2[\delta(j_1)..j]) \oplus \omega(T_1[i], \lambda)] \\ \oplus [F_Meas(T_1[\delta(i_1)..i], T_2[\delta(j_1)..j-1]) \oplus \omega(\lambda, T_2[j])] \\ \oplus [F_Meas(T_1[\delta(i_1)..i-1], T_2[\delta(j_1)..j-1]) \oplus \omega(T_1[i], T_2[j])] \end{aligned}$$

Otherwise,

$$\begin{aligned} F_Meas(T_1[\delta(i_1)..i], T_2[\delta(j_1)..j]) \\ = [F_Meas(T_1[\delta(i_1)..i-1], T_2[\delta(j_1)..j]) \oplus \omega(T_1[i], \lambda)] \\ \oplus [F_Meas(T_1[\delta(i_1)..i], T_2[\delta(j_1)..j-1]) \oplus \omega(\lambda, T_2[j])] \\ \oplus [F_Meas(T_1[\delta(i_1)..i-1], T_2[\delta(j_1)..j-1]) \oplus T_Meas_S(i, j)]. \end{aligned}$$

PROOF

We shall first prove the "If" part of the theorem. By Theorem 4.8, if $\delta(i) = \delta(i_1)$ and $\delta(j) = \delta(j_1)$, then the forests $T_1[\delta(i_1)..i-1]$ and $T_2[\delta(j_1)..j-1]$ are both empty. Thus,

$$F_Meas(T_1[\delta(i_1)..i-1], T_2[\delta(j_1)..j-1]) = F_Meas(\mu, \mu),$$

which is equal to I , the identity of $\oplus$. The first part of the theorem follows.

We now proceed to prove the "Otherwise" part of the theorem. From Theorem 4.8, we know that :

$$\begin{aligned}
& F_Meas(\delta(i_1)..i, \delta(j_1)..j) \\
&= [F_Meas(\delta(i_1)..i-1, \delta(j_1)..j) \otimes \omega(T_1[i], \lambda)] \\
&\quad \otimes [F_Meas(\delta(i_1)..i, \delta(j_1)..j-1) \otimes \omega(\lambda, T_2[j])] \\
&\quad \otimes [F_Meas(\delta(i_1)..i-1, \delta(j_1)..j-1) \otimes F_Meas(\delta(i)..i-1, \delta(j)..j-1) \\
&\quad \quad \otimes \omega(T_1[i], T_2[j])].
\end{aligned}$$

Let us consider the case when $T_1[i]$ is substituted for by $T_2[j]$. By the definition of F_Meas , we have

$$\begin{aligned}
& F_Meas(\delta(i_1)..i, \delta(j_1)..j) \\
&= [F_Meas(\delta(i_1)..i-1, \delta(j_1)..j-1) \otimes F_Meas(\delta(i)..i-1, \delta(j)..j-1) \\
&\quad \otimes \omega(T_1[i], T_2[j])] \\
&= \bigotimes_{M' \in \tilde{\Gamma}_{(T_1[\delta(i_1)..i-1], T_2[\delta(j_1)..j-1])}} \left[\bigotimes_{(x_i, y_i) \in M'} \omega(x_i, y_i) \right] \\
&\quad \otimes \bigotimes_{M' \in \Gamma_{(T_1[\delta(i)..i-1], T_2[\delta(j)..j-1])}} \left[\bigotimes_{(x_i, y_i) \in M'} \omega(x_i, y_i) \right] \\
&\quad \otimes \omega(T_1[i], T_2[j]). \tag{4.13}
\end{aligned}$$

Now, since the abstract measure over all extended mapping M' between the subtree rooted at i and the subtree rooted at j such that $(T_1[i], T_2[j]) \in M'$ is denoted by $T_Meas_S(i, j)$, we have :

$$T_Meas_S(i, j) = \bigotimes_{M' \in \Gamma_{(T_1[\delta(i)..i], T_2[\delta(j)..j])}} \left[\bigotimes_{(x_i, y_i) \in M'} \omega(x_i, y_i) \right]$$

subject to the condition that $(T_1[i], T_2[j])$ is in M' . Thus, by using the distributivity and associativity of $\otimes$, we have

$$\begin{aligned}
& T_Meas_S(i, \cdot) \\
&= \sum_{M' \in \Gamma_{(T_1[\delta(i)..i], T_2[\delta(j)..j])}} \left[\prod_{(x_i, y_i) \in M' - \{(T_1[i], T_2[j])\}} \omega(x_i, y_i) \otimes \omega(T_1[i], T_2[j]) \right] \\
&= \sum_{M' \in \Gamma_{(T_1[\delta(i)..i-1], T_2[\delta(j)..j-1])}} \left[\prod_{(x_i, y_i) \in M'} \omega(x_i, y_i) \right] \otimes \omega(T_1[i], T_2[j])
\end{aligned}$$

By comparing this equation with equation (5.7), we can observe that

$$F_Meas(\delta(i)..i-1, \delta(j)..j-1) \otimes \omega(T_1[i], T_2[j]) = T_Meas_S(i, j), \quad (4.14)$$

and the result follows directly from this equality. ***

4.5.2 A Dynamic Programming Algorithm to Compute $\Omega(T_1, T_2)$

Theorem 4.9 suggests that we can use a dynamic programming flavoured algorithm to solve the general tree comparison problem. First of all, note that the second part of Theorem 4.9 suggests that if we are to compute the quantities $T_Meas(i_1, j_1)$, as well as $T_Meas_S(i_1, j_1)$, we have to have available the quantities $T_Meas(i, j)$ and $T_Meas_S(i, j)$ for all i and j , where the nodes i and j are all the descendants of i_1 and j_1 except nodes on the path from i_1 to $\delta(i_1)$ and the nodes on the path from j_1 to $\delta(j_1)$. Furthermore, the theorem asserts that the distances associated with the nodes which are on the path from i_1 to $\delta(i_1)$ get computed (as a by-product⁷) in the process of computing the F_Meas between the trees rooted at i_1 and j_1 . Indeed, the set of nodes for which the computation of T_Meas and T_Meas_S must be done independently before the T_Meas and T_Meas_S associated with

⁷The reason why this is obtained as a by-product will be clear when the algorithm is formally presented. Indeed, whenever an F_Meas is computed, if the forests are trees, it is retained as a T_Meas .

their ancestors can be computed is called the set of "Essential_Nodes", and these are merely those nodes for which the computation would involve the "Otherwise" case of Theorem 4.9 as opposed to the "If" case. Thus, the T_Meas and T_Meas_S can be computed for the entire tree if T_Meas and T_Meas_S of the Essential_Nodes are computed, and using these stored values, the rest of the T_Meas and T_Meas_S can be computed. This suggests a bottom-up approach for computing the T_Meas and T_Meas_S between all pairs of subtrees. To formally present the algorithm, we define the set of Essential_Nodes described above.

Definition : We define the set Essential_Nodes⁸ of tree T as follows :

$$Essential_Nodes(T) = \{k \mid \text{there exists no } k' > k \text{ such that } \delta(k) = \delta(k')\}.$$

Informally speaking, if k is in Essential_Nodes(T) then either k is the root or k has a left sibling. Intuitively, this set will be the roots of all subtrees of tree T that need separate computations. As an example, consider the trees as shown in Figure 3.5, the Essential_Nodes(T_1) = {2,6,7,8} and the Essential_Nodes(T_2) = {4,5,6}.

As in Algorithm Zhang and Shasha, we assume that the function $\delta()$ and the set Essential_Nodes() is pre-computed (in linear time), and that the corresponding arrays $\delta[]$ and Essential_Nodes[] are maintained with the elements in Essential_Nodes [] being sorted in an increasing order as per the postorder representation.

⁸The set of nodes which we refer to as the set of Essential_Nodes happens to be exactly the same as the set of nodes defined in Section 3.5 [ZS89] as the LR_keyroots set. Although these sets are identical, the implication of a node being in the sets is slightly different. In [ZS89] $i \in LR_keyroots(T_1)$ and $j \in LR_keyroots(T_2)$ implies that the corresponding tree weights associated with the trees rooted at these nodes need precomputation. In our case, $i \in Essential_Nodes(T_1)$ and $j \in Essential_Nodes(T_2)$ implies that the corresponding abstract tree measure involving $\oplus$ and $\otimes$ between the trees rooted at these nodes need precomputation with the additional constraint that the last operation involved in the comparison is a substitution.

ALGORITHM Abstract_T_Measure**INPUT :** Trees T_1 and T_2 and the set of elementary edit measures.**OUTPUT :** $\Omega(T_1, T_2)$.**ASSUMPTION :**

The algorithm assumes a procedure **Preprocess** (T_1, T_2) which preprocesses the trees and yields the $\delta[]$ and **Essential_Nodes**[] arrays for both trees. These quantities are assumed to be global. It also invokes the procedure **Compute Abstract_T_Meas()** given subsequently which globally computes the arrays $T_Meas(i, j)$ and $T_Meas_S(i, j)$ for $1 \leq i \leq |T_1|$, $1 \leq j \leq |T_2|$.

BEGINPreprocess (T_1, T_2);FOR $i' = 1$ to $|Essential_Nodes_1[]|$ DOFOR $j' = 1$ to $|Essential_Nodes_2[]|$ DO $i = Essential_Nodes_1[i']$; $j = Essential_Nodes_2[j']$;Compute_Abstract_T_Meas (i, j);

ENDFOR

ENDFOR

 $\Omega(T_1, T_2) = T_Meas[|T_1|][|T_2|]$ **END ALGORITHM Abstract_T_Measure.**

In the succeeding computation we shall attempt to evaluate $T_Meas(i, j)$ and $T_Meas_S(i, j)$ and store it in two *permanent* two-dimensional arrays T_Meas and T_Meas_S . From Theorem 4.9, we observe that to compute the quantities $T_Meas(i, j)$ and $T_Meas_S(i, j)$ the quantities which are involved are precisely the terms $F_Meas([\delta(i)..h], [\delta(j)..k])$ defined for a particular input pair (i, j) , where h and k are the internal nodes of $Tree_1(i)$ and $Tree_2(j)$ satisfying, $\delta(i) \leq h \leq i$, $\delta(j) \leq k \leq j$. Our intention is to store *these* values using a *single temporary* two-dimensional array $F_Meas[...]$. In order to achieve this, the base indices of the two-dimensional array $F_Meas[...]$ will have to be adjusted each time the procedure is invoked, so as to allow the possibility of utilizing the *same* memory allocations repeatedly for every computation. This is achieved by assigning the base values b_1 and b_2 as : $b_1 = \delta_1(i) - 1$, and $b_2 = \delta_2(j) - 1$.

Thus, for a particular input pair (i, j) , the same memory allocations $F_Meas[...]$ can be used to store the values in each phase of the computation by assigning :

$$F_Meas[x_1, y_1] = F_Meas([\delta(i)..\delta(i)+x_1-1], [\delta(j)..\delta(j)+y_1-1])$$

for all $1 \leq x_1 \leq i - \delta(i) + 1$, $1 \leq y_1 \leq j - \delta(j) + 1$.

Consequently, we note that for every x_1 and y_1 in any intermediate step in the algorithm, the quantities $T_Meas()$ and $T_Meas_S()$ that have to be stored in the permanent arrays can be obtained by incorporating these base values again, and have the form $T_Meas[x_1+b_1, y_1+b_2]$ and $T_Meas_S[x_1+b_1, y_1+b_2]$, respectively. The algorithm to compute the arrays T_Meas and T_Meas_S is shown on the next page.

After the array $T_Meas[.,.]$ has been computed, the value of the abstract measure $\Omega(T_1, T_2)$ between the trees T_1 and T_2 is stored in the array $T_Meas[|T_1|, |T_2|]$.

4.5.3 Remarks

As mentioned in the Section 4.1, our algorithm is similar (in spirit) to the earlier foundational work of Zhang and Shasha [ZS89]. However, we emphasize that our scheme is not merely a straightforward extension of [ZS89] in which the operators and the elementary distances are correspondingly generalized. Unlike their analytic results, which were primary derived for the case of the edit distance between the trees, our results are true for all operators $\otimes$ and $\oplus$, and for all well-defined elementary measures $\omega(.,.)$. Thus the *modus operandus* by which we have proved our results is completely different from the way by which their results have been derived [ZS89]. Furthermore, unlike the elementary edit distance, the abstract measure defined in this paper is defined by repeatedly using $\oplus$ over all extended mappings; hence, in the corresponding proofs, care must be taken to ensure that every such mapping is accounted for *exactly* once.

However, the fundamental difference between our algorithm and the algorithm in [ZS89] is the fact that unlike [ZS89], in our case, the F_Meas and the T_Meas are not directly related. This is because, if the corresponding quantities relating intermediate

PROCEDURE Compute_Abstract_T_Meas**INPUT :** Indices i, j and the quantities assumed global in **Abstract_T_Measure**.**OUTPUT :** The procedure essentially has no output. But for a given input pair (i, j) , it globally computes the quantities $T_Meas(i_1, j_1)$ and $T_Meas_S(i_1, j_1)$ for $\delta_1(i) \leq i_1 \leq i, \delta_2(j) \leq j_1 \leq j$. These are maintained in the arrays $T_Meas[i_1, j_1]$ and $T_Meas_S[i_1, j_1]$.**BEGIN**N = $i - \delta_1(i) + 1$; /* size of subtree rooted at $T_1[i]$ */M = $j - \delta_2(j) + 1$; /* size of subtree rooted at $T_2[j]$ */ $b_1 = \delta_1(i) - 1$; /* adjustment for nodes in subtree rooted at $T_1[i]$ */ $b_2 = \delta_2(j) - 1$; /* adjustment for nodes in subtree rooted at $T_2[j]$ */

F_Meas[0][0] = 0;

FOR $x_1 = 1$ to N DOF_Meas [x_1][0] = F_Meas [$x_1 - 1$][0] $\otimes \omega(T_1[x_1 + b_1] \rightarrow \lambda)$;T_Meas_S [$x_1 + b_1$][0] = I , the identity of $\otimes$;T_Meas [$x_1 + b_1$][0] = F_Meas [x_1][0];

ENDFOR

FOR $y_1 = 1$ to M DOF_Meas [0][y_1] = F_Meas [0][$y_1 - 1$] $\otimes \omega(\lambda \rightarrow T_2[y_1 + b_2])$;T_Meas_S [0][$y_1 + b_2$] = I , the identity of $\otimes$;T_Meas [0][$y_1 + b_2$] = F_Meas [0][y_1];

ENDFOR

FOR $x_1 = 1$ to N DOFOR $y_1 = 1$ to M DOIF $\delta_1(x_1 + b_1) = \delta_1(x_1)$ and $\delta_2(y_1 + b_2) = \delta_2(y_1)$ THENF_Meas [x_1][y_1]= [F_Meas [$x_1 - 1$][y_1] $\otimes \omega(T_1[x_1 + b_1] \rightarrow \lambda)$] $\otimes$ [F_Meas [x_1][$y_1 - 1$] $\otimes \omega(\lambda \rightarrow T_2[y_1 + b_2])$] $\otimes$ [F_Meas [$x_1 - 1$][$y_1 - 1$] $\otimes \omega(T_1[x_1 + b_1] \rightarrow T_2[y_1 + b_2])$]T_Meas_S [$x_1 + b_1$][$y_1 + b_2$]= F_Meas [$x_1 - 1$][$y_1 - 1$] $\otimes \omega(T_1[x_1 + b_1] \rightarrow T_2[y_1 + b_2])$;T_Meas [$x_1 + b_1$][$y_1 + b_2$] = F_Meas [x_1][y_1];

ELSE

F_Meas [x_1][y_1]= [F_Meas [$x_1 - 1$][y_1] $\otimes \omega(T_1[x_1 + b_1] \rightarrow \lambda)$] $\otimes$ [F_Meas [x_1][$y_1 - 1$] $\otimes \omega(\lambda \rightarrow T_2[y_1 + b_2])$] $\otimes$ [F_Meas [a][b] $\otimes$ T_Meas_S [$x_1 + b_1$][$y_1 + b_2$]]where $a = \delta_1(x_1 + b_1) - 1 - b_1, b = \delta_2(y_1 + b_2) - 1 - b_2$.

ENDIF

ENDFOR

ENDFOR

END PROCEDURE Compute_Abstract_T_Meas.

inter-tree measures of [ZS89] (referred to as *treedist* in Lemma IV of [ZS89]) were transferred to our perspective, it would be seen that the resulting measures would "overestimate" the quantities currently being computed in this abstract domain. Thus the previous scheme would yield erroneous results, especially when the arithmetic addition operation is used as a concrete example for $\oplus$. In our case, we have introduced a new auxiliary measure T_Meas_S in which all the intermediate inter-tree computations have to be maintained by utilizing a constrained inter-tree measure in which the extended mappings require that the roots of the corresponding trees are substituted.

4.6 Time and Space Complexities

We shall now analyze the time and space complexities of the algorithm. From [ZS89], we have the following result :

THEOREM 4.10

If for a tree T , $Span(T)$ is defined as the $Min(Depth(T), Leaves(T))$, the time and space complexities of this algorithm are :

Time : $O(|T_1| * |T_2| * Span(T_1) * Span(T_2))$ operations of $\oplus$ and $\otimes$.

Space : $O(|T_1| * |T_2|)$.

PROOF

For space complexity, we use two permanent arrays for T_Meas and T_Meas_S , and a temporary array for F_Meas . Clearly, each of these three arrays requires space $O(|T_1| * |T_2|)$.

With regard to the time complexity, we first observe that the preprocessing takes linear time to precompute the arrays $\delta[]$ and $Essential_Nodes[]$ for both trees. Also, note that if the array $T_Meas[...]$ is computed, the abstract measure, $\Omega(T_1, T_2)$, can be computed in time $O(1)$ by looking at the content of the array directly. Hence, the dominant term involves computing the arrays $T_Meas(i,j)$ and $T_Meas_S(i,j)$ for all relevant i , and j .

This dynamic programming algorithm involving the subtrees rooted at i and j , respectively, involves : $O(\text{Size}(i) * \text{Size}(j))$ computations. Therefore, total time required is :

$$\begin{aligned} & \sum_{i=1}^{|\text{Essential_Nodes}(T_1)|} \sum_{j=1}^{|\text{Essential_Nodes}(T_2)|} \text{Size}(i) * \text{Size}(j) \\ \leq & \sum_{i=1}^{|\text{Essential_Nodes}(T_1)|} \text{Size}(i) * \sum_{j=1}^{|\text{Essential_Nodes}(T_2)|} \text{Size}(j) \end{aligned}$$

Since the sets `Essential_Nodes` is identical to the set defined in [ZS89] as the `LR_keyroots`, we see from Theorem 2 of [ZS89] that :

$$\sum_{i=1}^{|\text{Essential_Nodes}(T)|} \text{Size}(i) \leq |T| * \text{Min}\{\text{Depth}(T), \text{Leaves}(T)\}$$

Since $\text{Span}(T)$ is equal to $\text{Min}\{\text{Depth}(T), \text{Leaves}(T)\}$, and since every single computation here requires at most three invocations of $\otimes$ and two invocations of $\oplus$, the time complexity of the algorithm is $O(|T_1| * |T_2| * \text{Span}(T_1) * \text{Span}(T_2))$, and the theorem is proved. ***

4.7 Conclusions

In this chapter we have defined an abstract measure of comparison, $\Omega(T_1, T_2)$, between two trees T_1 and T_2 . This measure has been presented in terms of a set of elementary inter-symbol measures $\omega(.,.)$ and two abstract operators $\oplus$ and $\otimes$. By appropriately choosing the concrete values for these two operators and for $\omega(.,.)$, this measure can be used to define various numeric similarity and dissimilarity measures as

shown in Table 4.1 below. Apart from the definition and formulation of the abstract measure $\Omega(T_1, T_2)$, we have also derived its recursive properties and presented an iterative dynamic programming scheme for its computation.

The algorithm presented in this chapter which computes the abstract measure has been implemented and rigorously tested for various concrete metrics. Also, a time and space analysis of the scheme has been derived in Section 4.6.

The actual physical quantities represented by $\Omega(T_1, T_2)$ is summarized by Table 4.1 below in which we have tabulated various concrete values for the operators $\oplus$ and $\otimes$, and for the function $\omega(\dots)$.

Problem	$\tau = (T, \oplus, \otimes, \theta, I)$	$\omega(\dots)$
Size of LCsT	$(Z_p', \text{MAX}, +, 0, 0)$	$\omega(a,b) = 1$, if $a=b \neq \lambda$, $\omega(a,b) = 0$, if $a \neq b \neq \lambda$, $\omega(a,\lambda) = 0$, and $\omega(\lambda,a) = 0$.
Size of SCST	$(Z_p', \text{MIN}, +, \infty, 0)$	$\omega(a,b) = 1$, if $a=b$, $\omega(a,b) = \infty$, if $a \neq b$, $\omega(a,\lambda) = 1$, and $\omega(\lambda,a) = 1$.
Probability of erroneous trees with substitution and deletion errors.	$([0, 1], +, *, 0, 1)$	$\omega(a,b) = p(b a)$.
Likelihood of erroneous trees with all three types of errors. From this the <i>a posteriori</i> probabilities can be defined.	$([0, 1], +, *, 0, 1)$	$\omega(a,b) = q(a \rightarrow b)$.
Distance ⁹	$(Z_p', \text{MIN}, +, 0, 0)$	$\omega(a,b) = 0$, if $a=b \neq \lambda$, $\omega(a,b) = 1$, if $a \neq b \neq \lambda$, $\omega(a,\lambda) = 1$, and $\omega(\lambda,a) = 1$.

Table 4.1 : Summary of the use of generalized operators and generalized edit measures to evaluate various Tree-to-Tree similarity measures.

⁹In this case our algorithm (with the omission of the array T_Meas_S) would reduce to the one presented in [ZS89].

V. Constrained Tree Editing

5.1 Introduction

As mentioned in Chapter I, all of the existing tree editing algorithms consider the editing of one tree, say T_1 , transforming it to another tree T_2 , with the edit processes being unconstrained. In this chapter we consider the problem of editing T_1 to T_2 subject to any general edit constraint. This constraint can be arbitrarily complex as long as it is specified in terms of the number and type of edit operations to be included in the optimal edit transformation. For the sake of clarity we present some examples of constrained editing problems.

EXAMPLE 5.1 :

Below are some constrained editing problems :

- (a) What is the optimal way of editing T_1 to T_2 using no more than k deletions ?
- (b) How can we optimally transform T_1 to T_2 using exactly k substitutions ?
- (c) Is it possible to transform T_1 to T_2 using exactly k_i insertions, k_e deletions and k_s substitutions ? If so, what is the distance between T_1 to T_2 subject to this constraint ?

In this chapter, we first present a consistent method of specifying arbitrary edit constraints. This method is analogous to (though slightly different from) the one shown in [Oo86] and in Section 2.2.5 which is specified by a constraint set, τ . We will then discuss the computation of $D_\tau(T_1, T_2)$, the edit distance between T_1 and T_2 subject to this constraint. The latter quantity is computed by first evaluating elements of a three-dimensional array using dynamic programming techniques, and then combining certain

elements of this three-dimensional array to yield $D_{\tau}(T_1, T_2)$. As in Chapter III and IV, if $\text{Span}(T)$ is defined as the $\text{Min} \{ \text{Depth}(T), \text{Leaves}(T) \}$, it will be shown that the algorithm requires $O(|T_1| \cdot |T_2| \cdot \text{Min} \{ |T_1|, |T_2| \}^2 \cdot \text{Span}(T_1) \cdot \text{Span}(T_2))$ time and $O(|T_1| \cdot |T_2| \cdot \text{Min} \{ |T_1|, |T_2| \})$ space. Using the three-dimensional array referred to earlier, the optimal sequence of edit operations required to edit T_1 to T_2 subject to the constraint τ can then be obtained by backtracking.

5.2 Notations and Definitions

Unless otherwise specified, the notation used in this chapter is similar to that of Chapter III. Also the definitions of the edit operations, mappings, and the distance between trees are identical to the definitions shown in Section 3.2. Finally, throughout this chapter, the postorder numbering will be used to number the nodes in the tree under consideration.

5.3 Edit Constraints

We now present a consistent method of specifying arbitrary edit constraints. This method is analogous to the one shown in [Oo86] but differs slightly. The reasons for the difference will be clarified later on.

5.3.1 Permissible and Feasible Edit Operations

Consider the problem of editing T_1 to T_2 , where $|T_1| = l$, and $|T_2| = M$. Suppose we edit a postorder-forest of T_1 into a postorder-forest of T_2 using exactly i insertions, e deletions, and s substitutions. Since the number of edit operations has been specified, this corresponds to editing $T_1[1..e+s]$ into $T_2[1..i+s]$.

As in Section 2.2.5, to obtain bounds on the magnitudes of variables i , e , s , we observe that they are constrained by the sizes of trees T_1 and T_2 . Thus, if $r=e+s$, $q=i+s$, and $R=\text{Min}\{N, M\}$, these variables will have to obey the following constraints :

$$\text{Max}\{0, M-N\} \leq i \leq q \leq M,$$

$$0 \leq e \leq r \leq N,$$

$$\text{and } 0 \leq s \leq R.$$

Values of triples (i, e, s) which satisfy these constraints are termed *feasible values* of the variables. Let

$$H_i = \{j \mid \text{max}\{0, M-N\} \leq j \leq M\},$$

$$H_e = \{j \mid 0 \leq j \leq N\},$$

$$H_s = \{j \mid 0 \leq j \leq \text{Min}\{M, N\}\}.$$

H_i , H_e , and H_s are called the set of *permissible values* of i , e , and s .

The following theorem specifies the permitted forms of feasible triples which are encountered in editing $T_1[1..r]$, the postorder-forest of T_1 of size r , to $T_2[1..q]$, the postorder-forest of T_2 of size q .

THEOREM 5.1

To edit $T_1[1..r]$, the postorder-forest of T_1 of size r , to $T_2[1..q]$, the postorder-forest of T_2 of size q , the set of feasible triples is given by :

$$\{(q-s, r-s, s) \mid 0 \leq s \leq \text{Min}\{M, N\}\}.$$

PROOF

Consider the constraints imposed on feasible values of i , e , and s . Since we are interested in editing $T_1[1..r]$ to $T_2[1..q]$, we have to consider only those triples (i, e, s) in which $i+s=r$ and $e+s=q$. However, the number of substitution can take any value from 0 to $\text{Min}\{r, q\}$. Therefore, for every value of s in this range, the feasible triple (i, e, s) must have exactly $r-s$ deletions since $r=e+s$. Similarly, the triple (i, e, s) must have exactly $q-s$ insertions since $q=s+i$, giving the desired result. ***

5.3.2 Specification of Edit Constraints

An edit constraint is specified in terms of the number and type of edit operations that are required in the process of transforming T_1 to T_2 . It is expressed by formulating the number and type of edit operations in terms of three sets Q_i , Q_e , and Q_s which are subsets of the sets H_i , H_e , and H_s defined in last subsection. We will now clarify this using the three constraints given in the following example.

EXAMPLE 5.2 :

(a) To edit T_1 to T_2 performing no more than k deletions, the sets Q_s and Q_i are both equal to ϕ , the null set. Further,

$$Q_e = \{j \mid j \in H_e, j \leq k\}$$

(b) To edit T_1 to T_2 performing exactly k_i insertions, k_e deletions, and k_s substitutions yields

$$Q_i = \{k_i\} \cap H_i, \quad Q_e = \{k_e\} \cap H_e, \quad \text{and} \quad Q_s = \{k_s\} \cap H_s. \quad ***$$

THEOREM 5.2

Every edit constraint specified for the process of editing T_1 to T_2 can be written as a unique subset of H_s .

PROOF

Let the constraint be specified by the sets Q_i , Q_e , and Q_s . Every element $j \in Q_i$ requires editing to be performed using exactly j insertions. Since $|T_2| = M$, from Theorem 5.1, this requires that the number of substitutions be $M-j$.

Similarly, if $j \in Q_e$, the edit transformation must contain exactly j deletions. Since $|T_1| = N$, Theorem 5.1 requires that $N-j$ substitutions be performed. Let

$$Q_e^* = \{N-j \mid j \in Q_e\},$$

and

$$Q_i^* = \{M-j \mid j \in Q_i\}.$$

Clearly, for any arbitrary constraint, the number of substitutions that are permitted is given by

$$Q_s \cap Q_e^* \cap Q_i^*$$

which is obviously a subset of H_s .

EXAMPLE 5.3 :

Let T_1 and T_2 be 2 trees as shown in Figure 3.5. Suppose we want to transform T_1 to T_2 by performing at most 5 insertions, at least 3 substitutions and exactly 3 deletion. Then

$$Q_i = \{0,1,2,3,4,5\}, \quad Q_e = \{3\}, \quad \text{and } Q_s = \{4,5,6\}.$$

Hence

$$Q_e^* = \{5\}, \quad Q_i^* = \{1,2,3,4,5,6\}$$

Thus,

$$\tau = Q_s \cap Q_e^* \cap Q_i^* = \{5\}.$$

Hence, the optimal transformation must contains *exactly* 5 substitutions.

5.3.3 Differences between the Constrained String and Tree Editing Problem Specification

At this juncture, it is appropriate to distinguish between the constrained string editing problem and the constrained tree editing problem. The points of similarity are few. First of all, the similarity includes the technique by which the permissible and feasible edit operations can be defined. Indeed, by virtue of this similarity, Theorem 5.1 can be seen to be analogous to the corresponding theorem (Theorem 5.1 of [Oo86]) in the constrained string editing problem. The second similarity involving the specification of the edit constraints. In the case of the constrained string editing problem, this set was defined in terms of the subset of the number of the feasible insertions. It is interesting to note that it could, just as effectively, have been specified as a subset of the feasible substitutions. In

this case, we have opted to use the latter formulation primarily because the postorder representation permits us to always increase the number of substitutions performed more directly.

Although the similarities are few, the differences between these two problems are far more pronounced. First of all, it should be noted that the constrained tree editing problem is not even well-defined if the mappings are not consistent. Thus, for every edit S -derivation, not only must the number of substitutions be feasible, but also the corresponding mapping must be well-defined. Additionally and more importantly, two stringent constraints, namely the sibling and the parent relationship must also be satisfied. Thus, the constrained tree editing problem can be considered as a “conditionally optimized” constrained string editing problem: in which the postorder string of T_1 is edited to the postorder string of T_2 subject to H_s and further subject to the sibling/parent conditions. This is the reason why the problem currently studied is far more complex.

We shall refer to the edit distance subject to the constraint τ as $D_\tau(T_1, T_2)$. By definition, $D_\tau(T_1, T_2) = \infty$ if $\tau = \emptyset$, the null set. This is merely a way of expressing that it is impossible to edit T_1 to T_2 subject to the constraint τ . We now consider the computation of $D_\tau(T_1, T_2)$.

5.4 Constrained Tree Editing

Since edit constraints can be written as unique subsets of H_s , we will denote the distance between forest $T_1[i'..i]$ and forest $T_2[j'..j]$ subject to the constraint that exactly s substitutions are performed by $\text{Const_F_Wt}(T_1[i'..i], T_2[j'..j], s)^{10}$ or more precisely by $\text{Const_F_Wt}([i'..i], [j'..j], s)$ if the context is clear. The distance between $T_1[1..i]$ and $T_2[1..j]$ subject to the constraint that exactly s substitutions are used is sometimes denoted by $\text{Const_F_Wt}(i, j, s)$ since the starting index of both trees is unity. As opposed to this,

¹⁰This quantity is a generalization of the quantity F_Wt in Chapter IV. It is a constrained “ F_Wt ” with the constraint that exactly s substitutions are used in the edit process.

the distance between the *subtree* rooted at i and the *subtree* rooted at j subject to the constraint that exactly s substitutions are used is denoted by $\text{Const_T_Wt}(i,j,s)$. The difference between Const_F_Wt and Const_T_Wt is subtle. Indeed,

$$\text{Const_T_Wt}(i,j,s) = \text{Const_F_Wt}(T_1[\delta(i)..i], T_2[\delta(j)..j], s),$$

and compares segments of T_1 and T_2 which are exactly trees.

5.4.1 The Constrained Editing Algorithm

To develop the Constrained Tree Editing Algorithm, we now present the following results :

LEMMA 5.1a

Let $i_1 \in \text{Anc}(i)$ and $j_1 \in \text{Anc}(j)$, then

- (i) $\text{Const_F_Wt}(\mu, \mu, 0) = 0.$
- (ii) $\text{Const_F_Wt}(T_1[\delta(i_1)..i], \mu, 0)$
 $= \text{Const_F_Wt}(T_1[\delta(i_1)..i-1], \mu, 0) + d(T_1[i], \lambda).$
- (iii) $\text{Const_F_Wt}(\mu, T_2[\delta(j_1)..j], 0)$
 $= \text{Const_F_Wt}(\mu, T_2[\delta(j_1)..j-1], 0) + d(\lambda, T_2[j]).$
- (iv) $\text{Const_F_Wt}(T_1[\delta(i_1)..i], T_2[\delta(j_1)..j], 0)$
 $= \text{Min} \begin{cases} \text{Const_F_Wt}(T_1[\delta(i_1)..i-1], T_2[\delta(j_1)..j], 0) + d(T_1[i], \lambda) \\ \text{Const_F_Wt}(T_1[\delta(i_1)..i], T_2[\delta(j_1)..j-1], 0) + d(\lambda, T_2[j]). \end{cases}$

PROOF

Case (i) requires no edit operations. In cases (ii) and (iii), the distance corresponds to the cost of deleting and inserting nodes in $T_1[\delta(i_1)..i]$ and $T_2[\delta(j_1)..j]$, respectively. In case (iv), since no substitution is allowed, the minimum cost mapping can be extended to $T_1[i]$ and $T_2[j]$ by either inserting $T_2[j]$ or deleting $T_1[i]$ only, and the minimum of these two costs will be the desired distance. ***

LEMMA 5.1b

Let $i_1 \in \text{Anc}(i)$ and $j_1 \in \text{Anc}(j)$, then

- (i) $\text{Const_F_Wt}(T_1[\delta(i_1)..i], \mu, s) = \infty$ if $s > 0$.
- (ii) $\text{Const_F_Wt}(\mu, T_2[\delta(j_1)..j], s) = \infty$ if $s > 0$.
- (iii) $\text{Const_F_Wt}(\mu, \mu, s) = \infty$ if $s > 0$.

PROOF

If $s > 0$, there must be exactly s nodes in both trees which are involved in the substitution. Since, none of the above three cases satisfies this, all of the distances equal ∞ , implying that it is impossible to edit the trees subject to this constraint. ***

THEOREM 5.3

Let $i_1 \in \text{Anc}(i)$ and $j_1 \in \text{Anc}(j)$, then

$$\begin{aligned} & \text{Const_F_Wt}(T_1[\delta(i_1)..i], T_2[\delta(j_1)..j], s) \\ &= \text{Min} \left\{ \begin{array}{l} \text{Const_F_Wt}([\delta(i_1)..i-1], [\delta(j_1)..j], s) + d(T_1[i], \lambda) \\ \text{Const_F_Wt}([\delta(i_1)..i], [\delta(j_1)..j-1], s) + d(\lambda, T_2[j]) \\ \text{Min}_{1 \leq s_2 \leq \text{Min}\{\text{Size}(i); \text{Size}(j)\}} \left\{ \begin{array}{l} \text{Const_F_Wt}([\delta(i_1)..i-1], [\delta(j_1)..j-1], s-s_2) \\ + \text{Const_F_Wt}([\delta(i)..i-1], [\delta(j)..j-1], s_2-1) \\ + d(T_1[i], T_2[j]) \end{array} \right\} \end{array} \right. \end{aligned}$$

PROOF

We are trying to find a minimum cost mapping M between $T_1[\delta(i_1)..i]$ and $T_2[\delta(j_1)..j]$ such that exactly s substitutions are performed. The map can be extended to $T_1[i]$ and $T_2[j]$ in the following three ways :

- (i) If $T_1[i]$ is not touched by any line in M , then $T_1[i]$ is to be deleted. Thus, since the number of substitutions in $\text{Const_F_Wt}(\dots)$ remains unchanged, we have :

$$\begin{aligned} & \text{Const_F_Wt}(T_1[\delta(i_1)..i], T_2[\delta(j_1)..j], s) \\ &= \text{Const_F_Wt}(T_1[\delta(i_1)..i-1], T_2[\delta(j_1)..j], s) + d(T_1[i], \lambda). \end{aligned}$$

(ii) If $T_2[j]$ is not touched by any line in M , then $T_2[j]$ is to be inserted. Again, since the number of substitutions in $\text{Const_F_Wt}(\dots)$ remains unchanged, we have :

$$\begin{aligned} & \text{Const_F_Wt}(T_1[\delta(i_1)..i], T_2[\delta(j_1)..j], s) \\ &= \text{Const_F_Wt}(T_1[\delta(i_1)..i], T_2[\delta(j_1)..j-1], s) + d(\lambda, T_2[j]). \end{aligned}$$

Observe that in both the above cases the numbers of substitutions done remains unchanged.

(iii) Consider the case when both $T_1[i]$ and $T_2[j]$ are touched by lines in M . From Theorem 4.8, we know that $(i, j) \in M$. Again, by the ancestor property of M , any node in the subtree rooted at $T_1[i]$ can only be touched by a node in the subtree rooted at $T_2[j]$. This situation is identical to that depicted by Figure 4.6 shown in Chapter IV.

Since exactly s substitutions must be performed in this transformation, the total number of substitutions used in the sub-transformation from $T_1[\delta(i_1)..i-1]$ to $T_2[\delta(j_1)..j-1]$ and the sub-transformation from $T_1[\delta(i)..i-1]$ to $T_2[\delta(j)..j-1]$ must be equal to $s-1$ (the last substitution being the operation $T_1[i] \rightarrow T_2[j]$). Let s_2-1 be the number of substitutions used in the sub-transformation from $T_1[\delta(i)..i-1]$ to $T_2[\delta(j)..j-1]$, then s_2 can take any value between 1 to $\text{Min}\{\text{Size}(i), \text{Size}(j), s\}$. Hence, we have :

$$\begin{aligned} & \text{Const_F_Wt}(T_1[\delta(i_1)..i], T_2[\delta(j_1)..j], s) \\ = & \min_{1 \leq s_2 \leq \text{Min}\{\text{Size}(i); \text{Size}(j); s\}} \begin{cases} \text{Const_F_Wt}(T_1[\delta(i_1)..i-1], T_2[\delta(j_1)..j-1], s-s_2) \\ + \text{Const_F_Wt}(T_1[\delta(i)..i-1], T_2[\delta(j)..j-1], s_2-1) \\ + d(T_1[i], T_2[j]) \end{cases} \end{aligned}$$

Since these three cases exhaust the possible ways for yielding $\text{Const_F_Wt}(\delta(i_1)..i, \delta(j_1)..j, s)$, the minimum of these three costs yields the result. ***

It is easy to construct a recursive algorithm by using the above theorem. However, both the time and space complexities of the algorithm will be prohibitively large - probably

of an octic time complexity. Improvements in both the time and space complexities can be obtained by utilizing the following result involving the quantity $\text{Const_T_Wt}(\dots)$ described earlier. Just as in the case of Theorem 4.9 of Chapter IV, we can indeed show that the Const_F_Wt reduces to become a Const_T_Wt under certain circumstances. This is also a generalization of Theorem 4.9 (for the case when $\oplus$ is the Min operator, $\oplus$ is the arithmetic addition, and ω_1 is the set of elementary edit distances) except that, in this case, we impose the additional constraint that the edit process must execute exactly s substitutions.

THEOREM 5.4

Let $i_1 \in \text{Anc}(i)$ and $j_1 \in \text{Anc}(j)$. Then the following is true :

If $\delta(i) = \delta(i_1)$ and $\delta(j) = \delta(j_1)$ then

$$\begin{aligned} & \text{Const_F_Wt}(T_1[\delta(i_1)..i], T_2[\delta(j_1)..j], s) \\ &= \text{Min} \begin{cases} \text{Const_F_Wt}(T_1[\delta(i_1)..i-1], T_2[\delta(j_1)..j], s) + d(T_1[i], \lambda) \\ \text{Const_F_Wt}(T_1[\delta(i_1)..i], T_2[\delta(j_1)..j-1], s) + d(\lambda, T_2[j]) \\ \text{Const_F_Wt}(T_1[\delta(i_1)..i-1], T_2[\delta(j_1)..j-1], s-1) + d(T_1[i], T_2[j]) \end{cases} \end{aligned}$$

Otherwise,

$$\begin{aligned} & \text{Const_F_Wt}(T_1[\delta(i_1)..i], T_2[\delta(j_1)..j], s) \\ &= \text{Min} \begin{cases} \text{Const_F_Wt}(T_1[\delta(i_1)..i-1], T_2[\delta(j_1)..j], s) + d(T_1[i], \lambda) \\ \text{Const_F_Wt}(T_1[\delta(i_1)..i], T_2[\delta(j_1)..j-1], s) + d(\lambda, T_2[j]) \\ \text{Min}_{1 \leq s_2 \leq \text{Min}\{\text{Size}(i); \text{Size}(j); s\}} \left\{ \text{Const_F_Wt}(T_1[\delta(i_1)..i-1], T_2[\delta(j_1)..j-1], s-s_2) \right. \\ \left. + \text{Const_T_Wt}(i, j, s_2) \right\} \end{cases} \end{aligned}$$

PROOF

By Theorem 5.3, if $\delta(i) = \delta(i_1)$ and $\delta(j) = \delta(j_1)$, then the forests $T_1[\delta(i_1)..i-1]$ and $T_2[\delta(j_1)..j-1]$ are both empty. Thus,

$$\text{Const_F_Wt}(T_1[\delta(i_1)..i-1], T_2[\delta(j_1)..j-1], s-s_2) = \text{Const_F_Wt}(\phi, \phi, s-s_2)$$

which is equal to zero if $s_2 = s$, or is equal to ∞ if $s_2 < s$. The first part of the theorem follow:.

For the second part, since the distance is the cost of the minimal cost mapping, we know that:

$$\begin{aligned} & \text{Const_F_Wt } (T_1[\delta(i_1)..i], T_2[\delta(j_1)..j], s) \\ & \leq \quad \text{Const_F_Wt } (T_1[\delta(i_1)..i-1], T_2[\delta(j_1)..j-1], s-s_2) \\ & \quad + \text{Const_T_Wt } (i, j, s_2). \end{aligned}$$

This is because the latter formula represents a particular mapping of $T_1[\delta(i_1)..i]$ to $T_2[\delta(j_1)..j]$ in which the forest $T_1[\delta(i_1)..i-1]$ is transformed into the forest $T_2[\delta(j_1)..j-1]$ and subsequently the tree rooted at i is transformed into the tree rooted at j . Thus the second term in the above expression is a Const_T_Wt and not an Const_F_Wt . For the same reason, we have :

$$\begin{aligned} & \text{Const_T_Wt } (i, j, s_2) \\ & \leq \quad \text{Const_F_Wt } (T_1[\delta(i_1)..i-1], T_2[\delta(j_1)..j-1], s_2-1) + d(T_1[i], T_2[j]). \end{aligned}$$

Theorem 5.3 and these two inequalities justify the substituting of $\text{Const_T_Wt } (i, j, s_2)$ for the corresponding Const_F_Wt expressions, and the result follows. ***

As in Sections 3.5 and 4.5.2, Theorem 5.4 suggests that we can use a dynamic programming flavoured algorithm to solve the constrained tree editing problem. As in Chapter IV, we observe that to compute the quantity $\text{Const_T_Wt}(i_1, j_1, s)$ we need the quantities $\text{Const_T_Wt } (i, j, s_2)$ for all i and j and for all feasible values of $0 \leq s_2 \leq s$, where the nodes i and j are all the descendants of i_1 and j_1 except nodes on the path from i_1 to $\delta(i_1)$ and the nodes on the path from j_1 to $\delta(j_1)$. Thus, a scheme similar in principle to the one used in Chapter IV can be used except that the minimization process will have to include an additional parameter, namely, s , the number of substitutions. Indeed, just as in the case of the results derived in Chapter IV, the set of nodes for which the computation of Const_T_Wt must be done independently before the Const_T_Wt associated with their

ancestors can be computed turns out to be the identical set referred to as the set of "Essential_Nodes"¹¹ in Chapter IV. These are merely those nodes for which the computation would involve the second case of Theorem 5.4 as opposed to the first. Thus, as before, the Const_T_Wt can be computed for the entire tree if Const_T_Wt of the Essential_Nodes are computed, and using this stored values, the rest of the Const_T_Wts can be computed.

In order to describe the algorithm formally, we again assume the existence of the arrays $\delta[]$ and Essential_Nodes [], where the latter is sorted in the increasing order of the postorder numbering of the nodes.

PROCEDURE T_Weights

INPUT : Trees T_1 and T_2 and the set of elementary edit distances.

OUTPUT : Const_T_Wt (i, j, s), $1 \leq i \leq |T_1|$, $1 \leq j \leq |T_2|$, and $1 \leq s \leq \text{Min}\{|T_1|, |T_2|\}$.

ASSUMPTION :

The algorithm assume a procedure Preprocess (T_1, T_2) which preprocesses the trees and yields the $\delta []$ and Essential_Nodes [] arrays for both trees. These quantities are assumed to be global. It also invokes the procedure Compute_Const_T_Wt given subsequently.

BEGIN

Preprocess (T_1, T_2) ;

FOR $i' = 1$ to | Essential_Nodes₁ [] | DO

FOR $j' = 1$ to | Essential_Nodes₂ [] | DO

$i = \text{Essential_Nodes}_1 [i']$;

$j = \text{Essential_Nodes}_2 [j']$;

Compute_Const_T_Wt (i, j);

ENDFOR

ENDFOR

END PROCEDURE T_Weights

¹¹The set of nodes which we refer to as the set of Essential_Nodes here happens to be exactly the same as the set of nodes defined in [ZS89] as the LR_keyroots set and the set of nodes defined as the set of Essential_Nodes in Chapter IV. Although these sets are identical, the implication of a node being in the sets is slightly different. In [ZS89] $i \in \text{LR_keyroots}[T_1]$ and $j \in \text{LR_keyroots}[T_2]$ implies that the corresponding tree weights associated with the trees rooted at these nodes need precomputation. In this case, $i \in \text{Essential_Nodes}[T_1]$ and $j \in \text{Essential_Nodes}[T_2]$ implies that the corresponding constrained tree weights rooted at these trees need precomputation, and this precomputation must be achieved for all feasible values of s which are relevant.

In order to minimize the space requirements, techniques similar to those used in Section 4.5.2 can be utilized. The details of maintaining and utilizing the corresponding arrays are omitted here, but the reader will find it a trivial exercise to fill in these details. After the array $\text{Const_T_Wt} [\dots]$ has been computed, the distance $D_\tau(T_1, T_2)$ between the trees T_1 and T_2 subject to the constraint τ can be directly evaluated using the ALGORITHM $\text{Constrained_Tree_Distance}$ presented thereafter.

PROCEDURE Compute_Const_T_Wt

INPUT : Indexes i, j and the quantities assumed global in **T_Weights**.

OUTPUT : $\text{Const_T_Wt} [i_1, j_1, s]$, $\delta_1(i) \leq i_1 \leq i$, $\delta_2(j) \leq j_1 \leq j$, $0 \leq s \leq \text{Min}\{\text{Size}(i), \text{Size}(j)\}$.

BEGIN

$N = i - \delta_1(i) + 1;$ /* size of subtree rooted at $T_1[i]$ */
 $M = j - \delta_2(j) + 1;$ /* size of subtree rooted at $T_2[j]$ */
 $R = \text{Min}\{M, N\};$
 $b_1 = \delta_1(i) - 1;$ /* adjustment for nodes in subtree rooted at $T_1[i]$ */
 $b_2 = \delta_2(j) - 1;$ /* adjustment for nodes in subtree rooted at $T_2[j]$ */

$\text{Const_F_Wt} [0][0][0] = 0;$

FOR $x_1 = 1$ **to** N **DO**

$\text{Const_F_Wt} [x_1][0][0] = \text{Const_F_Wt} [x_1-1][0][0] + d(T_1[x_1+b_1] \rightarrow \lambda);$

$\text{Const_T_Wt} [x_1+b_1][0][0] = \text{Const_F_Wt} [x_1][0][0];$

ENDFOR

FOR $y_1 = 1$ **to** M **DO**

$\text{Const_F_Wt} [0][y_1][0] = \text{Const_F_Wt} [0][y_1-1][0] + d(\lambda \rightarrow T_2[y_1+b_2]);$

$\text{Const_T_Wt} [0][y_1+b_2][0] = \text{Const_F_Wt} [0][y_1][0];$

ENDFOR

FOR $s = 1$ **to** R **DO**

$\text{Const_F_Wt} [0][0][s] = \infty;$

$\text{Const_T_Wt} [0][0][s] = \text{Const_F_Wt} [0][0][s];$

ENDFOR

FOR $x_1 = 1$ **to** N **DO**

FOR $y_1 = 1$ **to** M **DO**

$\text{Const_F_Wt} [x_1][y_1][0]$

$= \text{Min} \begin{cases} \text{Const_F_Wt} [x_1][y_1-1][0] + d(\lambda \rightarrow T_2[y_1+b_2]) \\ \text{Const_F_Wt} [x_1-1][y_1][0] + d(T_1[x_1+b_1] \rightarrow \lambda) \end{cases}$

$\text{Const_T_Wt} [x_1+b_1][y_1+b_2][0] = \text{Const_F_Wt} [x_1][y_1][0];$

ENDFOR

ENDFOR

```

FOR  $x_1 = 1$  to  $N$  DO
  FOR  $s = 1$  to  $R$  DO
    Const_F_Wt [ $x_1$ ][0][ $s$ ] =  $\infty$ ;
    Const_T_Wt [ $x_1+b_1$ ][0][ $s$ ] = Const_F_Wt [ $x_1$ ][0][ $s$ ];
  ENDFOR
ENDFOR
FOR  $y_1 = 1$  to  $M$  DO
  FOR  $s = 1$  to  $R$  DO
    Const_F_Wt [0][ $y_1$ ][ $s$ ] =  $\infty$ ;
    Const_T_Wt [0][ $y_1+b_2$ ][ $s$ ] = Const_F_Wt [0][ $y_1$ ][ $s$ ];
  ENDFOR
ENDFOR
FOR  $x_1 = 1$  to  $N$  DO
  FOR  $y_1 = 1$  to  $M$  DO
    FOR  $s = 1$  to  $R$  DO
      IF  $\delta_1(x_1+b_1) = \delta_1(x)$  and  $\delta_2(y_1+b_2) = \delta_2(y)$  THEN
        Const_F_Wt [ $x_1$ ][ $y_1$ ][ $s$ ]
          = Min {
            Const_F_Wt [ $x_1-1$ ][ $y_1$ ][ $s$ ] +  $d(T_1[x_1+b_1] \rightarrow \lambda)$ 
            Const_F_Wt [ $x_1$ ][ $y_1-1$ ][ $s$ ] +  $d(\lambda \rightarrow T_2[y_1+b_2])$ 
            Const_F_Wt [ $x_1-1$ ][ $y_1-1$ ][ $s-1$ ]
              +  $d(T_1[x_1+b_1] \rightarrow T_2[y_1+b_2])$ 
          }
        Const_T_Wt [ $x_1+b_1$ ][ $y_1+b_2$ ][ $s$ ] = Const_F_Wt [ $x_1$ ][ $y_1$ ][ $s$ ];
      ELSE
        Const_F_Wt [ $x_1$ ][ $y_1$ ][ $s$ ]
          = Min {
            Const_F_Wt [ $x_1-1$ ][ $y_1$ ][ $s$ ] +  $d(T_1[x_1+b_1] \rightarrow \lambda)$ 
            Const_F_Wt [ $x_1$ ][ $y_1-1$ ][ $s$ ] +  $d(\lambda \rightarrow T_2[y_1+b_2])$ 
            Min_{ $1 \leq s_2 \leq \min\{c; d; s\}}$  { Const_F_Wt [ $a$ ][ $b$ ][ $s-s_2$ ]
              + Const_T_Wt [ $x_1+b_1$ ][ $y_1+b_2$ ][ $s_2$ ]
            }
          }
        where  $a = \delta_1(x_1+b_1) - 1 - b_1$ ,  $b = \delta_2(y_1+b_2) - 1 - b_2$ ,
               $c = \text{Size}(x_1+b_1)$  and  $d = \text{Size}(y_1+b_2)$ .
      ENDIF
    ENDFOR
  ENDFOR
ENDFOR
END PROCEDURE Compute_Const_T_Wt.

```

ALGORITHM Constrained_Tree Distance

INPUT : The array Const_T_Wt [..., ...] computed using Algorithm T_Weights,
and the constraint set τ .

OUTPUT : The constrained distance $D_\tau(T_1, T_2)$.

BEGIN

$D_\tau(T_1, T_2) = \infty$;

FOR all $s \in \tau$ DO

$D_\tau(T_1, T_2) = \text{Min} \{D_\tau(T_1, T_2), \text{Const_T_Wt} [|T_1|][|T_2|][s]\}$

ENDFOR

END ALGORITHM Constrained_Tree Distance.

THEOREM 5.5

The basic algorithm, Algorithm Compute Const_T_Wt, is correct.

PROOF

We will prove that the following two invariants hold for all pairs (i, j) such that $i \in \text{Essential_Nodes}(T_1)$ and $j \in \text{Essential_Nodes}(T_2)$:

(i) Immediately before the computation of Const_T_Wt (i, j, s) for all valid values of s , all distances Const_T_Wt (h, k, s') are available, where $\delta(i) \leq h \leq i$, $\delta(j) \leq k \leq j$, $0 \leq s' \leq \text{Min}\{\text{Size}(h), \text{Size}(k)\}$, and either $\delta(i) \neq \delta(h)$ or $\delta(j) \neq \delta(k)$. This, of course, is true because the values of h and k are either contained in $\text{Essential_Nodes}(T_1)$ and $\text{Essential_Nodes}(T_2)$ respectively, or can be computed from them. In other words, Const_T_Wt (h, k, s') is available if h is in the subtree of Tree(i) but not in the path from $\delta(i)$ to i , and k is in the subtree of Tree(j) but not in the path from $\delta(j)$ to j .

(ii) Immediately after the computation of Const_T_Wt (i, j, s) , all distances Const_T_Wt (h, k, s') are available, where $\delta(i) \leq h \leq i$, $\delta(j) \leq k \leq j$, and $0 \leq s' \leq \text{Min}\{\text{Size}(h), \text{Size}(k)\}$. In other words, all Const_T_Wt (h, k, s') are available including those nodes in the path from $\delta(i)$ to i , and in the path from $\delta(j)$ to j .

We will show that if (i) is true, then (ii) is true. From Theorem 5.4 and (i), we know that all required subtree-to-subtree distances are available. We compute each Const_T_Wt (h, k, s') , where $\delta(h) = \delta(i)$, $\delta(k) = \delta(j)$, and $0 \leq s' \leq \text{Min}\{\text{Size}(h), \text{Size}(k)\}$ using the IF part of Theorem 5.4 and subsequently include them in the permanent array Const_T_Wt. So (ii) holds.

Let us show that (i) always holds. Suppose $\delta(h) \neq \delta(i)$. Let h' be the lowest ancestor of h such that $h' \in \text{Essential_Nodes}(T_1)$. Clearly, such an ancestor exists since the root of T_1 is in $\text{Essential_Nodes}[]$. Since $\delta(h') = \delta(h) \neq \delta(i)$, we conclude that $h' \neq i$. Further, $i \in \text{Essential_Nodes}(T_1)$, we have $h' \leq i$. Combining the latter two assertions we obtain $h' < i$. Similarly, we have $k' < j$. This means that Const_T_Wt (h', k', s'') will be

computed for all valid values of s before $\text{Const_T_Wt}(i, j, s)$ because elements in $\text{Essential_Nodes}(T_1)$ and $\text{Essential_Nodes}(T_2)$ are stored in their increasing orders. Hence, $\text{Const_T_Wt}(h, k, s')$ is available for all valid values of s' after $\text{Const_T_Wt}(h', k', s'')$ is computed for all valid values of s'' . So (i) always holds and this proves the theorem. ***

5.5 Time and Space Complexities

We shall now analyze the time and space complexities of the algorithm.

THEOREM 5.6

If for a tree T , $\text{Span}(T)$ is defined as the $\text{Min}\{\text{Depth}(T), \text{Leaves}(T)\}$, the time and space complexities of this algorithm are :

$$\text{Time : } O(|T_1| \cdot |T_2| \cdot \text{Min}\{|T_1|, |T_2|\}^2 \cdot \text{Span}(T_1) \cdot \text{Span}(T_2))$$

$$\text{Space : } O(|T_1| \cdot |T_2| \cdot \text{Min}\{|T_1|, |T_2|\}).$$

PROOF

For space complexity, we use a permanent array for Const_T_Wt and a temporary array for Const_F_Wt . Each of these two arrays requires space $O(|T_1| \cdot |T_2| \cdot \text{Min}\{|T_1|, |T_2|\})$ since the first component of these arrays is $O(|T_1|)$, the second component of these arrays is $O(|T_2|)$, and the last component of these arrays is the maximum number of substitution that are feasible, which is $O(\text{Min}\{|T_1|, |T_2|\})$.

With regard to the time complexity, we first observe that the preprocessing takes linear time to precompute the arrays $\delta[]$ and $\text{Essential_Nodes}[]$ for both trees. Also, note that if the array Const_T_Wt is computed, the constrained tree editing distance, $D_t(\dots)$, can be computed using Algorithm *Constrained_Tree_Distance* in time ld , which in the worst case is $O(\text{Min}\{|T_1|, |T_2|\})$ which is also linear. Hence, the dominant term involves computing the array $\text{Const_T_Wt}(i, j, s)$ for all relevant i, j , and s . This constrained

distance dynamic programming algorithm involving the subtrees rooted at i and j , respectively, involves : $O(\text{Size}(i) \cdot \text{Size}(j) \cdot \text{Min}\{\text{Size}(i), \text{Size}(j)\}^2)$ computation.

Therefore, total time required is :

$$\begin{aligned}
 & \sum_{i=1}^{|\text{Essential_Nodes}(T_1)|} \sum_{j=1}^{|\text{Essential_Nodes}(T_2)|} \text{Size}(i) \cdot \text{Size}(j) \cdot \text{Min}\{\text{Size}(i), \text{Size}(j)\}^2 \\
 & \leq \sum_{i=1}^{|\text{Essential_Nodes}(T_1)|} \sum_{j=1}^{|\text{Essential_Nodes}(T_2)|} \text{Size}(i) \cdot \text{Size}(j) \cdot \text{Min}\{|T_1|, |T_2|\}^2 \\
 & \leq \text{Min}\{|T_1|, |T_2|\}^2 \cdot \sum_{i=1}^{|\text{Essential_Nodes}(T_1)|} \text{Size}(i) \cdot \sum_{j=1}^{|\text{Essential_Nodes}(T_2)|} \text{Size}(j)
 \end{aligned}$$

Since the sets **Essential_Nodes** is identical to the set defined in [ZS89] as the **LR_keyroots**, we see from Theorem 2 of [ZS89] that :

$$\sum_{i=1}^{|\text{Essential_Nodes}(T)|} \text{Size}(i) \leq |T| \cdot \text{Min}\{\text{Depth}(T), \text{Leaves}(T)\}$$

Since $\text{Span}(T)$ is equal to $\text{Min}\{\text{Depth}(T), \text{Leaves}(T)\}$, the time complexity of the algorithm is $O(|T_1| \cdot |T_2| \cdot \text{Min}\{|T_1|, |T_2|\}^2 \cdot \text{Span}(T_1) \cdot \text{Span}(T_2))$, and the theorem is proved. ***

5.6 Conclusions

In this chapter, we have considered the problem of editing a tree T_1 to a tree T_2 subject to a set of specified edit constraints. The edit constraint is fairly arbitrary and can be specified in terms of the number and type of edit operations desired in optimal

transformation. The way by which constraint τ can be specified has been proposed. Also, the technique to compute $D_\tau(T_1, T_2)$, the edit distance subject to constraint τ , has been presented. This set, τ , is typically a subset of the number of possible substitutions.

Given the trees T_1 and T_2 , $D_\tau(T_1, T_2)$ the array of constrained edit distances $\text{Const_T_Wt}(i, j, s)$ can be computed using dynamic programming, where $\text{Const_T_Wt}(i, j, s)$ is the constrained distance between the subtrees rooted at i and j subject to the constraint that exactly s substitutions are performed. If for a tree T , $\text{Span}(T)$ is defined as the $\text{Min}\{\text{Depth}(T), \text{Leaves}(T)\}$, the scheme to compute this array requires $O(|T_1| \cdot |T_2| \cdot \text{Min}(|T_1|, |T_2|)^2 \cdot \text{Span}(T_1) \cdot \text{Span}(T_2))$ time. The space required for this computation is $O(|T_1| \cdot |T_2| \cdot \text{Min}(|T_1|, |T_2|))$. From the array $\text{Const_T_Wt}(i, j, s)$ the constrained edit distance $D_\tau(T_1, T_2)$ can be obtained in linear time.

The algorithm presented in this chapter has been implemented and rigorously tested.

VI. Summary and Open Problems

In this thesis, we have considered a problem that is fundamental in syntactic and structural pattern recognition. The problem is that of quantifying the similarity (or dissimilarity) between pairs of patterns which are represented as strings or trees.

To provide the proper context, we initially studied the string editing problem. Various similarity and dissimilarity measures involving two strings were discussed and a variety of representative algorithms which computed these indices were described.

The heart of the thesis considered the tree editing problem which has not been studied extensively in the literature. In Chapter III, all the reported solutions for solving the tree editing problem have been discussed. The solutions and their corresponding limitations are listed below :

- (i) Selkow *et. al.* [Se77] presented a solution which solved the problem when the editing was restricted to the leaves of tree.
- (ii) Tai [Ta79] reported a scheme which solved the problem whenever the deletion and insertion operations were not permitted at the root of the tree, but was permitted at all the internal nodes.
- (iii) The final algorithm which tackles the general tree editing problem is due Zhang and Shasha [ZS89]. This algorithm too has been explained in detail in Chapter III.

The entire problem of tree editing was thereafter re-visited in Chapter IV. Rather than merely evaluate the distance between two trees, we have attempted to define various families of similarity and dissimilarity measures between the trees in question. This has been done by defining an abstract measure, $\Omega(T_1, T_2)$, of comparison between two trees

T_1 and T_2 . This measure has been defined in terms of a set of elementary inter-symbol measures, $\omega(\dots)$, and two abstract operators $\oplus$ and $\otimes$. By appropriately choosing the concrete values of these two operators and for $\omega(\dots)$, this measure can be used to define the following quantities involving the trees T_1 and T_2 :

- (i) The Size of their Largest Common Sub-Tree,
- (ii) The Size of their Smallest Common Super-Tree,
- (iii) The Confusion Probability ($\text{Prob}(T_2|T_1)$), the probability of receiving T_2 given that T_1 was transmitted across a channel causing independent substitution and deletion errors, and,
- (iv) The quantity $P(T_i|Q)$, the probability of T_i being the transmitted tree given that Q is the received tree containing independent substitution, insertion and deletion errors and that the transmitted tree is an element from a finite dictionary $\mathcal{H}$.

Apart from the above quantities being formally defined in terms of the abstract measure, in Chapter IV we also gave a common computational scheme by which all these quantities can be evaluated.

Finally, in Chapter V, we have studied a far more complicated tree editing problem, namely, the Constrained Tree Editing Problem. Unlike the string editing problems, all existing tree editing algorithms consider the editing of one tree, T_1 , and transforming it to another tree T_2 , with the edit processes being absolutely unconstrained. In this chapter we studied the problem of editing T_1 to T_2 subject to the constraint that the number of individual edit operations done is specified by a user-defined constraint. A general fast algorithm which solves this problem has been derived.

6.1 Open Problems

Apart from the direct application of the techniques which we have introduced in this thesis to a practical pattern recognition problem, we shall now list the two most outstanding of the theoretical open problems that remain with regard to tree editing.

6.1.1 Non-numeric Similarity Measures between Two Trees

The first open problem that we see is related to defining and computing non-numeric measures relating the trees T_1 and T_2 . Examples of these measures are the set of their sub-trees and their super-trees and the set of their Largest Common Sub-trees and their Smallest Common Super-Trees. Interestingly enough, in [KO83], the abstract formulation for the measure involving strings could capture all the numeric and non-numeric measures that involved these strings. We believe that techniques parallel to those introduced in Chapter IV in this thesis can be utilized to define and compute analogous non-numeric (symbolic) measures between two trees. This is not so directly obtained as in the case of strings [KO83], because, in the case of trees, it appears as if we need two *independent* abstract operators for " $\otimes$ " to handle the insertion operation. Indeed, this is to ensure that although the physical insertion operation maintains the ancestor and sibling properties of the trees consistently, the corresponding abstract operator " $\otimes$ " used to define the abstract measure must be defined in such a way so as to simultaneously maintain these properties. We are currently investigating how these operators can be defined.

6.1.2 Probabilistic Correction of Trees

The second open problem that we see is related to defining and computing the transmission-reception probability of receiving T_2 given that T_1 is transmitted over a channel which causes all the three types of errors. This is analogous to the result derived for strings in [KO82] in which the distribution for the number of insertions can be completely arbitrary. Thus we consider the problem of transmitting a tree T_1 through a

noisy channel which causes substitution, deletion and insertion errors. Since the channel can deform T_1 by all these three types of errors, T_2 , the noisy tree received at the receiving end of the channel can be of any arbitrary size N , $N \geq 0$. Of the three types of errors that can occur, insertion errors are distinct from both deletion and substitution errors. This is because, if the tree we transmitted is of size $M \geq 0$, the number of deletions and substitutions that can take place is at most M . However, the number of insertion errors that can occur can be indefinite. Hence, when we try to compute the probability of receiving T_2 given that T_1 is transmitted across this type of noisy channel, the number of trees that could potentially be encountered is infinite. This is why this problem is more complicated than the one presented in Chapter IV. However we believe that technique analogous to those introduced in *both* Chapter IV and Chapter V can be augmented to define and solve the "Probabilistic Correction of Trees" Problem.

Although the same problem has been studied by Kashyap and Oommen [KO82] for a pair of strings, their results cannot be directly applied to the case of trees. This is because of the complex issues encountered if we required that every *position* for the insertion errors (which the noisy channel creates) is "equal-likely". Indeed, in this case, the problem of even modelling "how" the errors are caused is far from trivial.

One of the unsuccessful strategies by which we have tried to tackle this problem is to try to resort to the underlying technique used by Kashyap and Oommen [KO82]. Instead of transmitting the tree itself, let us suppose that we transmitted the postorder sequence of the tree under consideration through the channel. From the point of view of the channel, since deletion and substitution errors can be done independently, all we have to explicitly take care of is the modelling of the insertion errors. Of course, as in [KO82], the number of insertion errors caused can be modelled to have any valid probability distribution. But if the position of these insertions is uniformly distributed, one would have hoped that we could randomly select the locations between any two consecutive node symbols in the postorder sequence of the tree transmitted. In other words, if the tree transmitted is of size M , since

the postorder sequence is also of length M , the number of different locations for the insertion errors *in the postorder sequence* is equal to $M+1$. Thus, it would have been desirable if the insertions could have been modelled to have occurred uniformly at one of the $(M+1)$ locations as depicted in Figure 6.1.

Input Tree :

Postorder Sequence : b d e c a

Possible positions of generating
a single insertion error :

$$\text{Prob \{ Insertion at position } p, 1 \leq p \leq 6 \} = \frac{1}{6} .$$

Figure 6.1 : Pictorial description of the "Postorder Sequence" approach to Modelling the Channel generating the noisy trees.

By virtue of the above, it is clear that the position of a single insertion error in the postorder sequence can be obtained in an "equal-likely" fashion by the model described above. Unfortunately, the laws of probability do not ensure that the probability of a single insertion error being generated, is **uniformly** distributed when the received postorder

sequence is mapped back into a tree structure. This is because each erroneous postorder sequence obtained does not map to the same number of tree structures. Thus, even though the probability of each resultant garbled postorder sequence being received is uniform, the trees which are obtained as a consequence of this model are not uniformly distributed.

Example 6.1 clarifies this point. In this case the original tree is of size five. Although a single insertion can be caused in the postorder sequence at one of the six possible locations, a straightforward enumeration shows that 16 possible resultant erroneous trees can be generated as a consequence of this single insertion error. The example clearly shows that not all of these are equally likely.

EXAMPLE 6.1 :

Original Tree

Postorder Sequence : b d e c a

Possible positions of generating a single insertion error :

16 Possible trees which can be obtained after inserting the node "f" :

(a)

Postorder sequence :
fbdeca.

(b)

Postorder sequence :
bfdeca.

(c)

Postorder sequence :
bdfeca.

(d)

Postorder sequence :
bfdeca.

(e)

Postorder sequence :
bdfeca.

(f)

Postorder sequence :
bdefca.

(g)

Postorder sequence :
bdfeca.

(h)

Postorder sequence :
bdefca.

(i)

Postorder sequence :
bdefca.

(j)

Postorder sequence :
fbdeca.

(k)

Postorder sequence :
bfdeca.

(l)

Postorder sequence :
bdecfa.

(m)

Postorder sequence :
bfdeca.

(n)

Postorder sequence :
bdecfa.

(o)

Postorder sequence :
bdecfa.

(p)

Postorder sequence :
bdecfa.

Possible postorder sequences
which can result from
inserting the node symbol "f"

No. of different trees
which can be generated

fbdeca	2
bfdeca	4
bdfeca	3
bdefca	3
bdecfa	3
bdecfa	1

If we used the Postorder technique for modelling the insertions we observe the following :

$$\text{Prob \{ Insertion occurs at location (1) \}} = \frac{1}{6},$$

Number of tree represented by the Postorder sequence "fbdeca" = 2.

$$\text{This implies that Prob \{ resulting tree is (a) \}} = \frac{1}{6} * \frac{1}{2} = \frac{1}{12}.$$

Arguing in a similar way we see that :

$$\text{Prob \{ Insertion occurs at location (2) \}} = \frac{1}{6},$$

Number of tree represented by the Postorder sequence "bfdeca" = 4.

This, in turn, implies that $\text{Prob} \{ \text{resulting tree is (b)} \} = \frac{1}{6} * \frac{1}{4} = \frac{1}{24}$.

But if an equally likely distribution was desirable we would have required that:

$$\text{Prob} \{ \text{resulting tree is (a)} \} = \text{Prob} \{ \text{resulting tree is (b)} \} = \frac{1}{16}.$$

Thus we see that the modelling of the process of generating insertions using the postorder sequence of the underlying tree is inadequate. ***

This problem of modelling the noisy channel and computing the transmission-reception probabilities of the channel thus seems to be irreducible to that of the underlying string transmission-reception problem. We are currently investigating other approaches to evaluating this probability.

References

- [AHD76] A. V. Aho, D. S. Hirschberg, and J. D. Ullman, "Bounds on the complexity of the longest common subsequence problem", *J. ACM*, Vol 23: pp. 1-12 (Jan 1976).
- [Al67] C. N. Albigra, "String similarity and misspellings", *Comm. ACM*, Vol 10 : pp. 302-313 (May 1967).
- [AG87] A. Apostolico and C. Guerra, "The longest common subsequence problem revisited", *Algorithmica*, Vol : 2, pp. 315-336 (1987).
- [ADKF70] V. L. Arlazarov, E. A. Dinic, M. A. Kronrod, and I. A. Faradzer, "On economic construction of the transitive closure of a directed graph", (English translation) *Soviet Math. Dokl.* Vol 11 : No. 5, pp. 1209-1210 (1970).
- [BSSU74] W. A. Beyer, M. L. Stein, T. F. Smith, and S. M. Ulam, "A molecular-sequence metric and evolutionary trees", *Mathematical Biosciences*, Vol 19 : pp. 9-25 (1974).
- [Da64] F. J. Damerau, "A technique for computer detection and correction of spelling errors", *Comm. ACM*, Vol 7 : pp. 171-176 (March 1964).
- [Fu68] K. S. Fu, *Sequential methods in pattern recognition and machine learning*, Academic Press, New York, 1968.
- [Fu82] K. S. Fu, *Syntactic Pattern Recognition and Application*, Prentice-Hall, 1978.
- [GK82] W. B. Goad and M. I. Kanehisa, "Pattern recognition in nucleic acid sequences. I. A general method for finding local homologies and symmetries", *Nucleic Acids Research*, Vol : 10, No. 1, pp. 247-263 (1982).

-
- [GT78] R. C. Gonzalez and M. G. Thomason, *Syntactic Pattern Recognition - An Introduction*, Addison-Wesley, 1978.
- [Ha88] F. Hadlock, "Minimum detour methods for string or sequence comparison", *Congr. Numer.*, Vol 61 : pp. 263-274 (1988).
- [HD80] P. A. V. Hall and G. R. Dowling, "Approximate string matching", *Comput. Sur.*, Vol 12 : pp. 381-402 (Dec 1980).
- [Hi75] D. S. Hirschberg, "A linear space algorithm for computing maximal common subsequences", *Comm. ACM*, Vol 18 : pp. 341-343 (June 1975).
- [Hi77] D. S. Hirschberg, "Algorithms for the longest common subsequence problem", *J. ACM*, Vol 24 : pp. 664-675 (Oct 1977).
- [Hi78] D. S. Hirschberg, "An information-theoretic lower bound for the longest common subsequence problem", *Information Processing Letter*, Vol 7 : pp. 40-41 (Jan 1978).
- [HS77] J. W. Hunt and T. G. Szymanski, "A fast algorithm for computing longest common subsequences", *Comm. ACM*, Vol 20 : pp. 350-353 (May 1977).
- [KC72] L. Kanal and B. Chandrasekaran, "On linguistic, statistical and mixed models for pattern recognition", in *Frontiers of Pattern Recognition* (S. Watanabe, ed.), Academic Press, New York, 1982.
- [KO82] R. L. Kashyap and B. J. Oommen, "Probabilistic correction of strings", *Proceedings of the I.E.E.E. Trans. on Pattern Recognition and Image Processing*, pp. 28-33 (June 1982).
- [KO83a] R. L. Kashyap and B. J. Oommen, "The noisy substring matching problem", *I.E.E.E. Trans. of Software Eng.*, Vol SE-9, No3 : pp. 365-370 (May 1983).
- [KO83b] R. L. Kashyap and B. J. Oommen, "A common basic for similarity measures involving two strings", *Intern. J. Computer Math.*, Vol 13 : pp. 17-40 (1983).

-
- [KO84] R. L. Kashyap and B. J. Oommen, "Spelling correction using probabilistic methods", *Pattern Recognition Letters*, Vol 2 : pp. 147-154 (1984).
- [KT82] M. Kunze and G. Thierrin, "Maximal common subsequences of pairs of strings", *Congressus Numerantium*, Vol 34 : pp. 299-311 (1982).
- [Le66] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals", *Sov. Phys. Dokl.*, pp. 707-710 (1966).
- [LW75] R. Lowrance and R. A. Wagner, "An extension of string-to-string correction problem", *J. ACM*, Vol 22: pp. 177-183 (April 1975).
- [Lu79] S. Y. Lu, "A tree-to-tree distance and its application to cluster analysis", *I.E.E.E. Trans. Pattern Analysis and Machine Intelligence*, Vol : PAMI 1, No. 2, pp. 219 - 224 (April 1979).
- [Lu84] S. Y. Lu, "A tree-matching algorithm based on node splitting and merging", *I.E.E.E. Trans. Pattern Analysis and Machine Intelligence*, Vol : PAMI 6, No. 2, pp. 249 - 256 (March 1984).
- [Ma78] D. Maier, "The complexity of some problems on subsequences and superSequences", *J. ACM*, Vol 25 : pp. 322-336 (April 1978).
- [MP80] W. J. Maser and M. S. Paterson, "A faster algorithm computing string edit distances", *J. of Comp. and Syst. Sci.*, Vol 20 : pp. 18-31 (1980).
- [MF70] J. M. Mendel and K. S. Fu, *Adaptive, Learning and Pattern Recognition System : Theory and Applications*, Academic Press, New York, 1970.
- [Me72] W. Meisel, *Computer-Oriented Approaches to Pattern Recognition*, Academic Press, New York, 1972.
- [Mu80] A. Mukhopadhyay, "A fast algorithm for the Longest-Common-Subsequence Problem", *Inform. Sci.*, Vol: 20, pp. 69-82 (1980).
- [My86] E. W. Myers, "An $O(ND)$ difference algorithm and its variations", *Algorithmica*, Vol 1 : pp. 251-266 (1986).

-
- [NKY82] N. Nakatsu, Y. Kambayashi, and S. Yajima, "A longest common subsequence algorithm suitable for similar text string", *Acta Informatica*, Vol : 18, pp. 171-179 (1982).
- [Ni65] N. J. Nilsson, *Learning Machines : Foundations of trainable Pattern-Classifying System*, McGraw-Hill, New York, 1965.
- [Oo86] B. J. Oommen, "Constrained string editing", *Inform. Sci.*, Vol 40 : pp. 267-284 (1986).
- [Oo87] B. J. Oommen, "Recognition of noisy subsequences using constrained edit distances", *I.E.E.E. Trans. on Pattern Anal. and Mach. Intell.*, Vol PAMI-9, No. 5 : pp. 676-685 (Sept 1987).
- [OF91] Oommen, B.J. and Floyd E.T., "An Improved Algorithm for the Recognition of Noisy Subsequences", *Proceedings of the 1991 IASTED International Symposium on Artificial Intelligence Applications and Neural Networks*, July 1991, Zurich, pp. 145-147.
- [Sa72] D. Sankoff, "Matching sequences under deletion-insertion constraints", *Proceedings of the National Academy of Sciences of the U.S.A.*, Vol 69 : pp.4-6 (1972).
- [SK83] D. Sankoff and J. B. Kruskal, *Time wraps, string edits, and macromolecules : Theory and practice of sequence comparison*, Addison-Wesley (1983).
- [Se77] S. M. Selkow, "The tree-to-tree editing problem", *Information Processing Letters*, Vol 6, No. 6, pp. 184 - 186 (Dec. 1977).
- [Ser74] P. H. Sellers, "An algorithm for the distance between two finite sequences", *J. of Combin.Theory (A)* Vol 16 : pp. 253-258 (1974).
- [SW80] T. F. Smith and M. S. Waterman, "New stratigraphic correlation techniques", *Journl of Geology*, Vol 88 : pp. 451-457 (1980).
- [Ta79] K. C. Tai, "The tree-to-tree correction problem", *J. Assoc. Comput. Mach.* Vol 26, pp. 422 - 433, (1979).

-
- [Uk85a] E. Ukkonen, "Finding approximate patterns in strings", *J. Algorithm*, Vol 6 : pp. 132-137 (1985).
- [UK85b] E. Ukkonen, "Algorithms for approximate string matching", *Information and Control*, Vol : 64, pp. 100-118 (1985).
- [WF74] R. A. Wagner and M. J. Fischer, "The string-to-string correction problem", *J. ACM*, Vol 21 : pp. 168-173 (Jan 1974).
- [WL83] W. J. Wilbur and D. J. Lipman, "Rapid similarity searches of nucleic acid and protein data banks", *Proceedings of the National Academy of Science of the U.S.A.*, Vol 80 : pp. 726-730 (1983).
- [WC76] C. K. Wong and A. K. Chandra, "Bounds for the string editing problem", *J. ACM*, Vol 23 : pp. 13-16 (Jan 1976).
- [WMM91] S. Wu, U. Manber, and E. W. Myers, *Improving the running times for some string-matching problems*, TR91-20, Department of Computer Science, The University of Arizona (1991).
- [WMMM90] S. Wu, U. Manber, E. W. Myers, and W. Miller, "An $O(NP)$ sequence comparison algorithm", *Information Processing Letters*, Vol : 35, pp. 317-323 (1990).
- [ZS89] K. Zhang and D. Shasha, "Simple fast algorithms for the editing distance between trees and related problems", *SIAM J. Comput.*, Vol : 18, No. 6, pp. 1245 - 1262 (Dec. 1989).

END

2/3·12·92

FIN